Sequential Optimization of Asynchronous and Synchronous Finite-State Machines: Algorithms and Tools

Robert Mack Fuhrer

Submitted in partial fulfillment of the

requirements for the degree

of Doctor of Philosophy

in the Graduate School of Arts and Sciences

COLUMBIA UNIVERSITY

1999

Sequential Optimization of Asynchronous and Synchronous Finite-State Machines: Algorithms and Tools

Approved by
Dissertation Committee:

This thesis is dedicated to:

the gift of music

pizza

Beef Wellington

the wines of Bordeaux

mi amore

Contents

List of Tables

List of Figures

Acknowledgments

I would like to acknowledge several people who have made significant contributions to my life during the course of this thesis, in the form of technical discourse, and in other ways.

First, I'd like to thank my advisor, Steven Nowick, whose guidance and encouragement of my research efforts has been indispensable. Steve's unrelenting pursuit of excellence prodded me at times when I needed it; and his patient, methodical approach helped me slow down when I needed that too. The breadth of his knowledge in the field of CAD for asynchronous and synchronous circuits has been inspiring, and has also served as a source of pointers to timely and useful information.

Thanks also go to IBM, the National Science Foundation, and the Alfred P. Sloan Foundation, whose generous grants partially funded the work which this dissertation reports.

Earlier in my graduate education, several faculty members provided me with inspiration and motivation to pursue scientific research in general, and my chosen field of CAD for VLSI circuits in particular.

Courses in VLSI given by Charles Zukowski early in my studies fascinated me, and spurred me on to learn more about VLSI, and especially, computer-aided design.

Steven Unger's no-nonsense, intuitive approach to asynchronous circuits and systems, along with his wealth of knowledge of the early history of the area, were and are an inspiration. His graduate course in asynchronous circuits was one of the best courses

I encountered, and was responsible for my awareness of asynchronous circuits and the rich intellectual challenges they present.

I'd also like to thank Henryk Wozniakowski and John Kender, whose obvious dedication to the art of teaching resulted in some of the best educational experiences I've had. It was courses like theirs that gave me some of my first insights into the sheer joy of scientific discovery.

Tiziano Villa graciously agreed to be on the defense committee at the last moment, when an emergency necessitated a change. In spite of the timing, Tiziano deftly assimilated the material, and offered useful criticism of both the work and the dissertation. I am very grateful for his efforts.

Sal Stolfo provided kind words of encouragement regarding my dissertation, at a time when I needed them. To Sal, I give my thanks.

The students in my research group (John Cheng, Luis Plana, Montek Singh, Michael Theobald, and Tibi Chelcea) have been a genuine pleasure to work with.

My colleagues at IBM, including Sesh Murthy, Fred Wu, Bob Risch, Fateh Tipu, Marshall Schor, David Jameson, Jim Wright, Steve Abrams, Don Pazel, Danny Oppenheim, Bill Jecusco, and Brian White, have been both good friends and stimulating associates. I value them all highly.

The IBM Research Division itself deserves thanks as well, for providing a top-notch research environment within which to hone my skills and develop my research potential. The practical experience I gained at IBM has been instrumental in my development as a scientist.

I thank my family, who patiently awaited my re-emergence from the isolation that the demands of concurrent work and graduate studies required.

A special note of appreciation goes to Terry Rico O'Reilly, who has been my mentor in many ways. Her support has been of immeasurable value to me. Her passion for and knowledge of music, the arts and humanities has been a veritable wellspring of

nourishment for my artistic side. More importantly, she has helped me find my path to happiness. I still strive to put her lessons into practice.

Last but by no means least, I owe the most to my wife Cindy. In every sense, I can only speak of her place in my life as one would speak of a vital organ; one cannot begin to comprehend what life would be like without it. Cindy has been the most positive element in my life, and I am eternally grateful. I now begin with zeal the most pleasant "task" I have ever faced: to repay her inasmuch as it is possible! To Cindy, all my love and devotion!

ABSTRACT

Sequential Optimization of Asynchronous and Synchronous Finite-State Machines: Algorithms and Tools

Robert Mack Fuhrer

Asynchronous digital systems have significant potential over their synchronous counterparts. For example, they offer the potential to achieve high performance and low power consumption, embodied in modular implementations. As such, they address several difficult problems faced by the designers of large-scale synchronous digital systems: power consumption, worst-case timing constraints, and engineering issues associated with the use of a global clock. Moreover, while for synchronous systems these problems are exacerbated by increasing system size, asynchronous systems promise to scale more gracefully.

The design of asynchronous systems poses unique challenges, however, which has impeded designers' acceptance of asynchronous implementations for mainstream digital system development. In particular, these challenges make hand design even less practical for asynchronous circuits than for synchronous circuits. Thus, computer-aided design (CAD) tools are of paramount importance to asynchronous systems designers. Unfortunately, the development of suitably potent CAD algorithms, tools and complete synthesis paths for asynchronous circuits has lagged those of the synchronous domain. Designers have had little choice but to relegate asynchronous circuits to niche applications.

This thesis therefore consists of three contributions to the field of sequential optimizations for finite-state machines: 1) provably optimal algorithms for the synthesis and optimization of asynchronous finite state machines (FSM's), 2) practical software implementations of these algorithms, and 3) a complete CAD package binding these tools into a state-of-the-art technology independent synthesis path for burst-mode asynchronous circuits. Throughout, real-world industrial designs are used as benchmark circuits to validate the tools' usefulness. As an additional benefit, some of the theory and tools developed provide methods for the optimization of synchronous FSM's.

Chapter 1

Introduction

This thesis addresses several problems in *sequential optimization*, that is, transformations on finite-state machines which produce machines having equivalent behaviour, but whose implementations are either more economical, faster, or both. In particular, the focus of this thesis is a suite of algorithms and tools which synthesize small, high-performance realizations of finite-state machines, given a suitable specification. First and foremost, the emphasis is on asynchronous controller circuits.

This chapter proceeds as follows. First, Section 1.1 provides some basic motivation on why asynchronous designs are promising alternatives to traditional synchronous designs. Next, Section 1.2 describes various forms of asynchronous controllers, which are the focus of the bulk of the thesis. Section 1.3 offers a general overview of sequential synthesis and traditional approaches. Then, Section 1.4 presents one of the central themes of this thesis: obtaining stronger synthesis methods by merging steps. Section 1.5 gives some insight into the unique problems asynchronous designs present to the synthesis process. A brief sketch of the state of the art in asynchronous synthesis is also presented. The main contributions of the thesis are then described in Section 1.6. Finally, the structure of the remainder of the thesis is presented in Section 1.7.

1.1 The Case for Asynchronous Circuits

Sequential digital systems are typically constructed as *synchronous* systems, that is, as systems which have a single synchronizing clock signal. The clock signal pulses at regular intervals, and is distributed throughout the system, thus serving to keep all of the components operating in lock-step. All processing in each step must complete within the clock period, or the circuit will fail to function properly.

This implementation strategy simplifies the system's timing, by quantizing time into uniform, discrete steps. As a result, one can largely ignore timing issues while deriving an implementation for a given behavioural specification, and concentrate on the intended functionality.

However, the use of a global clock presents several increasingly challenging problems [141][36]:

clock skew The clock must reach every component very nearly simultaneously in order for synchronization to be achieved. In very large circuits, such as modern microprocessors, the distribution of the clock signal to tens or hundreds of thousands of components produces complicated signal paths of varying length. This variation in length produces a corresponding variation in propagation delay, known as clock skew. If this skew is more than a small fraction of the clock period itself, the system is likely to fall out of sync and malfunction. Although clock distribution networks have been constructed for large circuits using clock rates upwards of 500 MHz, the engineering effort required is considerable [11][50][20], which impacts time-to-market. Moreover, the problem becomes more difficult to solve at ever-higher clock speeds. To make matters worse, it is expected that future systems-on-a-chip will require multiple clock domains,[1] with even more complex skew problems.

power consumption The distribution network for the clock signal itself, as well as registers used to hold data stable in between clock events, consumes power. In fact, in circuits using very fast clocks, such as the 500 MHz DEC Alpha, this can be a considerable portion (up to 40%! [50]) of the circuit's total power consumption. Moreover, in CMOS circuits, almost all of the power consumed is dissipated during transitions. Hence, in synchronous CMOS systems, the clock signal causes many CMOS components to consume power even when they are not performing "useful work". Clock gating [59] has been recently used to reduce power consumption by preventing clock pulses from stimulating circuits unnecessarily. Unfortunately, this technique must be designed into the circuit, and often introduces additional clock skew. By comparison, asynchronous designs achieve the same effect "for free", at arbitrary granularity.

worst-case timing In order to use a single clock signal, all components in the system must operate within the same period of time. As a result, slower components rob the system of the opportunity to take advantage of faster components. Typically, therefore, synchronous designers struggle to balance the system so that the "critical" (slowest) path does not slow the system intolerably [41].

electromagnetic interference and noise In a synchronous system, the clock signal triggers periodic bursts of activity. These bursts induce peaks of power-supply current, with concomitant peaks in power-supply noise and electromagnetic (EM) emissions. Such noise and EM bursts can be particularly problematic for sensitive analog components that are nearby, e.g., digital-to-analog converters or radio receivers [141].

asynchronous interfacing Even synchronous systems frequently must interface to inherently asynchronous components, e.g., memories, external devices, and so on. Synchronous system designers must take great pains to ensure that such hybrid interfaces operate correctly under all possible circumstances.

modularity Because a synchronous subsystem must meet the global timing constraint to work alongside other subsystems, it is often difficult or impossible to re-use its implementation in a context where timing requirements are more stringent (i.e. with a shorter clock period). Likewise, introducing a faster component into a synchronous system does nothing to speed up the operation of the entire system, unless *all* subsystems are made to operate at the faster speed. The trend toward systems-on-a-chip and the desire to re-use large circuit implementations (e.g. microprocessor cores) make modularity a critical design goal.

In addition, many of these difficult problems are exacerbated by the continuing trend to produce increasingly large-scale digital systems.

An *asynchronous* system, by comparison, uses no clock to synchronize the operation of its various components. In essence, each component performs *local* synchronization, as needed, with the neighboring components with which it directly communicates. As a result, asynchronous systems promise to either eliminate (as in the case of clock skew or worst-case timing) or ameliorate (as in the case of power consumption) the above problems. For the same reasons, asynchronous systems also tend to scale better to larger sizes than do synchronous ones.

Several recent successes substantiate the benefits of asynchronous design:

- A *fully asynchronous 80C51 microcontroller*, designed by Philips Research in collaboration with Philips Semiconductor [142], which consumes *one-quarter of the power* of the synchronous implementation. This design has been incorporated into a line of integrated circuits (IC's) now on the market for use in pagers.

- An experimental *asynchronous instruction decoder* developed at Intel, exhibiting *three to four times the performance* of the highly-tuned synchronous version [21]. This design demonstrates the potential for asynchronous circuits to exploit data-dependent computational delays to reduce average-case delay.

[1] i.e., distinct portions of the chip's circuitry will operate under different clocks

- The *Amulet2e*, an embedded micro-processor chip developed at the University of Manchester [57], whose asynchronous components include an ARM-compatible execution core and a level-1 cache. Although its power dissipation is only somewhat lower than the synchronous versions of this chip, the absence of a clock allows *microwatt power consumption when idle.* The asynchronous chip also features near instantaneous reaction to interrupts when idle. By comparison, the synchronous versions take considerable time re-synchronizing the clock when awakening.

- A *digital filter bank*, part of a synchronous digital hearing aid, was re-implemented as a fully asynchronous system by the Technical University of Denmark and Oticon, Inc. [100]. The asynchronous design dissipates *five times less power* than its synchronous counterpart.

- A *self-timed RISC processor design*, **STRiP** [41], based on the MIPS-X architecture, exhibiting a *factor of two speedup* over the MIPS-X, due solely to its asynchronous characteristics.

Additional examples include a high-performance asynchronous differential equation solver [151], a high-performance asynchronous decompression engine for embedded microprocessors [7], a low-power Reed-Solomon error corrector for digital audio [10], a high-performance self-timed divider circuit [148], an infrared communications chip [86], and a high-performance adaptive routing chip for parallel processing [132].

1.2 Asynchronous Controllers

This section describes various classes of asynchronous machines and classical modes of operation. Particular attention is paid to asynchronous controllers, which are the focus of this thesis. Finally, more extensive detail is given on the design style known as *burst mode*, which is the target of the various optimization algorithms in the thesis. For an overview of some of the major design and implementation problems associated with asynchronous synthesis, refer to Chapter 2.

1.2.1 Classical Asynchronous FSM Models

Asynchronous circuits do not use a clock signal to synchronize their operation; hence, this synchronization must be accomplished in various other ways. Essentially, a spectrum of basic approaches exists, ordered according to the degree of timing assumptions required in order to assure correct operation (see [36] for a survey). Broadly speaking, the fewer timing assumptions are made, the more robust is the component's operation with respect to its environment. In particular, the most robust approaches are exceedingly tolerant of variations in ambient conditions (e.g. temperature and supply voltage), manufacturing process parameters (which affect gate delays), as well as latencies and skews in the surrounding circuitry.

Unfortunately, as robustness increases, the difficulty in designing and implementing the individual components also rises, and, frequently, performance is sacrificed. Further, the most robust class of circuits even place constraints on communication protocols among components[2]. Thus, these trade-offs give rise to several classes of asynchronous machines, which operate under varying degrees of timing constraints.

Delay Models

Each class assumes of one of several *delay models*, which prescribe allowable values for a circuit's delays. The *unbounded delay model* allows any finite delay, while the *bounded delay model* requires that a delay be shorter than some specific finite value. Finally, the *fixed delay model* (the least common of the three) assigns a delay a specific value.

[2] E.g., delay-insensitive circuits require an acknowledgement for *every* output transition.

Classes of Asynchronous Circuits

At one end of the spectrum are *delay-insensitive* (DI) circuits [26], which operate correctly under arbitrary gate and wire delays (i.e., under the unbounded delay model). Such circuits are quite difficult to design from simple gates, but have formal properties which facilitate verification of DI networks [46] and synthesis by translation from programming language specifications [48], for example. The difficulty in using simple gates has led some [48] to fashion complex gates that make use of internal timing assumptions, in order to construct larger circuits than would otherwise be possible.

Quasi-delay-insensitive circuits are DI circuits that make use of *isochronic forks*. An isochronic fork is a fan-out from a single component to several destinations, for which the delay to all destinations is assumed equal. This timing assumption is disallowed in true DI circuits, where the delays even to different destinations of a single wire can not be assumed in any way correlated. This slight weakening of delay insensitivity, however, allows a larger class of circuits to be built [9, 87] from simple gates than is the case for pure DI circuits.

Next, *speed-independent* (SI) circuits [96] can tolerate arbitrary gate delays, but assume negligible wire delays.

Self-timed circuits [127] are essentially DI circuits composed of circuit *elements*, defined by so-called "equipotential regions." These regions have controlled or negligible wire and gate delays, and therefore internally make use of timing assumptions (e.g., they are locally SI). However, self-timed circuits impose no timing constraints on the communications among elements, and so are globally delay insensitive.

The most general class of circuits are *asynchronous circuits* [139]. This class often makes timing assumptions regarding the interactions among components, in addition to assumptions on component-internal timing.

1.2.2 Asynchronous Controller Design Styles

There are several design styles for asynchronous controllers, the most important of which divide roughly in three categories. The first of these specifies circuits in one of several concurrent programming languages, and synthesizes circuits by syntax-directed translation. Second is a set of methods based on state transition graphs (or Petri Nets) as the specification form. The third class broadly subsumes state-machine-based specifications. This latter design style is the focus of the thesis.

1.2.3 Programming in Silicon

One set of popular design styles derives from Hoare's *Communicating Sequential Processes* (CSP) [65]. These styles all make use of specifications written in a concurrent programming language. Specifications in the style of Martin, for example, describe a set of concurrent processes which communicate over channels. Each program is automatically [17] translated into a network of circuits. This is the origin of the term "programming in silicon" [87, 9, 15, 48]. Channel communications are implemented by handshaking circuits observing a *four-phase* protocol. Martin and Burns have used this approach to design an entire microprocessor [89], as well as numerous other circuits.

A similar approach has been taken by Ebergen [48], who defined a specification form known as *commands* based on Udding et al.'s trace theory [131]. Traces describe permissible interleavings of transitions on the inputs and outputs. A notable feature of this formal approach is its natural ability to verify implementations constructed by the composition of smaller circuits.[3]

Although these methods offer reasonably high-level specification forms for control circuits, they synthesize circuits which generally retain the structure of the original

[3] Ebergen's presentation actually concentrates on synthesis by the reverse procedure of decomposition. Although provably-correct synthesis by decomposition is attractive, it seems quite difficult to automate. To my knowledge, Ebergen has no such method. Verification by composition, by contrast, is easily, if not efficiently, automated.

specification. Typically, optimization is limited to local ("peephole") methods of modest potential. In particular, methods for more global and rigorous optimizations, such as state minimization, state encoding, or logic minimization, are lacking. It is partly for this reason that circuit implementations tend to be expensive and slow.

1.2.4 State Transition Graphs

Petri nets [110] and related formalisms have served as a foundation for a number of asynchronous specifications and synthesis methods [143, 94, 24, 127]. Petri Nets have long been used in a variety of contexts (e.g. distributed systems, communications protocol specification) as a convenient form for specifying concurrent behaviour. This is due primarily to their ability to concisely represent the possible interleavings of events in a distributed system, as well as synchronization. As with CSP-based methods, Petri nets offer an attractive high-level representation for asynchronous behaviour.

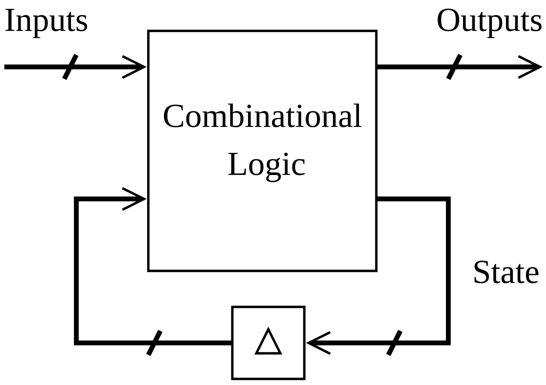
The conciseness of Petri nets makes robust global optimizations difficult to construct. Nevertheless, some progress has been made in recent years, e.g. in state encoding [28, 22] and logic minimization [112]. Unfortunately, the optimality of these methods is hard to quantify.

1.2.5 Asynchronous State Machines

Much of the earliest work on the design of asynchronous controllers focused on asynchronous state machines, and includes seminal work by Huffman [67], Unger [139], as well as more recent contributions by Davis [35], Nowick [102], and Yun [153]. Asynchronous state machines are finite-state sequential machines using no clock signal. This specification style is somewhat lower-level than the two previous styles. These machines are often specified in terms of a flow table, such as that shown in Figure 1.1. In effect, they trade the more explicit representation of concurrency, such as that available in STG's or CSP's, for a simpler representation which is more amenable to the synthesis of

	inputs xy			
	00	01	11	10
a	**a**,0	**a**,0	c,1	**a**,0
b	**b**,0	d,1	**b**,0	a,0
c	b,0	-,-	**c**,1	d,0
d	**d**,0	**d**,1	c,1	**d**,0

(a) Flow table



(b) Huffman machine

Figure 1.1: Simple asynchronous flow table and Huffman implementation

high-performance implementations.

In contrast to synchronous state machines, the lack of a clock implies that only some states are *stable*. A stable machine state results in no state transition, e.g., total state $\langle 00\,a \rangle$ in Figure1.1 (a). Stable states are typically shown in bold.[4]

A block diagram of a Huffman machine [67], a common implementation for asynchronous state machines, is also shown in Figure 1.1. The circuit consists of a block of combinational logic and a set of feedback paths, on which the machine's internal state is stored. No registers or latches are used, although delay elements are often added to prevent the manifestation of sequential hazards.

Within the class of asynchronous state machines, Unger [139] defined several subclasses, based on restrictions of input/output behaviour. The spectrum of classes encompasses increasingly general behaviours, but with simultaneously more complex implementation constraints.

For example, *single-input change* (SIC) machines allow only a single input to change, after which the machine must be allowed to stabilize before any further input changes. Although this operating mode is capable of expressing a limited set of behaviours, it is more easily implemented than the more general classes. In particular,

[4]or circled, as in some of the classic literature

hazard elimination is greatly simplified.

A more flexible class is that of *multiple-input change* (MIC) machines. Here, several inputs may change at once before the machine is allowed to settle. This flexibility is tempered by the fact that MIC machines require that all input changes arrive within some maximum time period. Also, hazard elimination requires more care than in SIC mode.

The most general class of machine is the *unrestricted-input change* (UIC) machine. Machines in this class allow arbitrary input changes, with the constraint that no input may change more than once within a prescribed time period. Implementing such flexible specifications has proved problematic, particularly with respect to metastability and the use of expensive inertial delay elements.

1.2.6 Burst-Mode Machines

The last design style, and the focus of this thesis, is *burst-mode*, a generalization of Unger's MIC mode. Burst-mode was first formalized by Nowick [101], who also developed a systematic synthesis method for hazard-free implementations. This design style is based on more ad-hoc methods used earlier by Davis et al.. [37].

Burst-mode machines allow multiple inputs to change concurrently, but, unlike MIC machines, in any order and at any time. Hence, burst-mode removes the maximum time period constraint for input changes. This relaxation considerably reduces the timing constraints placed on the environment, but nonetheless allows economical and high-performance implementations. In particular, applying Nowick's method for exact MIC two-level hazard-free logic minimization [106] yields low-area, high-performance circuits.

Burst-mode has been successfully used by both academia and commercial interests to design and implement a number of significant circuits, for example, at Stanford, UCSD, HP, AMD and Intel.

Details on burst-mode machines, their operation, and synthesis appear in Chapter 2.

Extended Burst-Mode

An important variation of burst-mode asynchronous circuits, known as *extended burst-mode* (XBM), was proposed by Yun et al. [153]. This form of asynchronous specification provides two useful extensions to basic burst-mode semantics. First, signal *levels* can be sampled, whereas plain burst-mode is only sensitive to signal transitions. Second, a more flexible interleaving of input and output changes is allowed, which allows for greater concurrency between a pair of communicating asynchronous machines. The combination of these extensions makes extended burst-mode especially well suited to the specification of interfaces between clocked and asynchronous circuits.

1.3 Sequential Synthesis

Computer-aided design (CAD) plays a vital role in the development of digital VLSI systems. The demand for greater functionality, higher speed, smaller size, and lower power in digital systems produces several trends in digital design. These trends include increasing complexity (many interacting components), larger numbers of transistors per chip, and approaching physical device and fabrication limits. Each of these trends in turn places great demands on the development process, contributing to the need for effective CAD tools. The desire to decrease the time-to-market simply amplifies this need.

CAD tools for VLSI circuit development address these important issues by facilitating the flow from specification to physical realization. In particular, current practice uses automated methods for synthesis, functional verification, and optimization, to help ensure some degree of correctness in the end result.

This thesis focuses on the synthesis of *sequential* digital systems, i.e., on the derivation of an arrangement of logic gates and memory elements which implements a given specification of sequential behaviour. Moreover, we address *technology-independent synthesis*, which exclusively uses generic, somewhat idealized logic gates (e.g. AND, OR, NOT) for its implementations.

The following section describes the classic sequential synthesis approach, and gives some detail on the individual steps that constitute the synthesis process.

1.3.1 Classic Synthesis Trajectory

This section describes the sequential synthesis problem, and the traditional approach to this problem as a sequence of decoupled steps. A brief recount of the history of sequential synthesis and the current state of the art for both synchronous and asynchronous FSM's is also given.

Sequential synthesis is the process of creating a suitable implementation for a given FSM. It is an extremely difficult problem, for several reasons. First, the number of possible implementations for even modest-sized specifications is staggering. Second, although the cost criterion is approximated by many tools as a simple metric, in reality it is often a complex, multi-dimensional function. Moreover, trade-offs involving several characteristics of the solution (e.g. area, power or latency) are typical. Further, addressing even a simplified cost metric often requires algorithms of high computational complexity. In fact, many of the problems described below are NP-complete.

As a result, the synthesis problem is typically broken into a sequence of decoupled sub-problems, as shown in Figure 1.2. We first sketch the overall flow, and then give some more detailed background on certain key steps.

The process starts with a specification of the FSM's behaviour in some form, such as a flow-table. The machine's internal states are normally identified by symbols, so that the specification focuses as much as possible on the intended behaviour and not some particular implementation.

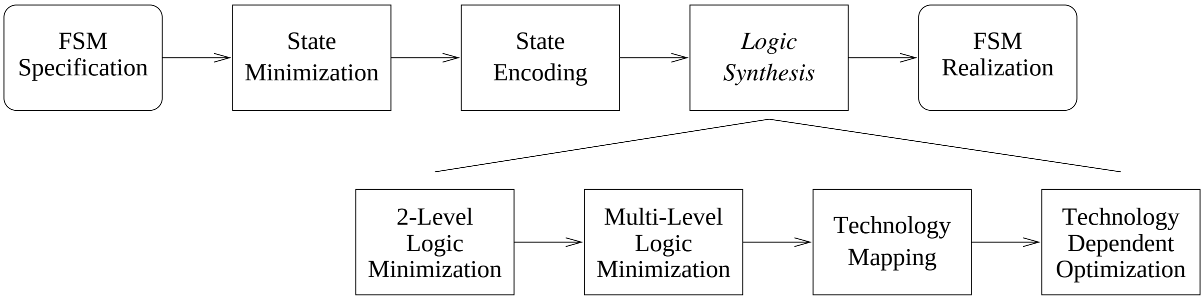


Figure 1.2: Traditional sequential synthesis trajectory

State Minimization

The first step in the flow, *state minimization*, is an optimization that makes use of a classic observation [64] that don't-cares in an incompletely-specified FSM specification often permit a realization with fewer states. In fact, even when the FSM is completely specified, states having indistinguishable behaviour can be "merged", producing smaller equivalent machines. Often, the FSM's resulting from state minimization are fundamentally simpler than their unminimized counterparts, and can thus be implemented with better logic, regardless of the cost metric involved.

The importance of state minimization to finding good implementations was recognized early on [64], and has been studied by many researchers over several decades [60, 139, 62]. The bulk of this research has striven to find efficient algorithms for solving the classic problem as stated by Luccio and Grasselli [60], both exactly and heuristically. A small minority of recent work has attempted to solve the more difficult problem of *optimal state minimization*, which targets logic complexity [4, 18], rather than the fewest number of states. This line of research met with very limited success until the approach described in Chapter 4.

State Encoding

Obviously, a realization in binary logic is required; hence, *state encoding* derives a suitable assignment of binary codes to the symbolic states. The encoding so derived effectively transforms the symbolic FSM specification into a set of pure binary-valued Boolean ("combinational") functions.

Nominally, any encoding which maintains the distinction among the various states is valid[5]; however, certain encodings permit more economical or faster realizations than do others. This realization spurred significant research into the problem of *optimal state encoding* [40, 146, 44, 81]. The resulting algorithms offer a wide variety of heuristic [128] and exact approaches. Typically, the more successful approaches incorporate models of the desired logic structure and/or cost functions. Hence, a variety of methods exists targetting, for example, 2-level logic [40, 146] vs. multi-level logic [44, 81], or area vs. power consumption [69, 8, 138].

Logic Implementation and Minimization

The wide variety of applications, design constraints, design complexity, production scales, and so on, has given rise to a similar variety of implementation technologies. For example, low-power applications favor the use of CMOS, which dissipates almost no power while inactive. High-frequency applications, on the other hand, might require GaAs technology. For various reasons, each technology most naturally supports a different suite of basic logic elements, or *gates*. In another dimension, small, critical circuits may permit carefully manually tuned ("full-custom") designs, while limited productions of medium-sized circuits may employ more regular structures (e.g. PLA's, FPGA's) to reduce design cost. All of these forces shape the specific set of gates ("cells") available to the implementation, collectively known as a *technology library*.

The task of *logic synthesis* is to produce an optimal implementation of the given combinational function(s) using the suite of logic gates offered by the technology library in use. This too is such a complex process that it is normally decomposed into several substeps that are more readily solved. These steps are 2-level logic minimization, multi-level

[5] for synchronous FSM's

logic optimization, technology mapping, and finally, technology-dependent optimization.

Two-level logic forms are important both as final implementations for certain regular structures (such as PLA's) and as a starting point for subsequent multi-level optimizations. Pioneering work by Quine [113] and McCluskey [90] proposed exact methods for two-level minimization. Later exact methods were developed largely as a refinement of their ideas, notably, Espresso-Exact [118], and Scherzo [30]. The latter of these introduced the powerful notion of *implicit representations* based on so-called *characteristic functions*, which greatly extended the reach of exact minimization. Another interesting exact method by McGeer [93] departs considerably from the Quine-McCluskey method, by avoiding the computation of the complete set of prime implicants, a bottleneck in other approaches. Heuristic methods were also the subject of considerable research [66, 119, 30] and address various cost functions, such as area, power, and testability. Recent developments include re-factoring for low power based on pre-computation [95], and other techniques.

Multi-level logic is often capable of much smaller area, and offers a richer space of trade-offs, than does 2-level logic. Hence, a large number of methods has been developed, using algorithmic algebraic manipulations [13, 14, 117], as well as rule-based systems [33]. Just as for 2-level optimization, algorithms for delay [130], area and testability [5, 61] optimization abound. In particular, a significant amount of attention has focused on the computation of don't-care information, for use in both local and global network optimization [136, 32, 5]. Similarly, the *false path* problem, central to accurate delay estimation, has been investigated in depth [12, 42, 92].

Finally, technology mapping is the process by which the network of idealized gates used by previous steps is transformed into a network of cells chosen from the technology library. Methods include early rule-based systems such as Lss [33], as well as seminal work by Keutzer [71] based on graph matching and covering, which was further developed by Rudell [117]. Much work has been done on mapping to specialized technologies, such as

FPGA's (which present somewhat unique problems), XOR-based implementations, and so on. Using the rich information available on the technology gates, many optimizations are possible, including detailed power and delay analysis, and area minimization.

1.4 Toward Global Solutions to Optimal Synthesis

After much research into the above synthesis problems, it has become apparent that the linear flow of isolated steps, while simpler to engineer, produces globally sub-optimal solutions, even when optimum solutions are found at each step. A typical response has been to introduce additional quasi-local optimizations, such as retiming [79, 85] and re-encoding [63]. Other approaches, such as technology-dependent and layout-oriented logic optimizations [84, 27], early power estimation [99, 114] and so on, represent a more concerted effort to move away from local maxima. These optimizations help, but are by no means globally robust solutions.

Optimal state encoding represents a partial (and early) attempt to remedy this situation. A critical insight by De Micheli [40] states that optimal state encoding requires *symbolic* logic minimization. In other words, a better encoding method results when encoding and logic minimization steps are combined.

This thesis embodies a stronger and more general principle that derives from De Micheli's insight: more powerful synthesis methods are obtained by integrating steps to a higher degree. The integration is achieved by embedding knowledge of the structure of the solution in early transformations. By doing so, early transformations can be instrumented with knowledge of the rich trade-offs inherent in sequential synthesis, thereby allowing them to better target the desired cost function.

In other words, this thesis is, in large part, an effort to apply this philosophy to the synthesis of asynchronous (and synchronous) FSM's. Later chapters will present a progression in which each successive method captures more and more of the synthesis flow. For example, Chapter 3 presents HFMIN, which performs logic minimization, and CHASM, which integrates both state encoding and logic minimization. Optimist, presented in Chapter 4, merges three steps, and thus embodies simultaneous state minimization, state encoding, and logic minimization.

Much work remains to be done in this area in order to understand and harness the complex relationships among the many transformations and the myriad possible optimizations.

1.5 Asynchronous Sequential Synthesis

This section highlights some of the unique problems confronting the implementation of asynchronous sequential circuits. More on these problems can be found in Chapter 2. Then, it gives a brief overview of the asynchronous controller synthesis path, and the state of the art.

Asynchronous circuit implementations must satisfy several constraints above and beyond those of synchronous systems, in order to ensure correct operation. In general, because an asynchronous circuit is nearly always sensitive to changes in its inputs, many implementation forms require that the output changes generated be free of undesired transitions, or *glitches*. This is a pervasive issue in asynchronous synthesis, and appears under several guises.

For combinational asynchronous circuits, *combinational hazards* [106] can cause output glitches to manifest. A combinational hazard is said to exist for a given input transition when a particular arrangement of delays reliably produces an output glitch. In some cases, the Boolean function under consideration can never be implemented without hazards; then, a *function hazard* is said to exist. Alternatively, a particular implementation may exhibit a *logic hazard* for that transition.

Both kinds of combinational hazards must be addressed in most robust synthesis methods. We will see in Chapter 2 that early synthesis stages will often be charged with ensuring *function-hazard-freedom*. Subsequent stages will then be concerned with finding a suitable *logic-hazard-free* implementation for that function.

Sequential asynchronous circuits also face difficulties beyond those of their synchronous counterparts. These difficulties revolve around implementation malfunctions caused by two classes of *sequential hazards*, namely, *critical races* [137] and *essential hazards* [139]. Both malfunctions manifest in an implementation's settling in an incorrect destination state for a given transition.

Critical races [68, 139] exist in an asynchronous machine's implementation when an input transition causes two changing state variables to "race," such that the state into which the machine settles depends on which variable wins the race. Critical races can be prevented by judicious encoding [137]. The process of critical race-free encoding is described in detail in Chapter 2.

The second type of sequential hazard, the *essential hazard* [139],[6] occurs in certain machines when an arrangement of circuit delays allows a state change to complete before the input change is fully processed. For this type of hazard, neither a judicious encoding nor logic implementation avoids the problem — the problem is an inherent property of the sequential function [139]. In fact, one can only solve the problem by adding delays (e.g. to the feedback path) so as to "fix the race" [139]. These delays ensure that the input change is completely absorbed before the state changes propagate.

1.5.1 Asynchronous Synthesis Path

Asynchronous controller synthesis follows a flow similar to that of synchronous synthesis; however, it presents unique problems requiring significantly different solution methods. Like synchronous synthesis, the synthesis trajectory is divided for tractability's sake into several steps: state minimization, state encoding, two-level logic minimization, multi-level and testability transformations, and so on. Each of these steps can be modeled roughly after its synchronous counterpart, but poses additional complications. We now

[6] a kind of steady-state hazard [139]

review each step, outlining the basic problems unique to asynchronous synthesis.

The task of *state minimization* is to find a closed state cover for the original burst-mode specification. The result is a reduced machine realizing the original specification [64]. As with synchronous machines, this problem can be solved by first forming a set of compatibles and then forming a binate covering problem expressing the two basic sets of constraints (covering and closure) [60]. Asynchronous machines, however, require *different forms of state compatibles* in order to be assured of the existence of a hazard-free logic implementation [102].

State encoding produces a set of binary codes for the symbolic states of the reduced machine. For synchronous machines, all encodings which distinguish the states are valid; however, typically this is performed judiciously, so as to minimize logic area [40], improve performance, or reduce power consumption. By contrast, asynchronous machines must be encoded so as to avoid *critical races* [137]. Further, if optimal logic is to be obtained, *logic hazards* [139][105] must be taken into account [55].

Finally, to ensure correct operation, *two-level logic minimization* for burst-mode asynchronous machines must also take care to avoid logic hazards. Recent developments in this area include exact multi-valued-input/multi-output minimization [55], fast heuristic minimization [134], and exact implicit minimization [135].

An additional issue facing asynchronous synthesis is the potential for using fed-back outputs to reduce the number of state variables and the overall implementation complexity. In this *machine implementation style*, primary outputs are fed back as additional input variables, which help to identify the machine's present state, thereby reducing the need for distinct state variables. The loading on the outputs may be minimal (only a short path to a feedback buffer is added to its fan-out), but the savings in overall logic complexity can be dramatic. Care must be taken, however, in various synthesis steps, in order to ensure that the use of fed-back outputs does not introduce hazards or critical races.

1.5.2 State of the Art

This section provides a brief overview of the state of the art in asynchronous sequential synthesis.

Although asynchronous circuits were first studied long ago (see for example [68, 97, 139, 26]), only recently has enough progress been made that their use is starting to become practical. This is due to the resurgence of interest in asynchronous circuits as a solution to the problems faced by synchronous design. In turn, the interest has spurred a significant amount of research in three areas, among others:

- Specification forms (e.g. signal transition graphs (STG's) [23], CSP-based [48, 88], burst-mode [104, 155])
- Synthesis algorithms (both technology-independent [106], [75], [77], [24], [72] and technology-dependent [17], [15], [140], [129])
- Verification (both functional [46], [115], [47] and timing [43], [116])

In short, these research areas have seen considerable progress in a relatively short period, and the results are encouraging. However, current asynchronous CAD tools and algorithms still leave much room for improvement.

The asynchronous community has struggled to find a suitably expressive and capable specification form which is amenable to the automated synthesis of high-quality implementations. Unfortunately, the various specification forms differ enough that a distinct body of synthesis research has emerged for each. For example, specialized methods for both state minimization and state encoding exist for STG's[77, 144], state machines (e.g. [150]), and translation-based methods [17]. Further, the theoretical frameworks for each method differ enough that meaningful quantitative comparisons are difficult. Also, the insights gained in one domain are typically inapplicable to another.

Thus, the diversity of asynchronous specification forms, coupled with the relative youth of automated asynchronous synthesis, leaves the asynchronous CAD community lagging far behind its synchronous cousin. The lag manifests in several ways:

- No robust optimal methods yet exist for several key synthesis problems, e.g. state encoding, multi-level logic optimization, and so on.
- There are few high-quality, usable CAD tools which provide the kind of *flexibility* designers need in order to craft solutions to domain-specific applications within domain-specific constraints.
- There are no software frameworks which serve as a complete synthesis environment, as well as a backplane onto which new tools and methods can be easily grafted.

With this in mind, the following section describes the contributions made by the present body of work.

1.6 Thesis Contributions

This thesis makes contributions on three fronts.

First, the thesis offers a trio of practical algorithms for producing optimal two-level implementations for burst-mode asynchronous machines. Hfmin is an exact method for hazard-free two-level logic minimization. The second, Chasm, is the first known optimal state encoding algorithm for asynchronous state machines. Finally, Optimista is the first optimal state minimization algorithm for asynchronous machines of any kind.

Second, the thesis presents a suite of high-quality CAD tools embodying these algorithms. The tools were implemented in C++, and make use of highly-tuned algorithms for key substeps. They are capable of handling almost all known industrial benchmark designs.

Third, MINIMALIST is a new software CAD framework which binds these tools into a state-of-the-art technology-independent path for burst-mode synthesis. MINIMALIST is both a complete technology-independent synthesis path with state of the art tools, and a flexible framework for incorporating new tools.

Although the main thrust of this body of research is asynchronous synthesis, it makes contributions to the state of the art in synchronous FSM synthesis as well. In particular, one highlight of this thesis is a pair of algorithms (and corresponding tools) for the state minimization of synchronous FSM's: OPTIMIST, described in Chapter 4, and OPTIMISTO, described in Chapter 5. These are first-of-a-kind algorithms, and are significant beyond their use as a foundation for the asynchronous method of Chapter 6.

1.6.1 CAD Algorithms and Tools

The thesis offers three new algorithms for technology-independent asynchronous sequential synthesis, along with practical software implementations for each. All three algorithms target two-level logic realizations.

The first of the tools, **HFMIN**, is a *hazard-free two-level logic minimizer*. It embodies the first exact hazard-free two-level logic minimization algorithm [55] which is capable of minimizing functions with *symbolic* inputs. This ability is vital to both of our other algorithms. Notably, HFMIN offers several user-selectable operating modes to accommodate different applications. For example, HFMIN supports single-output, output-disjoint, or multi-output implementation styles, exact and quasi-exact modes, and both literal and product count minimization. The algorithm makes use of the highly-optimized algorithms ESPRESSO [118] and MINCOV[118] for key substeps, and is capable of minimizing a large variety of industrial circuits.

Second, we present **CHASM** [55], the first general method for the *optimal state assignment* of asynchronous state machines. CHASM is a state-of-the-art algorithm which produces state encodings that result in optimum or near-optimum logic implementations. CHASM features both an exact and an effective heuristic mode, and supports all of HFMIN's logic implementation styles. In its exact mode, CHASM produces exactly optimum two-level output logic *over all possible encodings.* In its heuristic mode, CHASM is capable of producing near-optimum output logic with more compact next-state logic. Further, CHASM supports the use of outputs that are fed-back as state variables. This frequently significantly reduces the next-state logic complexity. CHASM's unique combination of features allows designers to explore various trade-offs and find the solution that best fits their application's requirements. For constraint solution, CHASM uses existing highly-optimized tools, namely, DICHOT [121] for exact solution, and NOVA [146] for heuristic solution. These tools help CHASM encode large asynchronous FSM's.

Finally, **OPTIMIST** [56] is the first *state minimization algorithm* for incompletely-specified finite state machines (FSM's) to *directly and accurately target logic complexity.* Three state minimization methods are proposed, all based on the same theoretical foundation: two for synchronous FSM's and another for asynchronous burst-mode machines.

The first synchronous method, **OPTIMIST**, described in Chapter 4, addresses both output and next-state logic quality. Its computational complexity is high, and hence it is currently capable of minimizing only relatively small synchronous FSM's.

The second synchronous method, **OPTIMISTO**, described in Chapter 5, takes advantage of our method's precise modeling of outputs by focusing exclusively on output logic complexity. The result is the first truly exactly optimum state minimization algorithm for output logic. This method has the added benefit of a far lower computational complexity than the first method. The improvement in the cost of output logic over the leading tool is quite dramatic in some cases.

The third state minimization method, **OPTIMISTA**, described in Chapter 6, is the *only* known optimal state minimization algorithm for asynchronous machines of any form. Moreover, it guarantees exactly minimum cardinality hazard-free output logic over *all possible state minimizations and encodings.* Because of the more demanding correctness requirements on asynchronous implementations, it is somewhat more complex than its synchronous counterpart. Nonetheless, experimental results show that it is capable of handling most of the industrial benchmark circuits available today.

1.6.2 CAD Framework

The MINIMALIST software package constitutes a state-of-the-art synthesis path for asynchronous machines. MINIMALIST combines a simple, yet powerful software framework for tool integration with top-flight synthesis tools to yield a uniquely effective and extensible synthesis environment.

The MINIMALIST CAD framework incorporates a C++ class library, an extensible command interpreter, and a graphical user interface. The framework constitutes approximately 35k lines of C++ (including certain core libraries), and offers key building blocks for crafting new tools and features. Together, the components provide a uniquely powerful framework for binding CAD tools into a robust and usable synthesis path for burst-mode asynchronous machines. For example, incorporating new tools into the framework is typically as easy as writing a short shell script, or perhaps a couple of dozen lines of C++.

Unlike other existing asynchronous synthesis packages (e.g. [150, 102]), MINIMALIST is able to produce implementations in a variety of styles, and under various cost functions. This feature supports the exploration of the complex trade-offs inherent in sequential synthesis. For example, a designer can choose an implementation targetted to overall performance, area, output latency, or some hybrid. Because a single cost metric can hardly expect to accommodate all applications, this ability should prove uniquely effective for a wide spectrum of designers' needs.

In addition to the tools described above, MINIMALIST integrates several other state-of-the-art tools, e.g., for extremely fast exact two-level hazard-free logic minimization, asynchronous combinational verification, and synthesis-for-testability.

1.6.3 Scope of the Thesis

This section discusses the scope and limitations of the present body of work.

This thesis focuses exclusively on two-level technology-independent synthesis. Although such implementations are only directly usable for PLA-based implementations (and hence require technology mapping in most other situations), this is actually a reasonable first choice in the context of burst-mode synthesis. First, two-level forms serve as a good starting point for multi-level transformations and technology-dependent logic optimizations. Second, asynchronous machines are typically quite small, so that two-level logic is often close to optimum, even for area. Also, output latency is the critical performance parameter for burst-mode machines, and two-level forms have lower latency than multi-level forms. Finally, it would have been difficult to target multi-level logic in optimal state encoding and optimal state minimization without first understanding the two-level problem.

Nevertheless, it is clear that handling multi-level logic forms would increase the path's value considerably. For example, it is well-known that where area is the primary cost metric, multi-level realizations have a clear advantage over two-level forms. Excessive fan-in can also necessitate the use of multi-level logic. Likewise, a mechanism for technology mapping and technology-dependent logic optimization should be incorporated into the framework, for it to serve as a complete package for asynchronous burst-mode synthesis. However, both of these abilities can be added in modular fashion, without disturbing either the framework or the upstream portions of the path.

One limitation of the present work is its restriction to plain burst-mode [104] specifications. Although plain burst-mode specifications encompass a large number of useful asynchronous designs, they do not allow sampling level-based signals, and require a strict alternation of input and output bursts. Extended burst-mode specifications [155] avoid these restrictions and as a result can be used to describe a wider range of behaviours. We believe that an extension of each of the methods presented herein to extended burst-mode design is relatively straightforward. This is an area for future work.

In addition, the algorithms in this thesis are all based on a single encoding model: the input encoding model [40] (described in some detail in Chapter 2). This limits their effectiveness under certain cost metrics,[7] because input encoding is only an approximation of optimal state encoding. As such, we expect that considerably better logic would result under these metrics from using a more precise model. However, the use of input encoding does not constitute a structural deficiency of the work, but rather a modular decision. In particular, a more potent model, namely output encoding [45], can be substituted in the various steps, without disturbing the remainder of the synthesis path. Also, input encoding precisely models output logic, which is the critical component of burst-mode system performance. As a result, input encoding is actually quite effective for this application.

Finally, certain of the optimization algorithms presented here are computationally too expensive for the full range of sequential machines encountered in practice. Notably, OPTIMIST would especially benefit from a more efficient implementation. We believe that implicit methods, such as those used by SCHERZO [31] and IMPYMIN [135], would extend its capacity enough to handle any practical design.

1.7 Outline of Dissertation

The structure of the rest of the thesis is as follows.

First, in Chapter 2, we review the background material necessary to several of the subsequent chapters. Chapters 3 through 6 materialize this thesis' philosophy of "step-merging" in several tools that encompass progressively more of the synthesis path. In Chapter 3, we present an encoding tool for asynchronous state machines, CHASM, along with some substantiating experimental results and several important theoretical results justifying its correctness and optimality. Chapter 4 presents the basic results for the

[7] such as total area or machine cycle time

Optimist state minimization method in the synchronous domain. Chapter 5 introduces Optimisto, a synchronous algorithm, which focuses Optimist on output logic complexity, and achieves some novel theoretical results as well as experimental results. The results of Chapters 4 and 5 are useful in their own right. More importantly to this thesis, however, they serve as a foundation for the work presented in Chapter 6 for asynchronous machines. Chapter 6 introduces a further extension of Optimist, Optimista, which performs optimal state minimization for asynchronous burst-mode machines. Chapter 7 describes the Minimalist framework and the suite of tools currently integrated into it, and compares it to two existing synthesis packages for burst-mode asynchronous machines. Finally, in Chapter 8, we offer some concluding remarks on the relative success of the research and suggest several avenues for continued work on the subject.

Chapter 2

Background

This chapter reviews some basic technical material that is common to the later chapters. Three major areas are reviewed: finite state machines, logic functions and their synthesis, and a related pair of abstract combinatorial problems that find numerous applications in CAD.

Section 2.1 presents basic definitions related to finite state machines. Section 2.1.1 deals with synchronous automata, while Section 2.1.2 describes burst-mode asynchronous state machines, the design style addressed by this thesis. Section 2.2 defines Boolean functions and two-level logic minimization for both binary functions (in Section 2.2.1) and for symbolic functions (in Section 2.2.2). Additionally, Section 2.2.3 defines the problem of hazard-free two-level logic minimization, a pivotal problem for asynchronous synthesis. It also describes an exact method for its solution due to Nowick and Dill [106]. Section 2.3 discusses a class of problems known collectively as sequential hazards, along with corresponding solutions. Section 2.4 introduces a problem known as input encoding, which lies at the heart of this thesis. Finally, Section 2.5 defines a pair of abstract problems with many applications in CAD, namely, unate and binate covering.

2.1 Finite State Machines

This section starts by giving a broad definition of finite state machines (FSM's) and proceeds by defining two particularly interesting subsets, namely, non-deterministic and incompletely-specified finite state machines. Two commonly-used representations for FSM's are also shown.

2.1.1 Synchronous FSM's

Generally speaking, a finite state machine (FSM) is an automaton having a finite set of states, and a finite number of binary inputs and outputs. One or more states are identified as initial states, from which the machine starts. A set of mappings determines the next state and next output values, given the current machine state and input values.

Figure 2.1 shows a simple FSM, represented as a *state graph*. Here, each state is represented by a graph node, and each transition appears as a directed arc. Each transition is labelled by a set of input and output values. When in a given state (node) and the input values match those of a given transition (arc) rooted in that state, the machine generates the corresponding output values, and proceeds to the destination state.

Two common types of FSM are defined in the literature: *Mealy* machines, whose outputs depend upon the current inputs as well as the present state, and a sub-class of Mealy machines known as *Moore* machines, whose outputs depend only upon the present state. This thesis restricts its attention throughout to the more general class of Mealy machines.

More formally, a *completely-specified* or *deterministic* Finite State Machine (FSM) $\mathcal{M}$ is defined [145] by the 6-tuple $\langle \mathcal{I}, \mathcal{O}, \mathcal{S}, \mathcal{S}_0, \delta, \lambda \rangle$, where:

$\mathcal{I}$ is the input alphabet,

$\mathcal{O}$ is the output alphabet,

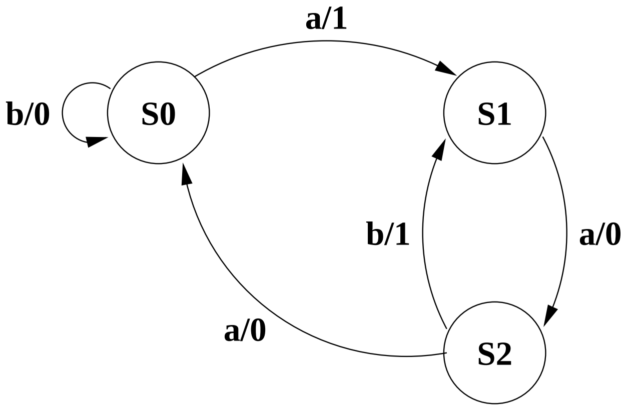


Figure 2.1: State transition graph for a simple FSM

$\mathcal{S}$ is the set of states,

$\mathcal{S}_0 \subseteq \mathcal{S}$ is the (set of) initial state(s),

$\delta = \delta(i, s) \in \mathcal{S}$, $i \in \mathcal{I}$, $s \in \mathcal{S}$ is the transition function, and

$\lambda = \lambda(i, s) \in \mathcal{O}$, $i \in \mathcal{I}$, $s \in \mathcal{S}$ is the output function.

The first argument $i \in \mathcal{I}$ to both δ and λ is normally referred to as the *present state*; the value of $\delta(i, s)$ is called the *next state.* A pair $\langle i, s \rangle$ for $i \in \mathcal{I}$ and $s \in \mathcal{S}$ is known as a *total state* of $\mathcal{M}$. The machine is completely specified in that, at each point, the next state and output values are uniquely defined.

A *non-deterministic* FSM (NDFSM) is defined similarly, except that $\delta \subset I \times S \times S$ and $\lambda \subset I \times S \times O$ are relations, rather than functions.[1] Thus, an NDFSM's specification allows for several possible outputs or next states for any given combination of input and present state.

Unless otherwise indicated, we restrict attention to the common subclass of NDFSM's known as *incompletely-specified* FSM's (ISFSM's) where, for each total state $\langle i, s \rangle$, the value of the transition function $\delta(i, s)$ is either a unique (singleton) state or the set of

[1] Where δ (respectively λ) maps some (i, s) onto a single element of $\mathcal{S}$ (respectively $\mathcal{O}$), a functional notation may be used, without loss of clarity.

all states. In the latter case, we say that the next-state is *unspecified* for the total state $\langle i, s \rangle$.

While the above definitions describe a machine having a single input and a single output (each taking its value from the given alphabet of symbols), it is often convenient to represent an FSM as having *several* inputs or outputs. Such a representation more adequately models the fact that, for example, the inputs may emanate from distinct sources in a network of interconnected FSM's. In this case, each symbol in the alphabet represents a *combination* of values, one for each input or output.

For convenience, finite state machines are frequently specified in one of two other equivalent forms: as flow tables, and in so-called cube-table format [39]. We describe and give examples of each, as these forms will appear throughout the thesis.

Flow tables are a human-readable form mainly suitable for FSM's with a small number of inputs. They consist of a single row for each machine state, and a single column for each input combination (or, alternatively, each input symbol). In each table entry, the specified next state and output values are listed for the given input and present state.

Figure 2.2 depicts the flow table representing a simple completely-specified FSM $\mathcal{M}$ having 2 binary inputs `x` and `y`, and a single output, `z`. In state `a`, under inputs `xy` $= 00$, $\mathcal{M}$ produces the output value $z = 0$ and remains in state `a`. In state `b` and under inputs `xy` $= 11$, $\mathcal{M}$ produces the output 1 and proceeds to state `c`. Note that the next state and output values are unique under all combinations of input and state. The flow table in Figure 2.3 is equivalent to that of Figure 2.2 under the mapping $\{\mathcal{I}_0, \mathcal{I}_1, \mathcal{I}_2, \mathcal{I}_3\} = \{00, 01, 11, 10\}$.

A flow table for an ISFSM is shown in Figure 2.4. Unspecified next states and output values are indicated with a dash ('-'). Note that it is permissible for a given entry to specify a next state but not an output value, as is the case in total state $\langle$`c`, 01$\rangle$. The reverse situation, a total state having an unspecified next state but a specified output, is

	inputs x,y			
	00	01	11	10
a	a,0	b,1	c,1	a,0
b	b,1	b,1	b,1	a,0
c	c,1	d,0	c,1	d,0
d	d,1	d,0	c,1	d,1

Figure 2.2: A flow table describing a simple FSM $\mathcal{M}$

	input $\mathcal{I}$			
	$\mathcal{I}_0$	$\mathcal{I}_1$	$\mathcal{I}_2$	$\mathcal{I}_3$
a	a,0	b,1	c,1	a,0
b	b,1	b,1	b,1	a,0
c	c,1	d,0	c,1	d,0
d	d,1	d,0	c,1	d,1

Figure 2.3: A flow table describing $\mathcal{M}$ using a symbolic representation of the inputs

allowed but rarely encountered in practice.

Cube-table format often compactly represents FSM's that are too large for a flow table representation, and are also particularly well suited for incompletely-specified FSM's. In order to define cube-table format, however, we must first define several basic concepts relating to Boolean functions and their representation. The following definitions, adapted from [39], are also relevant to the description and synthesis of FSM implementations.

	inputs x,y			
	00	01	11	10
a	a,0	b,1	a,1	-,0
b	b,1	b,1	-,-	-,0
c	c,1	d,-	-,-	d,1
d	-,-	d,-	a,1	d,1

Figure 2.4: A flow table describing an incompletely-specified FSM

2.1.2 Burst-Mode Asynchronous Specifications

We now describe burst-mode asynchronous specifications [101, 153], the asynchronous design style that is the focus of this thesis. We first describe the specifications, and then the corresponding flow-table representation.

The specifications are most easily illustrated by example. A burst-mode specification for a distributed mutual-exclusion controller with 3 inputs and 3 outputs is shown in Figure 2.5. The unique starting state (`S0`) is indicated by a `'v'`, and initial input and output values are either explicitly specified or (as in the figure) default to `0`. Each arc is labelled with a set of input and output transitions, known as *bursts,* separated by a `'/'`. Rising transitions are denoted by a `'+'`; falling transitions, by a `'-'`.

The operation of a burst-mode machine is as follows. Starting in a given state, the machine remains stable in that state until a complete input burst arrives. Individual inputs within that burst may arrive in any order and at any time. Once the last input arrives, the burst is complete. The machine then generates the corresponding output burst, if any, and moves to the specified next state. The environment allows the machine to settle, and the next cycle begins.

Figure 2.5 illustrates burst-mode operation. For example, consider the transition from `S2` to `S0`, with corresponding input and output bursts `LIN-,RIN-` and `LOUT-`, respectively. As a result, when in `S2`, if the pair of input changes `LIN-` and `RIN-` arrive at any order, and within any time window, the machine responds with a falling edge on `LOUT` and a transition to `S0`.

Burst-mode specifications must obey two important restrictions. First, input bursts must not be empty; in the absence of input changes, the machine remains stable in its current state. Second, the so-called *maximal set property* stipulates that no arc leaving a given state may possess an input burst that is a subset of any other arc leaving that state. This property guarantees that, at all times, the machine can unambiguously decide whether to follow a transition or remain stable.

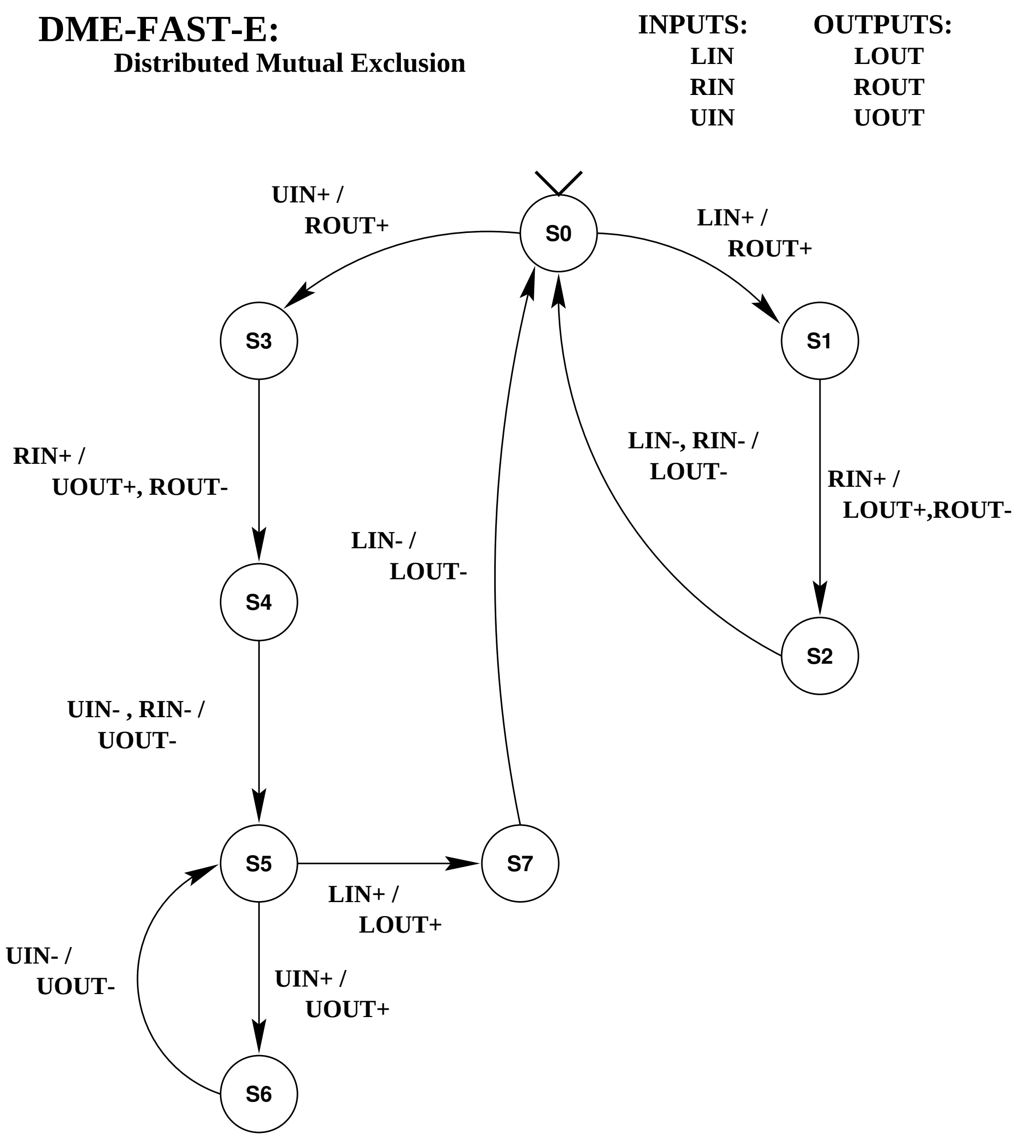


Figure 2.5: Burst-mode specification for a distributed mutual-exclusion controller

It is important to understand that, in a given burst-mode specification, *any unspecified input combinations are forbidden.* For example, the input burst `RIN+` in state `S0` in the specification of Figure 2.5 is prohibited. In other words, the surrounding circuitry must never generate that input combination. Any such combinations can thus be treated as don't-cares, and used to optimize the machine's implementation.

It is also conventional to adhere to a simple syntactic constraint, to make burst-mode specifications more readable. Each state must have a *unique entry point*, i.e., a unique set of input and output values upon entry. In other words, each state has a *single* total state that is the destination of one or more transitions. Note that this property is not necessary for proper burst-mode operation. However, it tends to produce machines which are more easily minimized, and which are more easily proven to have hazard-free implementations.

An equivalent textual burst-mode specification (as used by MINIMALIST and MEAT [35]) appears in Figure 2.6, and an equivalent flow table in Figure 2.7.

It is easy to see from Figure 2.7 that burst-mode specifications frequently offer significant opportunity for state minimization. This due to the unique entry point criterion, which generally results in states which bind the output and next-state functions in only a few input columns.

The following definitions relate specified transitions in burst-mode specifications to transitions in the corresponding primitive flow tables. Figure 2.8 shows a fragment of the flow-table for the burst-mode design of Figure 2.6. Also shown is a single specified transition from $s_4 \rightarrow s_5$, with input burst `RIN-,UIN-` and output burst `UOUT-`. The transition has the following components:

horizontal transition The portion of the transition corresponding to the input burst.

entry point The starting input column of the horizontal transition (011 in this case).

exit point The ending input column of the horizontal transition (000 in this case).

```
name DME_FAST_E

input LIN 0
input RIN 0
input UIN 0

output LOUT 0
output ROUT 0
output UOUT 0

0 1  LIN+ |  ROUT+
0 3  UIN+ |  ROUT+
1 2  RIN+ |  LOUT+ ROUT-
2 0  LIN- RIN- |  LOUT-
3 4  RIN+ |  UOUT+ ROUT-
4 5  UIN- RIN- |  UOUT-
5 6  UIN+ |  UOUT+
5 7  LIN+ |  LOUT+
6 5  UIN- |  UOUT-
7 0  LIN- |  LOUT-
```

Figure 2.6: Textual (**.bms**) burst-mode specification for **DME-FAST-E**

```
Inputs:  LIN, RIN, UIN;
Outputs: LOUT, ROUT, UOUT;
#Sn:    000    001    011    010    110    111    101    100
S0:  S0,000 S3,010  -,---  -,---  -,---  -,---  -,--- S1,010;
S1:   -,---  -,---  -,---  -,--- S2,100  -,---  -,--- S1,010;
S2:  S0,000  -,---  -,--- S2,100 S2,100  -,---  -,--- S2,100;
S3:   -,--- S3,010 S4,001  -,---  -,---  -,---  -,---  -,---;
S4:  S5,000 S4,001 S4,001 S4,001  -,---  -,---  -,---  -,---;
S5:  S5,000 S6,001  -,---  -,---  -,---  -,---  -,--- S7,100;
S6:  S5,000 S6,001  -,---  -,---  -,---  -,---  -,---  -,---;
S7:  S0,000  -,---  -,---  -,---  -,---  -,---  -,--- S7,100;
```

Figure 2.7: Asynchronous flow table for **DME-FAST-E**

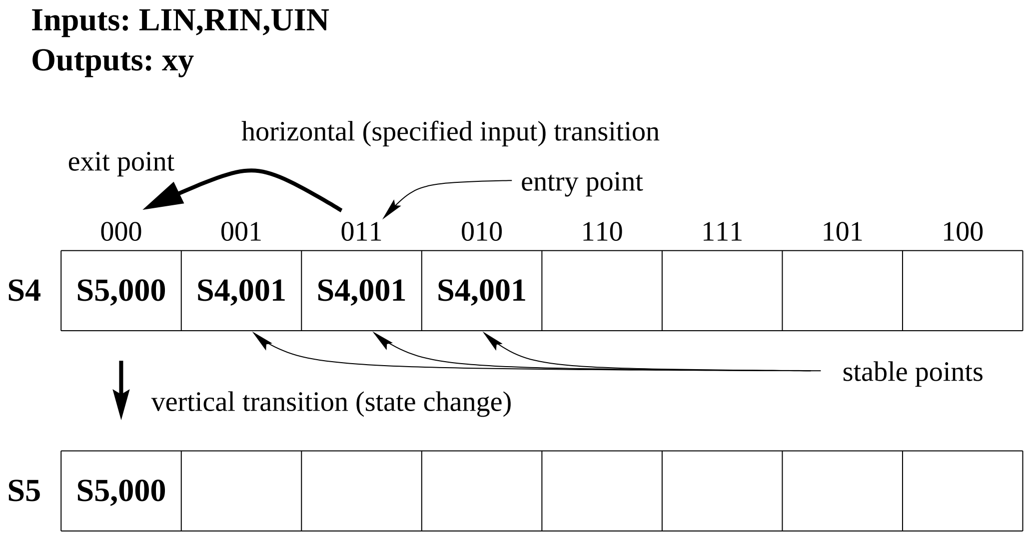


Figure 2.8: Horizontal and vertical transitions in a burst-mode flow table

stable points All points in the horizontal transition, exclusive of the exit point (in this case, 011, 001, and 010).

vertical transition The state change portion of the transition.

Two particular properties of primitive flow tables are worth noting, as they have an impact hazard-free logic implementation. First, stable points are so named because in proper burst-mode flow tables they *always* specify a stable next-state and unchanged output values (relative to the entry point values). That is, a stable point always identifies the present state as the destination state, and the current output values as the new output values. Second, both the outputs and next-state are stable throughout any vertical transition.[2]

[2]One variation of burst-mode specifications allows so-called "late output" semantics, in which outputs may change concurrently with the state change (during the vertical transition). This thesis deals exclusively with early output semantics, however.

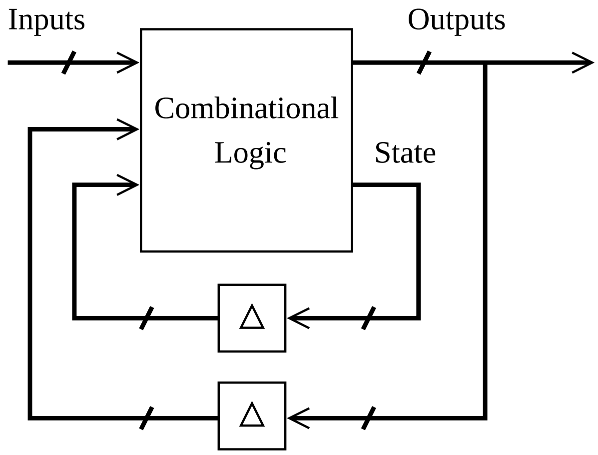


Figure 2.9: Huffman implementation with fed-back outputs

Burst-Mode Machines with Fed-Back Outputs

This section describes a special technique of feeding back outputs as internal state variables. This is purely an implementation technique, and does not affect the machine's externally-visible (input-output) behaviour. However, using this technique often helps to reduce the complexity of the next-state logic, since in many specifications the output values reflect the machine's internal state.

Figure 2.9 shows an FSM implementation (as a Huffman machine) that employs fed-back outputs. By this we mean that one or more outputs are fed back as inputs to the combinational logic, thereby acting as state variables.

The machine's operation is essentially the same as for one without fedback outputs. First, as input changes arrive, the machine remains stable in its present state. Once a valid input burst is complete, the combinational logic generates a set of output and state changes. These changes propagate through the feedback loop, and are presented to the combinational logic as a distinct set of input changes. This second set of changes triggers a set of static transitions on both outputs and next-state. The machine is allowed to settle, and the cycle begins again.

Figure 2.10 illustrates a transition in the primitive flow table of an FSM with 2 inputs ab and 2 fed-back outputs xy. In the figure, groups of columns corresponding to combinations of primary input values (i.e. original flow-table columns) are separated by thick borders. Individual columns within each group represent specific combinations of values on the fed-back outputs. A point in the total state space is represented by the triple $\langle S\,ab\,xy\rangle$. A transition $S_i \rightarrow S_j$ is shown, triggered by the input burst a^+b^+ (i.e. $00 \rightarrow 11$).

First, we describe the effect of the *horizontal* transition on the combinational logic inputs (which include the fed-back outputs) and logic outputs. The input change seen by the logic during the horizontal transition is $\langle S_i - - 11\rangle$, spanning the start point $\langle S_i\,00\,11\rangle$ and the end point $\langle S_i\,11\,11\rangle$. Notice in particular that the present-state and fed-back outputs are constant throughout this cube. Meanwhile, the next state S and primary outputs xy are stable at S_i and 11, respectively, throughout this horizontal transition, *up until* the exit point of the transition at total state $\langle S_i\,11\,11\rangle$. At that point, they change to S_j and 10, respectively. It is important to note also that the outputs and next-state are unspecified (don't-care) for intermediate total states lying outside the transition supercube, such as $\langle S_i\,01\,10\rangle$.[3]

Next, we describe the effect of the *vertical* transition on the combinational logic inputs and outputs. During the vertical portion of the transition, both outputs and state changes feed back to the logic inputs. The input change seen by the logic during this period is thus $\{S_iS_j\}\,11\,1-$, reflecting the state change from S_i to S_j and the change from 1 to 0 for the y output. In contrast to the horizontal transition, the primary inputs are now constant.

[3] This is true because the output logic holds the outputs stable at 11. Hence, total states with the fed-back outputs having other values cannot be reached during the horizontal transition.

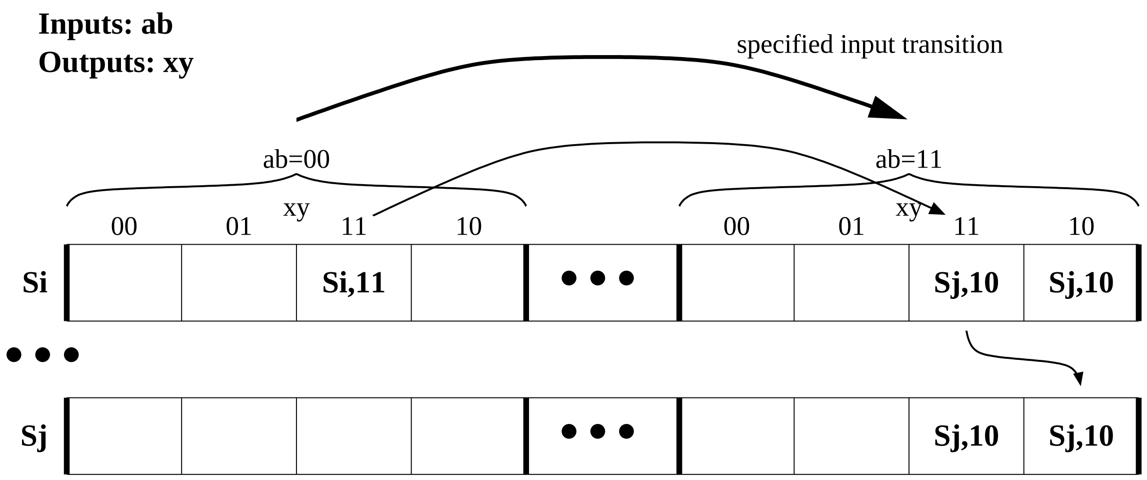


Figure 2.10: Specified transition in the presence of fed-back outputs

2.2 Boolean functions and Logic Synthesis

Having introduced synchronous and asynchronous finite-state machines, we now proceed to topics relevant to their implementation. The following section, Section 2.2.1, defines Boolean functions, along with certain concepts that serve as basic building blocks for logic synthesis. Section 2.2.2 then defines the class of multiple-valued input (mvi) functions.

2.2.1 Binary Functions

We denote the binary Boolean domain by $B = \{0, 1\}$. The n-dimensional Boolean space spanned by n Boolean-valued variables is denoted B^n. B^3 is depicted in Figure 2.11 for Boolean variables x, y, z.

A *completely-specified Boolean function* is a mapping $f : B^n \rightarrow B$. Likewise, an m-output completely-specified Boolean function is a mapping $f : B^n \rightarrow B^m$. A point in the domain of an n-input Boolean function is called a *minterm*.

An *incompletely-specified* Boolean function is a partial function, defined over a portion of the Boolean n-space. It is defined as a mapping $f : B^n \rightarrow \{0, 1, *\}^m$, where minterms mapped to $*$ are said to be *unspecified*. Such minterms are referred to as *don't-*

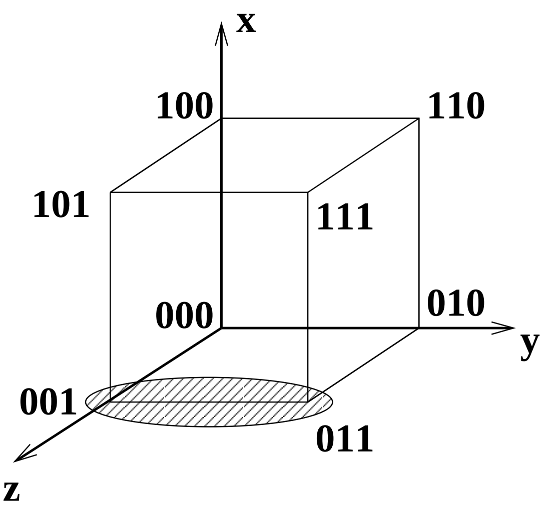


Figure 2.11: The domain $B^3 = xyz$ and the cube `0-1`

cares. Note that a given minterm may be unspecified for some of the m outputs and specified for others.

The *ON-*, *OFF-*, and *DON'T-care* sets of a given output are those sets of minterms for which that output is 1, 0, or unspecified, respectively.

Given a set $S \subseteq B$, we can define a *literal* X^S corresponding to a function $f_X(x) : B \to B$ which is 1 iff $x \in S$ and 0 otherwise. In Boolean expressions, $X^{\{0\}}$ is typically written as x', $X^{\{1\}}$ as simply x, and $X^{\{0,1\}}$ as x^* (or omitted altogether).

A *product term* is a Boolean product of literals $P = \prod_{i=1\ldots n} X_i$, corresponding to the function $f(x_1 \ldots x_n) : B^n \to B$. Now, $f = \prod_{i=1\ldots n} f_{X_i}(x_i)$, and so $f = 1$ iff $f_{X_i}(x_i) = 1, \forall i = 1 \ldots n$. A product P is said to *contain* a minterm $m \in B^n$ iff $f_P(m) = 1$. Because a product comprises a set of adjacent minterms that forms a complete[4] hypercube in B^n, the terms product and *cube* are often used interchangeably.

A *sum-of-products* is a Boolean sum of product terms corresponding to a function $f(x_1 \ldots x_n) : B^n \to B$ which is 1 for minterm m iff some product term contains m and 0 otherwise.

A sum-of-products is said to *cover* a function f iff it contains all of f's ON-set

[4]but possibly degenerate, if empty

minterms and none of f's OFF-set minterms. A cover may contain any or all of f's DC-set.

The 3 non-empty subsets of B, namely, $\{0\}$, $\{1\}$, and $\{0,1\}$, are often denoted simply 0, 1, and `-`. This allows a terse representation of non-empty cubes by a vector of characters. Figure 2.11, for example, shows the cube `0-1` comprising minterms `001` and `011`, corresponding to product $P = X_1^{S_1} X_2^{S_2} X_3^{S_3}$, where $S_1 = \{0\}$, $S_2 = \{0,1\}$ and $S_3 = \{1\}$. As an expression, P is written $x_1' x_3$.

2.2.2 Symbolic Functions

Because this thesis deals with sequential synthesis, a pivotal class of Boolean functions is that set of functions which operate over *symbolic* domains. This class contains the set of sequential functions involving symbolic states, as are encountered in FSM specifications.

Such symbolic variables are often represented by so-called *multiple-valued input* (or "mvi") variables [39], variables taking their values from sets of positive integers. We now extend the above definitions to describe functions of multiple-valued inputs, known as *mvi functions.*

First, define the sets $P_i = \{0, \ldots, p_i - 1\}$ for $i = 1 \ldots n$ and positive integers p_i. Thus, each set P_i has p_i members. The P_i define a multiple-valued domain $P_1 \times \cdots \times P_n$, whose points are *multiple-valued minterms.* For example, $n = 2$, $p_1 = 2$ and $p_2 = 4$ define the sets $P_1 = \{0,1\}$ and $P_2 = \{0,1,2,3\}$ and the multiple-valued domain $\{0,1\} \times \{0,1,2,3\}$. Note that in this example, P_1 identifies the Boolean domain B.

A *completely-specified mvi function* is a mapping $P_1 \times \cdots \times P_n \rightarrow B$, while an *incompletely-specified mvi function* is a mapping $P_1 \times \cdots \times P_n \rightarrow \{0, 1, *\}$.

Now, given set $S_i \subseteq P_i$, a *multiple-valued literal* $X_i^{S_i}$ defines a function $f_{X_i}(x_i) : P_i \rightarrow B$ which is 1 iff $x_i \in S_i$ and 0 otherwise.

An *mvi product term* is a Boolean product of literals $P = \prod\limits_{i=1 \ldots n} X_i^{S_i}$, corresponding to the function $f(x_1 \ldots x_n) : P_1 \times \cdots \times P_n \rightarrow B$, defined as above.

The definitions for sums of products, and so on, extend in similar fashion.

A common representation for multiple-valued inputs is *positional-cube notation* [39]. For each set having N possible members, an N-bit vector is used. A 1 in the i'th position indicates that the i'th member is present in the set; a 0 indicates its absence.

Example 2.1 For example, the set $S_1 = \{0, 1, 3\}$ taken from the domain $P_1 = \{0, \ldots, 4\}$ could be represented in positional-cube notation as `11010`. A vector of 1's corresponds to a "don't-care" input. Positional-cube notation has the advantage of making set operations simple and efficient:[5] set union is implemented as the bit-wise OR of positional cubes, set intersection, as the bit-wise AND, and so on.

The cube-table format for an FSM of n binary inputs and m binary outputs is a set of 4-tuples $\langle I, PS, NS, O \rangle$, where I and O are cubes in B^n and B^m, respectively, $PS \subseteq S$ is a set of present states, and NS is either a next state or the symbol `*`. Each 4-tuple indicates a single set of output values and next state for all total states contained by $\langle I, PS \rangle$. The symbol * indicates that the next state is unspecified in all contained total states. A total state not contained by any 4-tuple has unspecified next state and output values. Figure 2.12 shows a cube-table representation for the FSM of Figure 2.4. The entry in the fifth row indicates next-state `d` and any output for all total states in input column `01` and states `c` or `d`. Note that a given FSM typically has many distinct cube-table representations.

2.2.3 Hazard-free Logic Minimization

Having laid the foundation for combinational logic minimization in the synchronous domain, we turn to the examination of the analogous problem in the asynchronous domain. This section reviews a classic problem in asynchronous combinational logic implementation that is central to burst-mode synthesis: hazard avoidance. In particular, com-

[5] for the small sets typically encountered in FSM specifications

```
IN PS  NS OUT
00 a    a  0
0- b    b  1
00 c    c  1
01 a,b  b  1
01 c,d  d  -
11 a,d  a  1
10 a,b  *  0
10 c,d  d  1
```

Figure 2.12: A cube-table representation for the FSM of Figure 2.4

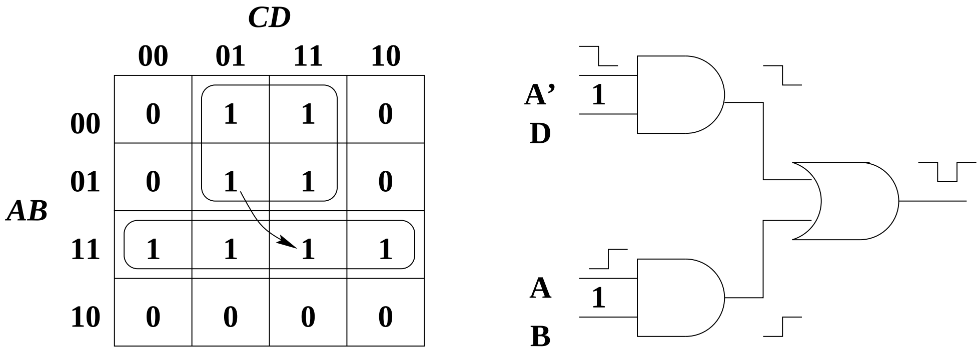


Figure 2.13: A hazardous circuit implementing $F = A'D + AB$

binational hazards are defined, along with a characterization of the conditions for their avoidance [106, 139, 49, 6].

For combinational asynchronous circuits, combinational hazards [106] can cause output glitches to manifest, as shown in the simple 2-level circuit of Figure 2.13. For that circuit, a transition from input vector `ABCD = 0101` to `1111` nominally induces a stable value of 1 on the OR gate's output. However, if the upper AND gate is sufficiently faster than the lower one, a glitch will appear on the OR gate's output. This spurious output change might cause an observing circuit to react, which might be an undesired response.

The circuit in Figure 2.14, however, never produces a glitch for the given input

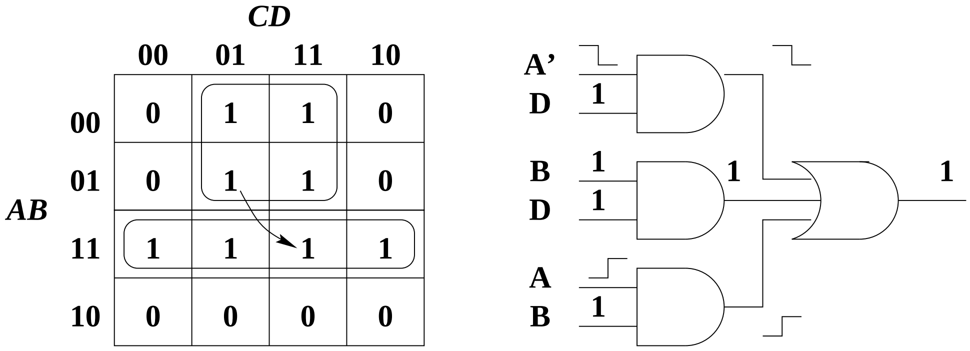


Figure 2.14: A hazard-free circuit implementing F

transition. A single AND gate maintains a steady 1 value throughout the transition, which in turn keeps the OR gate's output stable at 1. Thus, an observing circuit sees only the intended (or specified) transitions on the output.

We now define several basic terms, leading to precise definitions of function and logic hazards under multiple input changes. The following definitions are adapted from those in [101].

A *transition cube*[6] is the cube in an n-dimensional Boolean space spanning a given *start point* and an *end point* (both minterms in that space). Specifically, a transition t from start point A to end point B is denoted $[A,\, B]$. t embodies a set of variable changes, namely, those variables whose values differ between A and B. As expected, t contains all minterms that can be reached along any minimum-length path from A to B (i.e. under any ordering of the individual variable changes).

We define the *open transition cube* $[A,\, B)$ as the set of minterms $[A,\, B] - B$. This consists of the set of maximal sub-cubes spanning all minterms in $[A,\, B]$ except B.

A Boolean function $f(x)$ undergoes a *static transition* for $[A,\, B]$ iff $f(A) = f(B)$. On the other hand, when $f(A) \neq f(B)$, f is said to undergo a *dynamic transition*. Ordinarily, we only consider transition cubes over which f is completely defined; embedded

[6]We often use the terms *transition* and *transition cube* interchangeably, where no ambiguity arises.

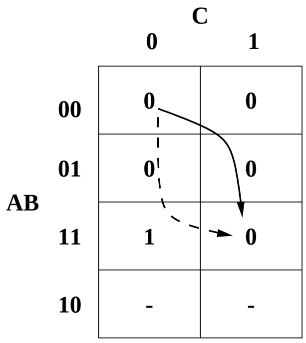


Figure 2.15: Function exhibiting a static-0 function hazard

don't-care points are not allowed. We speak of $0 \rightarrow 0$ transitions when $f(A) = f(B) = 0$, $1 \rightarrow 1$ transitions when $f(A) = f(B) = 1$, and so on. $0 \rightarrow 0$ transitions are often referred to as *static-0* transitions, and $1 \rightarrow 1$ transitions as *static-1* transitions.

A *function hazard* exists for function f during transition t iff f does not change (weakly) monotonically for some path through t.

Definition 2.2 *A* static function hazard *exists for f during $t = [A, B]$ iff a) $f(A) = f(B)$ and b) $\exists C \in [A, B]$ such that $f(C) \neq f(A)$.*

A dynamic function hazard *exists for f during transition $t = [A, B]$ iff*

1. $f(A) \neq f(B)$
2. $\exists C \in [A, B)$ such that $C \neq A$ and $f(C) = f(B)$ and
3. $\exists D \in [C, B)$ such that $D \neq C$ and $f(D) = f(A)$.

In other words, there exists a path from A to B for which f *must* change three times. Figure 2.15 and Figure 2.16 depict a pair of functions having static and dynamic function hazards. For each, the path that manifests that hazard is also shown.

If a function f has a function hazard for a given transition, it is *always* possible to find an assignment of gate delays that causes *any* implementation to trace the hazardous path, thereby producing an output glitch. As a result, in the sequel, we restrict attention to logic implementations of functions that are function-hazard-free.

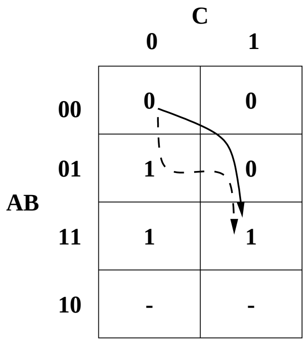


Figure 2.16: Function exhibiting a dynamic function hazard

A combinational logic circuit $\mathcal{C}$ implementing a function f has a *logic hazard* for transition t iff there exists some assignment of gate delays for which the output does not change (weakly) monotonically during t. That is, $\mathcal{C}$ has a logic hazard if its output will glitch during t, given an arbitrary arrangement of gate delays.

Definition 2.3 *Circuit $\mathcal{C}$ has a* static logic hazard *for $t = [A, B]$ iff*

1. $f(A) = f(B)$, and

2. $\mathcal{C}$'s output is not monotonic under some delay assignment

The circuit of Figure 2.13 has a static-1 logic hazard.

Definition 2.4 *Circuit $\mathcal{C}$ has a* dynamic logic hazard *for t iff*

1. $f(A) \neq f(B)$, and

2. $\mathcal{C}$'s output does not change monotonically under some delay assignment

It is worth noting that in general a given function f will be function-hazard-free for some transitions, but not others. Likewise, a circuit $\mathcal{C}$ implementing a function f may exhibit logic hazards only for certain transitions. The latter fact is quite important to the synthesis of asynchronous circuits.

Conditions for a Hazard-Free Two-level SOP Implementation

We now present the conditions for logic hazard-freedom of a two-level sum-of-products implementation of a Boolean function under multiple input changes, as formulated by Nowick and Dill [101].[7]

Theorem 2.5 *A sum-of-products $P = \sum p_i$ implementing f is always static-hazard-free for static-0 transition t.*

For example, the trio of products in Figure 2.17 is static-hazard-free for the static-0 transition from $000 \rightarrow 001$ (not shown in the figure).

Theorem 2.6 *A sum-of-products $P = \sum p_i$ implementing f is static-hazard-free for static-1 transition t iff there exists some product p_i which completely contains t.*

The trio of products in Figure 2.17 is static-hazard-free for the static-1 transition from $101 \rightarrow 111$ (not shown).

Theorem 2.7 *A sum-of-products $P = \sum p_i$ implementing f is dynamic-hazard-free for a $0 \rightarrow 1$ (resp.. $1 \rightarrow 0$) transition t iff every product p_i that intersects t contains the end (respectively, start) point.*

The pair of products $\{AC, BC\}$ in Figure 2.17 is dynamic-hazard-free for the $1 \rightarrow 0$ transition from $111 \rightarrow 001$.

As a consequence of the last condition, if P implements f, the maximal $1 \rightarrow 1$ sub-transitions of a dynamic transition t will be hazard-free if P is dynamic hazard-free for t. For a proof of this fact, see [101].

The following definitions are useful in setting up the covering problem in the next section.

[7]We restrict attention to sensible covers in which no product contains a literal and its negation.

Definition 2.8 *The* required cubes *of f for a set of transitions T are the transition cubes of Theorem 2.6 (for static-1 transitions) and the maximal static-1 sub-cubes described by Theorem 2.7 (for dynamic transitions).*

The required cubes for the dynamic transition of Figure 2.17 are the cubes AC and BC.

Required cubes are so named because each must be contained by some product for a two-level SOP cover to be logic-hazard-free.

Definition 2.9 *The* privileged cubes *of f for a set of transitions T are the transition cubes of Theorem 2.7.*

Privileged cubes serve to constrain the set of implicants that can be used in logic-hazard-free two-level SOP implementations. In particular:

Definition 2.10 *A product p* illegally intersects *a privileged cube π iff it intersects π but does not contain its start point.*

A product that illegally intersects a privileged cube may turn on *in the middle of a transition*, thereby introducing a glitch. In the circuit of Figure 2.17, for example, the product $A'B$ intersects the privileged cube at `011`, causing a glitch should the input changes trace a path through that point. Clearly, products that illegally intersect privileged cubes cannot be used in hazard-free SOP implementations.

Illegal intersections can sometimes be avoided in one of two ways: 1) by using a smaller implicant, or 2) by using a larger implicant. We examine each possibility in turn.

The first solution is to reduce the implicant to one that does not intersect the transition at all. In the case of Figure 2.17, the implicant $A'BC'$ avoids the illegal intersection. The smaller implicant is not hazardous because it does not turn on at all during the given transition. Due to the potentially complex relationships among transitions, however, it may be impossible to find a smaller product which has no illegal

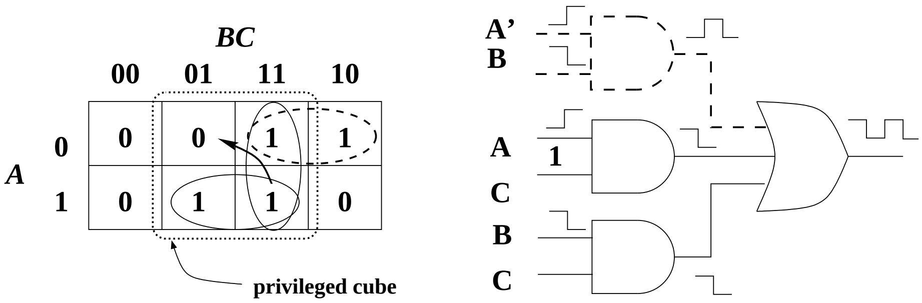


Figure 2.17: An illegal intersection and resulting dynamic logic hazard

intersections.[8]

The second solution is to use a larger implicant which contains the start point, avoiding the illegal intersection. Again, such an implicant does not always exist, as shown in our example; the larger product B is not even an implicant of f.

Some functions have no hazard-free implementation for a given set of transitions, even if they are function-hazard-free. Consider Figure 2.18, for example, in which a required cube for one transition illegally intersects the privileged cube of another. This property is a key difference between hazard-free and ordinary 2-level logic minimization; the latter always has a solution. However, many hazard-free minimization problems encountered in practice do have solutions.

Hazard-freedom is not solely the concern of logic minimization. This is so because up-stream transformations such as state minimization and state encoding play a role in defining the functions and transition sets presented to hazard-free logic minimization. These transformations must therefore be engineered so as to guarantee that a hazard-free logic implementation exists. For example, the UCLOCK, 3D, and MINIMALIST synthesis paths take great care in early synthesis steps to ensure hazard-free logic.

[8]For example, the smaller product may illegally intersect another transition which the larger one does not, if the larger product contains that transition's start point.

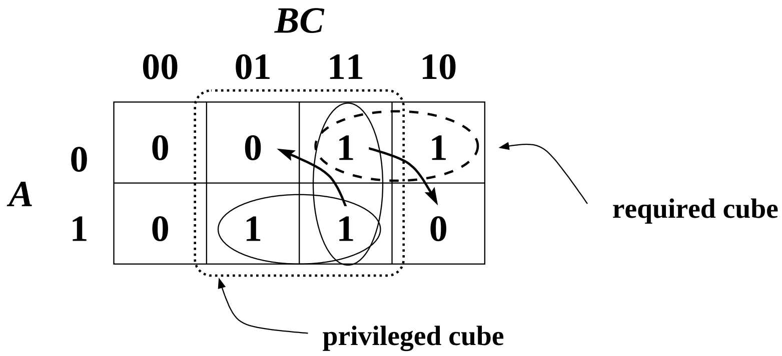


Figure 2.18: A function having no hazard-free implementation

Exact Two-Level Hazard-Free Logic Minimization

Given the above correctness conditions, we now define the problem of finding a minimal logic implementation, i.e., the problem known as hazard-free logic minimization. Because exact hazard-free two-level logic minimization is central to several parts of this thesis, we also briefly review a solution method developed by Nowick [101]. The formulation uses the framework of ordinary two-level logic minimization [117], but extends it in several significant ways.

The problem is defined thus:

Problem 2.11 *Given a Boolean function f and a set T of transitions, find a sum-of-products of minimum cardinality that implements f and is logic-hazard-free for all transitions in T, if any such sum-of-products exists.*

The classic Quine-McCluskey method for exact two-level logic minimization (the basis for most exact methods) uses prime implicants to implement the ON-set of the function under consideration. First, prime implicants are formed from the function's specification. Next, the selection of prime implicants to cover all ON-set minterms is cast as a *unate covering problem.* An implicant is said to cover a minterm if and only if the implicant contains the minterm.

Thus, the components involved in the covering problem are:

Covering objects Prime implicants

Objects to cover ON-set minterms

The solution to the covering problem is thus a minimum-cardinality selection of prime implicants that covers all ON-set minterms.

Exact two-level hazard-free logic minimization proceeds along the same path. Note however that the minimization problem is defined not only with respect to a Boolean function, but also *with respect to a set of specified transitions.*

The steps for two-level hazard-free logic minimization are as follows:

1. The set of required cubes and privileged cubes are formed from f and T. These cubes capture the covering requirements mandated by the above theorems.

2. A novel form of prime implicant, the *dynamic hazard-free prime implicant* (DHF-prime) is generated. DHF-primes are maximal implicants that do not illegally intersect the privileged cube of any transition.

3. A unate covering problem is formed, using the following components:

   Covering objects DHF prime implicants

   Objects to cover Required cubes

4. The unate covering problem is solved, resulting in a selection of DHF-primes which cover f without hazards, if a solution exists.

2.3 Sequential Hazards

In addition to the combinational hazards of the previous section, asynchronous synthesis must deal with sequential hazards. This section describes the two basic types of sequential hazards, namely, critical races and essential hazards. As mentioned earlier, critical races

	00	**01**	**11**	**10**	
S0	**S0,0**	**S0,0**	**S3,0**	**S0,0**	**000**
S1	**S1,0**	**S1,1**	***S1,1***	**S0,0**	***010***
S2	**S0,0**	**S1,1**	**S3,0**	**S2,0**	**110**
S3	**S3,1**	**S3,1**	**S3,0**	**S2,0**	**111**

Figure 2.19: Asynchronous flow table and an encoding exhibiting a critical race

are easily avoided through the use of a proper encoding. Tracey's classic method for deriving such an encoding is presented.

2.3.1 Critical Races

An asynchronous sequential circuit has a critical race [68, 139] when an input transition causes two changing state variables to "race," such that the machine's state depends on the outcome of the race. Critical races can be prevented by judicious encoding [137].

As an example, consider the asynchronous flow table and 3-bit state assignment in Figure 2.19. Notice that the transition in column `11` from `S0` to `S3` involves a change to *all three* state bits. Note also that state `S1` is stable, while `S2` is not. Thus, depending upon which state bit change propagates more quickly, either intermediate state `010` (corresponding to `S1`) or `110` (corresponding to `S2`) will be reached. In the former case, the machine incorrectly settles in `S1`; in the latter, the machine correctly settles in `S3`.[9]

A judicious (*critical race-free* [137]) encoding avoids this hazardous behaviour by ensuring that no two such potentially interfering transitions share intermediate states. Note further that *no* choice of logic implementation alone can prevent this problem.

Two distinct cases exist, both arising from the interference between two transitions in a given input column:

[9] In fact, depending on the nature of the delay elements used, it is also possible for the machine to oscillate between two states.

1. an unstable transition and another unstable transition [column `00` in Figure 2.19]

2. a stable transition and an unstable transition [column `11` in Figure 2.19]

Definition 2.12 ***["Tracey" Conditions]*** *An encoding for a given flow table is* critical race-free *iff in each input column, for all transitions $s_a \rightarrow s_b$ and $s_c \rightarrow s_d$ (where $s_b \neq s_d$), there exists some state bit which has a 0 value for s_a, s_b and 1 for s_c, s_d.*

The state bits satisfying the above necessary and sufficient condition ensure that the portions of the Boolean N-space spanned by the two transitions are kept distinct. Thus, the resulting machine is guaranteed to settle in the correct destination state.

The classic procedure for determining a minimum-length encoding satisfying the above constraints is presented in Chapter 3.

Types of State Assignment

There are many possible kinds of state assignments; for a partial taxonomy, see [139]. Perhaps the most common class is that of Unicode Single-Transition-Time (USTT) assignments, which:

1. assign a single code to each symbolic state (unicode), and

2. employ only direct transitions which do not multi-step through intermediate binary states (single-transition-time).

2.3.2 Essential Hazards

The third issue confronting asynchronous sequential synthesis is that of essential hazards. As mentioned earlier, the problem arises in certain machines when some arrangement of circuit delays allows a state change to complete before the input change is fully processed. This problem is an inherent property of the sequential function [139], and cannot be avoided by judicious encoding or logic implementation.

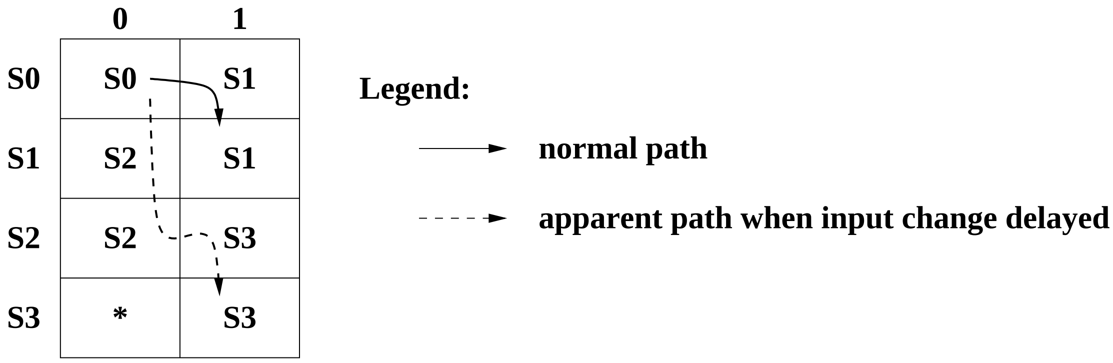


Figure 2.20: Asynchronous flow table having an essential hazard

Example 2.13 The simple flow table in Figure 2.20, taken from [139], illustrates the problem. If, during the transition from state s_0 to s_1 (upon input transition $0 \rightarrow 1$), the state change is seen by the next-state logic before the input change, the next-state logic will effect a change to state s_2. Once the original input change finally propagates through the next-state logic, an incorrect transition to state s_3 takes place. Observe that this entire sequence of events was initiated by a single valid transition on the circuit's input.

Fortunately, a simple solution is always possible. In particular, sufficient delays in the feedback path will ensure that the input change is completely processed *before* the state change. In terms of the example of Figure 2.20, the delay ensures that the next-state logic will see the total state $\langle 1\, s_0 \rangle$ rather than the unintended $\langle 0\, s_1 \rangle$.

2.4 Input Encoding

A central topic for this thesis is that of the input encoding problem, which plays a pivotal role in most of the algorithms developed herein: symbolic logic minimization, state assignment, and state minimization. This section introduces the problem, as it was first presented.

The *input encoding problem* was posed by De Micheli in [40] as one of encoding the

0	ADD	0							
0	SUB	1							
0	MUL	0							
1	ADD	0	ADD = 00						
1	SUB	1	SUB = 10				-	10	1
1	MUL	1	MUL = 11				1	1-	1

(a) a symbolic function (b) an encoding (c) the binary cover

Figure 2.21: A symbolic function, an encoding for the symbolic inputs, and corresponding binary cover

symbolic inputs of a combinational function such that output logic cardinality is exactly minimum, under a given cost function. Figure 2.21 depicts such a symbolic function with one binary input, the symbolic input to be encoded, and one binary output. To its right appear an optimum encoding and a 2-level logic cover for the binary function that results from the encoding. Although De Micheli's presentation dealt exclusively with 2-level SOP logic and product cardinality, the problem also extends naturally to multi-level logic and other forms.

Several methods have been proposed for solving the input encoding problem, including an exact method by De Micheli [40]. This approach uses mvi minimization [118] as a form of symbolic logic minimization to derive a symbolic logic cover, using a result due to Sasao [123]. This symbolic cover is then used as the basis for a constrained encoding step. In particular, encoding constraints in the form of dichotomies [137], [25] are generated that ensure the existence of a binary logic cover of identical cardinality and fundamentally the same structure. A straightforward instantiation of the symbolic cover with the encoding produces the binary logic cover. If exact mvi minimization was performed, the result is an exact solution to the input encoding problem. That is, the binary cover has exactly minimum cardinality over all possible encodings.

To illustrate the process, Figure 2.22 displays again the symbolic function of Figure 2.21. From this, a minimum-cardinality symbolic logic cover of two products is formed, shown to the function's right. Encoding constraints are generated which ensure

0	ADD	0			
0	SUB	1			
0	MUL	0			
1	ADD	0			
1	SUB	1	1	SUB,MUL	1
1	MUL	1	-	SUB	1

(a) a symbolic function (b) a minimum symbolic cover

ADD = 00				
SUB = 01	1	-1	1
MUL = 11	-	01	1

(c) an optimum encoding (d) the binary cover

Figure 2.22: A symbolic function, a minimum symbolic cover, an encoding, and instantiated cover

that the symbolic cover, when instantiated with the resulting codes, is a cover for the corresponding binary function. Such an encoding appears in Figure 2.22. The minimum-cardinality instantiated binary cover appears at right, and is obtained by merely "plugging" the state codes into the symbolic input field.

2.5 Unate and Binate Covering

This section defines the unate and binate covering problems. These abstract problems appear at the core of many VLSI CAD algorithms, for example, two-level logic minimization, multi-level logic minimization, state encoding, and state minimization.

The unate covering problem is stated as follows [117]:

Problem 2.14 (Minimum Unate Covering) *Given a binary matrix A and a cost c_j for each column of A, find a binary row vector x such that $A \cdot x^T \geq [1, 1, \ldots, 1]^T$ and the sum $\sum_{j=1}^{m} x_j c_j$ is minimum.*

Row r_i of A is said to be "covered" by those columns c_j for which $A_{i,j} = 1$. Thus, the problem can be thought of as finding a selection of columns (as signified by the 1's in the solution vector x) which cover all rows of A and has minimum cost.

$$A = \begin{bmatrix} 1 & 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 1 & 0 & 1 & 0 \\ 1 & 0 & 1 & 0 & 1 & 1 \\ 0 & 0 & 1 & 1 & 0 & 1 \end{bmatrix} \qquad E = (c_1 + c_2)(c_2 + c_3 + c_5)(c_1 + c_3 + c_5 + c_6)(c_3 + c_4 + c_6)$$

Figure 2.23: A unate covering matrix A and the equivalent POS expression E

Minimum unate covering is also frequently cast as a restricted form of the minimum-cost satisfiability problem.

Problem 2.15 (Minimum-Cost Satisfiability) *Given a set of decision variables* $X = \{x_1, \ldots, x_n\}$*, a corresponding set of weights* $W = \{w_1, \ldots, w_n\}$*, and a Boolean POS expression* E *over* X*, find the minimum-cost assignment of* X *for which* E *is true.*

In particular, for unate covering, each column is associated with a Boolean variable, for which a true value corresponds to the selection of the corresponding column. Hence, the POS expression has one disjunctive clause for each row r_i of A, of the form $\sum c_j$, $\forall j$ such that $A_{i,j} = 1$. It is easy to see that the expression so formed is in fact unate, giving rise to the name. Figure 2.23 shows a covering matrix and the corresponding POS expression.

The classic algorithm [117] for solving the unate covering problem applies various reductions, such as block partitioning, essential elimination, row and column dominance, and so on, until a *cyclic core* is reached. At this point, a branching operation considers selecting each remaining column in turn, discarding all but the lowest-cost alternative.

In part because unate covering is NP-complete [58], high-quality algorithms such as Mincov [118] (used within Espresso-Exact and Hfmin) use careful cost-bounding calculations and branching heuristics to prune the search tree as much and as early as possible. In fact, even recently-introduced powerful implicit methods [30, 70] use many of the same reduction and bounding operations, although the implementation details differ considerably, due to the nature of the data structures they employ.

An important generalization of the unate covering problem, **binate covering**, also appears in many places in VLSI design. Its ability to model implications among the selection of various objects, for example, is particularly useful. Most notably for the purposes of this thesis, it appears in the classic formulation of state minimization [60, 70], and is thus discussed in Chapter 4.

Chapter 3

CHASM: Optimal State Assignment for Asynchronous State Machines

This chapter presents the first contribution of this thesis, namely, **CHASM** (**C**oding for **H**azard-free **AS**ynchronous **M**achines), the first systematic method for the optimal state assignment of asynchronous state machines. In other words, CHASM chooses a state assignment for which the logic after encoding is optimum under some cost function. Because CHASM operates on asynchronous machines, it must ensure that the resulting machine implementation is both critical race-free and hazard-free.

The structure of the chapter is as follows. First, we define the generic problem of optimal state assignment, and relate it to the problem solved by CHASM in Section 3.1. Then, some background on the problem of optimal state encoding in the synchronous domain is given in Section 3.2. Particular attention is paid to two synchronous methods, KISS and NOVA, whose contributions are leveraged by CHASM. Section 3.2 also reviews previous work on state assignment for asynchronous machines. CHASM itself is the subject of subsequent sections. Section 3.3 defines the precise problem solved by CHASM, and offers an overview of the method. Next, Section 3.4 gives a brief introduction to multiple-valued input (mvi) hazard-free logic minimization, which is a cornerstone in the

CHASM method. Then, the method is described, and the key algorithms are presented in detail, in Section 3.5. The application of the method to implementations employing outputs as fed-back state variables appears in Section 3.6. Core theoretical results of CHASM's correctness and optimality appear in Section 3.7. Finally, experimental results in Section 3.8 demonstrate CHASM's performance relative to competitive methods.

3.1 Overview of CHASM

We begin the section by defining the optimal state assignment problem:

Definition 3.1 (Optimal State Assignment) *For a given FSM, find a state assignment whose corresponding implementation has minimum cost over all possible state assignments.*

Optimal state assignment is important because the state encoding can have a significant impact on the quality of the logic implementation [146, 44].

Many cost metrics are possible, including area, performance, power, as well as complex cost functions combining several simpler metrics. Likewise, many logic implementation structures are possible, such as SOP, XOR-based, and other multi-level forms. Because of the extremely large number of implementations when considering all logic forms, a single form is customarily selected. The minimum cost is then defined relative to the space of implementations using that form of logic.

Synchronous state assignment methods are inadequate for asynchronous designs, however, since their application may result in machines having both critical races and logic hazards. This section thus considers two related problems in the synthesis of asynchronous state machines: *critical race-free state encoding* and *hazard-free logic minimization.*

In many previously existing asynchronous synthesis trajectories [150, 102], these problems are solved separately, so that state assignment is typically performed without regard to the optimality of the logic implementation. Such a decoupling often leads to unnecessarily expensive or slow solutions. (A more detailed treatment of previous work is given in Section 3.3.1.)

CHASM builds on algorithms recently introduced to solve two constrained optimal state assignment sub-problems for asynchronous state machines [54]. The first solved an optimal critical race-free assignment problem, but ignored hazard issues. The second solved a combined hazard-free/critical race-free assignment problem limited to *single-input change (SIC)* asynchronous state machines. CHASM generalizes this work, and solves a combined hazard-free/critical race-free assignment problem for the class of multiple-input change (MIC) state machines known as burst-mode [103, 150, 102]. Further, CHASM addresses the optimal encoding of machines employing fed-back outputs, which often simplify the next-state implementation. Finally, CHASM offers a variety of operating modes to help best target the application's cost function.

A key contribution of this method is that it produces **exactly minimum hazard-free (two-level) output logic**, over *all* possible critical race-free assignments. This result is significant since the latency of an asynchronous machine is determined by its output logic: there are no clocks or latches. For next-state logic, the approach leads only to an approximate solution. However, in practice, high quality solutions are produced for next-state logic as well, ranging up to 17% overall improvement. This is the first general method for the optimal state assignment of hazard-free MIC asynchronous state machines.

3.2 Background

3.2.1 Optimal State Assignment for Synchronous Machines

This section reviews a successful approach to optimal state assignment for synchronous machines, KISS, on which CHASM is partly based.

The goal of the KISS method is to find a binary encoding of the machine's symbolic state that yields a minimum cardinality sum-of-products implementation.

In KISS [40], the optimal state assignment problem is cast as an instance of the *input encoding problem*. Input encoding, as described in Chapter 2, has been successfully applied by several researchers to optimal state assignment for synchronous FSM's; see, e.g., [40], [146], or [149].

Intuitively, the KISS method subjects the flow table to a simple transformation, after which the input encoding method described in Section 2.4 can be directly applied.

The KISS algorithm has four steps:

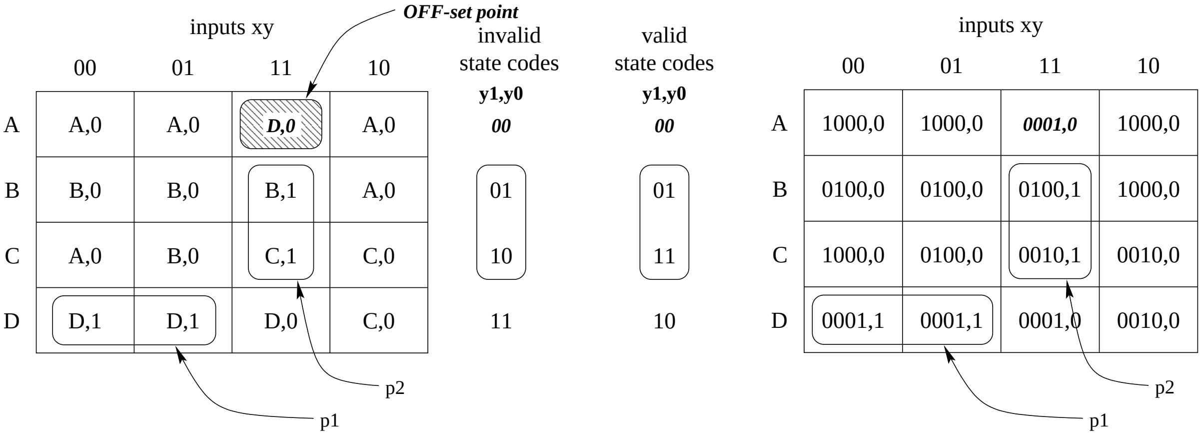
1. Transform flow table into "input-encoding form"
2. Generate a minimal symbolic logic cover
3. Generate a set of encoding constraints
4. Solve the encoding constraints to produce a state assignment

Flow Table Transformation

A simple transformation is needed before multiple-valued input minimization can be applied as it was in Section 2.4. This is because mvi minimization does not handle symbolic *outputs* such as the FSM's next-state in the present problem. Thus, the table is first transformed so that each next-state is treated as a distinct binary function.

The transformation is depicted in Figure 3.1 for a simple FSM. The next-state is "1-hot" encoded; that is, each of the N next-states is assigned a unique N-bit code consisting of a single 1 and $N-1$ 0's. For example, next-state A is assigned 1000; B is assigned 0100, and so on.

With the flow table now in "input encoding form", the procedure continues with the input encoding method, exactly as described in Section 2.4.



(a) A simple FSM and two encodings (b) FSM after transformation

Figure 3.1: A simple FSM and its input-encoding transformation

Symbolic Logic Minimization

Having cast the symbolic logic minimization problem as ordinary mvi minimization, step two solves the problem using *espresso-mv* [118]. An exactly minimum-cardinality symbolic cover, consisting of a set of symbolic implicants, is thus formed.

The key to the claim of optimality in this approach is the derivation of a binary implementation of the machine by a simple *one-for-one mapping* (called *instantiation*) of the minimum symbolic cover. In so doing, the binary implementation's cardinality is guaranteed to be equal to that of the symbolic cover. Conversely, the cardinality of any given symbolic cover is a direct measure of the cardinality of the corresponding binary implementation. Thus, symbolic logic minimization actually serves to minimize the cardinality of the binary implementation as well.

Because of the transformation applied above, however, input encoding can only approximate optimal state assignment. This is true in part because mvi minimization can express only a subset of the set of valid binary logic realizations. In particular, it cannot model logic implementations which share product terms across different next-state bits. This is the case because any minterm in the ON-set of a given binary next-state function is by definition in the OFF-set of every other next-state function. As a result, input encoding is an approximate, though useful, solution to optimal state assignment.

Clearly, for this approach to work properly, the instantiation process must produce a valid logic cover. Unfortunately, the instantiation of a given symbolic cover with arbitrary state codes often produces a faulty logic implementation. Specifically, instantiated implicants may hit the OFF-set.

The state table of Figure 3.1 and the given 2-variable state assignment illustrate the problem caused by instantiation with arbitrary encodings. A minimum-cardinality symbolic cover for the output consists of 2 symbolic implicants: $p_1 = \langle 0- \ \{D\}\rangle$ and $p_2 = \langle 11\ \{B, C\}\rangle$.[1] Implicant p_1 spans the single symbolic state, D, and its instantiation is therefore the binary product $p_1' = \langle 0*\ 11\rangle$ (by simply substituting D's code $y_0y_1 = 11$for p_1's present-state field). However, implicant p_2 spans the *pair* of symbolic states$\{B, C\}$, which forms a *state group*. The smallest binary cube which contains the codes assigned to B and C (called p_2's *group face*) is the *supercube* of those two codes, namely, $--$. The resulting instantiated binary product, $p_2' = \langle 11\ \ --\rangle$, is *not* an implicant, since it contains the OFF-set minterm $\langle 11\ 00\rangle$ corresponding to the OFF-set minterm at total state $\langle 11\ \{A\}\rangle$. Figure 3.2 shows the situation in the Boolean half-space corresponding to $y = 1$ (the middle two input columns of Figure 3.1 (b)).

Encoding Constraint Generation

The algorithm's third step solves the problem, by imposing *face embedding constraints* on the encoding [40]:

> *For each symbolic implicant p (with state group S_p) in the minimum symbolic cover, the group face of p must not contain the code of any state s not in S_p.*

Face embedding constraints solve the problem by preventing instantiated implicants from hitting the OFF-set. Recall that symbolic prime implicants are by definition *maximally*

[1] For simplicity, we consider only single-output implicants in this example; however, in general the method produces multiple-output implicants.

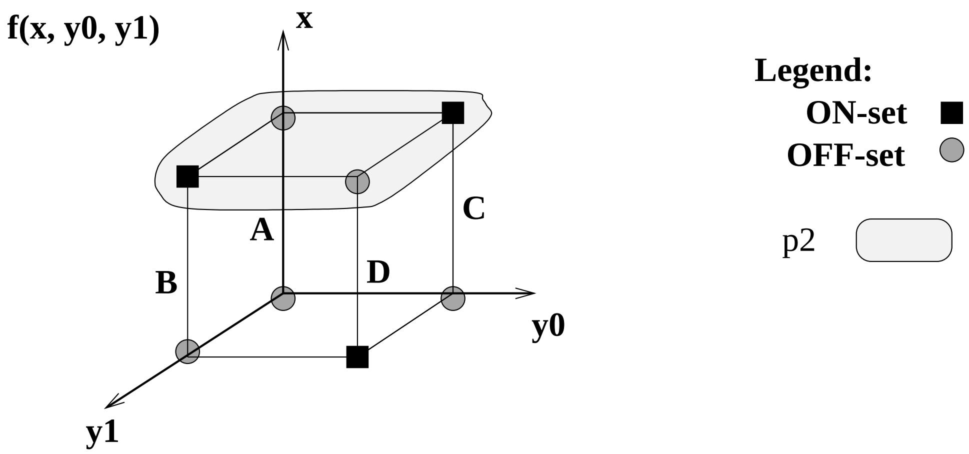


Figure 3.2: Diagram depicting improper encoding of table in Figure 3.1

expanded. In particular, any further expansion would result in an intersection with the OFF-set. Hence, they must not be allowed to span any state after encoding that they did not span before encoding.

Face embedding constraints can be described by *dichotomies* [137, 149].

Definition 3.2 (Dichotomy) *Given a set S (e.g. of states), a* dichotomy *is a bipartition $(T; U)$ of a subset S' of S.*

In a given state assignment, a binary state variable y_i *covers* the dichotomy $(T; U)$ if $y_i = 0$ for every state in T and $y_i = 1$ for every state in U (or vice-versa) [139, 137].

Example 3.3 We illustrate the set of dichotomies needed to ensure a valid binary cover using the table of Figure 3.1. The dichotomy constraints needed for the minimum output cover of that table are as follows. For p_1, whose state group $\{D\}$ is singleton, only trivial dichotomies result. That is, only encodings that do not assign a distinct code to each state could result in another state overlapping $\{D\}$. For p_2, whose state group is $\{BC\}$, the constraints $(BC; A)$ and $(BC; D)$ are generated. These ensure that p_2 does not hit the OFF-set points in states A and D, respectively.

Step three thus forms a set of *n-to-1 dichotomies*, i.e., between each state group S_p (of n states) and each disjoint state $s \notin S_p$, using Algorithm 1.

Algorithm 1 Face embedding constraint generation

```
for each symbolic implicant p in the minimum symbolic cover {
   for each state s not in state group S_p {
      generate face embedding constraint (S_p; s);
   } }
```

Encoding Constraint Solution

The fourth step finds a state assignment that satisfies the encoding constraints. Several exact dichotomy solvers have been developed which produce minimum-length assignments [149, 122].

Example 3.4 We now give an encoding which satisfies the face-embedding constraints. Specifically, the encoding labelled "valid" in Figure 3.1 satisfies the encoding constraints for the symbolic cover consisting of $\{p_1, p_2\}$. In particular, the constraint $(BC;\, A)$ is covered by state bit y_0, since y_0 is assigned 1 for states B and C, and 0 for state A. The constraint $(BC;\, D)$ is also covered by state bit y_0, whose value is 1 for B and C, and 0 for D.

After state assignment, instantiation produces the binary logic implementation. Instantiation proceeds by transforming the implicants in the minimum symbolic cover produced in step two, one by one, as described above. This produces a binary cover of cardinality equal to that of the minimum symbolic cover.

Example 3.5 The instantiated output cover for Figure 3.1 is $\{p_1', p_2'\}$, where $p_1' = \langle 0-\ \ 10\rangle$ and $p_2' = \langle 11\ \ -1\rangle$.

It is sometimes possible to produce a binary cover smaller than that obtained by instantiation by passing the instantiated machine through a binary logic minimizer [40]. This is a direct reflection of the fact that input encoding is only an approximation to optimal state encoding. In particular, although the encoding was not constructed to allow product sharing between state bits, the encoding produced may in fact allow it. If so, a smaller binary cover may result.

Optimal state assignment of synchronous machines has been an active area of research. De Micheli's formulation [40] as the input encoding problem approximates optimal state assignment for SOP logic implementations. Other formulations, e.g., as an *output encoding* or *input/output encoding problem* have also been developed [38, 147, 122, 149].

3.3 Problem Statement and CHASM Overview

We now formulate the synthesis problem to be solved by CHASM, and give an overview of the solution method. Details on the individual steps appear in Section 3.5.

Problem 3.6 (Optimal Critical Race-Free Assignment for Burst-Mode FSM's) *Find a USTT critical race-free assignment for a burst-mode flow table having a **hazard-free** sum-of-products implementation of minimum cost.*

CHASM's synthesis method follows the three basic steps of the KISS algorithm, but with modifications. Like the KISS [40] method, the problem is formulated as an *input encoding problem.*

In step 1, CHASM solves a **symbolic hazard-free minimization problem** for asynchronous synthesis. In this formulation, the symbolic functional specification is transformed as in KISS to a suitable mvi form, and minimized to obtain a minimum symbolic logic cover. Unlike KISS, however, a *hazard-free* mvi logic minimization procedure must be used, namely, HFMIN (described below in Section 3.4).

After symbolic minimization, a **constrained encoding step** is performed. Encoding constraints in the form of *dichotomies* [137, 122] are introduced, which must be satisfied in the context of MIC asynchronous state machines. These constraints are related to the critical race-free constraints introduced by Tracey [137] and the *face-embedding* constraints introduced by De Micheli [40], but *subsume both*.

Finally, the **encoding constraints are solved** using exact and heuristic techniques. The exact procedure makes use of an existing tool, DICHOT [122], while the heuristic procedure uses the simulated annealing mode of NOVA [147]. For the heuristic problem, CHASM uses a novel partitioning of constraints into *compulsory* and *non-compulsory* classes. A weighted annealing algorithm is used to ensure that all compulsory constraints are satisfied.

The following section briefly describes HFMIN, the symbolic logic minimization method used by CHASM in step one. As described earlier, ordinary mvi minimization cannot be used, because the logic must be hazard-free for the machine implementation to work properly.

3.3.1 State Assignment for Asynchronous Machines

The state encoding problem for asynchronous machines has been studied for over 30 years. Several methods have been proposed for minimum-length critical race-free assignment, which ignore logic complexity and hazards, e.g. Liu [83] and Tracey [137] targetted USTT codes, while Saucier [125] and Datta et al. [34] achieved shorter codes with non-STT assignments. Tan [133] and Saucier [124] proposed race-free STT encoding algorithms which heuristically minimize next-state logic; however, the former only provides a hazard-free implementation for SIC operation, and both ignored output logic.

More recent work by Fisher et al. [53] seeks race-free (but non-STT) assignments for large machines, again heuristically minimizing code length while ignoring logic complexity. Finally, Lam et al. [76] present a greedy algorithm which reduces the number of state bits, but only for a very limited class of delay-insensitive circuits.

3.4 Multiple-Valued Hazard-free Two-Level Logic Minimization

This section briefly presents the basic formulation of multiple-valued hazard-free two-level logic minimization. This formulation is the basis for a novel method for *symbolic* hazard-free logic minimization, HFMIN, which lies at the heart of CHASM. In particular, HFMIN plays the same role in CHASM that ESPRESSO plays in KISS: it obtains a symbolic logic cover from which to derive the instantiated binary cover. For a more detailed and theoretical treatment, refer to Appendix A.

The multiple-valued hazard-free logic minimization formulation given here is a straightforward extension of the binary-valued (bv) two-level hazard-free logic minimization problem posed and solved by Nowick and Dill [106]. It generalizes the key concepts of the binary-valued framework, e.g., multiple-input change transitions, privileged and required cubes, DHF implicants, and so on, to multiple-valued domains. With these core analogues in place, the remainder of the framework works identically.

The following makes use of the definitions of mvi minterms, product terms, containment, and so on, given in Section 2.2.

First, we generalize the notion of a multiple-input change (MIC) transition t to mvi domains. Such a transition spans all points in the multiple-valued domain that can be reached during the transition from the start point to the end point. This set of points constitutes a cube in the mvi domain, and is called the **multiple-valued transition cube**.

Example 3.7 Figure 3.3 depicts a binary function over a multiple-valued domain having two binary inputs (xy) and one four-valued symbolic input, with values A, B, C, D. It also shows three multiple-input change transitions, t_1, t_2 and t_3. t_1 has start point $\langle 01\ B \rangle$

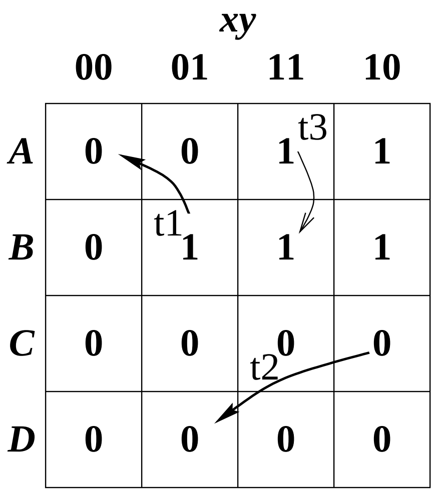


Figure 3.3: A multiple-valued input function and three multiple-valued transitions

and end point $\langle 00\ A\rangle$, while t_2 has start point $\langle 10\ C\rangle$ and end point $\langle 01\ D\rangle$, and t_3 has start $\langle 11\ A\rangle$ and end $\langle 11\ B\rangle$. The transition supercube of t_1 is $\langle 0-\ \ \{A, B\}\rangle$, that of t_2 is $\langle --\ \ \{C, D\}\rangle$, and that of t_3 is $\langle 11\ \{A, B\}\rangle$.

When a symbolic input undergoes a change, *only* the starting and ending values for that symbolic input are considered reachable. Thus, other symbolic values are not possible intermediate points. For example, the supercube of transition t_3 in Figure 3.3 spans states A and B, but neither C nor D.

Next, we classify an MIC transition over an mvi domain, according to the value of the output at the start and end points. The same four types are possible as in the bv case: $0 \rightarrow 0$ (**static-0**), $1 \rightarrow 1$ (**static-1**), and $0 \rightarrow 1$, $1 \rightarrow 0$ (**dynamic**).

Example 3.8 In Figure 3.3, t_1 is a dynamic $1 \rightarrow 0$ transition, t_2 is a static-0 ($0 \rightarrow 0$) transition, and t_3 is a static-1 ($1 \rightarrow 1$) transition.

Now, a **multiple-valued function hazard** is said to exist for a transition if and only if the binary output changes non-monotonically for some path from the start point to the end point lying wholly within the transition cube.

To generalize the notion of logic hazard-freedom to a Boolean sum (OR) of mvi

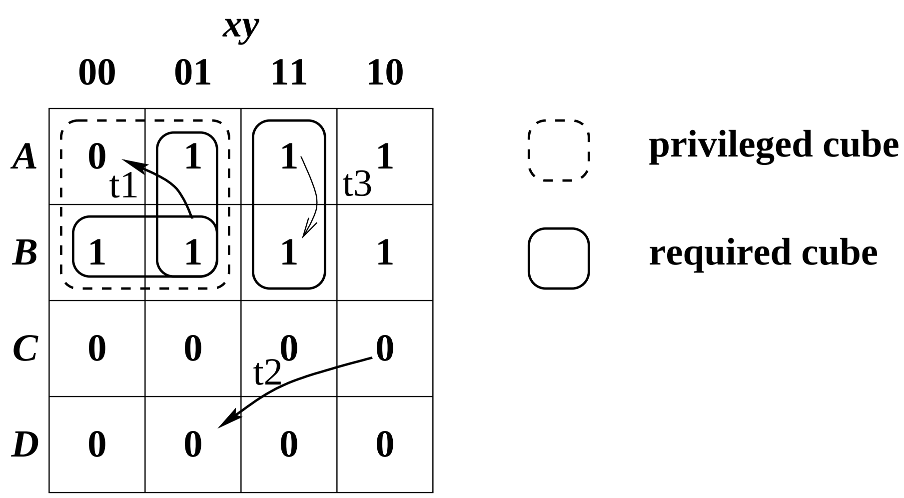


Figure 3.4: Transitions in a multiple-valued function and their privileged and required cubes

product terms, we start by extending the concepts of privileged and required cubes to the mvi domain. An mvi dynamic transition gives rise to an **mvi privileged cube** and one or more **mvi required cubes**, which are the maximal ON-set sub-cubes of the transition. Likewise, a static-1 mvi transition gives rise to a single mvi required cube spanning the entire transition, and no privileged cube. A static-0 mvi transition gives rise to neither a privileged cube nor any required cubes.

Example 3.9 In Figure 3.4, dynamic transition t_1 has privileged cube $\langle 0- \ \{A, B\}\rangle$ and required cubes $\{\langle 01 \ \{A, B\}\rangle, \langle 0- \ \ B\rangle\}$. Static-0 transition t_2 has neither privileged cubes nor required cubes. Static-1 transition t_3 has no privileged cube but a single required cube $\langle 11 \ \{A, B\}\rangle$.

With the above definitions[2], the remaining concepts generalize without further modification. For example, in Figure 3.5 mvi product $\langle -1 \ A\rangle$ **illegally intersects** the privileged cube of t_1 because it intersects t_1 but does not contain t_1's start point. However, mvi product p_3 is an **mvi DHF implicant** because it does not illegally intersect the privileged cube of any specified transition. On the other hand, p_3 is not an **mvi DHF**

[2]along with the standard mvi definitions for intersection, containment, and so on

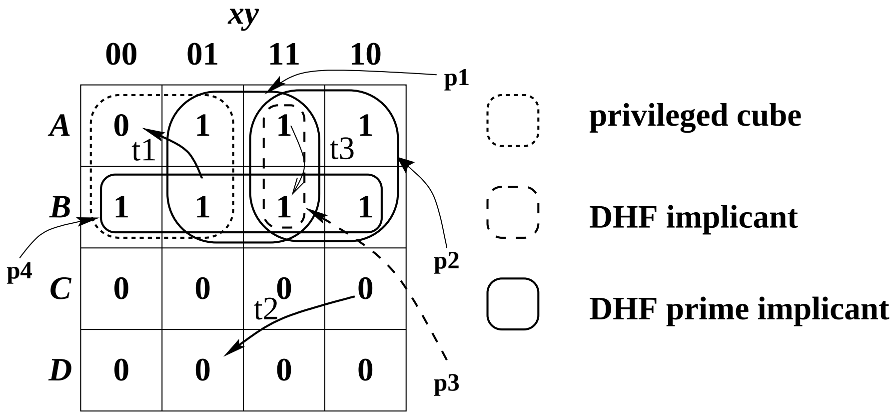


Figure 3.5: DHF implicants and DHF prime implicants for an mvi function

prime implicant since it is contained by another DHF implicant (p_1). p_1 and p_2 are mvi DHF prime implicants.

We now define the conditions for a hazard-free mvi cover. Specifically, a sum of mvi products C is a **hazard-free mvi cover** for a function f for specified input transitions if and only if: (a) no cube of C intersects the OFF-set of f, (b) each required cube of f is contained in some cube of C, and (c) no cube of C illegally intersects any privileged cube. For example, the pair of implicants $\{p_1, p_4\}$ in Figure 3.5 is a hazard-free cover for the given set of transitions.

It is worth pointing out a subtle but important distinction between the hazard-free covering problem as formulated here and the classic two-level logic covering problem. Specifically, the above conditions do *not* require that all ON-set minterms of the given Boolean function be covered. Rather, only those ON-set minterms that lie within a specified transition must be covered. In the example of Figure 3.5, the ON-set minterm at Axy' lies outside all specified transitions, and therefore need not be covered. In fact, it is not covered by the DHF prime cover $\{p_1, p_4\}$ mentioned earlier.

3.5 CHASM Method

We now describe each of the steps in CHASM's method for optimal encoding for asynchronous flow tables.

3.5.1 Symbolic Hazard-Free Logic Minimization

The first step in the CHASM procedure is symbolic hazard-free logic minimization. As mentioned earlier, this is similar in spirit to the mvi minimization used by KISS, but guarantees a hazard-free implementation. The flow table is first transformed as in KISS into "input encoding form," so that the next-state is represented as a set of distinct binary functions. Then, the method of Section 3.4 is applied, which produces a symbolic hazard-free logic cover for the flow table. The resulting cover serves as the basis for the encoding constraints described in the next section.

3.5.2 Encoding Constraints

In this step, encoding constraints are generated based on the symbolic cover. These constraints ensure that the cover will be correctly instantiated.

The face embedding constraints used by KISS for synchronous machines are insufficient for asynchronous machines for two reasons: (1) they do not consider the transient behavior of an asynchronous state machine, and (2) they do not consider hazard-free logic requirements. Therefore, face embedding constraints must be generalized. We consider these two problems in turn in the sequel.

Functional Correctness

A new condition concerns the *functional correctness* of the output and next-state implementations in the presence of state transitions.

Example 3.10 Consider a symbolic implicant $\langle I_1 \; s_0, s_1, s_2 \rangle$ for some binary output function z. Suppose there is a state transition $s_3 \rightarrow s_4$ in input column I_1 during which z should be held at `0`. With the state assignment

$$s_0 = 0000,\; s_1 = 1000,\; s_2 = 1100,\; s_3 = 0110,\; s_4 = 0101$$

the corresponding instantiated binary implicant is $\langle I_1 \; --00 \rangle$. As a result, during the $s_3 \rightarrow s_4$ transition, the state variables can reach the transient value `0100` which would turn on the given implicant, incorrectly forcing the output value to `1`. This problem occurs even though the face embedding constraints for state group $\{s_0, s_1, s_2\}$ are satisfied.

In general, during a transition of two or more state variables, transient points in the total state space are reached *which do not correspond to any symbolic state.* The possibility arises that the group face for some symbolic product term implementing a binary output may intersect such a transient point, thus inadvertently turning on the product term *during* the state transition. If the intended value of that output during the state transition is `0`, the output function will be incorrectly implemented.

A similar problem exists for the next-state function. This case requires a trivial generalization of the condition: if the value of the symbolic function (i.e. the destination state) during the transition differs from that which the product term implements, the machine will be incorrectly realized.

The solution is to add new dichotomy constraints to avoid problems with such state transitions. Unlike the face embedding constraints, these dichotomies are *N-to-2*. For example, the constraint needed for the above example is $\{s_0, s_1, s_2;\, s_3, s_4\}$. The constraint contains the state group of the given symbolic product ($\{s_0, s_1, s_2\}$) on the left-hand side, and the pair of states defining the state transition ($\{s_3, s_4\}$) on the right. The constraint ensures that the transition and the instantiated implicant lie in different halves of the total state space[3]. Hence, the instantiated product will not turn on during the transition.

The above constraints do not directly consider critical race-free encoding constraints per se. In Section 3.7, however, it will be shown that the above constraints in fact subsume all Tracey constraints, and therefore ensure a critical race-free assignment.

In summary, an asynchronous design differs from synchronous designs, since its logic is sensitive to the intermediary states traversed during state transitions. While face embedding constraints ensure that an implicant does not intersect an OFF-set *minterm*, generalized constraints are needed for asynchronous machines to ensure that an implicant does not intersect a *cube* of OFF-set minterms that may be traversed during a state change.

Logic Hazards

The second difference between asynchronous constraints and face embedding constraints addresses the need to avoid *dynamic logic hazards*. Recall that in asynchronous synthesis, a *DHF-prime implicant* must not illegally intersect any privileged cubes. It is possible, however, for a symbolic DHF implicant to intersect a privileged cube *after instantiation*.

The problem is solved by adding encoding constraints to ensure that a DHF mvi prime implicant has no illegal intersections after instantiation. For example, given a symbolic DHF-prime implicant with state group $\{s_0, s_1, s_2\}$ and a privileged cube spanning state s_3, a simple *N-to-1* dichotomy $\{s_0, s_1, s_2;\ s_3\}$ must be generated.

For the class of burst-mode machines under consideration, such hazard-free constraints are degenerate. As indicated earlier, in a burst-mode flow table, dynamic transitions only occur during input bursts: i.e., within a single state. Therefore, each privileged cube has a singleton state group. However, such a dichotomy is already generated as a face-embedding constraint. Therefore, no further constraints need to be generated.

[3]relative to the state bit that satisfies the dichotomy constraint

Constraint Generation Algorithm

The constraint generation algorithm for CHASM is as follows. In addition to the KISS face embedding constraints, Algorithm 2 is used. This algorithm generates *n-to-2* dichotomies, where t is a state transition from an unstable to a stable state.

Note that Algorithm 2 tacitly makes use of the fact that next-states are treated by input encoding as individual binary functions. In particular, an "output o that p implements" (in line 4 of the algorithm) can in fact refer to a next-state function. The correctness of the algorithm thus relies on the fact that a given next-state function is defined to be 0 wherever the machine's specified next-state differs.

Algorithm 2 `CHASM encoding constraint generation`

```
for each implicant p in the symbolic cover {
   for each state transition t {
      if p intersects the input column of t {
         if some output o that p implements has value 0 during t {
            generate dichotomy { stategroup(p); states(t) };
         } } } }
```

Encoding Constraints for Output-Targetted State Assignment

We now describe a special case of the above encoding algorithm, for which we can make stronger claims of optimality. Particularly, recall that input encoding precisely models output logic, but only approximates optimal state encoding. Thus, focusing attention exclusively on the output logic gives an even more potent method. This is accomplished by generating on behalf of next-state only those constraints that are necessary to ensure a valid implementation. In particular, Algorithm 2 can be modified, as shown in Algorithm 3. The sole difference between this and the earlier algorithm is the restriction to *binary outputs* in the conditional clause of the fourth line of the algorithm.

To these constraints are added both face embedding constraints (for output implicants) and Tracey constraints, which ensure a valid next-state implementation.

Algorithm 3 CHASM encoding constraint generation for output-targetted minimization

```
for each implicant p in the symbolic cover {
   for each state transition t {
      if p intersects the input column of t {
         if some binary output o that p implements has value 0 during t {
            generate dichotomy {stategroup(p); states(t)};
         } } } }
```

The modified constraint set works by focusing the cost function sharply on the output logic. As a result, the constraint satisfaction engine can focus on satisfying constraints that assure output logic quality, without the distraction of (approximated) next-state optimality constraints. In particular, when optimality constraints are treated as optional (as is done in fixed-length runs, described below), the less effective next-state constraints will not compete with the more effective output constraints.

3.5.3 Solving Constraints and Hazard-Free Logic Minimization

CHASM's third and final step is to solve the above encoding constraints.

Since all constraints are described as dichotomies, they are solved using off-the-shelf dichotomy solvers.

Constraints are solved using two methods: *exact solution* (using DICHOT [122]) and *heuristic solution* (using a slightly-modified version[4] of NOVA's simulated annealing mode [147]). The goal of the heuristic method is to solve as many constraints as possible given a fixed code-length.

The straightforward application of the heuristic method may result in an *incorrect* implementation. Asynchronous machines require as a bare minimum that the state assignment be critical race-free. Normally, critical race-free (Tracey) constraints are subsumed by CHASM's optimality constraints (see Section 3.7.2). Thus, CHASM need not

[4]The modified version allows passing in an arbitrary set of weighted dichotomies from the outside.

explicitly generate Tracey constraints when using the exact solution mode. However, since the heuristic constraint solver may not satisfy all dichotomies, some Tracey constraints may be left unsatisfied. The resulting state assignment would then have critical races.

The solution is to partition dichotomies into two classes: *compulsory* and *non-compulsory*. Critical race-free constraints are compulsory, and must be satisfied. The remaining constraints are concerned with logic optimality, and are thus non-compulsory, or optional. Because the simulated annealing engine treats *all* constraints as optional, we assign sufficiently large weights to the compulsory dichotomies, to ensure that they are all satisfied. In practice, such an approach has worked well on all available examples.[5]

Finally, once a state assignment is produced, the symbolic machine is instantiated with the resulting encoding. The resulting binary-valued function is then passed through a binary-valued hazard-free logic minimizer to produce a final machine implementation.

3.6 Optimal State Assignment for FSM's with Fed-back Outputs

This section indicates that CHASM's state assignment method works properly for machine implementations using fed-back outputs.

Because CHASM makes no assumptions regarding the form of the asynchronous flow table, CHASM can be used without modification to encode machine implementations using fed-back outputs. In particular, the functional specification used as input to CHASM is first transformed as described in Section 2.1.2, to account for the presence of the fed-back outputs. Then, CHASM's three steps (symbolic logic minimization, encoding constraint generation, and constraint solution) are performed exactly as described in

[5]It is possible that a solution will not satisfy all compulsory constraints. If this occurs, the relative weights can be modified, the run can be repeated to randomly explore another portion of the solution space, or the code length limit can be raised.

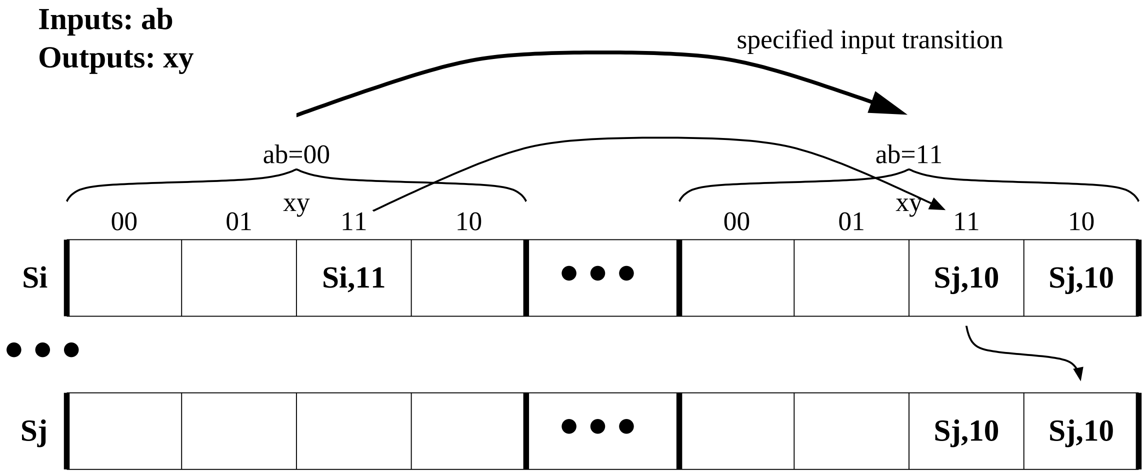


Figure 3.6: A specified transition in the presence of fed-back outputs

Section 3.5.

Actually, the constraint generation method, Algorithm 2, requires a trivial adjustment to work properly in this context. As shown in Figure 3.6, vertical transitions can span more than one input column. Thus, the algorithm must be aware of this fact when it examines DHF-implicant/transition pairs for potential interference. The modified algorithm is shown as Algorithm 4. The sole difference between this algorithm and Algorithm 2 is that the intersection test between the DHF implicant p and the transition t in the first `if` statement accounts for multi-column vertical transitions.

Algorithm 4 CHASM encoding constraint generation for fed-back outputs

```
for each implicant p in the symbolic cover {
   for each state transition t {
      if p intersects the input columns of t {
         if some output o that p implements has value 0 during t {
            generate dichotomy { stategroup(p); states(t) };
         }  }  }  }
```

3.7 Theoretical Results

This section sketches the basic theoretical results for the CHASM method. Specifically, a basic result is first established in Section 3.7.1 regarding the cardinality of the instantiated cover. This is analogous to a basic result from De Micheli's work [40]. Next, the instantiated binary cover is shown to be correct in Section 3.7.2. Finally, a theorem stating CHASM's exact optimality in its output-oriented mode is given in Section 3.7.5.

3.7.1 Machine Instantiation

First, a "pseudo-canonical" state assignment is defined, roughly analogous to the use of a "canonical" 1-hot assignment in KISS. Then, the instantiated asynchronous machine specification (encoded flow table) and binary implementation (cover) is formally defined under this assignment.

Pseudo-Canonical State Assignment

In [40], De Micheli indicates that, for synchronous machines, any symbolic minimized cover can be assigned a 1-hot canonical encoding. The result is a $1 \rightarrow 1$ mapping of symbolic to binary implicants, yielding a canonical cover whose cardinality is identical to that of the symbolic cover. For asynchronous machines, however, a 1-hot encoding is not in general critical race-free [139]; furthermore, it will not generally satisfy the encoding constraints of Section 3.5.2. As a simple alternative, to demonstrate theoretical results, we propose the following: solve the encoding constraints to produce an assignment.[6] This assignment will be called *pseudo-canonical* for the given machine.

[6] In fact, a state assignment which satisfies all possible N-to-1 and N-to-2 constraints can always be found, for a given number N of states. However, such an assignment is prohibitively expensive; for simplicity, we consider a more practical assignment here.

Symbolic Machine Instantiation

An encoding defines a mapping from a symbolic machine specification to an equivalent binary one. There are two components of an asynchronous machine specification: its functional specification and a set of specified transitions. For the functional specification, it is assumed that both ON-set and OFF-set are explicitly defined. The transitions are mapped in the obvious way: each symbolic startpoint (endpoint) $\langle in,\, present\rangle$ maps to the binary startpoint (endpoint) $\langle in,\, code(present)\rangle$.

We can view the functional specification as a set of ON-set and OFF-set cubes. Each symbolic product p (a 4-tuple $\langle in,\, present,\, next,\, out\rangle$), maps onto a binary product $\tilde{p}$, as follows:

$$\begin{array}{lcccc}
p: & in & present & next & out \\
 & \Downarrow & \Downarrow & \Downarrow & \Downarrow \\
\tilde{p}: & in & supercube(codes(present)) & code(next) & out
\end{array}$$

Example 3.11 [Product Instantiation]Under the state assignment $S_0 = 000$, $S_1 = 011$, $S_2 = 100$, and $S_3 = 101$, the symbolic product `<011| {S0,S2} S2 |100>` is mapped to the binary product `<011|-00 100|100>`.

With the above view, mapping an asynchronous symbolic flow table to an encoded table, column transitions require special care. In a symbolic table, a column transition is defined only at its symbolic startpoint and endpoint. However, in an encoded table with a USTT critical race-free assignment, all *intermediate* (transient) entries for the transition must be defined as well. This latter property can easily be guaranteed by constraining the symbolic specification: a *single product* must be used to specify each column transition. This constraint ensures that, for each column transition (*i.e.,* state change), some product will be instantiated which defines all intermediate states in the encoded transition.

Symbolic Cover Instantiation

Given a symbolic hazard-free cover and a resulting state assignment, symbolic implicants can be instantiated by substituting binary codes using the mapping described above, yielding a binary cover C. Note that instantiating a symbolic implicant may produce an empty binary implicant, if its symbolic next-state is mapped to the binary 0-vector. Such an implicant can be dropped from the binary cover.

Unfortunately, the sharing of 1-bits by different state codes may cause *static* transitions for next-state to appear in the binary machine where only *dynamic* transitions appeared in the symbolic machine. To avoid hazards, extra terms must be added to the binary cover: static-1 transitions must each be completely covered by some implicant, while the symbolic dynamic transitions clearly would not have been.

Example 3.12 To understand the problem, consider the input transition shown in Figure 3.7 for a machine with one output. No implicant in the symbolic cover can cover the entire transition, since both output and next-state undergo dynamic transitions. Recall that, under the input encoding model, each next-state is treated as a distinct binary function. Hence, the next-state function S_0 has a $1 \rightarrow 0$ transition, while the next-state function S_1 and the output both have $0 \rightarrow 1$ transitions. However, suppose that S_0 is assigned code 011 and S_1 is assigned 110. In the instantiated machine, the second state bit will then make a $1 \rightarrow 1$ transition. However, since no symbolic cube covered the entire transition, no instantiated binary cube will either, and the second state bit will have a static-1 hazard.

In sum, a naively instantiated cover will fail to properly implement certain static transitions of the next-state variables.

One solution is to add one product term to the instantiated cover for each such static-1 transition. For the above transition, the implicant <0- 011 010 0> would be

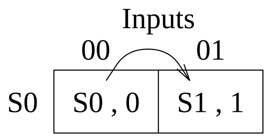


Figure 3.7: A simple transition in a single-output machine

added. In that implicant, the present-state field 011 corresponds to symbolic state S_0, while the next-state field 010 indicates that the implicant contributes only to the second next-state bit. As a result, the canonical cover may have *greater* cardinality than the symbolic cover:

Property 3.13 [Opt-HFCRF Cardinality of Cover]*Let* $\mid S \mid$ *be the cardinality of the symbolic cover,* $\mid C \mid$ *be the cardinality of the binary instantiated cover, and* k *is the number of unstable state transitions in the flow table; then* $\mid C \mid = \mathcal{O}(\mid S \mid + k)$*.*

Note that *this result is a theoretical upper bound only.* In practice, k additional products need not be added. Instead, the instantiated cover C is passed to a binary hazard-free minimizer and *re-run*, to improve results.

By analogy, KISS produces a theoretical upper bound on cardinality based a 1-hot-instantiated cover (although in KISS the upper bound is the cardinality $|S|$ of the symbolic cover; no added terms are required). This 1-hot-instantiated cover in KISS is *neither* guaranteed to have minimum number of products *nor* minimum code length [40]. In practice, shorter codes are sought, and the instantiated cover is likewise re-run through a binary minimizer to improve results [40, 147].

While in theory CHASM is only an approximate solution to the optimal state assignment problem, in practice, it obtains significant improvements over an arbitrary critical race-free encoding method (see Section 3.8).

3.7.2 Correctness of Binary Cover

In the following, let C be the instantiated cover, derived using the pseudo-canonical assignment.

Terms:

M_s	symbolic machine
M_ϵ	machine M_s instantiated by encoding ϵ
$x_j(p)$	j^{th} literal of product p
$\sigma(t)$	start point of privileged cube for transition t
$\tilde{c} = \text{map}(c)$	instantiation of symbolic cube c by some encoding
ϵ_k	the encoding assigned to state s_k
$\epsilon_k[i]$	bit i of ϵ_k
$\tilde{t} = \text{map}(t)$	instantiation of symbolic transition t by some encoding
$\text{map}(S)$	the member-wise mapping of a set S of cubes
C	minimal hazard-free symbolic cover of M_s
$\tilde{C}$	pseudo-canonical cover

We will at times use M_ϵ to denote also the combinational component of M_ϵ, when such usage is unambiguous.

We first need to establish certain characteristics of the instantiation process. In particular, we show that certain basic properties of M_s are preserved by the mapping performed by instantiation. These invariants will be used to prove the desired properties of our canonical implementation of M_ϵ, namely:

1. The critical race-freedom of M_ϵ (Theorem 3.14), and
2. The correctness of the pseudo-canonical logic implementation $\tilde{C}$ (Theorem 3.29).

3.7.3 Critical Race Freedom

Theorem 3.14 *Any state assignment satisfying the encoding constraints of Section 3.5.2 is critical race-free.*

Proof: (See also [54].) In each flow-table input column I, critical race-free constraints are needed to avoid interference between an unstable transition and either stable states (case 1), or other unstable transition (case 2).

Case 1: I contains an unstable transition $t : S_a \rightarrow S_b$, and a distinct stable state S_c. Tracey conditions demand that a *2-to-1* dichotomy constraint $d = \{S_a, S_b; S_c\}$ be satisfied [137]. At the same time, in our framework, the unstable transition t defines a symbolic required cube for next-state S_b, which must be covered. Therefore, some symbolic implicant, p, must cover the transition and implement the destination next-state function, S_b. Hence, present$(p) \supseteq \{S_a, S_b\}$. Note that $S_c \notin$ present(p), since the next-state in $\langle I\, S_c \rangle$ is S_c, while p implements S_b. Now, S_c is stable in I, and so the next-state function S_b is 0 there. Therefore, our encoding constraints include $\{$stategroup(p); $S_c\}$, which subsumes dichotomy d.

Case 2: I contains unstable transitions $t_1 : S_a \rightarrow S_b$ and $t_2 : S_c \rightarrow S_d$, where $S_b \neq S_d$. In this case, a *2-to-2* dichotomy constraint $d = \{S_a, S_b; S_c, S_d\}$ must be satisfied [137]. Again, each unstable transition defines a symbolic required cube for the respective next-state. Therefore, some implicant p covers transition t_1 and implements next-state S_b, so stategroup$(p) \supseteq \{S_a, S_b\}$. Again, $S_c \notin$ present(p) and $S_d \notin$ present(p), since the next-state in $\langle I\, S_c \rangle$ and $\langle I\, S_d \rangle$ is S_d, and p implements S_b. Meanwhile, next-state function S_b is 0 throughout transition t_2. Hence, our encoding constraints include $\{$stategroup(p); $S_c, S_d\}$, which subsumes dichotomy d. $\Box$

3.7.4 Logic Implementation

First the relationship between the specified transitions of M_s and those of M_ϵ is established. Then, the required cubes for the binary outputs and next-state functions of M_ϵ are determined. Finally, the pseudo-canonical cover $\tilde{\mathcal{C}}$ is proven to be a hazard-free cover for M_ϵ, in three steps.

Specified Transitions

Lemma 3.15 (Input transitions) *The specified input transitions of M_ϵ's combinational component are precisely the instantiated transitions of M_s, viz.,* $\{\tilde{t} = \mathrm{map}(t), t \in T\}$, *where T are the input transitions of M_s.*

Proof: Obvious, since instantiation does not affect the machine's primary inputs, and symbolic states have been mapped $1 \rightarrow 1$ onto encoded states. □

Lemma 3.16 (Mapping output functions) *For binary output o_i and minterm p_s in M_s' domain,* $\tilde{o}_i(\mathrm{map}(p_s)) = o_i(p_s)$.

Lemma 3.17 (Mapping output transitions) *M_ϵ's binary output transitions are precisely the instantiated binary output transitions of M_s. Further, their type (i.e. static-0/1, dynamic) is unaffected by the mapping.*

Proof: This follows directly from Lemmas 3.15 and 3.16: the machine's binary output values at the transition's endpoints are not affected by instantiation. □

Required Cubes We prepare to define the required cubes for M_ϵ's outputs by showing that the maximal ON-set subcubes of a horizontal dynamic transition map in the obvious way. We then determine the required cubes for the binary outputs and next-state bits of M_ϵ.

Lemma 3.18 *The maximal ON-set subcubes of horizontal transition t for binary output o_i map one-to-one onto the maximal ON-set subcubes* $\mathrm{map}(c)$ *of transition $\tilde{t}$, where* $\tilde{t} = \mathrm{map}(t)$.

Proof: t clearly spans a single symbolic state. Hence, its image on M_ϵ, $\tilde{t}$, likewise spans a single binary state, and each minterm within t maps to a corresponding minterm in $\tilde{t}$

having identical binary output values. Thus, each maximal ON-set subcube of t maps onto a unique maximal ON-set subcube of $\tilde{t}$. $\Box$

Lemma 3.19 (Binary output required cubes) *If binary output o_i of M_s has the set of required cubes $\mathcal{R}_{o_i}$, then binary output $\tilde{o}_i$ of M_ϵ has the set of required cubes $\tilde{\mathcal{R}}_{\tilde{o}_i} = \{\tilde{r}_{\tilde{o}_i} = \mathrm{map}(r_{o_i}), \forall r_{o_i} \in \mathcal{R}_{o_i}\}$.*

Proof: Essentially, this follows from Lemmas 3.17 and 3.18 and the "early output" implementation (which assures that all vertical output transitions are static). Specifically, the required cubes of a logic function [106] are the union of:

i. the supercubes of all static-1 transitions and

ii. the maximal ON-set sub-cubes of all dynamic transitions.

We address each of the 2 sets in turn.

static By Lemma 3.17, static-1 transition t for o_i maps onto static-1 transition $\tilde{t}$ for $\tilde{o}_i$. Hence, the corresponding required cube $c = t$ maps onto required cube $\tilde{c} = \tilde{t}$ for $\tilde{o}_i$.

dynamic Transition t maps onto dynamic transition $\tilde{t}$ for $\tilde{o}_i$. $\tilde{t}$ must be horizontal (corresponding to an input change), since the early output restriction disallows dynamic vertical output transitions. We need to show that the maximal ON-set subcubes of $\tilde{t}$ are the instantiations of the maximal ON-set sub-cubes of t, which was proven by Lemma 3.18.

$\Box$

We now examine the required cubes for the next-state bits of M_ϵ.

Lemma 3.20 *Static-*1 *transition t_s for symbolic state ns_j maps onto static (*0 *or* 1*) transition $\tilde{t}_s$ for each state bit $\tilde{s}_i$ for which $\epsilon_j[i] = 1$.*

	00	01	11	10	
S0	**S0,0**	**S0,0**	**S3,0**	**S0,0**	**000**
S1	**S1,0**	**S1,1**	***S1,1***	**S0,0**	**010**
S2	**S0,0**	**S1,1**	**S3,0**	**S2,0**	**110**
S3	**S3,1**	**S3,1**	**S3,0**	**S2,0**	**111**

Figure 3.8: An unstable state transition and the corresponding state bit transitions

Unlike the case for binary outputs, a dynamic transition t_d for symbolic next-state ns_j (corresponding to an unstable state transition) produces a mixture of static and dynamic transitions for the various next-state bits. This is true in part because the transition's source (ns_j) and destination (say, ns_k) states may share 0- or 1-bits (e.g. in order to avoid critical races). The state bit values held in common result in static transitions. For example, see Figure 3.8, in which the transition from s_2 to s_1 in column `01` produces one dynamic ($1 \rightarrow 0$) transition and two static transitions ($1 \rightarrow 1$ and $0 \rightarrow 0$) for the 3 state bits under the given encoding. The following lemma makes this notion precise.

Lemma 3.21 *Dynamic transition t_d for symbolic next-state ns_j is also dynamic for another symbolic state ns_k, $k \neq j$. Furthermore, t_d maps onto:*

$$\begin{cases} 0 \textit{ or more dynamic transitions } \tilde{t}_d \textit{ for } \tilde{s}_i \textit{ one for each } i \mid \epsilon_j[i] \neq \epsilon_k[i] \\ 0 \textit{ or more static transitions } \tilde{t}_s \textit{ for } \tilde{s}_i \textit{ one for each } i \mid \epsilon_j[i] = \epsilon_k[i] \end{cases}$$

From the above 2 Lemmas, we can see that there are 3 sources of required cubes for a given state bit $\tilde{s}_i$ of M_ϵ:

- stable transitions for symbolic next-states ns_j which assign $\tilde{s}_i = 1$,
- unstable transitions $ns_j \rightarrow ns_k$ when ns_j and ns_k assign different values to $\tilde{s}_i$, and
- unstable transitions $ns_j \rightarrow ns_k$ when ns_j and ns_k assign $\tilde{s}_i$ to 1.

More precisely:

Lemma 3.22 (State bit required cubes) *State bit $\tilde{s}_i$ of M_ϵ has required cubes:*

$$\tilde{\mathcal{R}}_{\tilde{s}_i} = \text{map}(S_{ns_j}) \cup \text{map}(S_D) \cup \text{map}(\mathcal{ON}_{max}),$$

where:

$\text{map}(S_{ns_j})$	the set $\left\{\tilde{t}_s = \text{map}(t_s)\right\}$, where t_s is a static-1 transition on symbolic state ns_j for which $\epsilon_j[i] = 1$.
$\mathcal{S}_D$	the set $\{\tilde{t}_d = \text{map}(t_d)\}$, where t_d is an unstable transition from ns_j to ns_k, and $\epsilon_j[i] = \epsilon_k[i] = 1$.
$\text{map}(\mathcal{ON}_{\max})$	the set $\{\tilde{c} = \text{map}(c),\ c \in \mathcal{ON}_{\max}(t_d)\}$, i.e. the maximal ON-set sub-cubes of all unstable transitions t_d from ns_j to ns_k, where $\epsilon_j[i] \neq \epsilon_k[i]$.

Proof: The required cubes of a logic function are the union of its static-1 transition supercubes and the maximal ON-set sub-cubes for all of its dynamic transitions [106]. Lemmas 3.20 and 3.21 show that there are 2 sources of static-1 transitions and 1 source of dynamic transitions for $\tilde{s}_i$. The first two items above describe precisely the two sets of static-1 transitions for $\tilde{s}_i$. Our single transition-time (STT) operation allows only *horizontal* dynamic transitions for next-state; hence, Lemma **??** applies, yielding the third set of required cubes. □

The following lemma states a property that is pivotal to establishing that $\tilde{\mathcal{C}}$ covers M_ϵ.

Lemma 3.23 (Preservation of Cube Containment) *Cube containment is preserved by instantiation. That is, if cube c_1 contains c_2, then* $\text{map}(c_1)$ *contains* $\text{map}(c_2)$.

Proof: $c_1 = \langle \text{in}_1, \text{pres}_1 \rangle$, $c_2 = \langle \text{in}_2, \text{pres}_2 \rangle$. We know $\text{in}_1 \supseteq \text{in}_2$ and $\text{pres}_1 \supseteq \text{pres}_2$. Now, $\text{map}(c_1) = \langle \text{in}_1, \text{supercube}(\{\epsilon_i, \forall i \in \text{pres}_1\}) \rangle$; $\text{map}(c_2) = \langle \text{in}_2, \text{supercube}(\{\epsilon_i, \forall i \in \text{pres}_2\}) \rangle$. Hence, $\text{map}(c_1) \supseteq \text{map}(c_2)$ iff $\text{supercube}(\{\epsilon_i, \forall i \in \text{pres}_1\}) \supseteq \text{supercube}(\{\epsilon_i, \forall i \in \text{pres}_2\})$.

Now, supercube$(\epsilon_{i_1}, \ldots, \epsilon_{i_n})$ is the smallest cube containing all of $\{\epsilon_{i_1}, \ldots, \epsilon_{i_n}\}$. So, given $\text{pres}_1 \supseteq \text{pres}_2$, clearly supercube$(pres_1) \supseteq$ supercube$(pres_2)$. $\Box$

The following three theorems prove that the pseudo-canonical cover $\tilde{\mathcal{C}}$ covers M_ϵ without hazards. First, $\tilde{\mathcal{C}}$ is shown to cover all of M_ϵ's required cubes. Next, $\tilde{\mathcal{C}}$ is shown not to intersect the OFF-set of M_ϵ. Finally, $\tilde{\mathcal{C}}$ is shown to be dynamic hazard-free for all of M_ϵ's specified transitions.

Theorem 3.24 *$\tilde{C}$ covers all required cubes of M_ϵ.*

Proof: We must prove this for the binary outputs and state bits of M_ϵ.

Binary outputs $\tilde{o}_i$: This follows from Lemma 3.19 ($\tilde{\mathcal{R}}_{o_i} = \text{map}(\mathcal{R}_{o_i})$), from the fact that $\tilde{C} \supseteq \{\tilde{c} = \text{map}(c),\ c \in C\}$, from Lemma 3.23 (preservation of cube containment), and from the fact that hazard-free symbolic logic minimization guarantees that C covers all required cubes R_{o_i} of M_s.

State bits $\tilde{s}_i$: We prove that each of the 3 sets of required cubes described in Lemma 3.22 are covered.

map(S_{ns_j}) For each static-1 transition t_s on ns_j, where $\epsilon_j[i] = 1$, clearly C contains some term c which covered t_s and contributed to ns_j. Hence $\tilde{C}$ contains $\tilde{c}$, which contributes to $\tilde{s}_k$ for all k for which $\epsilon_j[k] = 1$; in particular, $\tilde{s}_i$.

$\mathcal{S}_D$ This is the set of required cubes induced by new static-1 transitions on shared-1 bits in ns_j and ns_k for t. However, as described in section 3.7.1, $\tilde{C}$ contains terms for each such transition, thus covering the shared-1 bits.

map$(\mathcal{ON}_{\mathbf{max}})$ Dynamic transitions occur only in horizontal transitions t_H. Hence, Lemma 3.18 applies, and the maximal ON-set sub-cubes of $\tilde{s}_i$ for all such $\tilde{t}_H$ are map$(\mathcal{ON}_{\max}(ns_j))$. Since C covers $\mathcal{ON}_{\max}(ns_j)$, $\tilde{C}$ covers map$(\mathcal{ON}_{\max}(ns_j))$, by Lemma 3.23.

□

Theorem 3.25 *No product $\tilde{c}$ in $\tilde{C}$ intersects the OFF-set of M_e.*

Proof: We proceed by addressing first the portion of $\tilde{C}$ corresponding to map(C), and then the added cubes $\mathcal{RS}_1$ of section 3.7.1. Recall that machine behaviour is don't-care *outside specified transitions*; hence, all OFF-set points lie within specified transitions.

Binary output $\tilde{o}_i$: Let $\tilde{m}$ be an OFF-set minterm of $\tilde{o}_i$ in transition $\tilde{t}$ of M_ϵ. By Lemma 3.17, $\tilde{t} = \text{map}(t)$ for some t, and $\exists m \in t$ in the OFF-set of o_i.

For any implicant $\tilde{p} = \text{map}(p)$, $p \in C$, symbolic minimization ensures that p either does not contain m or does not contribute to o_i. Specifically, one of the following conditions holds true:

- p does not contribute to o_i,
- $\text{input}(t) \cap \text{input}(p) = \phi$,
- $\text{present}(t) \cap \text{present}(p) = \phi$

Only the first 2 conditions are guaranteed to hold after instantiation by an arbitrary encoding. We must show that any encoding that satisfies our constraints ensure that $\text{present}(\tilde{t}) \cap \text{present}(\tilde{p}) = \phi$ when the first 2 conditions fail to hold. Observe that o_i is either 1 throughout t (impossible, since $m \in t$ is an OFF-set minterm), or o_i is 0 somewhere during t. In the latter case, an encoding constraint is generated to ensure that $\text{present}(\tilde{p}) \cap \text{present}(\tilde{t}) = \phi$.

For any $\tilde{p} \in \mathcal{RS}_1$, $\tilde{p}$ never contributes to any binary output, by construction.

State bit $\tilde{s}_i$: Let $\tilde{m}$ be an OFF-set minterm of output $\tilde{s}_i$ in transition $\tilde{t}$ of M_ϵ. This corresponds[7] to some $m \in t$ in the ON-set of some ns_j for which $\epsilon_j[i] = 0$.

[7] perhaps not uniquely

For any implicant $\tilde{p} = \mathrm{map}(p)$, $p \in C$, symbolic minimization ensures that p either does not contain m or does not implement $\tilde{s}_i$. Specifically, one of the following conditions holds true:

- $\mathrm{input}(t) \cap \mathrm{input}(p) = \phi$,
- p does not implement next-state
- p implements ns_j
- p implements some distinct state ns_k, $k \neq j$
- $\mathrm{present}(t) \cap \mathrm{present}(p) = \phi$

The first two cases clearly present no problem after instantiation. In the third case, $\tilde{p}$ implements only those state bits $\tilde{s}_i$ for which $\epsilon_j[i] = 1$; but $\epsilon_j[i] = 0$, so $\tilde{p}$ does not implement $\tilde{s}_i$, and there can be no problem. In the fourth case, either $\epsilon_k[i] = 0$ or $\epsilon_k[i] = 1$. The sub-case $\epsilon_k[i] = 0$ offers no difficulty. If $\epsilon_k[i] = 1$, $\tilde{p}$ implements $\tilde{s}_i$; however, since $ns_j \neq ns_k$, we generate an encoding constraint to ensure that $\mathrm{present}(\tilde{t}) \cap \mathrm{present}(\tilde{p}) = \phi$.

For any $\tilde{p} \in \mathcal{RS}_1$, $\tilde{p}$ contributes only to a single state bit, and spans only a given static-1 horizontal transition of M_ϵ, so it cannot hit the OFF-set minterms of binary outputs or state bits. □

Lemma 3.26 *If binary output o_i of M_s has privileged cubes $\mathcal{P}_{o_i}$ with corresponding start points σ_{o_i}, binary output $\tilde{o}_i$ has privileged cubes*

$$\tilde{\mathcal{P}}_{\tilde{o}_i} = \{\tilde{p} = \mathrm{map}(p),\, p \in \mathcal{P}_{o_i}\}$$

and corresponding start points $\tilde{\sigma}_{\tilde{o}_i} = \mathrm{map}(\sigma_{o_i})$.

Proof: This follows directly from Lemma 3.17. □

Lemma 3.27 *If the symbolic next-states ns_j have privileged cubes $\mathcal{P}_{ns_j}$, with corresponding start points σ_{ns_j}, state bit $\tilde{s}_i$ has privileged cubes*

$$\begin{aligned}\tilde{\mathcal{P}}_{\tilde{s}_i} &= \{\tilde{p} = \text{map}(p),\, p \in \mathcal{P}_{ns_j},\, \forall j \mid \epsilon_j[i] = 1\} - \\ &\quad \{\tilde{p} = \text{map}(t),\, t:\, ns_j \rightarrow ns_k,\, \epsilon_j[i] = \epsilon_k[i] = 1\},\end{aligned}$$

and corresponding start points $\tilde{\sigma}_{\tilde{s}_i} = \text{map}(\sigma_{ns_j})$. In other words, $\tilde{\mathcal{P}}_{\tilde{s}_i}$ consists of the mapped privileged cubes for all unstable transitions from $ns_j \rightarrow ns_k$ that assign $\tilde{s}_i$ a 1, excluding those transitions where both source and destination state assign $\tilde{s}_i$ to 1.

Proof: This follows from Lemma 3.21: dynamic transitions for $\tilde{s}_i$ arise from dynamic transitions on ns_j and ns_k when $\epsilon_j[i] \neq \epsilon_k[i]$. □

Lemma 3.28 *For all privileged cubes $\tilde{c}$ for M_ϵ, $|\text{present}(c)| = 1$.*

Proof: This follows from the early output restriction (which ensures static state/output transitions in all vertical transitions), along with the use of STT encodings. □

Theorem 3.29 *Cover $\tilde{C}$ is hazard-free for every specified input transition.*

Proof: A hazard-free implementation requires that:

1. All required cubes are covered by some implicant, and

2. No implicant illegally intersects a privileged cube for any output to which it contributes.

Theorem 3.24 established the first point; the second remains to be demonstrated. We again treat $\tilde{p} \in \text{map}(C)$ and added cubes $\tilde{p} \in \mathcal{RS}_1$ separately.

Binary output $\tilde{o}_i$: By Lemma 3.26, $\tilde{o}_i$ has privileged cubes $\{\tilde{c}_p = \text{map}(c_p),\ c_p \in \mathcal{P}_{o_i}\}$.

- For any implicant $\tilde{p} \in \text{map}(C)$, symbolic hazard-free logic minimization ensures that one of the following holds:

  1. $o_i \notin \text{output}(p)$
  2. $\text{input}(p) \cap \text{input}(c_p) = \phi$
  3. $\sigma_{c_p} \in p$
  4. $\text{present}(p) \cap \text{present}(c_p) = \phi$

  In cases 1, 2, and 4, there is no intersection; in the third, the intersection is legal: the implicant contains the start point. Since instantiation does not affect the input and output fields of implicants, and by Lemma 3.23, one of the following three analogous conditions also holds, for *any* encoding:

  1. $\tilde{o}_i \notin \text{output}(\tilde{p})$
  2. $\text{input}(\tilde{p}) \cap \text{input}(\tilde{c}_p) = \phi$
  3. $\sigma_{\tilde{c}_p} \in \tilde{p}$

  The condition analogous to the 4^{th}, however, viz. $\text{present}(\tilde{p}) \cap \text{present}(\tilde{c}_p) = \phi$, is not guaranteed to hold for any arbitrary encoding. From Lemma 3.28, $|\text{present}(c_p)| = 1$, and hence the encoding constraint $\{\text{present}(p);\ \text{present}(c_p)\}$ that we impose is in fact a face embedding constraint, which is thus respected by our encoding. Thus, $\text{present}(\tilde{p}) \cap \text{present}(\tilde{c}_p) = \phi$.

- For any $\tilde{p} \in \mathcal{RS}_1$:

  $\tilde{p}$ never implements binary outputs; hence no illegal intersection is possible.

State bit $\tilde{s}_i$: By Lemma 3.27, $\tilde{s}_i$ has privileged cubes $\{\tilde{c}_p = \text{map}(c_p), c_p \in \mathcal{P}_{ns_j}, \forall j \mid \epsilon_j[i] = 1\}$.

- For any implicant $\tilde{p} \in \text{map}(C)$, symbolic hazard-free logic minimization ensures that either:

  1. $ns_j \notin \text{output}(p)$
  2. $\text{input}(p) \cap \text{input}(c_p) = \phi$
  3. $\sigma_{c_p} \in p$
  4. $\text{present}(p) \cap \text{present}(c_p) = \phi$

  As for binary outputs, although the first three conditions are preserved by instantiation with any encoding, the last is not. The illegal intersection is then only avoided if $\text{present}(\tilde{p}) \cap \text{present}(\tilde{c}_p) = \phi$, which is guaranteed by an encoding constraint of the form: $\{\text{present}(p); \text{present}(c_p)\}$ which is in fact a face embedding constraint (by Lemma 3.28), and is thus respected by encodings satisfying our constraints.

- For any $\tilde{p} \in \mathcal{R}$, recall that $\tilde{p}$ covers some static-1 horizontal transition $\tilde{t}$, corresponding to some transition t for which $ns_k : 1 \rightarrow 0$, $ns_l : 0 \rightarrow 1$, and $\epsilon_k[i] = \epsilon_l[i] = 1$. By Lemma 3.28, $|\text{present}(c_p)| = 1$, and specifically, $\text{present}(p) = ns_j$. Further, an illegal intersection implies all of the following:

  - $\text{input}(p) \cap \text{input}(c_p) \neq \phi$
  - $\sigma_{c_p} \notin p$
  - $\tilde{s}_i \in \text{next}(\tilde{p})$

  Now, c_p corresponds to a dynamic horizontal transition t' on ns_j and $\epsilon_j[i] = 1$. However, note that the end point e of t lies strictly within t'. This would mean that the next-state at e is ambiguous: for t, it must be ns_l, while for t', it must be ns_j.

□

3.7.5 Optimality of Binary Cover

A final key result is that CHASM's algorithm produces state assignments and hazard-free realizations which are *exactly optimal* with respect to output logic (if outputs and next-state are minimized separately).

Property 3.30 [Opt-HFCRF Optimality of Output Cover] *The binary instantiated* output *cover $\mathcal{OC}$ (where outputs are minimized separately from next-state) has* exactly *minimum cardinality.*

This result is especially important for asynchronous state machines. Since asynchronous machines have no clocks or latches, the input-to-output latency is determined solely by output logic delay. CHASM's algorithm finds a USTT state assignment which results in a *hazard-free output cover with smallest cardinality over all possible critical race-free assignments.*

3.8 Experimental Results

A preliminary set of experiments was run on industrial examples using our optimal encoding and logic minimization algorithms. Results appear in Figure 3.8.

For each set of runs, the number of state variables (#b) and number of cubes (#c) in the final cover are reported. The column labelled *optimal* lists runs in which all constraints were solved. A parallel set of runs using a "random" (but minimal length) critical race-free encoding was done as well, labelled *base-crf*, for comparison with the optimal. Finally, a third set of runs, *opt-fixed*, was performed (for cases where *optimal* and *base-crf* differed in code length), using a fixed code length and partial constraint satisfaction. For this set, runs at or near the code length of the *base-crf* case were performed; the best of several iterations is reported. For all sets of runs, HFMIN was used for the binary hazard-free logic implementation step.

The *opt-fixed* algorithm achieves results at least as good as the *optimal* and *base-crf* algorithms. As in KISS [40] and NOVA [147], this phenomenon occurs because input encoding is itself an approximate formulation. Hence, by using partial constraint satisfaction with restricted code lengths, a large percentage of optimality constraints can be satisfied with less overhead in the next-state implementation. Improvements in cardinality of up to 17% are observed.

DESIGN	I/S/O	*opt-fixed*		*optimal*		*base-crf*	
		#b	#c	#b	#c	#b	#c
sbuf-read-ctl	3/3/3	2	7	3	9	2	8
sbuf-send-ctl	3/4/3	2	11	4	12	2	11
rf-control	6/6/5	3	13	6	15	3	15
it-control	5/5/7	3	15	6	15	3	15
pe-send-ifc	5/5/3	3	18	7	27	3	21
sd-control	8/13/12	5	29	10	34	4	35
dram-ctrl	7/3/6	-	-	2	22	2	22
pscsi-ircv	4/4/3	2	9	4	12	2	10
pscsi-isend	4/6/3	3	17	7	23	3	19
pscsi-trcv	4/4/3	3	9	4	13	2	11
pscsi-trcv-bm	4/4/4	2	12	4	15	2	14
pscsi-tsend	4/7/3	3	18	7	22	3	18
sscsi-isend-bm	5/4/4	2	21	5	22	2	24
sscsi-isend-csm	5/3/4	-	-	2	12	2	12
sscsi-trcv-bm	5/4/4	2	18	5	24	2	18
sscsi-trcv-csm	5/3/4	2	12	3	12	2	12
sscsi-tsend-bm	5/5/4	3	17	6	20	3	18
sscsi-tsend-csm	5/4/4	2	14	5	15	2	14
stetson-p1	13/12/14	4	53	19	–*	4	55
stetson-p2	8/13/12	4	31	10	37	4	36

*Exact logic minimization failed due to insufficient virtual memory in prime generation.

Figure 3.9: Experimental Results

Chapter 4

OPTIMIST: Optimal State Minimization for Synchronous State Machines

This chapter presents OPTIMIST, a novel method for state minimization of incompletely-specified finite state machines. Where classic methods simply minimize the number of states, this method directly addresses the implementation's logic complexity. The method is the first to produce an exactly optimum two-level implementation under an input encoding model.

OPTIMIST extends the step-merging synthesis philosophy used by CHASM in Chapter 3 by incorporating yet another step: state minimization. The result is a first-of-a-kind concurrent state minimization, state encoding, and two-level logic minimization algorithm. In a sense, OPTIMIST can be thought of as playing a role for state minimization analogous to that of KISS for state encoding. Although OPTIMIST as described here applies only to synchronous FSM's, its framework serves as a springboard for a novel algorithm applicable to asynchronous machines, which is presented in Chapter 6.

OPTIMIST currently makes use of an input encoding model within its minimization procedure. Nevertheless, it is capable of significantly reducing logic complexity for some machines. It also appears capable of being extended to encompass more powerful encoding models, such as output encoding.

The OPTIMIST method incorporates optimal *state mapping*, i.e., the process of reducing the symbolic next-state relation which results from state splitting to an optimal conforming symbolic function. Further, it offers a number of convenient sites for applying heuristics to reduce time and space complexity, and is amenable to implementation based on implicit representations.

4.1 Introduction

State minimization is the problem of finding a machine realizing the input/output behaviour of a given FSM, with fewer internal states [60, 109, 62]. This is an important step in sequential synthesis: implementing unminimized FSM's often leads to considerably larger and/or slower implementations. However, it is well known that the classic formulation for state minimization expresses a *heuristic* — reducing the number of states only *tends* to decrease logic complexity. Early on, Hartmanis observed [52] that this heuristic sometimes fails; realizations having more states may be simpler to implement. Moreover, there may be many minimum-state realizations of a given FSM, and their logic complexity can vary significantly [111, 91]. Hence, simply targeting any minimum-state solution is insufficient.

The major contribution of this chapter is a state minimization method which, in contrast to existing ones, directly targets logic complexity. In particular, it defines and solves the *optimal state minimization problem*, that of finding for a given FSM a realization having minimum 2-level logic complexity over all realizations.

Classic sequential synthesis comprises several steps: state minimization, state encoding, 2-level logic minimization, multi-level optimization and technology mapping. Each step has traditionally been treated as an isolated problem, which limits early steps most severely. In [40], a key insight into optimal state encoding was presented: *symbolic* logic minimization can be performed *concurrently* with state encoding. More recent methods for optimal encoding have been developed [147, 45] based on the same insight, but yielding even better results.

OPTIMIST borrows this insight, as did CHASM, and takes it one step further, by performing symbolic logic minimization concurrently with *both* state minimization *and* state encoding. The method is cast as a unique form of generalized prime implicant minimization [45]. Specifically, symbolic prime implicants are generated, and a binate covering problem is formed and solved, yielding a reduced machine and logic cover.

This chapter offers a novel theoretical framework for formulating and solving the optimal state minimization problem. It demonstrates that the method identifies optimal solutions which are inaccessible to existing tools, due to their focus on minimum cardinality state covers. In addition, it provides initial results of a CAD tool implementation.

The structure of the chapter is as follows. Section 4.2 provides background on state minimization, state assignment and related work. Section 4.3 then gives a general overview of our method, with a focus on the major issues. Sections 4.4, 4.5 and 4.6, provide greater detail on the three major components of our method: symbolic prime generation, binate constraint generation and solution, and symbolic instantiation, respectively. Two examples in Section 4.7 demonstrate the procedure and show results unattainable by existing methods. The key theoretical results of this chapter are given in Section 4.8. Efficient algorithms for symbolic prime generation are described in Section 4.9. Finally, Section 4.10 provides some experimental results, and Section 4.11 presents conclusions and future work.

4.2 Background and Related Work

The following sections review some basic background material related to the problem of optimal state minimization. Specifically, the classic state minimization problem and some of its solutions are first presented. Next, the related problem of state mapping is described, an issue that is key to obtaining optimal next-state logic. Finally, the body of previous work on state minimization is reviewed.

We continue with the notation presented earlier in Chapter 2. Again, we restrict attention to the common subclass of ISFSM's where, in all total states $\langle i, s \rangle$, the next-state $\delta(i, s)$ is either a singleton state or else is completely unspecified (denoted $\delta(i, s) = \mathcal{S}$). Likewise, it is assumed that output $\mathcal{O}(i, s)$ is either a single value or unspecified.

4.2.1 State Minimization

We now review basic definitions for state minimization [60]. The first several of these are sufficient to formulate the problem under consideration. The remainder is used in describing the classic exact solution proposed by Grasselli [60].

Central to state minimization is the notion of *state compatibility*, on which basis states are merged in order to form a reduced machine. State compatibility is based on the compatibility of the output behaviour that results from starting in a given state and applying a given input sequence. Clearly, the output behaviour depends on the state behaviour (i.e. the sequence of states visited under the given input sequence). To capture this effect, the transitive closure of the next-state relation is mirrored by an inductive definition of state compatibility.

Definition 4.1 *A pair of states is* output-compatible *iff corresponding output values agree wherever both are specified.*

In the flow table of Figure 4.1, state pairs (s_0, s_1) and (s_2, s_3) are each output-compatible, while the pair (s_0, s_2) is not, because of the different output values in input column 00.

It is worthwhile noting that the output compatibility relation is both reflexive and symmetric, but not transitive.

Definition 4.2 *A pair of states* (s_a, s_b) implies *the state pair* (s_c, s_d) under input $\mathcal{I}_k$ *iff*

	00	01	11	10
	inputs xy			
s_0	$s_0, 1$	$s_2, 1$	$s_0, -$	$s_3, 0$
s_1	$s_1, -$	$s_2, 1$	$s_1, 0$	$s_2, 0$
s_2	$s_2, 0$	$s_3, 0$	$s_1, 1$	$s_3, 0$
s_3	$s_3, -$	$s_2, 0$	$s_0, -$	$s_3, 0$

Figure 4.1: Flow table illustrating state compatibility

$\{\delta(\mathcal{I}_k, s_a),\ \delta(\mathcal{I}_k, s_b)\} = \{s_c, s_d\}$.

For example, in Figure 4.1, states (s_0, s_1) imply (s_2, s_3) under input 10, since the destination states for s_0 and s_1 in that column are s_3 and s_2, respectively.

Implication is significant because a merged state such as (s_0, s_1) must specify a single valid destination state in each input column. At the same time, the merged state's output behaviour must conform to that of both the original states. This in turn requires merging the respective destination states. For example, if (s_0, s_1) were to be merged, the states (s_2, s_3) would have to be merged as well, due to the implication in column 10.

Clearly, there are cases where an implied set of states is not itself output-compatible; hence the following definition.

Definition 4.3 *A pair of states is* compatible *iff they are output-compatible and imply no incompatible pair of states.*

We now extend the above definitions to state sets having more than two members.

Definition 4.4 *A set of states is a* compatible *iff it consists only of pairwise-compatible states.*

The set of compatibles for the table of Figure 4.1 is $c_0 = \{s_0\}$, $c_1 = \{s_1\}$, $c_2 = \{s_2\}$, $c_3 = \{s_3\}$, $c_4 = \{s_0, s_1\}$, and $c_5 = \{s_2, s_3\}$. However, the state set $\{s_1, s_2, s_3\}$ is not a compatible because s_1 and s_2 are incompatible.

Definition 4.5 *Compatible* c_a implies *compatible* c_b under input I_k *iff* $c_b = \{\delta(\mathcal{I}_k, s), \forall s \in c_a\}$.

Definition 4.6 *The* implied set $P(c)$ *of a compatible* c *is the set of compatibles implied by* c *over all inputs, excluding singleton states, subsets of* c*, and proper subsets of compatibles in* $P(c)$*.*

The above definition effectively ignores those implications that would be trivially satisfied (i.e. that would never require the merging of another set of states). In particular, reflexive implications such as $(s_0, s_1) \rightarrow (s_0, s_1)$ (which occurs in column 00) are omitted from the implied set. Likewise, degenerate implications such as $(s_0, s_1) \rightarrow (s_2, s_2)$ are omitted.

Example 1 The implied sets for the table of Figure 4.1 are:

$P(c_0) = \phi$	$P(c_1) = \phi$
$P(c_2) = \phi$	$P(c_3) = \phi$
$P(c_4) = \{c_5\}$	$P(c_5) = \{c_4\}$

The following two definitions capture the two essential characteristics of any valid selection of state compatibles: closure and covering.

Definition 4.7 *(Closure) A set* C *of compatibles is* closed *iff every element of the implied set of each compatible* $c \in C$ *is contained by some compatible in* C*.*

Definition 4.8 *(Covering) A set* C *of compatibles is a* cover *for* $\mathcal{M}$ *iff for every state* $s \in \mathcal{S}$ *there exists a compatible* $c \in C$ *such that* $s \in c$*.*

The concept of state covering allows us to define precisely whether the selected set of state compatibles represents a complete realization of $\mathcal{M}$.

Note that the above definition does *not* require that each implied compatible is itself selected; rather, it requires selecting *some* compatible which *contains* the implied compatible. Containment is sufficient because a set of states always has output behaviour compatible with any subset of itself.

Using the above conditions, it is now possible to define the state minimization problem precisely.

Problem 4.9 *State Minimization Find a minimum cardinality closed cover of compatibles of $\mathcal{M}$.*

Exact State Minimization

Given a machine $\mathcal{M}$ and the set $\mathcal{C}$ of all of its compatibles, we can now form a simple covering problem to express all of the constraints (covering and closure). There are two types of constraints:

Covering Each state must be covered by some selected compatible.

$\forall$ states s of $\mathcal{M}$

$c_{i_1} + \cdots + c_{i_n}$, where $s \in c_{i_k}$.

Here, variable c_{i_k} represents the selection of compatible c_{i_k}. This constraint ensures that, for each state s in the unminimized machine $\mathcal{M}$, some compatible c_{i_k} is selected which contains s. Note that each such constraint is unate, since it consists of a sum of positive literals.

Closure Every member of every selected compatible's implied set must be covered.

$\forall$ compatibles c of $\mathcal{M}$

$\forall\, c' \in P(c)$

$\overline{c} + c_{i_1} + \cdots + c_{i_n}$, where $c_{i_k} \supseteq c'$

This constraint ensures that, for each selected compatible c, each member of c's implied set is contained by some selected compatible c_{i_k}. Note that each such constraint is binate, since it consists of a sum of both positive and negative literals. The closure constraints as shown make use of the following well-known equivalence in Boolean logic: $p \rightarrow q \Leftrightarrow \overline{p} + q$.

$\mathcal{M}'$	00	01	11	10
$s'_0 = \{s_0, s_1\}$	$s'_0, 1$	$s'_1, 1$	$s'_0, 0$	$s'_1, 0$
$s'_1 = \{s_2, s_3\}$	$s'_1, 0$	$s'_1, 0$	$s'_0, 1$	$s'_1, 0$

Figure 4.2: Table of Figure 4.1, after reduction by $\{\{s_0, s_1\}, \{s_2, s_3\}\}$

Unlike the unate covering problems encountered in Chapter 3, this problem is *binate*[60, 117]. That is, it corresponds to a product of Boolean clauses containing both complemented and uncomplemented literals.

As an example, given the table of Figure 4.1 and the set of compatibles $c_0 = \{s_0\}$, $c_1 = \{s_1\}$, $c_2 = \{s_2\}$, $c_3 = \{s_0, s_1\}$, $c_4 = \{s_1, s_2\}$, the following constraint clauses result:

Covering:

$c_0 + c_4$	Cover s_0
$c_1 + c_4$	Cover s_1
$c_2 + c_5$	Cover s_2
$c_3 + c_5$	Cover s_3

Closure:

$\overline{c_4} + c_5$	c_5 in implied set of c_4
$\overline{c_5} + c_4$	c_4 in implied set of c_5

The minimum-cardinality solution given these constraints is the assignment $\{c_0 = 0,\ c_1 = 0,\ c_2 = 0,\ c_3 = 0,\ c_4 = 1,\ c_5 = 1\}$, corresponding to the compatibles $\{c_4, c_5\}$. It is easy to verify that this set covers all states and is closed. The resulting reduced machine has 2 states corresponding to these 2 selected compatibles, and is shown in Figure 4.2.

More Efficient Solutions

One can fashion a state cover from the set of all possible compatibles, as was just shown. However, for the problem at hand, it is possible (and desirable, especially for large machines) to restrict attention to a smaller set of compatibles.[1] We describe two such classes: *maximal compatibles* and *prime compatibles.*

Definition 4.10 *Compatible c is a* maximal compatible *iff it is a proper subset of no other compatible.*

It is always possible to find a closed cover of maximal compatibles for any machine $\mathcal{M}$ (see Unger [139]).

If a minimum cardinality cover were not the goal, maximal compatibles would suffice. Unfortunately, larger compatibles may possess more implications, and thus complicate the closure requirements. For this reason, so-called *prime compatibles* [60] are defined, which take into account not only the size of the compatible, but the closure requirements as well (in the form of their implied sets).

Definition 4.11 *Compatible c_a* excludes *c_b iff $c_b \subset c_a$ and $P(c_a) \subseteq P(c_b)$.*

Informally, exclusion expresses the notion that one compatible is larger than another, but has no more restrictive closure requirements.

Definition 4.12 *Compatible c is a* prime compatible *iff it is excluded by no other compatible.*

Since maximal compatibles are by definition prime and a closed cover of maximal compatibles can always be found, a solution consisting solely of prime compatibles also always exists.

Problem 4.13 *Classic State Minimization Find a minimum cardinality closed cover of prime compatibles of $\mathcal{M}$.*

In the sequel, a reduced machine corresponding to $\mathcal{M}$ is designated $\mathcal{M}'$.

[1] It will be shown later, however, that this commonly-used smaller set of compatibles will not suffice for *optimal* state minimization.

$\mathcal{M}$	0	1
s_0	$s_0, 0$	$s_2, 0$
s_1	$s_1, 0$	$s_1, -$
s_2	$s_1, -$	$s_0, 1$

$\mathcal{M}'$	0	1
$s'_0 = \{s_0, s_1\}$	$s'_0, 0$	$s'_1, 0$
$s'_1 = \{s_1, s_2\}$	$\{s_1\}, 0$	$s'_0, 1$

Figure 4.3: State table before and after minimization

4.2.2 The State Mapping Problem

This section describes an issue pivotal to optimal state minimization, because it is partially responsible for defining the next-state function after state minimization.

Given an incompletely-specified FSM, a state reduction often defines a *set* of compatible realizations [109, 60]. Specifically, a given unreduced state s may be *split*, i.e., two or more states covering s may be selected. In that case, a reduced next-state entry referring to s may be bound to any of the covering states of $\mathcal{M}'$. When there are two or more such covering states, a choice for the next-state binding exists (even when the next-state was uniquely specified in the original specification). Thus, the next-state behaviour of the resulting ISFSM forms a *relation*. This flexibility gives rise to the *state-mapping problem* [82], in which the symbolic relation must be reduced to a conforming symbolic function. The function is obtained by choosing a specific next-state wherever a choice exists. This choice clearly has a direct impact on logic quality, and thus constitutes an important issue for optimal state minimization.

The state mapping problem is illustrated in Figure 4.3, taken from [82]. A choice of next-state exists in the reduced machine $\mathcal{M}'$ in total state $\langle \mathtt{0}\ s'_1 \rangle$. Specifically, the next state can be assigned to either of the reduced states s'_0 or s'_1, since each of these two states covers the unreduced state s_1. The best cover achievable[2] when state mapping s_1 to s'_1 in that total state has 4 terms. In contrast, state mapping to s'_0 yields a 3-term cover. Thus, the state mapping choice has a direct impact on the optimality of the implementation. The impact of state mapping on logic has been investigated, and

[2]assuming an input-encoded symbolic implementation

several approaches to selecting a good mapping have been suggested [82][62].

4.2.3 Previous Work

The topic of state minimization has been researched extensively over several decades. Hartmanis et al. observed in [52] that the minimum cardinality solution does not always yield the best implementation. The classic problem was later formulated as a search for a closed cover *of minimum cardinality* and solved exactly in [60]. In [64], the relationship of state reduction to implementation complexity was explored through an elegant theoretical framework, which unfortunately did not provide a solution to the problem.

Several recent methods have been focused on producing minimum cardinality covers. Efficient algorithms have been produced for solving the problem exactly ([62], [70]), and heuristics ([62], [2], [3]) have been developed for inexact solutions. These two fronts have seen considerable progress.

Only a few recent attempts have been made to address the more general problem of *optimal state minimization.* In STAMINA [62], some attention is paid to implementation complexity, but no attempt at direct or exact solutions was made. A more direct approach was taken by Avedillo et al. [4], but results were less than encouraging (they were not even compared with a state reduction tool, but rather with NOVA, a state assignment tool, which did better on already-minimized FSM's), and no theoretical results are given. Finally, Calazans [18] offers a framework within which both optimal encoding and state minimization can be expressed. In it, state reduction is modeled as the assignment of two or more states to the same code. This insight constitutes one of the cornerstones of our method. Unfortunately, the only solution method given is a simple, greedy method which fares relatively poorly. Worse, it is not evident how to express a high-quality solution method within that formulation.

Two recently developed CAD optimization techniques are also relevant to this work, though created for other purposes. First, Devadas and Newton [45] address the problem of exact state encoding through *GPI minimization*. In this method, *generalized* symbolic prime implicants (*GPI's*) are formed, and a constrained binate covering problem is then solved which both selects a set of GPI's and produces a compatible state encoding. The basic flow — symbolic prime generation followed by constrained covering — is used in OPTIMIST, but with considerably different symbolic primes and constraints.

Second, Lin and Somenzi [82] introduce a technique for the exact minimization of symbolic relations. Of particular interest is its application to exact state encoding, incorporating an elegant method for state mapping. The symbolic relation they minimize, however, is the *result* of state minimization; state minimization itself is not addressed in their work.

4.3 Optimal State Minimization: Overview

This section provides a general overview of OPTIMIST's method for optimal state minimization. A more detailed presentation appears in subsequent sections.

Optimal state minimization is defined as finding, for an unminimized machine $\mathcal{M}$, a reduced machine $\mathcal{M}'$ with compatible behaviour and minimum logic complexity. Our method is cast as a form of symbolic GPI (generalized prime implicant) minimization which encapsulates both state minimization *and* state encoding. The method has 5 steps:

1. Generate state compatibles
2. Generate symbolic primes
3. Generate binate constraints
4. Solve constraints
5. Instantiate symbolic cover

First, state compatibles are formed by any standard method (e.g., STAMINA [62]). Next, a novel form of symbolic prime called restricted GPI's (*RGPI's*), based on GPI's [45], is generated. A set of binate constraints which identify the valid realizations is then formed. Finally, the constraints are solved so as to minimize logic cardinality using a binate solver (e.g., Scherzo [31]).

The solution is a set of selected compatibles and a set of selected RGPI's. These are trivially combined during cover instantiation, to produce the reduced machine $\mathcal{M}'$ and its symbolic two-level implementation. From there, input encoding constraints can be immediately generated and solved to produce an optimal encoding.

The following sections first describe the relationship between the encoding model to the rest of the process, and then give an overview of each of the 3 steps unique to our method: symbolic prime generation, binate constraint generation, and cover instantiation. Further details are provided in sections 4.4 through 4.6.

4.3.1 Optimal State Minimization and Input Encoding

The input encoding model provides a framework for symbolic logic minimization that is applied in OPTIMIST in a novel way to an ISFSM. In this way, OPTIMIST captures state reduction, state assignment, and two-level logic minimization.

The abstract flow of the approach is as follows. OPTIMIST starts with an ISFSM, and transforms it exactly as does CHASM, turning the next-state into a set of distinct output functions. In other words, it produces an mvi description consistent with the input encoding model. It then performs *symbolic logic minimization*, using a new form of symbolic prime implicants (described in the following section) that is capable of modelling the collapsing of compatible states.

The symbolic logic minimization problem is effectively cast as a search through the space of all valid reduced machines for a machine having an exactly minimum-cardinality symbolic logic cover. The symbolic logic cover consists of mvi products which, after state reduction, each contribute to at most one next-state. As a result, the cover so derived will be suitable for use as the basis of an input encoding process. From there, one can derive an encoding, and instantiate the symbolic cover, resulting in a binary cover of exactly minimum cardinality over all possible state reductions and input encodings.

4.3.2 Symbolic Primes

This section describes the unique form of symbolic prime implicants that lie at the heart of the OPTIMIST method. The concepts are defined and illustrated with several examples.

A *symbolic product* is a product of literals over an mvi domain [39]. Each value taken by a symbolic output is called a *symbolic part*. For example, each state s_i of a machine $\mathcal{M}$ is a symbolic part of the symbolic output known as the next-state.

A *symbolic implicant* is a symbolic product which satisfies the following two properties: (i) it contains no OFF-set minterm of any *binary output* to which it contributes; (ii) for each *symbolic output* to which it contributes, it asserts all symbolic parts specified in the total states which it contains. A symbolic implicant of an FSM is expressed as a 4-tuple $p : \langle \texttt{in ps ns out} \rangle$, denoting the input, present-state, next-state and output fields. For example, the symbolic product $p_a : \langle \texttt{0} \;\; s_0, s_1 \;\; s_0, s_1 \;\; \texttt{0} \rangle$ in Figure 4.3 is a symbolic implicant. Observe that p_a satisfies property (ii) by contributing to the symbolic parts of the next-state (namely, s_0 and s_1) that are specified in the total states (namely, $\langle 0 \; s_0 \rangle$ and $\langle 0 \; s_1 \rangle$) that p_a spans.

Our procedure forms a set of symbolic implicants on the unreduced machine, which can be used to cover portions of various reduced machines, after a suitable transformation. These implicants are maximal over $\mathcal{M}$ in an intuitive sense. Specifically, we define a novel type of symbolic prime implicant, called a *restricted generalized (prime) implicant*, or *RG(P)I*, a variant of a GPI, formed on the unreduced machine.

GPI's were introduced in [45] as a kind of symbolic prime implicant, "tagged with" (i.e., having a field comprising) a *set* of next-states. The tag contains *all* next-states which are specified in all total states the GPI contains. For example, in Figure 4.3, the symbolic implicant $g_a : \langle -\ \ s_0\ \ s_0, s_2\ \ \mathtt{0}\,\rangle$ is a GPI, where $\mathtt{ns}=\{s_0, s_2\}$.

We now formally define RGI's, and also an easily-calculated subset of RGPI's called *RGPI seeds*.

Definition 4.14 *An **RGI** is a symbolic implicant whose next-state field consists of compatible states.*

Intuitively, the selection of an RGI corresponds to making a state-mapping choice. Note that the next-state field of an RGI (if non-empty) is a set of states, i.e. a *compatible*, in the unreduced machine. This compatible, if selected, will appear as a *single state* in the reduced machine. Thus, the RGI, if selected, will be used to bind a cube-shaped region in the reduced machine $\mathcal{M}'$ with a single uniform next-state choice — the row in $\mathcal{M}'$ corresponding to the compatible. Moreover, the RGI's next-state field will then refer to (at most[3]) one symbolic next-state in the reduced machine $\mathcal{M}'$, as required by the input encoding formulation.

Definition 4.15 *An RGI p_1 **contains** RGI p_2 iff each field of p_1 contains the corresponding field of p_2.*

Definition 4.16 *An **RGPI seed** is an RGI which is contained* only *by RGI's with unequal next-state fields.*

Example 4.17 The flow table of Figure 4.3 has the following RGPI seeds:

[3] some RGPI's contribute only to the FSM's binary outputs

$p_0 : \langle 0\ s_0\ s_0\ 0\rangle$ $\quad$ $p_1 : \langle 0\ s_0, s_1, s_2\ s_0, s_1\ 0\rangle$

$p_2 : \langle 0\ s_1, s_2\ s_1\ 0\rangle$ $\quad$ $p_3 : \langle 0\ s_2\ s_1\ 1\rangle$

$p_4 : \langle 1\ s_0\ s_2\ 0\rangle$ $\quad$ $p_5 : \langle 1\ s_1\ s_1\ 1\rangle$

$p_6 : \langle -\ s_1\ s_1\ 0\rangle$ $\quad$ $p_7 : \langle 1\ s_0, s_1\ s_1, s_2\ 0\rangle$

$p_8 : \langle 1\ s_1, s_2\ s_0, s_1\ 1\rangle$ $\quad$ $p_9 : \langle 0\ s_1, s_2\ s_1, s_2\ 0\rangle$

$p_{10} : \langle -\ s_1, s_2\ s_0, s_1\ 0\rangle$ $\quad$ $p_{11} : \langle -\ s_2\ s_0, s_1\ 1\rangle$

$p_{12} : \langle 1\ s_2\ s_0\ 1\rangle$ $\quad$ $p_{13} : \langle 0\ s_2\ s_1, s_2\ 1\rangle$

$p_{14} : \langle -\ s_1\ s_1, s_2\ 0\rangle$ $\quad$ $p_{15} : \langle 1\ s_1\ s_1, s_2\ 1\rangle$

Note that $\{s_0, s_1\}$ is a compatible, but $\{s_0, s_2\}$ is not. Therefore, p_1 is an RGI, but GPI $g_a : \langle -\ s_0\ s_0, s_2\ \mathtt{0}\rangle$ is not. Furthermore, $p_b : \langle \mathtt{0}\ s_0, s_1\ s_0, s_1\ \mathtt{0}\rangle$ is an RGI, but is contained by p_1: each field of p_b is contained by the corresponding field of p_1. Hence, p_b is not an RGPI, since p_1 contains it and has the same next-state field. On the other hand, although p_1 contains p_0, its next-state field is different, and so p_0 is a distinct RGPI.

RGPI seeds can be generated by a slightly modified version of a GPI generation algorithm [45], which will be described in section 4.4. A more sophisticated and efficient algorithm is also presented in that section as well.

While RGPI seeds are the basic covering objects which will be used, it will be shown in section 4.4 that a more general class, RGPI's, is in fact needed.[4]

Definition 4.18 *An **RGPI** is an RGI which is contained* only *by RGI's with unequal input or next-state fields (or both).*

Example 4.23 shows several RGPI's that are not RGPI seeds.
The class of RGPI's includes RGPI seeds, as well as smaller RGPI's which result from reducing seeds in the input dimension, to allow finer-grained control over state mapping. The various relationships are depicted in Figure 4.4.

[4] Note that all RGPI seeds are GPI's, while RGPI's in general are not.

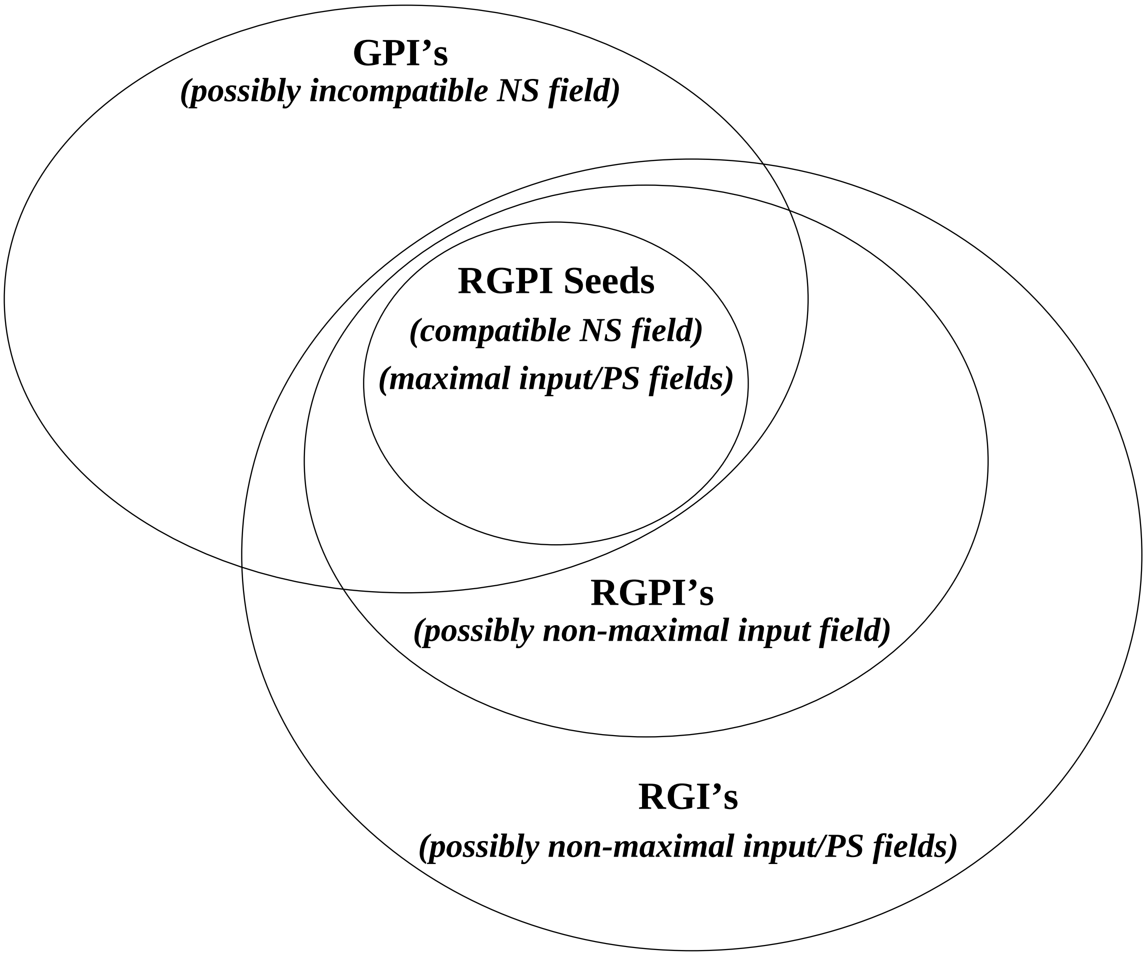


Figure 4.4: The relationships among the various classes of RGI’s and GPI’s

4.3.3 Constraint Generation

Once RGPI's are generated, constraints are formulated to insure a valid and optimum implementation. The solution is a set of selected compatibles and RGPI's. There are 3 main objectives. First, reduced machine $\mathcal{M}'$ must be a *realization* of $\mathcal{M}$. Second, the selected set of RGPI's must constitute a *symbolic cover* for $\mathcal{M}'$. Third, the resulting cover must have *minimum cardinality*, under the input encoding model. We now outline the constraints; details are provided in section 4.5.

The first objective is ensured by two sets of constraints. Each corresponds to a classic state minimization constraint: 1) the selected state compatibles form a cover, and 2) the resulting cover is closed. These state covering constraints are precisely as described in [60].

The second objective, forming a symbolic logic cover for $\mathcal{M}'$, is met by a novel set of constraints. Each selected compatible identifies a unique state s' in the reduced machine. These constraints ensure that each symbolic ON-set minterm of each reduced state s' in the reduced machine $\mathcal{M}'$ is covered by some selected RGPI.

The third objective, minimum logic cardinality, is assured by the binate solver, which finds a minimum-cost solution. To make cost a straightforward calculation, we introduce one extra variable per RGPI. Only these variables have non-zero cost, so that the cost function is simply the *cardinality* of the solution; that is, the number of selected RGPI's.

4.3.4 Constraint Solution

The binate constraints are solved by any of the various standard binate solvers, such as Scherzo [30]. The solution constitutes a selection of compatibles and RGPI's which define both a reduced machine $\mathcal{M}'$ and a symbolic logic cover for $\mathcal{M}'$.

4.3.5 Symbolic Instantiation

Symbolic instantiation is the process by which selected RGPI's are transformed one-for-one into a *symbolic realization* (a 2-level symbolic cover) of $\mathcal{M}'$. This symbolic realization is then used as the starting point for the encoding process, according to the classic KISS method.

To describe the RGPI instantiation process, we consider the transformation of a single RGPI. We consider separately the mapping of each of the RGPI's 4 fields.

Formally, for RGPI $p = \langle IN,\ PS,\ NS,\ OUT \rangle$ we define

$$p' = \text{Instantiate}(p) \ = \ \langle IN,\ PS',\ NS',\ OUT \rangle$$

The input and output fields of p are unchanged by instantiation. The next-state field of p is a *compatible* of $\mathcal{M}$. Note that this compatible corresponds to a *single row* in the reduced machine. Hence, an RGPI's next-state field identifies a unique state of $\mathcal{M}'$, and is mapped trivially:

$$NS' = \text{the unique state of } \mathcal{M}' \text{ corresponding to } NS$$

This step simply identifies the *compatible* given by the next-state field in p with the corresponding reduced row in $\mathcal{M}'$. The role of the RGPI in the reduced machine is thus to contribute to (at most) a *single* symbolic next-state. This role reflects our use of an input encoding formulation, where each next-state in the reduced machine is treated as a distinct function. Thus, a symbolic implicant which contributes to next-state can contribute to *only* one next-state. Therefore, each RGPI embodies a *uniform state mapping* over some cube of $\mathcal{M}'$ to a *specific* reduced state of $\mathcal{M}'$.

Finally, the present-state field of an RGPI contains one or more symbolic states. To a first approximation, an RGPI will be mapped so as to cover *all selected* compatibles

c which are contained by the present state field of that RGPI. That is, the RGPI is regarded as covering the class of compatibles which are contained within its present state field.

Example 4.19 RGPI $p_8 : \langle \texttt{1} \;\; s_1, s_2 \;\; s_0, s_1 \;\; \texttt{1} \rangle$ in Figure 4.3 contributes to (compatible) next-states $\{s_0, s_1\}$ of $\mathcal{M}$, and has present states $\{s_1, s_2\}$. Therefore, in the reduced table $\mathcal{M}'$, p_8 maps to the product $p'_8 : \langle \texttt{1} \;\; s'_1 \;\; s'_0 \;\; \texttt{1} \rangle$. The resulting present state field contains $s'_1 = \{s_1, s_2\}$; the next-state field consists of the single reduced next-state s'_0 (which corresponds to the original compatible set $\{s_0, s_1\}$).

As indicated, for the present state, we can include in PS' all selected compatibles contained by PS. This scheme works in some cases, but fails to capture the full flexibility of state mapping in others. In section 4.6, we define the precise mapping for PS which circumvents this problem.

Example 4.20 One complete solution to the constrained covering problem for Figure 4.3 consists of (i) compatibles $s'_0 \equiv \{s_0, s_1\}$ and $s'_1 \equiv \{s_1, s_2\}$, and (ii) RGPI's $\{p_1, p_7, p_8\}$ shown below.

$p_1 : \langle \texttt{0} \;\; s_0, s_1, s_2 \;\; s_0, s_1 \;\; \texttt{0} \rangle$ $\qquad$ $p'_1 : \langle \texttt{0} \;\; s'_0, s'_1 \;\; s'_0 \;\; \texttt{0} \rangle$

$p_7 : \langle \texttt{1} \;\; s_0, s_1 \;\; s_1, s_2 \;\; \texttt{0} \rangle$ $\qquad$ $p'_7 : \langle \texttt{1} \;\; s'_0 \;\; s'_1 \;\; \texttt{0} \rangle$

$p_8 : \langle \texttt{1} \;\; s_1, s_2 \;\; s_0, s_1 \;\; \texttt{1} \rangle$ $\qquad$ $p'_8 : \langle \texttt{1} \;\; s'_1 \;\; s'_0 \;\; \texttt{1} \rangle$

The selected RGPI's, $\{p_1, p_7, p_8\}$ can be instantiated as symbolic implicants $\{p'_1, p'_7, p'_8\}$ of reduced machine $\mathcal{M}'$. It is easy to verify that the result is a cover for $\mathcal{M}'$. Observe that the input and output fields are unchanged; only the present-state and next-state fields are transformed. Further, each mapped implicant contributes to at most 1 state. Note that the state mapping choice in $\langle \texttt{0} \; s'_1 \rangle$ is resolved to s'_0 by the binding effected by

p'_1.

Now the method flow has been outlined, details for each of the three steps are given in the following sections.

4.4 Symbolic Primes: RGPI's

This section completely characterizes the full set of RGPI's used by OPTIMIST, and presents simple algorithms for their generation.

The structure of this section is as follows. First, we present a naive algorithm for generating RGPI seeds. Next, we highlight the problem arising from restricting solutions to RGPI seeds. Then, we describe the solution to this problem — use of a larger set of symbolic primes, the complete set of RGPI's. A later section (section 4.9) describes a much more efficient RGPI generation algorithm, based on the use of characteristic functions.

4.4.1 Generating RGPI Seeds

RGPI seeds can be generated by a modified version of the GPI "k-cube" algorithm presented in [45]. The k-cube algorithm starts by generating small cubes, and then merges them iteratively to generate the complete set of GPI's.

The modified algorithm for RGPI seed generation works as follows. First, "0-cubes" are generated, which essentially record the next-state and output in each total state. Then, an iterative "merge-and-dominate" step is performed, in which pairs of distance-1 cubes at level i are merged to form a new cube at level $i+1$. When a merged cube has the same output and next-state field as either of its (smaller) parent cubes, the parent cube is not an RGPI seed, and is "dominated away". Otherwise, both parents and child are added to the set of RGPI seeds.

The k-cube algorithm appears as Algorithms 5, 6, and 7.

$\mathcal{M}$	0	1
s_0	$s_2, -$	$s_2, 0$
s_1	$s_2, 1$	$s_2, 0$
s_2	$s_1, 1$	$s_1, -$
s_3	$s_1, 1$	$s_2, 0$

Figure 4.5: Example of table requiring post-processing step

The algorithm as shown only produces a subset of the RGPI seeds. In particular, it produces RGPI seeds with "tight-fitting" next-state fields, which exactly equal the set of specified next-states within each seed. However, the set of RGPI seeds may include cubes that are nearly identical to cubes generated by the k-cube algorithm, but whose next-state fields specify a larger set of compatible states.

For example, Figure 4.5 shows a table and the RGPI seed $p_1 = \langle - \; s_0, s_1 \; s_2 \; 0 \rangle$. p_1 contributes only to next-state s_2, the specified next-state throughout the total states spanned by p_1. Note that p_1 would be generated by the k-cube algorithm. $\{s_2, s_3\}$ is also a compatible, and properly contains $\{s_2\}$. Hence, $p_2 = \langle - \; s_0, s_1 \; s_2, s_3 \; 0 \rangle$ is also an RGPI seed; however, p_2 would not be generated by the k-cube algorithm.

A trivial post-processing step (not shown) produces the remaining RGPI seeds, by expanding the next-state field of each seed to all possible properly containing compatibles.

Algorithm 5 Function ***merge()***

$$
\textbf{merge}(s_a,\, s_b) \;:=\; \begin{array}{ll} s_a \cup s_b & \text{if compatible}(s_a, s_b) \text{ and neither } s_a \text{ nor } s_b \text{ are } \phi \\ s_a & \text{if } s_b = DC \\ s_b & \text{if } s_a = DC \\ \phi & \text{otherwise} \end{array}
$$

4.4.2 Non-Seed RGPI's

It is not always possible to construct an optimum solution using only RGPI seeds. The reason for this is the incompatibility of RGPI's which intersect and implement next-state,

Algorithm 6 Procedure ***generate-0-cubes()***

```
generate-0-cubes() {
    cubes_0 := φ;
  foreach total state τ = ⟨IN, PS⟩ do {
      OUT := {o_i | output i ≠ 0 in τ};
      NS := {s}, iff δ(IN, PS) = s, else DC;
      cubes_0 := cubes_0 ∪ { ⟨IN, PS, OUT, NS⟩ };
  } }
```

Algorithm 7 Procedure ***generate-k-cubes()***

```
generate-k-cubes() {
    generate-0-cubes();
    for k := 1 to k_max do {
        cubes_k := φ;
        for each pair p_i, p_j in cubes_{k-1} do {
            if distance(p_i, p_j) = 1 then {
                IN' := IN_i ∪ IN_j;
                PS' := PS_i ∪ PS_j;
                NS' := merge(NS_i, NS_j);
                OUT' := OUT_i ∩ OUT_j;
                if (not empty(NS') OR not empty(OUT')) then {
                    cubes_k := cubes_k ∪ { ⟨IN', PS', NS', OUT'⟩ };
                    if (OUT' = OUT_i) AND (NS' = NS_i) then
                        mark p_i dominated;
                    if (OUT' = OUT_j) AND (NS' = NS_j) then
                        mark p_j dominated;
        } } } } }
```

but disagree on the next-state. Their incompatibility results from their commitment to *bind* the next-state entries of contained total states to *conflicting* next-states. Lacking finer-grained cubes, this interference results in cases where only sub-optimal solutions are produced, or worse, where no cover exists.

Example 4.21 The following table, when reduced using compatibles $\{s_1, s_2\}$ and $\{s_2, s_3\}$, cannot be covered by RGPI seeds alone:

$\mathcal{M}$	00	01	11	10
s_0	$s_1, -$	$s_2, -$	$s_3, -$	$s_4, 1$
s_1	$s_1, 1$	$s_1, 1$	$s_4, -$	$s_4, -$
s_2	$s_2, 1$	$s_2, -$	$s_{4,0}$	$s_4, 0$
s_3	$s_2, -$	$s_3, 0$	$s_4, 0$	$s_4, 0$
s_4	$s_4, 0$	$s_4, 1$	$s_4, 1$	$s_4, 1$

Consider the RGPI seeds $p_1 : \langle\, \texttt{0}- \;\; s_0, s_1, s_2 \;\; s_1, s_2 \;\; \texttt{1} \,\rangle$ and $p_2 : \langle\, -\texttt{1} \;\; s_0 \;\; s_2, s_3 \;\; \texttt{1} \,\rangle$. These seeds are incompatible: they intersect in total state $\langle\, \texttt{01}\; s_0 \,\rangle$, but have different next-state fields ($\{s_1, s_2\}$ vs. $\{s_2, s_3\}$), representing conflicting state mappings. Yet, p_1 and p_2 are both essential for covering minterms at $\langle\, \texttt{00}\; s_0 \,\rangle$ and $\langle\, \texttt{11}\; s_0 \,\rangle$, respectively. Hence no solution exists that uses RGPI seeds alone.

Clearly, RGPI seeds are not fine-grained enough to express the full flexibility of state mapping. In some cases, no solution consisting solely of RGPI seeds exists; in others, no optimum solution exists. In general, an optimum solution requires a combination of RGPI seeds and finer-grained symbolic cubes.

To see the kind of cubes that are needed, consider the following reduction of the above machine.

Example 4.22

$\mathcal{M}'$	00	01	11	10
$s'_0 = \{s_0\}$	$s'_1, -$	$\{s_2\}, -$	$s'_2, -$	$s'_3, 1$
$s'_1 = \{s_1, s_2\}$	$s'_1, 1$	$s'_1, 1$	$s'_3, 0$	$s'_3, 0$
$s'_2 = \{s_2, s_3\}$	$\{s_2\}, 1$	$s'_2, 0$	$s'_3, 0$	$s'_3, 0$
$s'_3 = \{s_4\}$	$s'_3, 0$	$s'_3, 1$	$s'_3, 1$	$s'_3, 1$

There are two state mapping choices in $\langle\, \texttt{01}\ s'_0 \,\rangle$: s'_1 and s'_2. If we choose mapping s'_1, we require an implicant which covers the isolated s'_2 minterm at $\langle\, \texttt{11}\ s'_0 \,\rangle$; however, no RGPI seed maps onto such an implicant. If we choose s'_2 instead, two distinct implicants are required to cover the minterms of s'_1 at $\langle\, \texttt{00}\ s'_0 \,\rangle$ and $\langle\, \texttt{01}\ s'_1 \,\rangle$, since the only RGPI seeds that contain both total states are incompatible with the state mapping choice s'_2. Unfortunately, no RGPI seed maps onto either implicant.

We need smaller RGPI's to gain finer control over state-mapping. Specifically, we require a set of implicants with smaller input and/or present-state fields.

Example 4.23 An optimum cover can be constructed if the set of RGPI seeds is augmented with the following smaller RGPI's:
$p_{1a} : \langle\, \texttt{00}\ \ s_0, s_1, s_2\ \ s_1, s_2\ \ \texttt{1} \,\rangle$,
$p_{1b} : \langle\, \texttt{0}-\ \ s_1, s_2\ \ s_1, s_2\ \ \texttt{1} \,\rangle$, and
$p_{2a} : \langle\, \texttt{11}\ \ s_0\ \ s_2, s_3\ \ \texttt{1} \,\rangle$.

The following paragraphs show how such non-seed cubes as p_{1a}, p_{1b}, and p_{2a} above can be derived from RGPI seeds by restricting their input and/or present-state fields. Two distinct approaches are used, for reasons described below. For cubes restricted in the *present-state field*, we introduce decision variables into constrained covering which determine the selected state mapping in each total state. The instantiation process then restricts the present-state dimension of each cube so as to ensure consistency with the chosen mapping. For cubes restricted in the *input field*, we simply refine RGPI seeds in the input dimension. In the following sections, we describe both approaches.

Present State Field Non-seed RGPI's which span fewer reduced states (such as p_{1b} above) are obtained by associating a set of Boolean decision variables $\{\gamma_{p,c}\}$ with each RGPI p. Each variable $\gamma_{p,c}$ is set to true iff RGPI p is to be instantiated so as to span the reduced state corresponding to c. Thus, control over the instantiation of p's present-state field is incorporated into the constrained covering step. The $\gamma_{p,c}$ assignments made during the binate covering solver are then used in symbolic instantiation to map the selected set of RGPI's onto a proper symbolic cover for the reduced machine $\mathcal{M}'$.

This approach is preferable to one which, for example, explicitly enumerates all combinations (subsets) of the states in the present-state field. Since any set of symbolic states can be made adjacent with a proper encoding, a single implicant can always be made to span those states after encoding.[5] Moreover, N binary variables $\{\gamma_{p,c}\}$ for an RGPI p with N candidate reduced states can express any of the 2^N possible subsets of those N states.

Naturally, a variable $\gamma_{p,c}$ is only meaningful for those compatibles c that are contained in $PS(p)$, since including any other compatible c could never result in an implicant of $\mathcal{M}'$.[6]Therefore, more formally, we define the set of compatibles $\Gamma_p = \{c \mid c \subseteq PS(p),\ c \in C\}$ (those whose corresponding reduced states can be considered for inclusion in an instantiation of p), and reserve a $\gamma_{p,c}$ for each c in Γ_p.

Input Field Non-seed RGPI's which are smaller in the input dimension (such as p_{1a} and p_{2a} above) are obtained by simply *refining* RGPI seeds, i.e., reducing them in all possible input dimensions, and adding the resulting sub-cubes to the RGPI set. A trivial algorithm produces such cubes readily.[7]

[5] Recall that a solution always exists to *any* combination of face embedding constraints [40].

[6] because an RGPI p is maximal in that any further expansion in the present-state field would cause it to hit the OFF-set of some output

Unlike the case for the present-state field, an optimum state mapping for the region spanned by a given RGPI seed may necessitate the use of *multiple* implicants to cover any specific next-state. This can happen because there is no guarantee that optimal state mapping will produce a cube-shaped sub-region for any given next-state. Consider for example the region of $\langle\, \texttt{0}- \; s'_0, s'_1 \,\rangle$ (the region corresponding to seed p_1) when $\langle\, \texttt{01} \; s_0 \,\rangle$ is mapped to s'_2. That state mapping choice effectively "breaks up" the single cube p_1 into two smaller RGPI's, one (p_{1a}) spanning only column 00, and another (p_{1b}) spanning columns 00 and 01 in row s'_1.

There seems to be no convenient way to encode a subset of the 3^N possible sub-cubes of an RGPI seed spanning 2^N input columns (i.e., a seed with N full input literals). Hence, these sub-cubes are instead explicitly represented in the set of RGPI's.

The additional cubes increase the complexity of the covering problem, in exchange for the control necessary to ensure an exactly optimum solution to the state mapping problem.

4.5 Constraint Generation

Once the sets of state compatibles and RGPI's have been generated, constraints must then be generated.

This section describes the novel set of binate constraints used by Optimist in the third step of the process, after RGPI generation. These constraints ensure that the selection of RGPI's and compatibles constitutes a valid (symbolic) realization of the original machine $\mathcal{M}$.

In the following subsections, the decision variables involved are first introduced. Next, the cost model is briefly described, and the constraints themselves are presented in detail. Finally, a mock recursive descent walk-through of the constraint solution process

[7] This algorithm is not shown, because in reality Optimist uses the much more efficient RGPI generation method presented in section 4.9.

helps illustrate the interactions among the various types of constraint.

4.5.1 Constraint Matrix Variables

The covering problem can be expressed as a constraint matrix, where each row is a constraint and each column is a decision variable. The covering matrix contains three kinds of columns: state compatibles, RGPI's, and "γ" variables (which modulate the instantiation of a selected RGPI's present-state field, as discussed in Section 4.4.2).

Specifically, the columns in the covering matrix are:

variable		**description**	**cost**
c_i	—	include compatible c_i in the state cover	0
p_i	—	include RGPI p_i in the symbolic logic cover	1
γ_{p_i,c_i}	—	make RGPI p_i span the reduced state c_i in $\mathcal{M}'$	0

4.5.2 Cost Model

Optimist's cost model for optimal state minimization, namely, the cardinality of the 2-level logic cover, is reflected in the column costs shown above. In particular, the cost of selecting an RGPI is 1; all other columns have 0 cost.

Given a binate solver capable of handling a two-tiered cost function, a more appropriate cost model for optimal state minimization would assign RGPI selection to the higher tier, while assigning compatible selection to the lower tier. This cost would have the effect of finding a closed state cover that has lowest cardinality, among all state covers yielding minimum cardinality logic covers.

4.5.3 Constraints

There are five distinct sets of constraints, each addressing a specific requirement on valid realizations. Two sets (sets 1 and 3 below) correspond directly to classic state minimization constraints, which ensure a closed cover of state compatibles. One set (set

2) ensures that the ON-set of the reduced machine $\mathcal{M}'$ is covered by the selected set of RGPI's. An additional set (set 4) ensures that the machine is state-mapped consistently by the selected RGPI's. The final set of constraints (set 5) provides the solver with a trivial means of determining the cover's cost.

The constraint clauses, which correspond to covering matrix rows, are shown below, grouped in five sections according to purpose. Each section consists of a set of similar clauses, all of which must be satisfied. The final Boolean expression is a conjunction of all rows/clauses in all five sections.

Below, some constraints are shown in the form $a \rightarrow b$, to emphasize the implication between the choices a and b. Such constraints are equivalent to the Boolean binate expression $\overline{a} + b$.

1. *State Covering*

   Each unreduced state must be covered by some state compatible (reduced state).

   $\forall$ states s of $\mathcal{M}$

   $$c_{i_1} + c_{i_2} + \cdots + c_{i_N}$$

   where $s \in c_{i_k}$. This corresponds to a classic closure constraint [60].

2. *Compatible Selection $\Rightarrow$ RGPI Selection*

   ("Functional Covering")

   Each ON-set minterm of each output and next-state of $\mathcal{M}'$ must be covered. Since selecting a compatible corresponds to adding a row in the reduced table, this constraint ensures that every ON-set minterm in the reduced row is covered.

   $\forall$ compatibles c_k

   $\forall$ minterms m' lying in c_k

   $$c_k \rightarrow \gamma_{p_1,c_k} + \gamma_{p_2,c_k} + \cdots + \gamma_{p_N,c_k}$$

   Here, m' is an ON-set minterm lying in c_k, corresponding to either an output or the next-state in the *reduced* machine $\mathcal{M}'$. That is, m' lies in some total state $\langle \mathcal{I}_i, s'_k \rangle$ of $\mathcal{M}'$, where s'_k corresponds to c_k. $p_1 \ldots p_N$ are those RGPI's that cover m'. If we select compatible c_k, we must cover minterm m', and so must select some RGPI p_i. Because of the need to control the present-state extent of instantiated RGPI's, however, we do not directly select p_i. Rather, we use γ variables to delimit the present-state extent of each selected RGPI. As a result, each clause does not select p_i, but instead selects γ_{p_i,c_k}, in order to ensure that RGPI p_i will be mapped over c_k. This in turn will guarantee that m' is covered in $\mathcal{M}'$.

Although the minterms m' in row c_k of the reduced machine are not yet explicitly available, they can easily be derived. Let $\mathcal{I}_i \in \mathcal{I}$ be any input column. A minterm $m' = \langle \mathcal{I}_i,\, s'_k \rangle$ is in the ON-set of a binary *output* j in reduced machine $\mathcal{M}'$ iff some unreduced state $s \in c_k$ specifies a 1 for output j; i.e., $\lambda_j(s, \mathcal{I}_i) = 1$.

Similarly, minterm m' is considered an ON-set minterm of the *symbolic next-state* of the reduced machine $\mathcal{M}'$ iff for some unreduced state $s \in c_k$, $\delta(s, i) = \tilde{s}$, for some singleton state $\tilde{s}$; that is, the next-state is specified in s. In this case, the next-state in m' will also be specified.

For each ON-set minterm, m', those γ_{p_i,c_k} are included in the above constraints for which (i) p_i contributes to the corresponding output or next-state, and (ii) also contains the minterm (i.e. intersects input column i).

3. *RGPI Selection ⇒ Compatible Selection*

   If an RGPI which implements next-state is selected, the corresponding state compatible must also be selected.

   $\forall$ RGPI's $p_i \mid NS(p_i) \neq \phi$

   $$p_i \rightarrow c_k$$

   where compatible $c_k = NS(p_i)$.

   This constraint corresponds to a classic closure constraint [60]. An RGPI p_i which implements next-state identifies a unique reduced state to which the next-state of all contained total states is uniformly state mapped. Hence, RGPI selection implies a commitment to select the compatible c_k corresponding to its next-state field.

Thus, if RGPI p_i implements next state, and has next-state field equal to c_k, c_k must be selected if p_i is selected. Note that an RGPI implementing only outputs has no such constraint.

4. *Implicant Incompatibility*

   Two RGPI's are incompatible if they intersect in some selected compatible, but disagree on next-state.

   Recall that each RGPI corresponds to a state mapping choice for the total states that it contains. This set of constraints states that overlapping RGPI's cannot be simultaneously selected if they represent conflicting state mapping choices.

   $\forall$ RGPI's $p_i,\, p_j \;\mid i \neq j,\; IN(p_i) \cap IN(p_j) \neq \phi,$

   $NS(p_i) \neq \phi,\; NS(p_j) \neq \phi,\; NS(p_i) \neq NS(p_j),$

   $\forall$ compatibles $c_k \subseteq PS(p_i) \cap PS(p_j)$

   and $\delta(IN(p_i) \cap IN(p_j),\, c_k) \neq \mathcal{S}$

   $$c_k \;\rightarrow\; \overline{\gamma_{p_i,c_k}} + \overline{\gamma_{p_j,c_k}}$$

   If two RGPI's implement next-state and disagree, and intersect in some compatible c_k, they can both be mapped over c_k *only* if the next-state is unspecified ($\delta(\mathcal{I}_a, s) = \mathcal{S}$) throughout the region of intersection. Otherwise, there is a conflict: a total state would be simultaneously mapped in two ways. We must either not select c_k, or not map one of the RGPI's onto reduced state c_k.

   The above constraint thus identifies the precise conditions under which 2 RGPI's p_i and p_j overlap and specify conflicting next-states. When such a conflict exists, the binate constraint ensures that at least one of the RGPI's is not mapped over the reduced state.

5. *Implicant Cost*

   Mapping an RGPI over a reduced state implies selecting that RGPI.

   $\forall$ RGPI's p_i, $\forall$ compatibles $c_k \in \Gamma_{p_i}$

   $$\gamma_{p_i,c_k} \;\rightarrow\; p_i$$

This is a bookkeeping device to make it easier for the solver to determine the solution cost (logic cover cardinality). The variable p_i is made true if p_i is mapped over *at least one* reduced state.

The rationale for this constraint is as follows. A given RGPI p_i can be mapped over several reduced states, if several $\gamma_{p_i,c}$ are set to 1. However, no matter how many reduced states p_i spans, it contributes only 1 to the cardinality of the logic cover. This constraint ensures that each p_i is counted only once.

4.5.4 Flow of Constraint Solution

The implications interconnecting the various constraints can be envisioned in a "flow" from, e.g., state covering (set 1) to functional covering (set 2) to state implication (set 3), and so on. To better illustrate the relationships, we offer the following solution scenario.[8]

1. We start by selecting a compatible c_i to cover some state, say, s_0. This satisfies the corresponding clause from set 1.

2. The selection of compatible c_i implies covering conditions on the ON-set minterms m' of $\mathcal{M}'$ lying in c_i (set 2). We choose one such minterm, and map an eligible RGPI p over c_i to cover it, by setting γ_{p,c_i} to true.

3. As a result, the corresponding RGPI, p is added to the cover (set 5).

4. Once p is selected, the closure requirement demands the selection of the compatible c_j associated with p's next-state field, if any (set 3). If c_j was not previously selected, new covering constraints must be satisfied, and we recur on step 2.

5. If there are uncovered minterms in c_i, continue at step 2.

6. If there are uncovered states, continue at step 1.

[8]which can be regarded as a crude recursive-descent algorithm

At any point, the currently selected RGPI's may be incompatible (set 4) with RGPI's mentioned in covering clauses (set 2) for a specific compatible. If so, the latter RGPI's cannot be mapped over that compatible, and the corresponding γ's are removed from consideration in this sub-tree. There may then be no RGPI eligible to cover some minterm in step 2. If so, the sub-problem has no solution, and backtracking occurs.

4.5.5 Efficient Constraint Solution

The constraints used by OPTIMIST are all unate and binate. Hence, in practice, an efficient general-purpose binate solver such as Coudert's Scherzo [30] is used to solve these constraints.

4.6 Symbolic Instantiation

Once all constraints are solved, the result is a selected set of compatibles and symbolic implicants (RGPI's). The final step produces a symbolic logic cover for the chosen reduced machine by instantiating the selected RGPI's. Section 4.3.5 gave a somewhat intuitive but incomplete description of the process of symbolic instantiation.

We now define symbolic instantiation precisely, taking into account the γ variables associated with each RGPI. γ variables were introduced in section 4.4 in order to gain finer control over the shape of instantiated implicants.

Symbolic instantiation is defined with respect to a selected set of compatibles (and hence a reduced machine $\mathcal{M}'$), along with the set of γ variable assignments identifying specific reduced states to span. Specifically,

$$p' = \text{Instantiate}_{\gamma_p}(p)$$

where γ_p is a set of Boolean variables associated with p. Each member of γ_p is associated with a compatible contained in $PS(p)$. We let

$$p' = \text{Instantiate}_{\gamma_p}(p) = \langle I,\ PS',\ NS',\ O \rangle$$

with NS' as before, but, for the present state field:

$$\begin{aligned} PS' = \{\text{reduced states } s' \mid \ & \gamma_{p,c} = 1 \text{ and} \\ & s' \text{ corresponds to } c\ \} \end{aligned}$$

That is, the present state of p' contains all reduced states whose corresponding γ variables were assigned a 1 value by the binate solver. Any other selected compatible c', which could be covered by p but was not selected for covering by p (i.e. $\gamma_{p,c'} = 0$) will not be included in PS'. In this way, the γ assignments determine the precise extent of the instantiated implicant p', i.e., the set of reduced rows p' will span.

4.7 Examples

This section presents two examples to illustrate the Optimist method.

The first example illustrates the mechanics of the procedure by giving complete results for each step of the procedure, using the simple table of Figure 4.3. The set of compatibles, RGPI's, and all constraints are shown. We show both the optimum logic cover, corresponding to the optimum state mapping, and a sub-optimal logic cover, resulting from sub-optimal mapping.

The second example focuses on the results obtained by the Optimist method. In particular, it demonstrates an optimum result that is not available to other existing methods. In this case, Optimist wins by relaxing both of the common restrictions of classic state minimization methods: (i) the search for only minimum-cardinality state covers, and (ii) the use of prime compatibles.

Example 4.24 (from Figure 4.3)

$\mathcal{M}$	0	1
s_0	$s_0, 0$	$s_2, 0$
s_1	$s_1, 0$	$s_1, -$
s_2	$s_1, -$	$s_0, 1$

Maximal compatibles:

$MC = \{\{s_0, s_1\}, \{s_1, s_2\}\}$

Prime compatibles:

$PC = \{c_0, \ldots, c_4\} =$

$\{\{s_0\}, \{s_1\}, \{s_2\}\} \cup MC.$

The RGPI seeds were given in section 4.3.2. There are no non-seed RGPI's. The complete set of constraints follows, in POS form, and grouped by section.

State Covering:

$(c_0 + c_3)(c_1 + c_3 + c_4)(c_2 + c_4)$

Functional Covering:

The first clause ensures that, when compatible c_0 is selected, the minterm for s_0 in $\langle\, \mathtt{0}\ s_0 \,\rangle$ is then covered. This clause is satisfied when, for example, γ_{p_0,c_0} is set to true, thereby mapping p_0 over c_0. Likewise, the second clause ensures the covering of the minterm for s_2 in $\langle\, \mathtt{1}\ s_0 \,\rangle$, when c_0 is selected.

$(c_0 \rightarrow \gamma_{p_0,c_0} + \gamma_{p_1,c_0})(c_0 \rightarrow \gamma_{p_4,c_0} + \gamma_{p_7,c_0})$

$(c_1 \rightarrow \gamma_{p_1,c_1} + \gamma_{p_2,c_1} + \gamma_{p_6,c_1} + \gamma_{p_9,c_1} + \gamma_{p_{10},c_1} + \gamma_{p_{14},c_1})$

$(c_1 \rightarrow \gamma_{p_5,c_1} + \gamma_{p_6,c_1} + \gamma_{p_7,c_1} + \gamma_{p_8,c_1} + \gamma_{p_{10},c_1} + \gamma_{p_{14},c_1} + \gamma_{p_{15},c_1})$

$(c_2 \rightarrow \gamma_{p_1,c_2} + \gamma_{p_2,c_2} + \gamma_{p_3,c_2} + \gamma_{p_9,c_2} + \gamma_{p_{10},c_2} + \gamma_{p_{11},c_2} + \gamma_{p_{13},c_2})$

$(c_2 \rightarrow \gamma_{p_8,c_2} + \gamma_{p_{10},c_2} + \gamma_{p_{11},c_2} + \gamma_{p_{12},c_2})$

$(c_2 \rightarrow \gamma_{p_8,c_2} + \gamma_{p_{11},c_2} + \gamma_{p_{12},c_2})$

$(c_3 \rightarrow \gamma_{p_1,c_3})(c_3 \rightarrow \gamma_{p_7,c_3})$

$(c_4 \rightarrow \gamma_{p_1,c_4} + \gamma_{p_2,c_4} + \gamma_{p_9,c_4} + \gamma_{p_{10},c_4})(c_4 \rightarrow \gamma_{p_8,c_4} + \gamma_{p_{10},c_4})(c_4 \rightarrow \gamma_{p_8,c_4})$

Compatible Implication:

Each clause ensures that when an RGPI that implements next-state is selected, the corresponding next-state is also selected.

$(p_0 \rightarrow c_0)(p_1 \rightarrow c_3)(p_2 \rightarrow c_1)(p_3 \rightarrow c_1)(p_4 \rightarrow c_2)(p_5 \rightarrow c_1)$

$(p_6 \rightarrow c_1)(p_7 \rightarrow c_4)(p_8 \rightarrow c_3)(p_9 \rightarrow c_4)(p_{10} \rightarrow c_3)(p_{11} \rightarrow c_3)(p_{12} \rightarrow c_0)$

$(p_{13} \rightarrow c_4)(p_{14} \rightarrow c_4)(p_{15} \rightarrow c_4)$

Implicant Incompatibility:

RGPI's p_0 and p_1 implement the reduced next-states $\{s_0\}$ and $\{s_0, s_1\}$, respectively, and overlap in total state $\langle\, \mathtt{0}\ s_0 \,\rangle$. Because of their disagreement, they cannot both be mapped over c_0. Hence, either c_0 must not be selected, or one of $\{\gamma_{p_0,c_0}, \gamma_{p_1,c_0}\}$ must not be.

$(c_0 \rightarrow \overline{\gamma_{p_0,c_0}} + \overline{\gamma_{p_1,c_0}})(c_0 \rightarrow \overline{\gamma_{p_4,c_0}} + \overline{\gamma_{p_7,c_0}})$

$(c_1 \rightarrow \overline{\gamma_{p_1,c_1}} + \overline{\gamma_{p_2,c_1}})(c_1 \rightarrow \overline{\gamma_{p_1,c_1}} + \overline{\gamma_{p_6,c_1}})(c_1 \rightarrow \overline{\gamma_{p_1,c_1}} + \overline{\gamma_{p_9,c_1}})(c_1 \rightarrow \overline{\gamma_{p_1,c_1}} + \overline{\gamma_{p_{14},c_1}})$

$(c_1 \rightarrow \overline{\gamma_{p_2,c_1}} + \overline{\gamma_{p_9,c_1}})(c_1 \rightarrow \overline{\gamma_{p_2,c_1}} + \overline{\gamma_{p_{10},c_1}})(c_1 \rightarrow \overline{\gamma_{p_2,c_1}} + \overline{\gamma_{p_{14},c_1}})$

$(c_1 \rightarrow \overline{\gamma_{p_6,c_1}} + \overline{\gamma_{p_9,c_1}})(c_1 \rightarrow \overline{\gamma_{p_6,c_1}} + \overline{\gamma_{p_{10},c_1}})(c_1 \rightarrow \overline{\gamma_{p_6,c_1}} + \overline{\gamma_{p_{14},c_1}})$

$(c_1 \rightarrow \overline{\gamma_{p_9,c_1}} + \overline{\gamma_{p_{10},c_1}})(c_1 \rightarrow \overline{\gamma_{p_{10},c_1}} + \overline{\gamma_{p_{14},c_1}})$

$(c_1 \rightarrow \overline{\gamma_{p_5,c_1}} + \overline{\gamma_{p_7,c_1}})(c_1 \rightarrow \overline{\gamma_{p_5,c_1}} + \overline{\gamma_{p_8,c_1}})(c_1 \rightarrow \overline{\gamma_{p_5,c_1}} + \overline{\gamma_{p_{10},c_1}})(c_1 \rightarrow \overline{\gamma_{p_5,c_1}} + \overline{\gamma_{p_{14},c_1}})(c_1 \rightarrow \overline{\gamma_{p_5,c_1}} + \overline{\gamma_{p_{15},c_1}})$

$(c_1 \rightarrow \overline{\gamma_{p_6,c_1}} + \overline{\gamma_{p_7,c_1}})(c_1 \rightarrow \overline{\gamma_{p_6,c_1}} + \overline{\gamma_{p_8,c_1}})(c_1 \rightarrow \overline{\gamma_{p_6,c_1}} + \overline{\gamma_{p_{10},c_1}})(c_1 \rightarrow \overline{\gamma_{p_6,c_1}} + \overline{\gamma_{p_{14},c_1}})(c_1 \rightarrow \overline{\gamma_{p_6,c_1}} + \overline{\gamma_{p_{15},c_1}})$

$(c_1 \rightarrow \overline{\gamma_{p_7,c_1}} + \overline{\gamma_{p_8,c_1}})(c_1 \rightarrow \overline{\gamma_{p_7,c_1}} + \overline{\gamma_{p_{10},c_1}})$

$(c_1 \rightarrow \overline{\gamma_{p_8,c_1}} + \overline{\gamma_{p_{14},c_1}})(c_1 \rightarrow \overline{\gamma_{p_8,c_1}} + \overline{\gamma_{p_{15},c_1}})$

$(c_1 \rightarrow \overline{\gamma_{p_{10},c_1}} + \overline{\gamma_{p_{14},c_1}})(c_1 \rightarrow \overline{\gamma_{p_{10},c_1}} + \overline{\gamma_{p_{15},c_1}})$

$(c_2 \rightarrow \overline{\gamma_{p_1,c_2}} + \overline{\gamma_{p_2,c_2}})(c_2 \rightarrow \overline{\gamma_{p_1,c_2}} + \overline{\gamma_{p_3,c_2}})(c_2 \rightarrow \overline{\gamma_{p_1,c_2}} + \overline{\gamma_{p_9,c_2}})(c_1 \rightarrow \overline{\gamma_{p_1,c_1}} + \overline{\gamma_{p_{13},c_1}})$

$(c_2 \rightarrow \overline{\gamma_{p_2,c_2}} + \overline{\gamma_{p_9,c_2}})(c_2 \rightarrow \overline{\gamma_{p_2,c_2}} + \overline{\gamma_{p_{10},c_2}})(c_2 \rightarrow \overline{\gamma_{p_2,c_2}} + \overline{\gamma_{p_{11},c_2}})(c_1 \rightarrow \overline{\gamma_{p_2,c_1}} + \overline{\gamma_{p_{13},c_1}})$

$(c_2 \rightarrow \overline{\gamma_{p_3,c_2}} + \overline{\gamma_{p_9,c_2}})(c_2 \rightarrow \overline{\gamma_{p_3,c_2}} + \overline{\gamma_{p_{10},c_2}})(c_2 \rightarrow \overline{\gamma_{p_3,c_2}} + \overline{\gamma_{p_{11},c_2}})(c_2 \rightarrow \overline{\gamma_{p_3,c_2}} + \overline{\gamma_{p_{13},c_2}})$

$(c_2 \rightarrow \overline{\gamma_{p_9,c_2}} + \overline{\gamma_{p_{10},c_2}})(c_2 \rightarrow \overline{\gamma_{p_9,c_2}} + \overline{\gamma_{p_{11},c_2}})(c_2 \rightarrow \overline{\gamma_{p_{10},c_2}} + \overline{\gamma_{p_{13},c_2}})(c_2 \rightarrow \overline{\gamma_{p_{11},c_2}} + \overline{\gamma_{p_{13},c_2}})$

$(c_4 \rightarrow \overline{\gamma_{p_1,c_4}} + \overline{\gamma_{p_2,c_4}})(c_4 \rightarrow \overline{\gamma_{p_1,c_4}} + \overline{\gamma_{p_9,c_4}})(c_4 \rightarrow \overline{\gamma_{p_2,c_4}} + \overline{\gamma_{p_9,c_4}})(c_4 \rightarrow \overline{\gamma_{p_2,c_4}} + \overline{\gamma_{p_{10},c_4}})$

$(c_4 \rightarrow \overline{\gamma_{p_9,c_4}} + \overline{\gamma_{p_{10},c_4}})$

Implicant Cost:

$(\gamma_{p_0,c_0} \rightarrow p_0)(\gamma_{p_1,c_0} \rightarrow p_1)(\gamma_{p_1,c_1} \rightarrow p_1)(\gamma_{p_1,c_2} \rightarrow p_1)(\gamma_{p_1,c_3} \rightarrow p_1)(\gamma_{p_1,c_4} \rightarrow p_1)$

$(\gamma_{p_2,c_1} \rightarrow p_2)(\gamma_{p_2,c_2} \rightarrow p_2)(\gamma_{p_2,c_4} \rightarrow p_2)(\gamma_{p_3,c_2} \rightarrow p_3)(\gamma_{p_4,c_0} \rightarrow p_4)$

$(\gamma_{p_5,c_1} \rightarrow p_5)(\gamma_{p_6,c_1} \rightarrow p_6)(\gamma_{p_7,c_0} \rightarrow p_7)(\gamma_{p_7,c_1} \rightarrow p_7)(\gamma_{p_7,c_3} \rightarrow p_7)$

$(\gamma_{p_8,c_1} \rightarrow p_8)(\gamma_{p_8,c_2} \rightarrow p_8)(\gamma_{p_8,c_4} \rightarrow p_8)(\gamma_{p_9,c_1} \rightarrow p_9)(\gamma_{p_9,c_2} \rightarrow p_9)(\gamma_{p_9,c_4} \rightarrow p_9)$

$(\gamma_{p_{10},c_1} \rightarrow p_{10})(\gamma_{p_{10},c_2} \rightarrow p_{10})(\gamma_{p_{10},c_4} \rightarrow p_{10})(\gamma_{p_{11},c_2} \rightarrow p_{11})(\gamma_{p_{12},c_2} \rightarrow p_{12})$

$(\gamma_{p_{13},c_2} \rightarrow p_{13})(\gamma_{p_{14},c_1} \rightarrow p_{14})(\gamma_{p_{15},c_1} \rightarrow p_{15})$

Selecting compatibles $\{s_0, s_1\}$ and $\{s_1, s_2\}$ gives two state-mapping choices in $\langle\, \mathtt{0}\ s_1' \,\rangle$, as observed earlier. The sub-optimal 4-RGPI cover $\{p_1, p_7, p_9, p_{11}\}$ corresponds to choosing s_1'. Our method instead finds the minimum cover $\{p_1, p_7, p_{11}\}$, with 3 RGPI's, which corresponds to the optimum state-mapping of s_0'.

Example 4.25 The following example demonstrates that Optimist, which considers *non-minimum* cardinality state covers, forms the optimum solution which other methods cannot find. Specifically, a reduction of $\mathcal{M}$ by a minimum-cardinality state cover (as other methods would require) results in a sub-optimal logic cover.

$\mathcal{M}$	00	01	11	10
s_0	$s_2, 1$	$s_1, 0$	$s_1, -$	$s_0, 0$
s_1	$s_2, 0$	$s_1, -$	$s_1, -$	$-, -$
s_2	$s_2, -$	$s_1, 1$	$s_1, 1$	$s_0, 1$

Prime compatibles:

$\{s_0\}$ and $\{s_1, s_2\}$.

$\mathcal{M}'$	00	01	11	10
s_0'	$s_1', 1$	$s_1', 0$	$s_1', -$	$s_0', 0$
s_1'	$s_1', 0$	$s_1', 1$	$s_1', 1$	$s_0', 1$

$\mathcal{M}$, reduced via $\{s_0\}$ and $\{s_1, s_2\}$.

The minimum logic covers for these two machines are shown below, for $\mathcal{M}$ on the left (RGPI's), and for $\mathcal{M}'$ on the right (instantiated RGPI's):

$p_1 : \langle 00 \;\; s_0, s_2 \;\; s_2 \;\; 1\rangle$
$p_2 : \langle -1 \;\; s_0, s_1, s_2 \;\; s_1 \;\; 0\rangle$
$p_3 : \langle 10 \;\; s_0, s_1, s_2 \;\; s_0 \;\; 0\rangle$
$p_4 : \langle 00 \;\; s_0, s_1, s_2 \;\; s_2 \;\; 0\rangle$
$p_5 : \langle -- \;\; s_2 \;\; - \;\; 1\rangle$

$p_1' : \langle 0- \;\; s_0', s_1' \;\; s_1' \;\; 0\rangle$
$p_2' : \langle 00 \;\; s_0' \;\; s_1' \;\; 1\rangle$
$p_3' : \langle -1 \;\; s_1' \;\; s_1' \;\; 1\rangle$
$p_4' : \langle 10 \;\; s_0', s_1' \;\; s_0' \;\; 0\rangle$
$p_5' : \langle 11 \;\; s_0', s_1' \;\; s_1' \;\; 1\rangle$
$p_6' : \langle 10 \;\; s_1' \;\; s_0' \;\; 1\rangle$

It is not hard to show that these two state and logic covers represent two distinct solutions to the covering constraints.

OPTIMIST produces the optimum (unreduced) implementation $\mathcal{M}$, with cover $p_1, \ldots, p_5$ and the 3 original states s_0, s_1 and s_2. Consider p_5, which is selected in the cover. Its use is critical to forming the minimum cover, since it covers all the ON-set minterms of the output in s_2. In contrast, p_5 cannot be used in the reduced machine $\mathcal{M}'$, since it cannot be mapped over s_1'. As a result, in $\mathcal{M}'$, 2 RGPI's are needed to cover the output's ON-set minterms in s_2. Our method therefore finds the minimum cover based on the selection of compatible $\{s_2\}$, and discards the sub-optimal solution based on $\{s_1, s_2\}$.

It is important to observe that the minimum logic cover (OPTIMIST's solution) could only be formed when using the *non-prime* compatibles ($\{s_1\}$ and $\{s_2\}$). Moreover, using these non-prime compatibles in turn necessitates the formation of a non-minimum cardinality state cover. Both of these abilities are unique to OPTIMIST; previously existing tools consider only minimum-cardinality state covers of prime compatibles. Thus, Optimist produces the optimum solution, which existing tools cannot find.

4.8 Theoretical Results

This section presents the major theoretical results regarding Optimist's method for optimal state minimization. First, it is established that every solution to the binate constraints constitutes a valid pair of a reduced machine and logic cover (after instantiation). Second, the optimality of the method is proven.

4.8.1 Correctness

The proof of correctness of the Optimist method has two parts. First, it is shown that a solution to the binate constraints describes a valid reduced machine, by verifying that the selected set of compatibles always forms a closed state cover. Second, the selected set of RGPI's is shown always to correspond to a proper symbolic logic cover for the given reduced machine.

Lemma 4.26 *Any solution to the constraints of section 4.5 describes a valid state cover.*

Proof: The unate constraints of part 1 are precisely the state covering constraints of Grasselli. Since all constraints must be satisfied by any solution, the set of selected compatibles forms a state cover. □

Lemma 4.27 *Any solution to the constraints of section 4.5 describes a valid state mapping for the given state cover.*

Proof: All state covering constraints in part 1 have been satisfied, by Lemma 4.26. Hence, for each (unate) state covering constraint, a column corresponding to some compatible (say, c_i) must have been selected. In the associated reduced state s_i', there is at least one state mapping choice for each reduced total state. Now, the constraints of part 2 contain one clause for each total state identifying all RGPI's that cover the next-state minterm, of the form $c_i \rightarrow \gamma_{p_{j_1},c_i} + \cdots + \gamma_{p_{j_n},c_i}$. The selection of c_i now necessitates the selection of one of the γ_{p_j,c_i}.

By the definition of the part 2 constraints, each γ_{p_j,c_i} identifies an RGPI p_j that contains the given reduced total state. By construction, p_j's next-state field (say, c_k) also covers the implied compatible in every total state it contains. Choosing γ_{p_j,c_i} guarantees that p_j will be instantiated so as to span c_i, thereby implicitly defining a state mapping to c_k in that total state.

Finally, the constraints of part 4 (implicant incompatibility) ensure that no two RGPI's which disagree on next-state are made to overlap in any reduced total state. Hence, the state mapping is consistent. □

Lemma 4.28 *Any solution to the constraints of section 4.5 describes a* closed *state cover.*

Proof: Given that all states are covered, and that a valid (implicit) state mapping has been chosen, we must establish that the implied compatible (if any) has been selected in all total states.

In the proof of Lemma 4.27, it was noted that every next-state minterm in the reduced machine is covered by the selection of some γ_{p_j,c_i}. Now, choosing γ_{p_j,c_i} forces the selection of p_j, by virtue of a constraint in part 5, of the form $\gamma_{p_j,c_i} \rightarrow p_j$. Finally, for each RGPI implementing next-state, there exists a constraint in part 4 of the form $p_j \rightarrow c_k$, which can now only be satisfied by choosing c_k. Thus, the state cover is closed, since the implied compatible in every total state has been selected. □

Theorem 4.29 *Any solution to the constraints of section 4.5 describes a valid reduced* *machine.*

Proof: This follows directly from Lemmas 4.26, 4.27, and 4.28. □

Denote the reduced machine identified by a given solution to the constraints (as described above) as $\mathcal{M}'$. Likewise, for a given RGPI p, denote the instantiation of p on $\mathcal{M}'$ as p', according to the definition of section 4.6.

Proving that the constraint solution embodies a logic cover for $\mathcal{M}'$ requires that we establish some relationships between RGPI's and the set of implicants on $\mathcal{M}'$.

Lemma 4.30 *Every RGPI p maps onto a (possibly empty) implicant for* any *reduced machine $\mathcal{M}'$, under a consistent γ assignment.*

By "consistent γ assignment", we mean an assignment of γ_{p,c_i} variables that does not cause p' to span any row of $\mathcal{M}'$ for which the state mapping is inconsistent with $NS(p')$.

Proof: p by construction does not hit the OFF-set of any output of $\mathcal{M}$ to which it contributes, and contributes only to compatible next-states. p' can only span a reduced state s' if p spanned the entire compatible c corresponding to s. Clearly, then, p' does not hit the OFF-set of any output to which it contributes in $\mathcal{M}'$. Moreover, a consistent γ assignment excludes from $PS(p')$ any state s' for which $NS(p')$ is incompatible with the chosen state mapping in some contained input column. Thus, p' implements the unique next-state consistent with the chosen state mapping throughout p' (or else p' has an empty PS field, and is an empty product). Therefore, p' does not hit the OFF-set of the next-state (if any) to which it contributes. Hence p' is an implicant of $\mathcal{M}'$. □

Theorem 4.31 *Any solution to the constraints of section 4.5 identifies a selection of RGPI's which, when instantiated, implements $\mathcal{M}'$.*

Proof: The functional covering constraints of part 2 directly enumerate and cover the ON-set minterms in $\mathcal{M}'$, row by row. Thus, some RGPI p has been selected to cover each minterm m' in $\mathcal{M}'$ (by assigning γ_{p,c_m} to 1). What remains to be proven is that p's instantiation, p', covers m'. This follows from the basis for an RGPI's qualification in the covering clause for m'. Specifically, because p spans the given input column in which m' resides, spans the entire compatible set of states, and because γ_{p,c_m} was assigned 1, p' will cover m'. □

Theorem 4.32 *A solution to the constraints of section 4.5 always exists.*

Proof: Because the bulk of the constraints (parts 1 through 3 and part 5) represent a monotonic set of implications, they can always be satisfied, as long as:

(i) there exists at least one compatible to cover each state,

(ii) there exists, for any given minterm of any reduced row in $\mathcal{M}'$, at least one RGPI which satisfies the covering requirement of part 2, and

(iii) the non-monotonic constraints (part 4) always leave some solution accessible.

The first requirement is true, assuming that a reasonable set of compatibles (e.g. prime, maximal, or all) is used.

The following discussion demonstrates point (ii) above. Although it examines a single output, it applies equally to both next-state and multiple outputs, mutatis mutandis.

Each ON-set minterm in $\mathcal{M}'$ derives from one or more ON-set minterms in $\mathcal{M}$. In particular, ON-set minterm m' in $\mathcal{M}'$ in reduced state s'_i derives from (at least) one ON-set minterm m in $\mathcal{M}$ in some unminimized state $s \in c_i$ that s'_i covers. Moreover, because every state in $\mathcal{M}$ must be covered in $\mathcal{M}'$, every minterm in $\mathcal{M}$ maps onto (at least[9]) one minterm in $\mathcal{M}'$. Hence, given that every minterm m in $\mathcal{M}$ is covered by at

[9]possibly more than one, due to state splitting

least one RGPI, we need only show that one of these RGPI's will cover m' in $\mathcal{M}'$ after instantiation (under an assignment of the associated γ_{p_j,c_i} to 1).

We now construct such an RGPI. For ON-set minterm m, construct an RGI p_0 whose extent is the total state in which m resides, and which covers m. Now, any reduced minterm m' in $\mathcal{M}'$ deriving from m resides in a reduced state s_i' corresponding to some compatible c_i. From the above, p_0 can be expanded into all states in c_i, yielding an RGI p_c. Now, if p_c is maximal in its present-state field, it is an RGPI; otherwise, choose any RGPI containing p_c. It is easy to see that the result, p, contains m' after instantiation.

It is important to note that the RGPI so constructed may not be an RGPI seed — it always spans a single input column and may thus not be maximal in the input dimension. This reflects the fact that solutions are not always available using RGPI seeds alone.

Finally, the constraints of part 4 (implicant incompatibility) assert the inconsistency of instantiating pairs of RGPI's that enforce different state mappings over the same reduced total state. However, for any reduced total state, at least one RGPI exists that is consistent with any given valid state mapping and spans only that input column (e.g. the RGPI constructed above). Clearly a cover for $\mathcal{M}'$ can be built solely from such column-wide RGPI's, restricting the assignment of γ_{p_j,c_i} so that only one RGPI spans any given reduced row in a given column. Such a cover clearly satisfies the consistency constraints of part 4, since only one RGPI contains any total state.$\Box$

Non-prime Instantiated Covers A subtle issue arises regarding instantiated logic covers: the instantiation process described in Section 4.6 does not guarantee a *prime* symbolic logic cover for the reduced machine $\mathcal{M}'$. Rather, it guarantees an exactly-minimum cardinality cover, possibly using non-primes. In particular, a given set of RGPI's and a given valid selection of $\gamma_{p,c}$ variables may result in non-prime implicants after instantiation. This will occur if, say, γ_{p_1,c_a} is selected to cover some ON-set minterm in c_a and p_1 is consistent with the next-state[10] of another state c_b, but γ_{p_1,c_b} is not selected because the ON-set minterms of c_b were already covered. So, although under OPTIMIST's cost model, the cost of the solution does not increase when selecting a $\gamma_{p,c}$ for an already-selected RGPI p, it may not contribute to the satisfaction of any additional constraints.

As we will see in the next section, this effect does *not* interfere with the optimality of logic covers corresponding to minimum-cost solutions to the binate constraints. (The next section will show that *any* prime cover of any $\mathcal{M}'$ has an image on $\mathcal{M}$ consisting of RGPI's.) A prime cover of identical cardinality can always be derived using, e.g., the EXPAND step of ESPRESSO [118] as a post-processing step.

4.8.2 Optimality

We now prove that the OPTIMIST method yields an optimum result under the input encoding model. That is, the reduced machine selected by OPTIMIST has a minimum-cardinality logic cover over all possible reduced machines and all possible input encodings.

In order to establish the optimality of the OPTIMIST method, we show that *any* symbolic prime cover (under the input encoding model) on any reduced machine $\mathcal{M}'$ can be formed by the instantiation of some solution to the binate constraints. Given this fact, since the cost function of our binate covering problem is the number of selected RGPI's, any reduced machine $\mathcal{M}'$ possessing a minimum-cardinality logic cover corresponds to a minimum-cost solution to the binate constraints.

OPTIMIST is optimal only within the context of the input encoding model. In particular, input encoding limits OPTIMIST's grasp of logic covers on $\mathcal{M}'$ to those which treat each symbolic next-state as a distinct function [40]. Logic covers that share product terms in the implementation of two or more next-states (e.g., as are captured by the output encoding model [42]) can not be expressed directly within the present OPTIMIST framework.[11] Nonetheless, input encoding has proven an effective model for a variety of

[10] and, obviously, with its outputs as well

[11] This might require, e.g., RGPI's whose next-state fields contain incompatible states (or groups of

practical problems.

The proof consists of several parts. First, it is shown that any symbolic prime implicant on any valid reduced machine can be formed by the instantiation of some RGPI. Next, we show that any valid state cover satisfies the binate constraints of section 4.5. Finally, we show that there exists a set of RGPI's that map onto the given prime cover, and materialize the assignment of γ_{p_j,c_i} that satisfies both the functional covering and implicant incompatibility constraints.

Note that, although the sequel makes frequent reference to a "reduced machine", nowhere does it depend on reduction (i.e., state merging) actually taking place. Thus, the original machine $\mathcal{M}$ itself qualifies as a solution to the constraints.[12]

Lemma 4.33 *Any symbolic prime implicant p' on a valid reduced machine $\mathcal{M}'$ is mapped onto by at least one RGPI, under an appropriate γ assignment.*

Proof: By construction. Given $p' = \langle IN',\ PS',\ NS',\ OUT' \rangle$ (where NS' specifies a single next-state) on $\mathcal{M}'$, we form the product $\tilde{p}$ on $\mathcal{M}$ thusly: $\tilde{p} = \langle IN,\ PS,\ NS,\ OUT \rangle$, where

$$\begin{aligned} IN &= IN' \\ OUT &= OUT' \\ NS &= \text{the unique compatible corresponding to } NS', \text{ or } \phi, \text{ if } NS' = \phi \\ PS &= \bigcup_i \{c_i \mid c_i \text{ corresponds to } s'_i \in PS'\} \end{aligned}$$

By definition, the prime implicant p' has maximal input, output, present state, and next-state fields, with respect to $\mathcal{M}'$. Clearly, $\tilde{p}$ is an RGI: it does not hit the OFF-set of any output to which it contributes (or else said OFF-set minterm would imply a mirroring OFF-set minterm in $\mathcal{M}'$), and its next-state field consists of compatible states. Now, maximally expand $\tilde{p}$ in the present-state dimension, so that any further expansion

compatible states).

[12]that is, when *all* compatibles are used; singleton compatibles are often not prime

would cause it to hit an OFF-set minterm for some output, or an ON-set minterm for some other next-state. The result, p, is an RGPI (though likely not an RGPI seed).

Finally, under the (partial) γ assignment

$$\begin{cases} \gamma_{p,c_i} = 1, & \forall c_i \text{ such that } c_i \subseteq PS(\tilde{p}) \\ \gamma_{p,c_i} = 0, & \text{otherwise} \end{cases}$$

p maps onto p'.$\square$

Lemma 4.34 *Any reduced machine $\mathcal{M}'$ and any prime symbolic cover Π' for it represent a solution to the binate constraints of section 4.5.*

Proof: The state cover corresponding to $\mathcal{M}'$ satisfies the constraints of part 1. The constraints of part 2 enumerate the ON-set minterms m' in each row in $\mathcal{M}'$. Each of these minterms is covered by some member p' of Π, and is also represented by some constraint in part 2. Using the construction of Lemma 4.33, construct RGPI p from implicant p'. Under the assignment $\gamma_{p,c_m} = 1$, the part 2 constraint is clearly satisfied. Whatever state mapping is in effect in that total state in $\mathcal{M}'$, the corresponding next-state must also be a state of $\mathcal{M}'$ (or else $\mathcal{M}'$ is not a valid reduction of $\mathcal{M}$). Now, given that the ON-set minterm m' of this next-state (say s'_j) is covered by p', p contributes to the corresponding compatible (say c_j). Thus, the only means of satisfying the part 3 implication constraint ($p \rightarrow c_j$) is by selecting c_j. Likewise, given the selection of γ_{p,c_m}, the part 5 constraint ($\gamma_{p,c_m} \rightarrow p$) must be satisfied by selecting p, so that the RGPI p does in fact appear in the logic cover. Finally, for $\mathcal{M}'$ to be a valid reduction of $\mathcal{M}$, it must be consistently state mapped. That is, for each total state τ' in $\mathcal{M}'$, there is a well-defined next-state. All members p' of Π' that contain τ' and implement next-state must agree with the chosen next-state. Clearly, then, the RGPI's p that are constructed from these p' will not disagree on next-state either, and hence the implicant incompatibility constraints of part 4 are satisfied. □

Theorem 4.35 *Any minimum-cost solution to the binate constraints of section 4.5 identifies a minimum-cardinality symbolic logic cover of $\mathcal{M}'$ under the input encoding model.*

Proof: By contradiction. Suppose there is a minimum-cardinality cover Π' which is lower than that of any solution to the constraints. Then either there is no corresponding RGPI cover Π (impossible given Lemma 4.34), or the solution has greater cardinality (also impossible, given that Π is derived from Π', one-for-one, and that the cost of any solution to the binate constraints is the cardinality of the set of selected RGPI's). □

4.9 Efficient RGPI Generation

This section first presents an algorithm for RGPI seed generation that is more efficient than the one described earlier. The algorithm is then generalized to produce the complete set of RGPI's.

In essence, both of these new algorithms recast the task of RGPI generation as an instance of ordinary prime implicant generation over an expanded domain (i.e., with additional variables). As a result, existing highly-optimized tools can be used to generate RGPI's much more efficiently.

4.9.1 Efficient RGPI Seed Generation

This section describes an efficient algorithm for RGPI seed generation that vastly outperforms the k-cube algorithm of Section 4.4. The technique used here is similar to that used for GPI generation [1], but is significantly different, owing to the unique nature of RGPI's.

The algorithm proceeds roughly as follows. First, it transforms the original flow table into an mvi function $\mathcal{C}$, whose prime implicants can be trivially transformed to the flow table's RGPI seeds. Next, a standard prime implicant generation algorithm, such as that provided by ESPRESSO-EXACT [118], is applied to $\mathcal{C}$. The prime implicants of $\mathcal{C}$ then undergo a simple transformation to recover the RGPI seeds.

The method described below deviates from that of GPI generation in two respects. First, recall that whereas GPI's can contribute to *any combination of* next-states[13], RGPI's can contribute only to *compatible* next-states. Thus, products are added to the specification of $\mathcal{C}$ to indicate the incompatibility of certain states. Second, $\mathcal{C}$ is represented as a characteristic function, for reasons described below.

Transformation of the Next-State Field

The following describes a simple transformation invented by Devadas et al. [1], which ensures that the next-state field behaves as needed during prime implicant generation.

If the next-state field is 1-hot encoded as usual for input encoding, it will not behave as needed during prime implicant generation. In particular, since each next-state is treated by input encoding as a distinct function, the ON-set minterms of any given next-state function are in the OFF-sets of every other next-state. Hence, no product can contribute to more than one next-state function. However, GPI's (and RGPI's) can by definition contribute to *one or more* states. Thus, the straightforward application of prime implicant generation to the standard mvi function used by input encoding would fail to generate RGPI's.

The solution is to represent the next-state field in positional-cube notation and then negate it bit-wise (a so-called "1-cold" encoding). Under this negation, a product spanning several total states finds in each total state an OFF-set point for the corresponding specified next-state. Hence, it can contribute to *all but* the set of specified

[13] because the relationship of the state codes is derived by a later step

next-states. After reversing the negation, the desired product, which contributes to all specified states, is retrieved.

A Characteristic Function for RGPI Seed Generation

We now describe in detail the form of the function $\mathcal{C}$ passed to the prime implicant generator.

$\mathcal{C}$ takes the form of a *characteristic function*[14] that captures $\mathcal{M}$'s functional behaviour, as well as its state incompatibility relation. Characteristic functions have proven to be a powerful means for representing and manipulating sets, particularly when implemented using ordered binary decision diagrams (OBDD's). [16].

Here, a characteristic function is *necessary* because the validity of merging a given pair of implicants is predicated on a non-local condition: the compatibility of the respective next-states. This condition cannot be detected by inspecting the implicants themselves.

The characteristic function $\mathcal{C}$ has two sets of products in its description:

1. Products describing the functional behaviour of $\mathcal{M}$
   (a) Products describing the ON-set of the outputs and next-state
   (b) Products describing the OFF-set of the outputs and next-state
2. Products asserting the incompatibility of various state pairs

The following sections describe each of these sets of products in turn.

Functional Specification of $\mathcal{M}$

First, each product in the specification of $\mathcal{M}$ (e.g. in cube-table format) is transformed as follows:

[14] A Boolean function C is said to be a characteristic function for the set $\mathcal{S}$ over domain $\mathcal{D}$ when, $\forall x \in \mathcal{D}$, $\mathcal{C}(x) = 1$ iff $x \in \mathcal{S}$.

$$\langle IN\ PS\ NS\ OUT \rangle \ \Rightarrow\ \langle IN\ PS\ \overline{NS}\ OUT\ 1 \rangle$$

In other words, the value of $\mathcal{C}$ is 1 for all products in the expanded domain consistent with the ON-set of the specified behaviour of $\mathcal{M}$.

Next, for each product in the specification of $\mathcal{M}$, a product is added for each output having a 0 value. The added products specify that any product intersecting that portion of the input space and implementing the given output hits the OFF-set of that output, and hence is in the characteristic function's OFF-set:

$$\langle IN\ PS\ NS\ OUT \rangle \ \Rightarrow\ \langle IN\ PS\ *\ OUT_{off}\ 0 \rangle$$

where OUT_{off} is 1 for some output having a 0 value, and 0 for all other outputs. Note that the don't-care entry for next-state (shown as a $*$) is in fact a field of 1's, the representation of a full literal for an mvi variable in positional-cube notation.

A similar set of products is added to constrain the next-state field. These products are somewhat different, however, because RGPI's are allowed to contribute to two or more compatible next-states. As a result, valid RGPI's frequently span total states for which the specified next-state s_i is a *subset* of the RGPI's next-state field NS. The RGPI thus appears to hit the OFF-set of the other next-states (those in the set $NS - \{s_i\}$).

The proper condition for the next-state field is as follows: a product is invalid if it contains a given total state, and contributes to next-state, but not to the *specified* next-state. In effect, this would violate the standard state closure requirement. Hence, the added products have the form:

$$\langle IN\ PS\ NS\ OUT \rangle \ \Rightarrow\ \langle IN\ PS\ NS_{on}\ *\ 0 \rangle$$

where NS_{on} is 1 for the specified next-state, 0 for some other next-state and 1 for all others.

State Incompatibility

The second set of products consists of one product for each incompatible state pair, of the form:

$$\langle * \; * \; NS_{inc} \; * \; 0 \rangle$$

where NS_{inc} has 0's in the bit positions corresponding to the incompatible pair of states, and $-$'s elsewhere (respecting the positional-cube notation for NS). Thus, any product contributing to these incompatible states[15] is in the OFF-set of the characteristic function, and does not belong to the set of RGPI seeds.

Example 2 Figure 4.6 shows a cube-table format specification of an FSM taken from [1], along with the corresponding characteristic function used for RGPI seed generation. Those products corresponding directly to products in the original specification are shown in bold. The sole incompatible state pair $\{\,\{s_0, s_2\}\,\}$ is represented by the final OFF-set product in the characteristic function's description.

Standard efficient techniques for the generation of prime implicants (e.g. ESPRESSO-EXACT [118]) are then applied to the characteristic function.

Recovering the RGPI Seeds from $\mathcal{C}$

To first order, the RGPI seeds of $\mathcal{M}$ are trivially retrieved from the prime implicants of $\mathcal{C}$ by reversing the transformation applied in step one. In particular, for each prime implicant of $\mathcal{C}$, the characteristic function output (the trailing bit) is dropped, and the next-state field is un-negated.

[15] after re-interpreting the negation of the next-state field

$\mathcal{M}$	0	1
s_0	$s_0, 0$	$s_2, 0$
s_1	$s_1, 0$	$s_1, -$
s_2	$s_1, -$	$s_0, 1$

0	s_0	s_0	0
1	s_0	s_2	0
0	s_1	s_1	0
1	s_1	s_1	-
0	s_2	s_1	-
1	s_2	s_0	1

0	**100**	**011**	**0**	**1**
0	100	111	1	0
0	100	101	*	0
0	100	110	*	0
1	**100**	**110**	**0**	**1**
1	100	111	1	0
1	100	011	*	0
1	100	101	*	0
0	**010**	**101**	**0**	**1**
0	010	111	1	0
0	010	011	*	0
0	010	110	*	0
1	**010**	**101**	**-**	**1**
1	010	011	*	0
1	010	110	*	0
0	**001**	**101**	**-**	**1**
0	001	011	*	0
0	001	110	*	0
1	**001**	**011**	**1**	**1**
1	001	101	*	0
1	001	110	*	0
*	*	010	*	0

Figure 4.6: Cube-table specification and corresponding characteristic function used in fast RGPI seed generation

However, two issues must be dealt with in order to arrive at the complete set of RGPI seeds. First, the primes that are generated always contribute to the minimum possible number of next-states, namely, the set of next-states specified in all total states that the prime contains. As a result, RGPI seeds which (validly) contribute to a proper superset of that set of states are not present. These are trivially generated by an "elaboration" post-processing step.

Second, it may occur that a prime implicant of the characteristic function is don't-care for a given output. This happens when the corresponding region of the table consisted solely of don't-care points for that output. In this case, an RGPI seed with either a 0 or 1 for that output would be consistent with the original flow table. Given the choice, the post-processing step arbitrarily chooses 1.

4.9.2 Efficient RGPI Generation

We now generalize the method of the previous section to generate *all* RGPI's.

The essential difference between RGPI seeds and RGPI's is that RGPI's with different input fields are considered *distinct*. Therefore, the primes of the characteristic function that is used must likewise maintain this distinction. The distinction is accomplished by augmenting the characteristic function with pseudo-outputs that mirror the input field, analogous to a technique employed by Devadas et al. [1] for fast generation of GPI's for state encoding.[16]

4.10 Experimental Results

We now present the results of a number of experimental trials pitting OPTIMIST against the de facto state minimization tool, STAMINA.

OPTIMIST was implemented in C++ and run under MkLinux. SCHERZO [30] was

[16] In that method the technique is actually applied to the present-state field, not the input field.

used to solve the core binate covering problem, and ESPRESSO [118] was used to generate prime implicants for the RGPI characteristic function as described in section 4.9.

Results for a number of FSM's, including several from the MCNC '91 benchmark suite, appear in table 4.1, using only prime compatibles. For each FSM, the number of inputs, unminimized states, outputs and prime compatible are shown. Two sets of OPTIMIST runs were done; one using only RGPI seeds, and another using all RGPI's. For each run, the number of RGPI's used, the number of covering constraints, the run-time, number of symbolic products after mvi minimization, and number of minimized states is given. For comparison, STAMINA [62] was used to minimize the same tables.

Table 4.1 shows the results when using only prime compatibles. OPTIMIST breaks even with STAMINA in product count in 7 of the 11 examples. In two examples, OPTIMIST's solution has a better product count than STAMINA. An outstanding result for the two FSM's `bbsse` and `sse` shows the power of optimal state assignment coupled with optimal state mapping, with an *85% reduction in product count.*

Frequently, OPTIMIST chooses a larger set of reduced states than does STAMINA, even when no gain in product cardinality results. E.g., in the case of `bbara`, OPTIMIST's state cover has 7 states compared to STAMINA's 4. This occurs because OPTIMIST uses a single-tiered cost function which does not incorporate state cover cardinality. This problem would be solved by using a binate solver capable of handling two-tiered cost functions.

Comparing the use of RGPI seeds versus all RGPI's, in no case did using all RGPI's produce any benefit. On the other hand, for these trials, run-time complexity was not increased appreciably either. In fact, generally the numbers of RGPI's and constraints was relatively close for most runs. This may suggest that the complexity of these FSMs was largely dominated by their next-state, rather than their output behaviour.

Table 4.2 shows a parallel set of runs, this time using all compatibles. STAMINA results are identical to those shown in table 4.1, and are repeated only for convenience.

FSM			OPTIMIST										STAMINA	
			SEEDS					ALL RGPI's						
name	i/s/o	pc	RGPI	cons	time	**prod**	**ms**	RGPI	cons	time	**prod**	**ms**	**prod**	**ms**
minst	1/2/3	2	13	54	<1	6	2	16	66	<1	6	2	6	2
lion9	2/3/1	5	79	227	2	8	4	81	243	3	8	4	8	2
lisom	1/5/1	5	28	130	<1	4	5	28	130	<1	4	5	4	2
bbara	4/10/2	7	75	358	3	25	7	160	501	11	25	7	25	4
bbsse	7/16/7	13	66	3704	192	31†	13	85	3788	230	31†	13	208	13
beecount	3/7/4	7	145	979	1240	10	7	170	1038	10	10	7	10	3
opus	5/10/6	9	33	1320	8	19	9	33	1320	8	19	9	19	5
s27	4/6/1	5	25	304	1	17	5	43	421	1	17	5	17	4
s8	4/5/1	1	207	245	76	1	1	308	374	110	1	1	4	1
sse	7/16/7	13	66	3704	176	31	13	85	3788	246	31	13	208	13
train11	2/11/1	17	318	3802	3318	6	11	‡	-	-	-		7	2

† Sub-optimal results due to necessity of using heuristic mode in binate solver

‡ Failed to produce a solution in a reasonable amount of time

Table 4.1: Results of minimization by OPTIMIST using prime compatibles

Clearly the run-time complexity of this set of trials is much higher than the previous set. As a result, three examples that were solvable using prime compatibles are not solvable by OPTIMIST in a reasonable amount of time when using all compatibles. In most cases, the bottleneck was in the binate solver. In only two cases did resorting to the binate solver's heuristic mode help. Interestingly, considerable variation appears to be present in the complexity of the binate covering problem. For example, Scherzo readily solves one problem having over one million constraints (that of `sse`), and yet has problems with far fewer constraints in other cases.

4.11 Conclusions and Future Work

This chapter has presented the first method for optimal state minimization which directly and accurately targets logic complexity, achieving a provably exact optimum result under input encoding. The method is computationally expensive, however, and would benefit greatly from heuristic and inexact variations. Unlike other methods, OPTIMIST provides several opportunities to reduce complexity while retaining the strong minimization framework, e.g. by using prime or maximal compatibles, RGPI seeds only, or by employing heuristic binate solvers. For more efficient exact solution, recent innovations in implicit methods [31, 70] hold particular promise. With all of these choices, this method offers a framework encompassing a spectrum of solutions.

A limitation to the current work is the large size of the RGPI set. Initial investigation suggests cases where the set can be pruned, while still guaranteeing an optimum solution. Similarly, it appears possible that compatibles may be pruned as well. These are important areas for further research.

Although this work presents only limited benchmarks, future experimentation should prove the method competitive with existing methods (e.g. STAMINA, SMAS) by offering improved results in exchange for longer run-times. Finally, it is anticipated that an extension to output encoding will yield a definitive solution to the problem of optimal

FSM			**OPTIMIST**										**STAMINA**	
			SEEDS					ALL RGPI's						
name	i/s/o	ac	RGPI	cons	time	**prod**	**ms**	RGPI	cons	time	**prod**	**ms**	**prod**	**ms**
minst	1/2/3	4	30	223	<1	5	3	37	265	3	5	3	6	2
lion9	2/3/1	20	263	14829	66	8	16	271	14959	76	8	18	8	2
lisom	1/5/1	5	28	130	<1	4	5	28	130	<1	4	5	4	2
bbara	4/10/2	21	250	2213	17	26†	19	535	3884	42	36†	21	25	4
bbsse	7/16/7	97	387	46095	‡	-	-	1949	311300	‡	-	-	30	7
beecount	3/7/4	11	204	4436	1240	14†	11	250	6229	28	14†	11	10	3
opus	5/10/6	11	66	2055	8	19	11	220	3863	12	19	11	19	5
s27	4/6/1	7	46	612	1	17	7	117	1312	4	17	7	17	4
s8	4/5/1	31	1647	1431676	76	1	1	2204	1732495	‡	-	-	4	1
sse	7/16/7	97	387	46095	‡	-	-	‡	-	-	-	-	208	13
train11	2/11/1	17	‡	-	-	-	-	‡	-	-	-	-	7	2

† Sub-optimal results due to necessity of using heuristic mode in binate solver

‡ Failed to produce a solution in a reasonable amount of time

Table 4.2: Results of minimization by OPTIMIST using all compatibles

state minimization.

Chapter 5

State Minimization for Exactly Optimum Two-Level Output Logic

In this chapter, the optimal state minimization method of the previous chapter is extended to encompass *output-targetted state minimization* for synchronous FSM's. In essence, the method of this chapter narrows OPTIMIST's focus to the complexity of the output logic, in order to obtain stronger results. Useful in its own right, this new state minimization method will further serve as a foundation for a similar method for asynchronous burst-mode machines in the next chapter.

This chapter makes two fundamental contributions. First, it presents a variation of OPTIMIST, OPTIMISTO, that is the first algorithm to produce *exactly minimum* cardinality two-level output logic over *all possible state minimizations, state encodings, and logic minimizations.* The second contribution is an interesting new theoretical result: the unminimized machine always possesses a minimum-cardinality two-level output logic implementation.

In the context of these two results, this chapter offers two choices: that of performing no reduction at all (useful if output logic is the sole interest), or using the constraint-satisfaction method to reduce next-state complexity (by performing state reduction), while guaranteeing exactly minimum-cardinality output logic. Last, it briefly explores the opportunities presented by the use of the output-targetted framework with alternative cost functions.

5.1 Introduction

OPTIMIST was a first attempt to address the global optimization issue broadly for technology-independent logic synthesis, by attacking the problem of *optimal state minimization* for incompletely specified FSM's. The problem was defined as selecting the reduced machine that has the best two-level logic implementation over *all possible state minimizations and input encodings.* To this end, OPTIMIST breaks with traditional approaches by performing state minimization and state encoding concurrently, using a powerful new form of symbolic logic minimization. However, it is an exact solution only under an input encoding model.

This chapter thus makes two significant contributions to the body of state minimization research.

First, it presents a new synthesis method: OPTIMISTO, an extension to the basic OPTIMIST approach, which targets output-logic. The method produces two-level output logic having exactly minimum cardinality over *all* possible state minimizations, state encodings, and logic minimizations. Unlike the original OPTIMIST algorithm, which is exact *only* under the input encoding model [40], this new method is the first exact solution to the optimal state minimization problem that is *independent* of the encoding model. Additionally, it has significantly less computational complexity than OPTIMIST.

Second, this chapter offers a novel theoretical result: the unminimized machine itself possesses exactly minimum-cardinality two-level output logic, over all possible minimizations and encodings. In other words, state minimization can never reduce two-level output logic cardinality. That is, the best cardinality solution for output logic can always be obtained on the unminimized state machine.

In this context, we propose two choices with regard to optimal state minimization. The first is to perform *no state reduction at all*, which is useful when output logic cardinality is of paramount importance. In this case, output logic is exactly optimum, but no attempt is made to minimize next-state complexity via state reduction. The second alternative is to use OPTIMISTO's binate constraint satisfaction method. The advantage of this method is that it can simultaneously guarantee exactly minimum-cardinality two-level output logic while reducing next-state complexity by performing state minimization. In fact, the latter choice can support a variety of cost functions, while the unminimized machine is exactly optimum with respect to only one cost function (output logic cardinality).

Although output-targetted minimization may be of somewhat limited utility in some synchronous applications, it serves as a foundation for the method of Chapter 6 for burst-mode asynchronous machines. In that context, it optimizes the key parameter determining asynchronous system performance — output latency.

The organization of the chapter is as follows. First, Section 5.2 presents the binate constraint satisfaction framework for output-targetted state minimization, in detail. The method is illustrated in full detail with a simple but interesting example in Section 5.3. Next, Section 5.4 demonstrates the method's soundness and optimality, and also proves the novel theoretical result regarding the unminimized machine. Then, a discussion of various cost functions and their interpretation within the constraint satisfaction framework appears in Section 5.5. Experimental results are presented in Section 5.6, in which the constraint satisfaction method is compared to STAMINA, the de facto state minimization CAD tool. Finally, some concluding remarks are made in Section 5.7.

5.2 Output-Targetted State Minimization for Synchronous FSM's

We now present an enhancement to the basic method of the previous chapter, extending the core theoretical framework to output-targetted state minimization for synchronous FSM's. This represents the first truly exact result for optimal state minimization that is independent of an encoding model.

This chapter, like its predecessors, restricts attention to the common subclass of NDFSM's known as incompletely-specified FSM's (ISFSM's) where, for each total state $\langle i, s\rangle$, the next-state $\delta(i, s)$ is either a singleton state or else is completely unspecified (denoted $\delta(i, s) = \mathcal{S}$). Likewise, it is assumed that the output $\lambda(i, s)$ is either a single value or is unspecified.

5.2.1 Overview of Problem Formulation

The input encoding model provides a framework for symbolic logic minimization that we apply in a novel way to the output logic of an ISFSM. In particular, our new method captures both state reduction and two-level output logic minimization.

Figure 5.1 illustrates the abstract flow of the approach. We start with the ISFSM (a), and extract from it the symbolic function representing its output behaviour (b), by discarding the next-state function. The problem is then modeled as a search, through the space of all valid reduced machines, for a machine having an exactly minimum-cardinality *symbolic* cover for the output functions. Having found such a machine (c), a straightforward input encoding problem now exists for the outputs, as described in Section 2.4.[1] One can thus derive an encoding, and instantiate the symbolic cover, resulting in a binary cover for the outputs of exactly minimum cardinality over all possible

[1]In fact, we already have the symbolic logic cover, so that symbolic logic minimization need not be performed again.

$\mathcal{M}$	0	1
s_0	$s_0, 0$	$s_2, 0$
s_1	$s_1, 0$	$s_1, -$
s_2	$s_1, -$	$s_0, 1$

(a) An ISFSM $\mathcal{M}$

$\lambda(\mathcal{M})$	0	1
s_0	0	0
s_1	0	$-$
s_2	$-$	1

(b) Output behaviour of $\mathcal{M}$

$\lambda(\mathcal{M}')$	0	1
$s'_0 = \{s_0, s_1\}$	0	0
$s'_1 = \{s_1, s_2\}$	0	1

(c) Output behaviour of $\mathcal{M}'$

Figure 5.1: An ISFSM, its output behaviour, and a reduced machine

state reductions and encodings.

With the above as an intuitive framework, our method proceeds as follows. First, it strips the ISFSM down to its output behaviour. It then subjects the symbolic function (b) to a form of mvi logic minimization over the *reduced* machine, driven by the construction of a reduced machine that satisfies both classic state covering and state closure constraints. In essence, it constructs the reduced machine a row at a time, forming portions of the logic cover as it goes. The goal is thus a reduced machine having a minimum-cost symbolic output logic cover *over all possible state reductions.* The result is a valid reduced machine, along with a globally minimum-cardinality output cover.

5.2.2 Overview of Method

We now give an overview of the output-targetted method, OPTIMISTO, which shares the OPTIMIST framework described in Chapter 4.

First, two sets of covering objects are generated from the unreduced machine $\mathcal{M}$. In particular, an ordinary set of *state compatibles* is generated, according to the classic compatibility relation [60]. Second, unlike the RGPI's of OPTIMIST, here an ordinary set of *symbolic prime implicants* is generated, *for the outputs only*; *no next-state implicants are generated.* Like the original method, these primes are formed *on the unreduced machine.* These symbolic primes are actually used to cover output ON-set minterms in various *reduced* machine states. This is made possible by a strong relationship (established in Section 5.4) between symbolic implicants on the unreduced and reduced machines.

Next, binate covering constraints are generated, using the given sets of state compatibles and symbolic primes. The covering constraints fall into two categories: (i) classic state reduction requirements (*identical* to those in [60]), and (ii) new logic covering constraints for the outputs of the reduced machine. As a result, state minimization and symbolic logic minimization are performed simultaneously. OPTIMISTO's constraints are far simpler than those used by OPTIMIST.

The constraints are solved using a binate solver (such as Scherzo [30]), under a cost model that targets logic cardinality (identical to that of OPTIMIST). The result is a closed state cover and a selection of symbolic primes having minimum cardinality over all valid state reductions.

Using the selected state compatibles and output primes, a symbolic cover for the reduced machine is formed by a straightforward instantiation process. This process maps the selected output primes one-for-one onto a set of symbolic implicants on the reduced machine. The result constitutes a logic cover for the outputs of the reduced machine.

At this point, the method is complete. The reduced machine, $\mathcal{M}'$, can then be passed to a state encoding tool, such as KISS or NOVA, to produce a binary implementation for the reduced machine. Alternatively, input encoding constraints can be generated directly from the instantiated symbolic cover, and an encoding generated, with the same result.

The proposed method is significantly simpler than the original method. Here, the next-state is considered only insofar as is necessary to ensure a correctly reduced FSM. Therefore, only classic state minimization constraints are required. Specifically, since a closed state cover guarantees the existence of a correct next-state implementation, the method needs only to ensure that a normal closed state cover is found. Notably, this focus allows us to ignore the issue of optimal state mapping, which is the source of considerable complexity in the basic OPTIMIST method.

Because the new method borrows extensively from the original OPTIMIST framework, in the sequel we sketch only the requisite changes to those steps which need them: symbolic prime generation and constraint generation.

5.2.3 Symbolic Primes

The symbolic implicants used by the output-targetted method are greatly simplified from the RGPI's used by the base OPTIMIST algorithm. In particular, a *normal set of symbolic prime implicants* is generated on the unreduced machine. Specifically, we generate prime implicants that implement *only* outputs (*output PI's*). No implicants are generated for next-state.

Thus we can use normal synchronous CAD tools, e.g., ESPRESSO [118], to generate all necessary symbolic prime implicants from the unreduced machine $\mathcal{M}$. Additionally, unlike the original method, no smaller cubes need to be generated; ordinary primes suffice. However, just as in OPTIMIST, symbolic implicants are generated once on $\mathcal{M}$ but can be used to cover *all possible* reduced machines.

The restriction to outputs and the exclusive use of ordinary primes together drastically reduce the number of symbolic primes from that required by the base method.

5.2.4 Binate Constraints

This section describes the binate constraints used for output-targetted state minimization, a much simpler set than that used by the base OPTIMIST method. In fact, where the original method has 5 constraint sets (state covering, functional covering, closure, implicant incompatibility, and implicant cost), the output-targetted method has only 3 constraint sets. Of these, two sets are exactly the classic state minimization constraints [60]. Beyond those, only functional covering constraints are required.

Constraint Variables

The variables in the covering problem and their assigned costs are:

variable	description	cost
c_i	select compatible c_i for state cover	0
p_i	select symbolic prime p_i for symbolic logic cover	1

As mentioned in Section 5.2.3, prime implicants are sufficient to guarantee optimality; no smaller cubes are necessary. Hence, the γ variables of the base method (which restrict the present-state extent of instantiated implicants) are likewise eliminated.

Constraint Clauses

The following paragraphs presents in detail the three sets of constraints needed for output-targetted state minimization.

1. *State Covering*

   Each unreduced state must be covered by a selected state compatible. This is precisely the classic state covering constraint of Grasselli [60].

   $\forall$ states s of $\mathcal{M}$

   $$c_{i_1} + c_{i_2} + \cdots + c_{i_N}$$

   where $s \in c_{i_k}$.

   For example, in the table of Section 5.3, state s_0 can only be covered by c_0; hence the state covering constraint for s_0 is simply (c_0). State s_1, on the other hand, is contained in two compatibles, namely, c_1 and c_2. Hence, the corresponding constraint is $(c_1 + c_2)$.

2. *Functional Covering*

   Each output ON-set minterm of $\mathcal{M}'$ must be covered by a selected output prime implicant. The ON-set minterms of $\mathcal{M}'$ in a given reduced state are identified and covered, once that reduced state (i.e. compatible) is selected.

$\forall$ compatibles c

$\prod_{m' \in c} \overline{c} + p_1 + p_2 + \cdots + p_N$

where p_i contains both m' and c.

Here, m' refers to an ON-set minterm *of some output* in some row of the reduced machine $\mathcal{M}'$. Each clause ensures that some prime implicant p_i will be selected to cover m'.

As with OPTIMIST, although the minterms m' in row c of the reduced machine are not explicitly available, they can be easily derived from the minterms of $\mathcal{M}$. Let $i \in \mathcal{I}$ be any input column. m' is then an ON-set minterm of output function λ_j in reduced machine $\mathcal{M}'$ iff, for some reduced state $s \in c$, the output $\lambda_j = 1$.

Example: For instance, in the example of Section 5.3, upon selecting compatible $c_1 = \{s_1, s_2\}$, the output ON-set minterm in column 01 deriving from the minterm in s_2 must be covered. Note that output prime p_5 does not span all of c_1, and is thus not a candidate. In fact, only p_3 will suffice. The resulting covering constraint is $(\overline{c_1} + p_3)$.

On the other hand, consider selecting compatible $c_0 = \{s_0\}$. This necessitates covering the minterm in column 00 deriving from the ON-set minterm in s_0. Only output prime p_1 spans c_0 and column 00; hence it is the sole candidate. The resulting constraint is $(\overline{c_0} + p_1)$.□

3. *State Closure*

The selected set of compatibles must be closed. This is the classic state closure constraint due to Grasselli [60].

$\forall$ compatibles c

$\prod_{c' \in P(c)} \overline{c} + c_{i_1} + c_{i_2} + \cdots + c_{i_N}$

where $P(c)$ is the implied set of c, c' is a member of that set, and $c_{i_j} \supseteq c'$.

Note that, for the example of Section 5.3, there are no non-trivial implications, and hence, no closure constraints.

We now qualitatively sketch the differences between the original OPTIMIST constraints and the constraints for the proposed output-targetted method.

Clearly, state covering requirements do not change. Functional covering constraints, however, are simplified in two ways. First, constraints for covering output minterms use implicants (p_i variables) directly, whereas the base OPTIMIST constraints used γ variables. Second, functional covering constraints for next-state are eliminated. In their place, we substitute an ordinary set of state closure constraints [60], to ensure the existence of a valid next-state implementation. Finally, both implicant incompatibility and implicant cost constraints can be removed: the former set ensured a consistent state mapping; the latter simplified cost function computation. So, we are left with 3 constraint sets: state covering, functional covering, and state closure.

5.2.5 Constraint Solution

As in OPTIMIST, the above constraints are all unate and binate. Hence, an efficient general-purpose binate solver (Coudert's Scherzo [30]) is used to solve these constraints. The result is a set of selected state compatibles, and a set of selected symbolic output implicants.

5.2.6 Symbolic Implicant Instantiation

Given a solution to the binate constraints, the selected output implicants are instantiated, thereby producing an output cover for the reduced FSM $\mathcal{M}'$. The instantiation process here is much simpler than that in base OPTIMIST, since there are no γ variables. Instead, we use a trivial mapping which we call the "Natural Mapping".

We illustrate the Natural Mapping with a simple example.

Example 5.1 Consider the instantiation of the output prime implicant $p_0 = \langle 00\, s_0, s_2 \rangle$ from the table of Section 5.3, under the closed state cover $\{\, s_0' = \{s_0\},\ s_1' = \{s_1, s_2\}\,\}$. The present state of p_0' consists of those reduced states that correspond to selected compatibles that are completely contained in $PS(p_0) = \{s_0, s_2\}$, namely, $\{s_0'\}$. Thus, $p_0' = \langle 00\, s_0' \rangle$.

Roughly, the Natural Mapping works as follows. Each selected output prime implicant p is mapped, one-for-one, onto a corresponding output implicant p' of $\mathcal{M}'$. The input and output fields of the instantiated implicant p' are identical to that of p. The present state field of p', however, is filled with the states of $\mathcal{M}'$ corresponding to those selected compatibles that are contained by $PS(p)$.

More formally, we define the *Natural Mapping* of an output product $p = \langle I\ PS\ O \rangle$ on the unreduced machine $\mathcal{M}$ onto a reduced machine $\mathcal{M}'$ to be the product $p' = \langle I\ PS'\ O \rangle$, where $PS' = \{s_i', \forall s_i'$ such that $c_i \subseteq PS\}$.

The Natural Mapping ensures that all instantiated products are in fact implicants of the reduced machine $\mathcal{M}'$. In essence, it excludes from the instantiated present-state of p' any reduced state that contains OFF-set minterms for some output that p' implements. In the above example, s_1' is excluded from $PS(p_0')$, because the corresponding compatible, $\{s_1, s_2\}$, is not contained in $PS(p_0) = \{s_0, s_2\}$. More specifically, the OFF-set minterm at $\langle 00\, s_1 \rangle$, which prevents the expansion of p_0 into s_1, produces an OFF-set minterm in the reduced machine $\mathcal{M}'$ at $\langle 00\, s_1' \rangle$. Thus, because p_0 cannot be made to span s_1, its image on $\mathcal{M}'$, p_0', cannot be allowed to span any reduced state which derives from s_1 (in this case, s_1'). In other words, were we to include s_1' in $PS(p_0')$, p_0' would no longer be an implicant.

The Natural Mapping makes use of this principle, and gives rise to the following extremely useful Lemma.

Lemma 5.2 *Any symbolic output product p which is an implicant of $\mathcal{M}$ is also an implicant of* any *corresponding reduced machine $\mathcal{M}'$ when instantiated under the Natural Mapping.*

Proof: Intuitively, the Natural Mapping assumes the primality of p, and conservatively

excludes from $PS(p')$ any reduced state suspected of containing an OFF-set minterm for some output of p' within the region of p'. In particular, since p is an implicant of $\mathcal{M}$, any state that p spans has no OFF-set minterms. Thus, if $PS(p) \supseteq c$ for some compatible c, then the reduced state s' corresponding to c does not contain an OFF-set minterm in any input column spanned by p. Thus, it is safe to include s' in $PS(p')$, and the Natural Mapping does so. In other words, p' is an implicant of $\mathcal{M}'$. □
The above Lemma allows output prime implicants that are generated on the *unreduced* machine to be used to cover ON-set minterms in the *reduced* machine.

5.2.7 Method Summary

In sum, the output-targetted method proceeds as follows. First, normal output prime implicants are generated on the unreduced machine $\mathcal{M}$. Next, binate constraints are generated, consisting of (i) classic state reduction constraints, and (ii) functional covering constraints for the outputs. The constraints are solved, producing a set of selected state compatibles and symbolic output prime implicants. Finally, the selected symbolic implicants are instantiated, resulting in an exactly minimum-cardinality symbolic logic cover for the outputs.

5.3 Example

The following simple example illustrates the operation of each step of the method. Specifically, it first lists all generated state compatibles. Next, the symbolic output prime implicants, are shown. Then, the three sets of binate constraints are given. Finally, the optimum solution and a sub-optimum solution to the constraints are shown, along with the instantiated symbolic logic implementations.

The example below demonstrates the advantage of the proposed method over standard state minimization methods, such as STAMINA. The optimum solution for this table is in fact the unminimized machine, $\mathcal{M}$, which STAMINA ignores, since STAMINA requires a minimum-cardinality *state* cover. The example also underscores our theoretical result (given in Section 5.4) that the unminimized machine *always* has a minimum-cardinality two-level logic cover.

$\mathcal{M}$	00	01	11	10
s_0	$s_2, 1$	$s_1, 0$	$s_1, -$	$s_0, 0$
s_1	$s_2, 0$	$s_1, -$	$s_1, -$	$-, -$
s_2	$s_2, -$	$s_1, 1$	$s_1, 1$	$s_0, 1$

The *prime compatibles* for $\mathcal{M}$ are $c_0 = \{s_0\}$ and $c_1 = \{s_1, s_2\}$. The *complete set of compatibles* used by our algorithm includes the remaining non-prime compatibles, namely, $c_2 = \{s_1\}$ and $c_3 = \{s_2\}$.

The *output prime implicants*, generated by ESPRESSO on the unreduced machine, are as follows: $p_1 : \langle 00\, s_0, s_2 \rangle$, $p_2 : \langle 11\, s_0, s_1, s_2 \rangle$, $p_3 : \langle -1\, s_1, s_2 \rangle$, $p_4 : \langle 1-\ s_1, s_2 \rangle$, and $p_5 : \langle --\ s_2 \rangle$.

The *constraints* generated by our algorithm are as follows:

State Covering: *(one constraint per unreduced state)*

$(c_0)(c_1 + c_2)(c_2 + c_3)$

The first constraint ensures that s_0 is covered (which can only be done using c_0); the second ensures that s_1 is covered, and so on.

Functional Covering: *(one constraint per minterm per reduced state)*

$(\overline{c_0}+p_1)(\overline{c_1}+p_3)(\overline{c_1}+p_2+p_3+p_4)(\overline{c_1}+p_4)(\overline{c_3}+p_3+p_5)(\overline{c_3}+p_2+p_3+p_4+p_5)(\overline{c_3}+p_4+p_5)$

The first constraint ensures that the output ON-set minterm in reduced total state $\langle 00\, \{s_0\} \rangle$ is covered, which can only be accomplished using p_1. The second constraint covers the ON-set minterm in total state $\langle 01\, \{s_1, s_2\} \rangle$. Note that there are no ON-set minterms in $c_2 = \{s_1\}$; hence no covering constraints are needed for that reduced state.

State Closure:

There are no non-trivial closure requirements, since the implied sets of all compatibles are empty.

Our method chooses a *minimum-cost solution* consisting of the state compatible cover $\{c_0, c_2, c_3\}$ and the prime implicant pair $\{p_1,\, p_5\}$. Interestingly, this solution performs no state merges (i.e. it chooses $\mathcal{M}$ itself), and uses non-prime compatibles.

For the original machine $\mathcal{M}$, corresponding to the selection of compatibles $\{c_0, c_1, c_2\}$, the minimum output logic cover, $\{p_1,\, p_5\}$ trivially maps onto itself under the Natural Mapping. Clearly it covers all ON-set minterms of $\mathcal{M}$.

STAMINA, on the other hand, requires prime compatibles and a minimum-cardinality *state* cover, and hence chooses the reduced machine $\mathcal{M}'$ below, corresponding to the state cover $\{c_0,\, c_1\}$. This choice requires the more expensive output cover with the 3 products $\{p_1,\, p_3,\, p_4\}$.□

$\mathcal{M}'$	00	01	11	10
s'_0	$s'_1, 1$	$s'_1, 0$	$s'_1, -$	$s'_0, 0$
s'_1	$s'_1, 0$	$s'_1, 1$	$s'_1, 1$	$s'_0, 1$

In summary, our method generates the best solution, which has 2 output implicants and 3 states, while STAMINA produces a solution having 3 output implicants and 2 states. Notably, our solution uses no state reduction at all, and further, uses non-prime compatibles. Neither of these two options is available to STAMINA, demonstrating the power of our method.

5.4 Theoretical Results

This section demonstrates the soundness and optimality of our output-targetted optimal state minimization method. First, it shows that any solution to the above binate constraints yields a valid reduced machine and a valid output logic cover. Then, it shows that our method yields a (possibly) reduced machine with minimum output logic cardinality over all possible state minimizations and encodings.

In the process, we prove an interesting and useful theoretical result in Section 5.4.2. A minimum-cardinality output logic cover for the unreduced machine itself has minimum cardinality *across all possible state minimizations and encodings.* A consequence of this is that state reduction can *never* reduce output logic complexity. (In fact, state reduction can actually *increase* output logic complexity, as Hartmanis et al. observed [64].)

Although the unminimized machine always possesses minimum-cardinality output logic, and requires no computation to derive, it is not always the ideal choice. In particular, it is not necessarily optimum under cost functions other than logic cardinality. For example, it is well-known that state reduction often helps simplify the next-state logic.[2] Thus, an optimum solution under a cost function incorporating, e.g., both output and next-state logic complexity, may require state reduction.

With these observations in mind, the constraint-satisfaction framework presented in Section 5.2 offers a powerful alternative method that is capable of addressing several cost functions. In particular, it is capable of reducing next-state logic complexity, while guaranteeing exactly minimum-cardinality two-level output logic. As such, several alternative cost functions are discussed in more detail in Section 5.5.

5.4.1 Correctness of the Constraint-Satisfaction Method

The soundness of our procedure is demonstrated in several steps. First, we show that any solution to the binate constraints of Section 5.2.4 yields a valid reduced machine. Next, we establish a relationship between the minterms of $\mathcal{M}$ and the minterms of $\mathcal{M}'$, and use it to relate functional covering on $\mathcal{M}$ to that on $\mathcal{M}'$. Finally, we prove that a solution to the binate constraints also produces a valid logic cover for the outputs.

Theorem 5.3 *Any solution to the binate constraints of Section 5.2.4 identifies a valid reduced machine* $\mathcal{M}'$.

[2] This is of course the classic motivation for state minimization.

Proof: Obvious, since the constraints of parts 1 and 3 are precisely the constraints of Grasselli [60]. Since any solution to the constraints satisfies every constraint, the selected compatibles form a closed state cover. □

Lemma 5.4 *Each ON-set or OFF-set output minterm m' of reduced machine $\mathcal{M}'$ is the image of one or more output minterms m of $\mathcal{M}$.*

Proof: Each ON-set or OFF-set minterm m' of $\mathcal{M}'$ resides in some reduced state s', and is the result of merging one or more minterms $\{m\}$ in the unreduced states that merged to produce s'. Thus, if m' is in the ON-set of an output of $\mathcal{M}'$, then one of $\{m\}$ must also have been an ON-set minterm. □

Lemma 5.5 *If output implicant p on $\mathcal{M}$ appears in a functional covering constraint for an output minterm m' of $\mathcal{M}'$, it covers m' when instantiated under the Natural Mapping.*

Proof: p appears in the given part 2 constraint iff its present-state field contains the compatible (row) c in which m' resides. Now, m' only exists once c is selected. Hence, c's selection implies that p's present-state field will be instantiated so as to span the reduced row corresponding to c. □

Theorem 5.6 *Any solution to the binate constraints of Section 5.2.4 identifies a valid output logic cover for the reduced machine $\mathcal{M}'$.*

Proof: This follows from the above two Lemmas, and from the fact that every output minterm in a reduced row has a covering constraint in part 2 that must be satisfied once the corresponding row has been selected. Also, note that because the set of implicants used is the set of output prime implicants, every minterm m in $\mathcal{M}$ is covered by some p. As a result, every functional covering constraint has at least one prime appearing on the right-hand side, and is thus satisfiable. □

5.4.2 Optimality of the Unminimized Machine

This section demonstrates that the unminimized machine $\mathcal{M}$ itself has the best possible two-level output logic cover, over all possible state minimizations and encodings. The burden of the proof lies in finding for every prime cover of any given reduced machine $\mathcal{M}'$, a corresponding cover for $\mathcal{M}$ of identical cardinality that maps onto it.

Lemma 5.7 *To every output implicant p' on some reduced machine $\mathcal{M}'$ there corresponds at least one implicant p on $\mathcal{M}$ which Naturally Maps onto p'.*

Proof: By construction. For $p' = \langle I\ PS'\ O\rangle$, construct the product $p = \langle I\ PS\ O\rangle$, where PS is the union of all the compatibles corresponding to states s' in PS'. Obviously, there are no OFF-set minterms inside p' for any output in O (or p' would not be an implicant of $\mathcal{M}'$). Here, there must be no OFF-set minterms inside p for any output in O. Hence, p is an implicant of $\mathcal{M}$. Further, p Naturally Maps onto p'. □

From the above, given any output logic cover Π' for some reduced machine $\mathcal{M}'$, we can construct, member-wise, a set of implicants (call it Π) on $\mathcal{M}$.

Lemma 5.8 Π *is a cover for $\mathcal{M}$.*

Proof: Every ON-set minterm m in $\mathcal{M}$ has at least one corresponding ON-set minterm m' in $\mathcal{M}'$ (or else $\mathcal{M}'$ is not a valid reduction of $\mathcal{M}$). Now, for every m' in $\mathcal{M}'$, there exists an implicant $p' \in \Pi'$ that covers it. Let p be the implicant in Π that corresponds to p'. Since p' contains m', p' spans a reduced state s' which contains the unreduced state s in which m lies. Hence, p contains m, by the above construction. □

Theorem 5.9 *The unminimized machine $\mathcal{M}$ possesses a two-level logic cover which has minimum cardinality across all possible state minimizations and encodings.*

Proof: Follows from the above Lemmas, since, for every output cover Π' for any reduced machine $\mathcal{M}'$, we can construct a cover of $\mathcal{M}$ of identical cardinality. □

5.4.3 Optimality of the Constraint-Satisfaction Method

The proof here is straightforward. Because any solution to the binate constraints represents a valid reduced machine and output logic cover, we need only establish that $\mathcal{M}$ itself is a minimum-cost solution to our constraints.

Lemma 5.10 *The unminimized machine $\mathcal{M}$, along with any minimum-cardinality output logic cover for it, represent a minimum-cost solution to the binate constraints of Section 5.2.4.*

Proof: Clearly $\mathcal{M}$ satisfies the state covering constraints of part 1. Also, since a singleton state compatible can have no non-trivial state implications, $\mathcal{M}$ satisfies the closure constraints of part 3. Finally, the functional covering constraints of part 2 are monotonic, so that there is always a solution, as long as there exists an implicant to cover any given output minterm. Because we form ordinary output primes on the unreduced machine, there is always a prime to cover any given minterm. So $\mathcal{M}$ is a solution to the binate constraints. That $\mathcal{M}$ and its minimum logic cover constitute a minimum-cost solution follows directly from the cost function (logic cover cardinality). □

Theorem 5.11 *A minimum-cost solution to the constraints of Section 5.2.4 identifies a reduced machine having an exactly minimum-cardinality two-level logic cover for its outputs.*

Proof: This result follows from Theorem 5.9 (the optimality of the unminimized machine), along with the above Lemma 5.10. Note that a minimum-cost solution to the constraints may in fact be a reduced machine, but it will have precisely the same cost as that of the unminimized machine. □

5.5 Cost Function

This section briefly examines the role of the cost function in our constraint-satisfaction framework of Section 5.2.4, and suggests some alternatives. These alternatives underscore the usefulness of the constraint-satisfaction framework, even over the trivial method suggested by Theorem 5.9 (that of no performing no state reduction).

The cost function directs binate constraint satisfaction, by instructing the solver to prune less desirable solutions, namely, those having higher cost. Given the two sets of decision variables ($\{p_i\}$ and $\{c_i\}$), one can form 4 distinct cost functions. We consider the behaviour of the above framework, under each of following cost assignments:

cost(p_i)	cost(c_i)	interpretation of minimum-cost solution
0	1	minimum-cardinality *state* cover
1	0	minimum-cardinality *logic* cover
1	1 (secondary)	minimum-cardinality *state* cover over all solutions having minimum-cardinality *logic* cover
1 (secondary)	1	minimum-cardinality *logic* cover over all solutions having minimum-cardinality *state* cover

Of the four alternatives, the first two are simple one-tiered cost functions. The third and fourth cost functions require a binate solver that can accommodate a two-tiered cost structure. I.e., the solver must be able to find a solution having lowest second-tier cost among all solutions of lowest first-tier cost.[3]

Under the first cost function, OPTIMIST would behave as STAMINA does (although at higher computational complexity, since functional covering constraints must be satisfied, but have no impact on the cost). The second is simply the previously-described cost model for output-targetted OPTIMIST: output logic cardinality.

The third choice, minimum-cardinality logic cover over all solutions having minimum-cardinality state covers, seems especially useful. It exactly minimizes state cover cardinality, under the requirement of a minimum-cardinality output logic cover over all possible state reductions. In other words, out of all solutions having exactly minimum output logic cardinality, it finds one with the fewest states. We hope to provide experimental results using such a cost function in the near future.

The fourth alternative, minimum-cardinality logic cover over all solutions having minimum-cardinality state covers, is useful in situations where next-state complexity is of greater importance than that of the output logic. For example, in applications where flip-flops are expensive, this cost function addresses the primary cost, while still offering low output logic complexity. Specifically, of all solutions having the fewest states, this cost function finds one with exactly minimum logic cardinality.

5.6 Experimental Results

The OPTIMISTO algorithm has been implemented and incorporated into the OPTIMIST tool. Table 5.1 shows the experimental results for a number of benchmark machines, including several taken from the MCNC '91 suite. For each FSM, we give the number of inputs, states and outputs, as well as the number of prime compatibles and the total number of all compatibles. Finally, the number of minimized states and output products is given for OPTIMISTO in both runs and for STAMINA.

The experimental set-up is as follows. STAMINA was run using default parameters,

[3] Such a solver is trivially derived from a standard one-tiered solver such as SCHERZO.

i.e., exact minimization and heuristic state mapping, and with an option which displays the two-level symbolic cover after minimization with ESPRESSO [118]. OPTIMISTO was run as described above, twice; first using only prime compatibles, and again, using all compatibles. Run-times for all examples but `tma` were under 2 seconds; `tma` required less than 1 minute in both modes.

Generally, OPTIMISTO achieves the same or slightly better cardinality than does STAMINA. In one case, however, the improvement is quite dramatic: for `tma`, OPTIMISTO achieves a solution with less than 50% of STAMINA's product count.

In several cases, e.g. `ex3`, OPTIMISTO chooses a significantly larger state cover (i.e. number of minimized states) than does STAMINA. This is sometimes simply an artifact of the binate solver's algorithm, coupled with OPTIMISTO's cost metric, product cardinality. Because selecting an additional compatible does not directly affect the cost of the binate constraint solution, the solver can choose to do so, even if a smaller state cover would suffice. A two-tiered cost function, in which the primary cost is product cardinality and the secondary cost is state cover cardinality, would more properly model the combined cost of output and next-state logic complexity.

5.7 Conclusions and Future Work

This chapter extends OPTIMIST's formulation, resulting in the first state minimization method that precisely targets output logic. It offers two fundamental choices for state minimization. First, it proposes the startling choice of performing *no reduction at all.* Surprisingly, this yields exactly minimum-cardinality output logic over *all* state reductions, state encodings, and two-level logic minimizations. Second, it provides a novel binate constraint-satisfaction framework, based on the framework used by OPTIMIST in the previous chapter.

Interestingly, this second method has far lower computational complexity than does OPTIMIST. At the same time, it supports a set of four useful cost functions, unlike

				OPTIMIST				STAMINA	
				prime compats		all compats			
design	**i/s/o**	#pc	#ac	min st	**prod**	min st	**prod**	min st	**prod**
minstateex	2/3/1	2	4	2	3	3	2	2	3
bbara	4/10/2	7	21	7	6	7	6	7	6
beecount	3/7/4	7	11	7	6	5	6	4	6
ex3	2/10/2	91	195	11	4	14	4	5	4
ex7	2/10/2	57	135	7	3	9	3	4	4
lion9	2/9/1	5	20	4	1	4	1	4	1
opus	5/10/6	9	11	9	12	9	12	9	13
s27	4/6/1	5	7	5	6	5	6	5	6
s8	4/5/1	1	31	1	1	1	1	1	1
tma	7/20/6	20	35	19	11	20	6	18	13
train11	2/11/1	17	85	5	1	5	1	4	1

Table 5.1: Results of minimization with both OPTIMISTO and STAMINA

the first choice (that of performing no reduction). As a result, the constraint-satisfaction method also suits applications requiring the best possible output logic and desiring reduction in next-state logic complexity.

The OPTIMISTO constraint satisfaction algorithm was implemented, and experimental results for one cost function (logic cardinality) demonstrate a significant reduction in logic complexity over STAMINA, for certain machines. Run-times are a fraction of that of OPTIMIST, and rarely exceeded two seconds for the benchmark suite used.

Given a binate solver capable of handling two-tiered cost functions, the current implementation could accomodate the other cost functions described in Section 5.5. Exploring the trade-offs in these choices is an interesting area for future work.

Chapter 6

OPTIMISTA: Output-Only OPTIMIST for Burst-Mode Asynchronous State Machines

This chapter builds on the results of the previous chapter to form OPTIMISTA, a method for output-targetted state minimization for burst-mode asynchronous machines.

The chapter makes two contributions. First, it presents the only known method for optimal state minimization of asynchronous machines of any form. Moreover, this method produces *exactly minimum* cardinality two-level hazard-free output logic over *all possible state minimizations, state encodings, and logic minimizations.* The second contribution is an interesting theoretical result, analogous to one from the previous chapter: the unminimized machine always possesses minimum-cardinality two-level hazard-free output logic.

Like the previous chapter, then, this chapter offers two choices: that of performing no reduction at all (useful when output logic is the sole interest), or using OPTIMISTA to reduce next-state complexity (by performing state reduction), while simultaneously guaranteeing exactly minimum-cardinality hazard-free output logic.

Output-targetted optimization is even more effective for asynchronous machines than it is for synchronous machines. This is the case because it addresses the key performance parameter for systems of asynchronous machines: output latency. In an asynchronous system, the input-to-output latency typically dominates performance, since state changes are not bound to a clock period. In practice, state changes are usually non-critical (see, e.g., [86]), and can therefore safely proceed in parallel with the propagation and processing of output changes.

OPTIMISTA borrows the framework and spirit of OPTIMISTO, adapting them to accommodate the unique requirements of burst-mode asynchronous synthesis. Just as CHASM's input encoding model made significant extensions to its synchronous counterpart in order to guarantee correct and optimal results, so OPTIMISTA significantly extends the basic objects (compatibles, symbolic implicants, and binate constraints) used by the OPTIMISTO framework.

This chapter proceeds as follows. First, the problem domain is presented in Section 6.1 and compared with the related problems addressed by MINIMALIST and OPTIMIST. State mapping gives rise to unique problems in the asynchronous domain, and is discussed in Section 6.2. Next, the flow of the method is described in Section 6.3. Each step in the flow is presented in detail in subsections 6.4 through 6.9. One class of binate covering constraints is particularly complex, and is treated separately in detail in Section 6.7. The soundness and optimality of the method are demonstrated in Section 6.10. More efficient algorithms for a key sub-step are presented in Section 6.11. Experimental results for a variety of industrial benchmark burst-mode circuits are given in Section 6.12. Finally, some concluding remarks are made in Section 6.13.

6.1 Problem Search Space

This section compares the problem solved OPTIMISTA with the problems solved by two other state minimization methods: OPTIMISTO, described in Chapter 5, and MINIMAL-

IST, described in Chapter 7. These comparisons highlight the unique challenges that OPTIMISTA faces. In particular, although OPTIMISTO solves the optimal state minimization problem, it does so in a different domain (synchronous synthesis) than OPTIMISTA. On the other hand, MINIMALIST solves the classic state minimization problem (ignoring logic complexity), but does so in the same domain as OPTIMISTA.

We start by reiterating the basic problem requirements for burst-mode synthesis. As detailed in Chapter 2, the following are necessary and sufficient conditions for a correct burst-mode Huffman implementation:

- Freedom from essential hazards
- A critical race-free encoding
- Hazard-free output and next-state logic

6.1.1 Comparison to OPTIMIST

Because the above correctness requirements are more stringent than their synchronous counterparts, there are several key differences between the OPTIMISTA method and the OPTIMISTO method presented in Chapter 5. Specifically, OPTIMISTA must:

1. use a different compatibility relation
2. use a different type of symbolic implicants
3. perform state mapping
4. constrain the state mapping

Each of these differences is addressed in the following paragraphs.

The standard synchronous compatibility relation cannot be used, as its use does not always permit a hazard-free logic implementation. The set of compatibles used by OPTIMISTA is thus smaller than those defined by the standard synchronous compatibility relation. This smaller set does not by itself guarantee a hazard-free implementation.

However, it does exclude an entire class of compatibles that can easily be determined to not permit *any* hazard-free logic implementation.

Output logic must be hazard-free; therefore, the symbolic implicants used must be dynamic hazard-free (DHF). Fortunately, ordinary DHF-primes (as described in Section A.7) suffice, as we will later show.

Interestingly, unlike the case in the previous chapter, state mapping must be performed in order to define output logic covering requirements. This is due to state mapping's affect on the present-state extent of output required cubes in the vertical portion of specified transitions.

Finally, state mapping must be constrained to ensure a hazard-free logic implementation for the next-state. This contrasts with synchronous machines, for which correct next-state logic can be synthesized for *any* state mapping.

6.1.2 Comparison to MINIMALIST's State Minimization Method

This section compares the standard state minimization algorithm used by MINIMALIST (described in Chapter 7) with the logic-optimal algorithm of OPTIMISTA.

Because OPTIMISTA targets two-level output logic optimality, it is different from MINIMALIST's method in several ways. Specifically, OPTIMISTA must:

1. use a different compatibility relation
2. perform logic (functional) covering
3. perform state mapping

Each of these differences is discussed in the following paragraphs.

First, the compatibility relation used by OPTIMISTA is less conservative than that used by the standard MINIMALIST state minimization method. This is necessary because MINIMALIST's basic state compatibility relation discards candidate compatibles on the basis of whether a "canonical logic cover" [108] is hazard-free. It ignores hazard-free logic covers of any other form. Thus, although perfectly safe, it may discard too many compatibles, thereby missing state covers that yield optimum output logic.

Second, unlike the MINIMALIST method, which simply attempts to minimize the number of states, OPTIMISTA searches for a closed state cover *also* having minimum-cardinality two-level output logic. Thus, in order to find a logic-optimum solution, OPTIMISTA performs logic covering *simultaneously* with state covering, where MINIMALIST which simply performs a standard binate covering to find a closed state cover.[1]

Third, OPTIMISTA must choose state mappings carefully, because of their effect on the output logic. In fact, MINIMALIST only handles state partitions, so that there is no flexibility in state mapping.

6.2 The Challenge of State Mapping for OPTIMISTA

This section explores the challenges of state mapping that arise from the correctness requirements for burst-mode implementations.

We now describe a trivial extension of the synchronous solution that might appear to be a plausible solution to the asynchronous problem, but in fact fails. In this approach, a set of DHF compatibles (say, those defined by UCLOCK [108]) is formed, and DHF output primes are generated on the unreduced machine. Then, the constraint satisfaction framework of the previous chapter is used, with simple modifications to functional covering constraints as needed to reflect hazard-free logic minimization.

In this candidate approach, functional covering constraints might be modified as follows. We start from the burst-mode specification with its set of specified input transitions. For each transition, we identify a corresponding transition in the reduced machine. This "reduced transition" gives rise to one or more required cubes in the reduced

[1] Technically, it currently only performs unate state covering, relegating state closure to a post-processing check.

machine. For each of these, a constraint is generated to ensure its covering by some DHF output prime.

In fact, such an approach fails, for several reasons. This failure stems from the requirement of a hazard-free logic implementation for both outputs *and* next-state, and from the strictures of burst-mode operation. In particular, there are three issues:

1. Hazard-free output covering requirements
2. Existence of hazard-free next-state logic
3. Proper burst-mode operation

These issues are explored in the following sub-sections.

6.2.1 Hazard-Free Output Covering

The trivial approach fails, because producing hazard-free output logic necessitates the consideration of state mapping. This is due to the fact that the vertical (state change) portion of a specified transition gives rise to required cubes for static-1 output transitions. These output required cubes span both the source and destination states. Thus, the destination state must be chosen for the extent of the required cubes to be known. In other words, each required cube's shape is dependent on the state mapping chosen for the given transition.

6.2.2 Existence of Hazard-Free Next-state Logic

This section gives a simple example to show that certain combinations of state mappings conspire to prevent hazard-free logic covering for next-state. In particular, under certain state mappings, some required cubes for next-state have no covering DHF implicant. The problem is addressed by the state mapping incompatibility constraints described in Section 6.6.

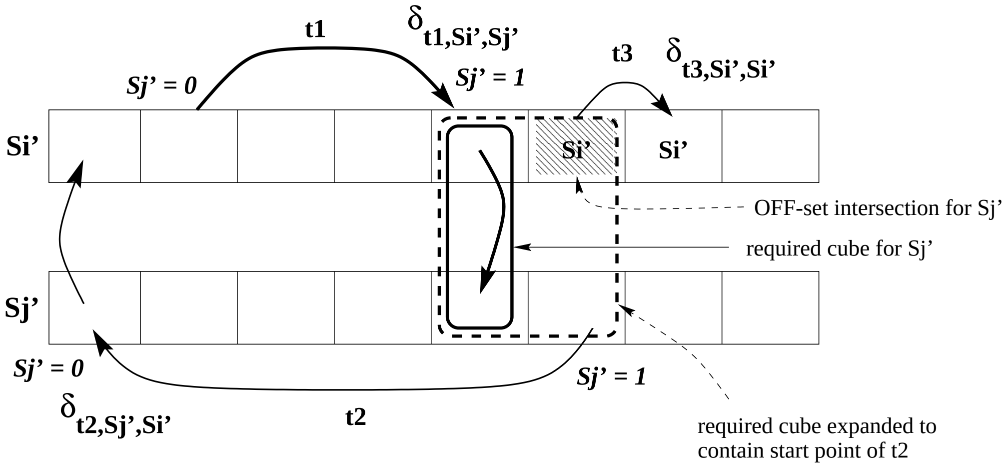


Figure 6.1: Flow table fragment and state mapping precluding a hazard-free cover

Example 6.1 Consider the state mapped transitions in the flow table fragment of Figure 6.1. The required cube for S'_j of transition t is not covered by any DHF implicant. This is due to the presence of the privileged cube in S'_j for the unstable transition t_2, and to the OFF-set point for S'_j belonging to the stably-mapped transition t_3.

6.2.3 Proper Burst-Mode Operation

This section describes a subtle problem that stems from the interaction of state mapping and proper burst-mode operation.

The problem arises as follows. Burst-mode operation, as described in Chapter 2, requires the machine to remain stable throughout the entire input burst. Thus, state mapping must be *uniform* throughout the horizontal portion (the so-called stable points) of a specified transition, and, further, *must be stable.*

Example 6.2 Consider the specified transition from $\langle 000\ S'_i \rangle$ to $\left\langle 011\ S'_j \right\rangle$ in the flow table fragment of Figure 6.2. If the stable points in total states $\langle 001\ S'_i \rangle$ and $\langle 010\ S'_i \rangle$ were unstably mapped (i.e. to some state other than S'_i), the machine would take an unintended transition to another state in the middle of the input burst. The implementation would thus violate proper burst-mode behaviour: the machine must remain stable

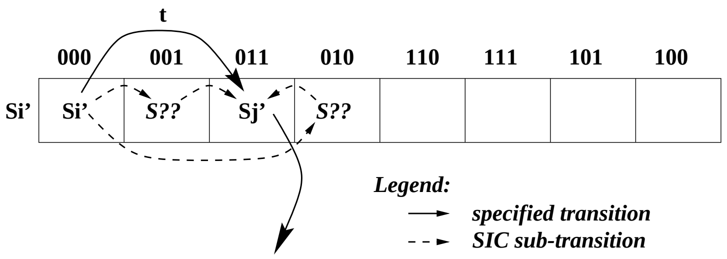


Figure 6.2: State mapping in the stable points of a specified transition

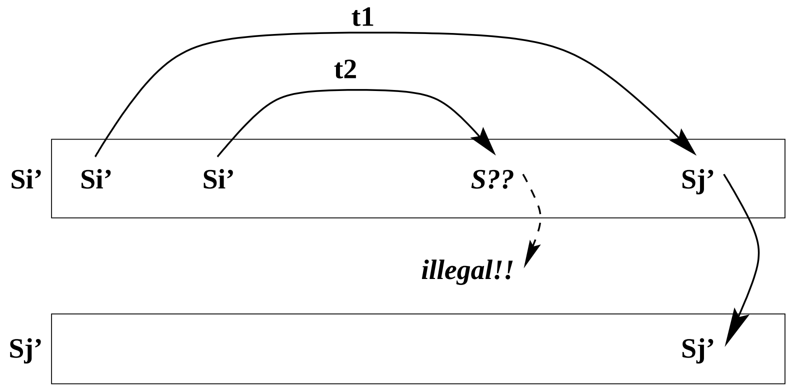


Figure 6.3: State mapping of an embedded exit point

throughout the entire input burst.

As a result, we are only free to choose a state mapping at *transition exit points*. In the previous example, only the exit point at $\langle 010\ S_i' \rangle$ has a state mapping choice.

It follows that in some cases, we must further constrain the state mapping for the exit point of a specified transition. In particular, the state mapping must be stable if state reduction causes that exit point to coincide with the stable point of another specified transition.

Example 6.3 For example, consider Figure 6.3. Transition t_1 from S_i' to S_j' contains transition t_2, also originating in S_i'. Because the t_2's exit point lies within the stable portion of t_1, t_2 must be stably state mapped. Otherwise, the machine will undergo an invalid state transition in the middle of the input burst for t_1.

The solution to this problem is to restrict the state mapping appropriately for such an embedded transition. In particular, we must exclude all unstable mapping alternatives from the set of valid state mappings for that transition. If a stable mapping is not an alternative, then no valid state mapping exists. In this case, the given compatible can be excluded from the set of candidate compatibles. Otherwise, as is done in OPTIMISTA, it can be pruned when the binate solver detects the absence of a valid state mapping in the state mapping constraint (see Section 6.6).

6.3 Method Flow

We now give an overview of OPTIMISTA, which shares the framework of OPTIMISTO, described in Chapter 5.

First, two sets of covering objects are generated from the original burst-mode specification. In particular, a set of state compatibles is generated, using the classic synchronous compatibility relation [60], followed by a filtering step. Then, an ordinary set of *DHF* symbolic output prime implicants is formed. These primes are formed *on the unreduced machine*. DHF implicants can be used to cover the reduced machine because of a relationship between implicants on the unreduced and reduced machines analogous to that present in the synchronous case. Again, *no next-state implicants are generated.*

Next, binate covering constraints are generated, using the given state compatibles and DHF primes. Covering constraints for OPTIMISTA are considerably more complex than those of the previous chapter. They fall into four categories: classic state reduction requirements (like those in [60]), state mapping, hazard-free logic covering requirements for the outputs of the reduced machine, and constraints to ensure the existence of a hazard-free implementation for the next-state.

The binate constraints are solved using Scherzo [30], under a cost model that targets logic cardinality. The result is a closed state cover, a state mapping, and a selection of DHF symbolic primes having minimum cardinality over all valid state reductions.

Using the selected state compatibles, state mappings, and output primes, a symbolic cover for the reduced machine is formed using the Natural Mapping of Chapter 5. The result is a hazard-free logic cover for the outputs of the reduced machine.

At this point, the method is complete. The reduced machine can then be passed to a burst-mode state assignment tool, such as CHASM, to produce a hazard-free binary implementation. Just as in Chapter 5, hazard-free input encoding constraints can alternatively be generated directly from the instantiated symbolic cover, and an encoding generated.

In summary, the OPTIMISTA algorithmic flow is as follows:

1. Generate state compatibles
2. Generate symbolic prime implicants
3. Generate binate covering constraints
4. Solve constraints
5. Generate an instantiated reduced table

Sections 6.4 through 6.9 detail each of the above steps for OPTIMISTA.

6.4 State Compatible Generation

This section first describes the requirements on the set of compatibles to be used, and then describes the means by which they are generated.

The goal of the following analysis is to identify the set of compatibles that, when used, guarantee the existence of hazard-free logic. We show that a tight definition of this set is not easily calculable. Instead, we propose a two-step solution. First, we identify a set of restrictions that are necessary but insufficient, and use this to prune the set of compatibles. Second, we indicate how the remaining conditions are folded into a later step: binate constraint generation.

A simple example demonstrates a problem with hazard-free logic caused by indiscriminate state merges. Next, the problem is analyzed to precisely identify the hazardous conditions. Unfortunately, it is shown that a pairwise compatibility relation cannot directly generate the appropriate set of state compatibles. The approach used by OPTIMISTA instead employs a standard synchronous pairwise compatibility relation, followed by a filtering step.

The compatibility relation has an impact on the existence of hazard-free logic because state merges can introduce unavoidable logic hazards. This is illustrated by the following example.

Example 6.4 Figure 6.4 illustrates the situation for a four-input, single-output machine. The Karnaugh map depicts the output function in a reduced state which results from merging 3 states $\{s_1, s_2, s_3\}$, containing transitions $\{t_1, t_2, t_3\}$, respectively. We ignore the next-state, as it does not affect the analysis of horizontal output transitions.
Note that the required cube corresponding to static-1 transition t_1 illegally intersects dynamic transition t_2's privileged cube, at minterm `0000`. The state merge leaves no way to expand t_1 to contain t_2's start point at `0100` without causing an intersection with t_3's OFF-set cube.

It is possible to use the standard synchronous compatibility relation and defer a check for the existence of a hazard-free cover to the binate covering step. However, in order to reduce computational complexity, OPTIMISTA uses a more restrictive compatibility relation instead.

UCLOCK's compatibility relation cannot be used for the present problem. UCLOCK defines a compatibility relation that avoids *all* function and logic hazards, but only for a specific form of logic cover [108]. This "canonical" form is simply a single-output cover consisting of the required cubes of the unminimized machine. In short, the compatibil-

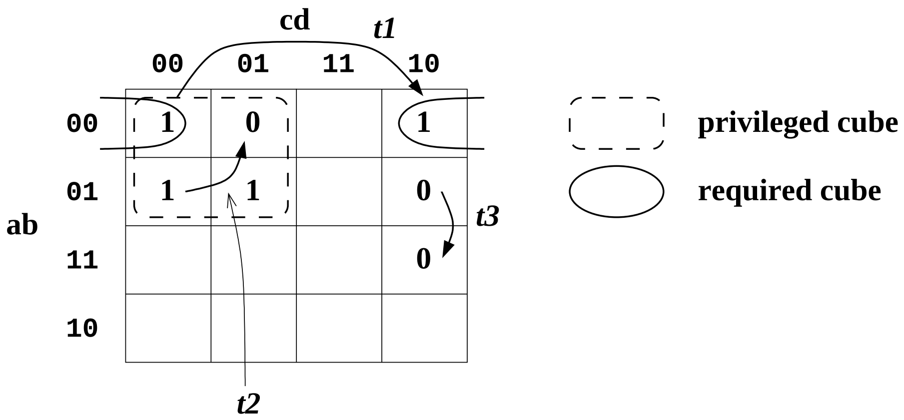


Figure 6.4: Karnaugh map for output function in reduced state $s' = \{s_1, s_2, s_3\}$.

ity relation expresses sufficient but not necessary conditions for hazard-freedom. Thus, UCLOCK's relation is too restrictive for OPTIMISTA's purposes, since it focuses on a single form of cover, thereby missing opportunities for better output logic.

There are five conditions that imply the incompatibility of a state set. The first four relate to logic hazard-freedom, and the last, to the aforementioned requirement of stably state mapping all of a transition's stable points.

1. Absence of a hazard-free cover for horizontal output transitions

2. Absence of a hazard-free cover for vertical output transitions (for *all* possible state mappings)

3. Absence of a hazard-free cover for horizontal next-state transitions (for *all* possible state mappings)

4. Absence of a hazard-free cover for vertical next-state transitions (for *all* possible state mappings)

5. Absence of a stable state mapping for some exit point which lies in the horizontal portion of another transition

In general, no hazard-free cover exists when some required cube r exists which a) illegally intersects a privileged cube p and b) cannot be expanded to contain p's start point without hitting the OFF-set.

Of the five conditions, OPTIMISTA directly detects only one: the first condition. This condition is readily detected without overly complicated analysis. The other conditions are folded into the covering constraints, so as to let the binate solver do the remaining analysis.

One issue remains: the generation of the new set of compatibles. In general, it is more efficient to generate the set of compatibles from a *pairwise incompatibility relation* (as is typically done for the synchronous case), rather than to identify the compatibles one by one. However, no pairwise relation can capture all of the incompatibilities of even the first kind above. We illustrate this problem with an example.

Example 6.5 Returning to the example of Figure 6.4, observe that, although merging states s_1 and s_2 produces no unsatisfiable hazard-free covering requirements, adding s_3 to the state set does. In fact, it is not possible to derive the set of compatibles satisfying condition 1 above from a set of incompatible state pairs.

OPTIMISTA therefore generates state compatibles using the standard synchronous pairwise incompatibility relation, and prunes the resulting set using Algorithm 8. This algorithm assumes the set of DHF output-primes on the unreduced machine are already available. Without loss of generality, we only show the analysis for a single output.

6.5 Symbolic Prime Implicant Generation

This section defines the set of symbolic implicants used by OPTIMISTA.

OPTIMISTA uses ordinary DHF output primes for symbolic hazard-free logic minimization. In particular, DHF output primes can be used to cover the "reduced required cubes" for the reduced machine's outputs. We will show in Section 6.9 that the Nat-

Algorithm 8 Compatible filtering algorithm for OPTIMISTA

```
function filter_compatibles(C) {
  C′ ← C;
  for each compatible c ∈ C′ {
    for each transition t originating in unreduced state s ∈ c {
      for each horizontal required output cube r of t {
        if no DHF prime contains r′ and spans c {
          // No hazard-free cover for r exists
          C′ ← C′ − {c};
} } } } }
```

ural Mapping can be used to instantiate these primes, without further regard for logic hazards.

6.6 Binate Constraint Generation

This section first presents the decision variables in the covering problem, and then presents the constraint categories, each in its own subsection.

6.6.1 Constraint Variables

The following table describes each of the decision variables involved in the binate covering constraints, and shows their assigned cost in the covering problem.

variable	**description**	**cost**
c_i	select compatible c_i for the state cover	0
δ_{t,c_i,c_j}	map the exit point of transition t in reduced state c_i to c_j	0
p_i	select symbolic DHF prime p_i for the symbolic logic cover	1

6.6.2 Constraint Roadmap

There are six types of constraints, summarized below. Subsequent subsections cover each type in detail.

1. State covering — Cover each state with at least one compatible
2. State mapping — Choose a destination state for each transition exit point
3. State mapping coherency — Choose at most one state for each state mapping choice
4. Functional covering (for output logic only)
   - Horizontal transitions: conditional on the selection of a reduced state
   - Vertical transitions: conditional on the selection of a state mapping for the transition's exit point
5. State closure — The set of selected compatibles must be closed
6. State mapping incompatibility — Ensure the existence of hazard-free next-state logic

In several cases, clauses are shown in implicative form, e.g., $a \rightarrow b$, to emphasize the conditional nature of the constraint. This form is exactly equivalent to the standard binate form $\overline{a} + b$.

6.6.3 State Covering

This set of constraints ensures that each unreduced state is covered by selecting some compatible that contains it. These constraints are identical to the state covering constraints in OPTIMISTO and Grasselli's classic formulation [60].

$$\begin{array}{l}\forall \text{ unreduced states } s \text{ of } \mathcal{M} \\ \quad c_{i_1} + c_{i_2} + \cdots + c_{i_n}\end{array}$$

where $\{c_i\}$ is the set of compatibles that contain s.

6.6.4 State Mapping

This set of constraints ensures that the exit point of each specified transition in the reduced machine is state-mapped appropriately. This is accomplished by selecting for transition t'_{c_i} a single destination state $s'_j = c_j$ from the set of valid choices. A set of decision variables $\{\delta_{t,c_i,c_j}\}$ is associated with each specified reduced transition t'_{c_i} in reduced state c_i, one for each valid state mapping choice c_j for that transition's exit point.

$\forall$ compatibles c *For each candidate compatible/reduced state*

$\forall$ specified transitions t of $\mathcal{M}$ such that $PS(t) \in c$

For each transition t_c corresponding to some t of $\mathcal{M}$

$$c \;\rightarrow\; \delta_{t,c,c'_1} + \delta_{t,c,c'_2} + \cdots + \delta_{t,c,c'_n}$$

where $\{c'\}$ are the compatibles which cover exitPoint(t).

The clause is conditional on the selection of compatible c, since only then does the transition t_c exist in the reduced machine.

As described in Section 6.1.1, if exitPoint(t) $\in$ stablePoints(t') for some t' lying in c, then the only valid choice is $\delta_{t,c,c}$ (which makes t a stable transition). Otherwise, non-burst-mode operation will result for t'. If in addition, $c \notin \{c'\}$, so that stable mapping is not an option, then *no* state mapping is valid. In that case, the clause becomes simply $(\overline{c})$, which can only be satisfied by *not* selecting compatible c.

6.6.5 State Mapping Coherency

This set of constraints ensures that *at most one* state mapping is selected for each reduced transition. It is necessary, because the set of N state mapping choices $\{c_j\}$ for reduced transition t_{c_i} is "encoded" in N distinct binary decision variables $\{\delta_{t,c_i,c_j}\}$. Thus, without the following constraints, more than one may be selected. To enforce the choice of a single mapping $\{\delta_{t,c_i,c_j}\}$, the following set of simple pairwise disjointness constraints is added.

$\forall$ compatibles c *For each candidate compatible/reduced state*

$\forall$ specified transitions t of $\mathcal{M}$ such that $PS(t) \in c$

For each transition t_c corresponding to some t of $\mathcal{M}$

$\forall$ distinct pairs of state mapping variables $\langle \delta_{t,c,c_a}, \delta_{t,c,c_b} \rangle$ for t_c

$\overline{\delta_{t,c,c_a}} + \overline{\delta_{t,c,c_b}}$ *Select at most one state mapping for t_c*

6.6.6 Functional Covering

This set of constraints effectively selects a set of DHF primes so as to functionally implement the outputs.

These constraints are a simple extension of the functional covering constraints of Chapters 4 and 5. However, in order to perform hazard-free covering, two essential differences arise. First, DHF prime implicants are used, and second, required cubes (cf. minterms) must be covered. The required cubes are identified reduced row by reduced row, just as the ON-set minterms were derived in OPTIMIST.

The next-state functions are not covered, since output logic quality is the sole cost metric. Instead, as mentioned earlier, the sole requirement for the next-state is to guarantee the existence of a hazard-free logic implementation for it. This is ensured by appropriately constraining the state mapping, an assurance provided by the state mapping incompatibility constraints, shown later in this chapter.

The following paragraphs deal with functionally covering the required cubes of a) horizontal and b) vertical portions of the specified transitions, respectively.

Horizontal Transitions

This section describes the constraints necessary to ensure the functional covering of the outputs in the horizontal portion of specified transitions.

State mapping is not needed to functionally cover required cubes in horizontal transitions. Since the horizontal (stable) portion of each specified transition *must* be stably state-mapped, all required cubes for the outputs during horizontal transitions are completely defined irrespective of any state mapping choices. Hence, functional covering of the required cubes outputs can be likewise performed without regard to state mapping. Consequently, the following constraint clauses do not depend on δ variables.

$\forall$ compatibles c *For each possible reduced state*

$\forall$specified transitions t of $\mathcal{M}$ such that $PS(t) \in c$

For each transition t_c corresponding to some tof $\mathcal{M}$

$\forall$ horizontal required cubes r of t

$c \rightarrow p_1 + p_2 + \cdots + p_N$

where $\{p_i\}$ is the set of DHF primes which a) contribute to the output, b) contain c, and c) contain r.

The horizontal required cubes $\{r\}$ are the required cubes of the transition t_c described in Section A.7. Specifically, they consist of the transition supercubes for all $1 \rightarrow 1$ output transitions, along with the maximal ON-set sub-cubes for all $1 \rightarrow 0$ and $0 \rightarrow 1$ output transitions.

This constraint is conditional on the selection of compatible c, since the reduced transition t_c only exists if compatible c is selected.

Each functional covering constraint for horizontal transitions always contains at least one eligible DHF prime. This is because any compatibles which have unsatisfiable covering constraints for horizontal output required cubes is pruned by the compatible filtering step described in Section 6.4. (The same can not be said of the "vertical functional covering constraints" that follow.)

Vertical Transitions

This section describes the constraints necessary to ensure the functional covering of the outputs in the vertical (state change) portion of specified transitions.

Unlike the case for horizontal transitions, state mapping is needed to functionally cover required cubes in vertical transitions. The shape of the required cubes for a static-1 vertical transition is defined in part by the state mapping chosen for the transition's exit point. In particular, if the next-state is mapped unstably, the required cube spans 2 states; otherwise, it spans only 1. As a result, we employ decision variables to record the state mapping decision made for each transition exit point. These variables are then used in forming the functional covering constraints for the exit points.

$\forall$ compatibles c *For each possible "reduced state"*
$\forall$specified transitions t of $\mathcal{M}$ such that $PS(t) \in c_t$
For each transition t_c corresponding to some $t$of $\mathcal{M}$
$\forall$state mapping choices $\delta_{t,c,c'}$ *For each choice of destination state*
$\forall$ vertical required cubes r of t when the exit point of t is mapped to c'
$\delta_{t,c,c'} \;\rightarrow\; p_1 + p_2 + \cdots + p_N$

where $\{p_i\}$ is the set of DHF primes which a) contribute to the output, b) span both c and c', and c) contain r.

The vertical required cubes $\{r\}$ consist solely of the vertical transition supercubes of all $1 \rightarrow 1$ output transitions. Recall that all vertical output transitions are static, i.e., either static-0 or static-1. Only the latter kind has any required cubes. Note that no constraint clause is generated for case where the transition is stably mapped (i.e. $\delta_{t,c,c}$), since in that case, the vertical portion of the transition collapses. This is because functional covering of the horizontal transition performed above takes care of all required cubes.

This constraint is conditional on the selection of state mapping variable $\delta_{t,c,c'}$, since the given vertical required cubes only exist if state mapping choice $\delta_{t,c,c'}$ is selected.

6.6.7 State Closure

This set of constraints ensures that the selected set of state compatibles, along with the chosen state mappings, is closed, in the classical sense.

Given the discussion in Section 6.1.1, the *only* non-unique state implications arise from vertical transitions. Hence, state mapping, as performed above using δ variables, along with the following connective clauses, guarantees state closure.

$\forall$ variables δ_{t,c_a,c_b} *For each state mapping choice*

$\delta_{t,c_a,c_b} \rightarrow c_b$ *If c_b is chosen as the destination state, must select it as well*

The above constraints guarantee state closure because:

- The *only* total states for which there is a choice are the specified transitions' exit points
- All other total states have only trivial implications (i.e. they imply only the reduced state in which they reside)
- *All* transitions' exit points are state mapped

6.6.8 State Mapping Incompatibility

State mapping incompatibility constraints ensure the existence of a hazard-free logic implementation for next-state, by disallowing those combinations of state mapping choices for which *no* hazard-free logic implementation exists. This is an important but subtle problem, and is analyzed in detail in the following section.

6.7 State Mapping Incompatibility Constraints

This section describes the process by which state mapping incompatibility constraints are generated.

The basic goal of this set of constraints is to guarantee the existence of a hazard-free logic implementation for the next-state. This is an issue because certain combinations of state mappings prevent the hazard-free covering of the next-state. Thus, this section analyzes the problem in detail to determine necessary and sufficient conditions under which a hazard-free cover fails to exist. These conditions are then used as the basis for a set of binate constraints, which are generated using a pair of algorithms.

The problem arises as follows. Hazard-free covering is a unate process; therefore, a hazard-free cover fails to exist if and only if no DHF implicant covers a particular required cube. For this to be the case, the required cube, along with all implicants that contain it, must illegally intersect some privileged cube. Now, state mapping determines the presence and shape of privileged cubes for specific next-states for a given transition. For example, suppose the valid state mappings of transition t in reduced row s_1' are s_1', s_2' and s_3'. The stable choice s_1' results in no privileged cubes at all, while the choice s_2' results in privileged cubes for s_1' and s_2', but not for s_3', and so on. Such privileged cubes may cause intersecting implicants to be dynamic-hazardous, and thereby prevent the covering of a given required cube.

The simple solution is to constrain the state mapping. In particular, we generate a set of constraints which captures the precise conditions for the existence of hazard-free next-state logic. There are two basic types of constraints: 1) constraints which restrict the state mappings of transitions in a single state, and 2) constraints which restrict the state mappings of transitions in two states.

Clearly, the set of all state mapping combinations across all possible state reductions can be very large. However, the set of combinations actually examined can be greatly pruned, by means of careful case analysis and by exploiting the structure of the problem. For example, the following analysis takes care to focus attention on those situations which can actually occur in proper burst-mode specifications.

The structure of the remainder of the section is as follows. First, a number of basic facts are presented, which begin the case analysis.

6.7.1 Basic Elements in the Case Analysis

We now present several facts regarding burst-mode operation and state reduction which will help in the case analysis to follow.

An essential issue throughout the case analysis is that the type of transition a given next-state function experiences is defined in part by the state mapping of the transition's exit point. Consider the transition of Figure 6.5, for instance. The candidate next-states are S'_i and S'_j; each choice has a different effect on the next-state function S'_i, as shown in Figures 6.6 and 6.7. The first choice, S'_i, represents a stable mapping, which yields a $1 \rightarrow 1$ transition for S'_i and a $0 \rightarrow 0$ for S'_j. The result is a single required cube for S'_i, and no privileged cubes. The alternative state mapping, S'_j, instead causes the S'_i function to undergo a $1 \rightarrow 0$ transition, and S'_j to undergo a $0 \rightarrow 1$ transition. In this event, there are two privileged cubes: one for S'_i, and another (trivial) one for S'_j, along with a vertical required cube for S'_j.

State mapping can collapse an unstable transition having a privileged cube into a stable transition having no privileged cube.

Since OPTIMISTA does not generate DHF implicants for the next-state functions, the following analysis constructs DHF implicants (not necessarily primes) on-the-fly, if possible, to verify the existence of a hazard-free cover for that cube.

Before proceeding with the analysis, we give the following definition and lemmas.

Definition 6.6 *A privileged cube is* non-trivial *iff it contains an ON-set point other than the corresponding transition's start point.*

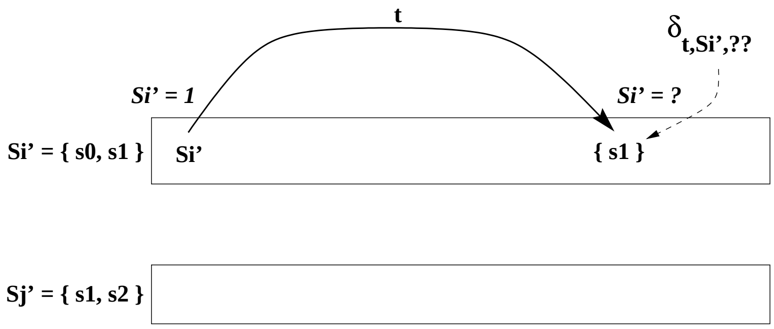


Figure 6.5: A specified transition with two possible state mappings

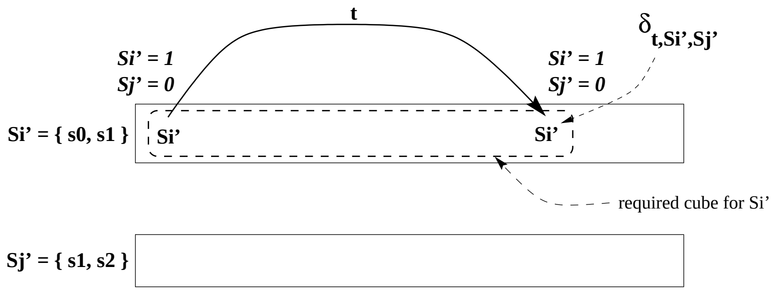


Figure 6.6: One possible state mapping for the transition of Figure 6.5

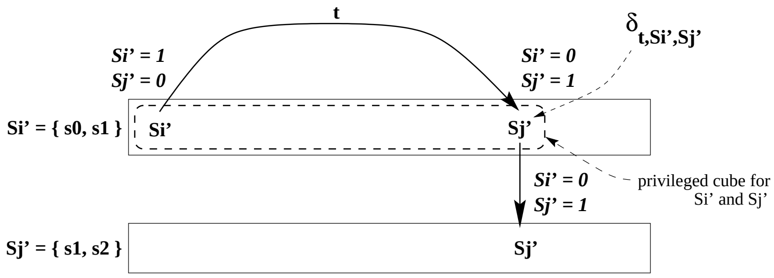


Figure 6.7: Another possible state mapping for the transition of Figure 6.5

Lemma 6.7 *The privileged cubes for the destination state of an unstably-mapped transition* are always trivial.

As shown in Figure 6.7, the destination state's sole ON-set minterm is at the transition exit point.

Corollary 6.8 *Only the source state of an unstable transition possesses a non-trivial privileged cube.*

These facts prove useful because trivial privileged cubes can *never* be illegally intersected by an implicant. Thus, the analysis need only concern itself with illegal intersections with the source state privileged cube.

The following lemmas help to limit the scope of state mapping choices which need to be examined.

Lemma 6.9 *Determining covering requirements for a horizontal next-state transition t' in s'_i requires state mapping information* only for transitions originating in the source state s'_i.

Proof: Any implicant used to cover a horizontal required cube r corresponding to t' only needs to span s'_i. Meanwhile, transitions t'' originating in any distinct state s'_j never have privileged cubes spanning s'_i, and hence cannot interfere with covering r. Further, their vertical portions can only interfere when t'' is state-mapped to s'_i (else there is no intersection), and only by virtue of an OFF-set cube for s'_i which prevents expansion to include the start point of t'. However, state-mapping t'' to s'_i implies that s'_i has ON-set points throughout the vertical portion of t''. $\square$

The next lemma extends the previous one, further pruning the search space.

Lemma 6.10 *Determining the feasibility of covering a horizontal next-state required cube only requires knowing* which transitions have been unstably mapped.

With the above in mind, we now proceed to the analysis of the four sources of next-state covering requirements in the reduced machine.

In the sequel, we refer to the stable state at the source of a given specified transition as the *source state*. Likewise, the *destination state* is the target state of the specified transition.

Clearly, the required cubes of both the source and destination state functions must be covered. This gives rise to the following taxonomy of covering requirements:

1. Horizontal required cubes

   (a) for the source state

   (b) for the destination state

2. Vertical required cubes (present only if the transition is mapped unstably)

   (a) for the source state

   (b) for the destination state

Each of the four cases is examined, in turn, in the following subsections. First, the covering requirements are identified, and any impossible situations noted. Following this, two simple algorithms are shown, one for the horizontal cases, and another for the vertical cases. Later, Section 6.11 presents a pair of optimized algorithms that have reduced run-time complexity and generate fewer, less restrictive constraints.

6.7.2 Horizontal Required Cubes for the Source State

We now analyze the covering requirements for the source state in the horizontal portion of the specified transition.

Figures 6.8 and 6.9 depict the situation involved in covering required cubes for the reduced source state s_i' in the horizontal portion of a specified transition t. If the

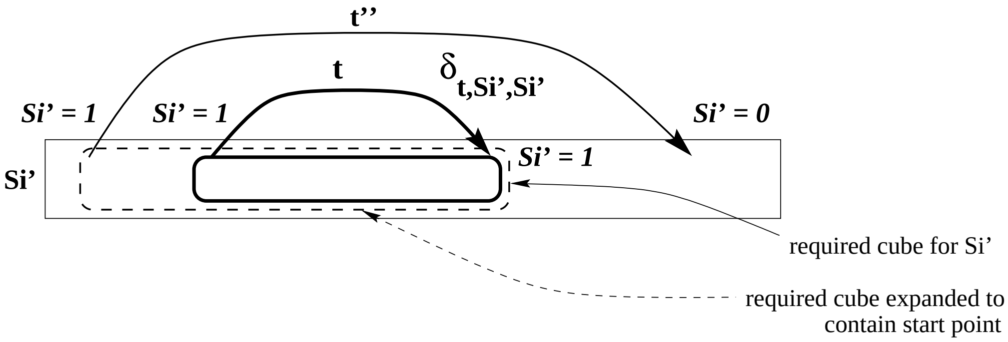


Figure 6.8: Horizontal required cubes for the source state in a stably-mapped transition

exit point of t is stably mapped, a single required cube (the supercube of t) results, as shown in Figure 6.8. There are no privileged cubes in this case. If t is unstably mapped, as shown in Figure 6.9, one or more smaller required cubes results (the maximal ON-set sub-cubes of t).

In the case of both stable and unstable mappings, the only possible interference with hazard-free covering arises from the presence of privileged cubes in s_i'. Now, privileged cubes for s_i' in s_i' can only result from unstable transitions in s_i'. Thus, Algorithm 9 only needs to map the source state in order to determine the feasibility of covering the source-state required cubes.

6.7.3 Horizontal Required Cubes for the Destination State

We now analyze the covering requirements for the destination state in the horizontal portion of the specified transition.

We show that no constraints need to be generated relative to the horizontal required cubes of the reduced destination state s_j'. Note that when t is stably mapped, all states s_j' other than s_i' undergo a $0 \rightarrow 0$ transition, and there are no required cubes for s_j'. (I.e., the destination state is s_i' itself.) If, however, t is unstably mapped, the destination state s_j' undergoes a $0 \rightarrow 1$ horizontal transition. The result is a trivial privileged cube, as shown in Figure 6.10. The sole maximal ON-set cube (a minterm) is subsumed by the

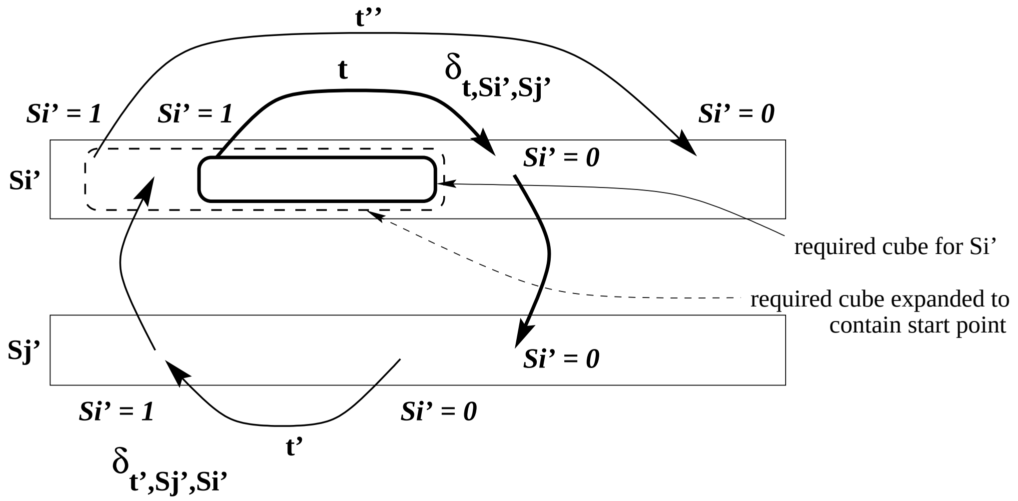


Figure 6.9: Horizontal required cubes for the source state in an unstably-mapped transition

corresponding static-1 vertical transition supercube, as shown.

6.7.4 Generating Horizontal State Mapping Incompatibility Constraints

This section presents a "naive" algorithm for identifying incompatible state mappings on behalf of required cubes in the horizontal portion of a specified transition.

We summarize the above analysis, as follows. First, we need only generate state mapping incompatibility constraints for horizontal required cubes involving the source state. Second, all constraints concerning source-state required cubes can be generated by considering state mapping combinations for the source state alone.

All "relevant" transitions must be state mapped in order to determine the location of the various privileged cubes. Hence, the following algorithm enumerates each complete combination of state mappings (for all transitions originating in the source state), and generates a constraint for each one that has no hazard-free cover for some required cube.

The set of relevant transitions can be pruned by even more careful case analysis. An optimized algorithm that embodies such an analysis is presented in Appendix B as

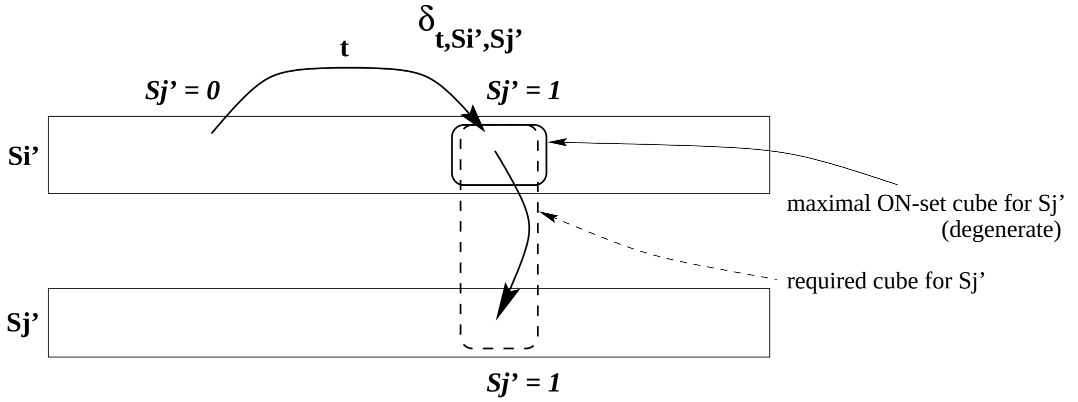


Figure 6.10: Required cubes for the reduced destination state in the horizontal portion of a specified transition

Algorithm 11.

Procedurally, we verify the existence of a suitable DHF implicant in each case by attempting its construction on-the-fly. This is necessary, since no DHF prime implicants are generated a priori for next-state. In fact, it does not appear possible to improve on this method. In particular, it would be *impossible* to generate a single set of such implicants a priori on the unreduced machine (analogous to the output DHF-primes). The next-state function is only well-defined *after* state mapping; hence, many sets of implicants, each targetting a particular reduced state and state mapping combination, would result.

The algorithm proceeds as follows. First, the outermost loop selects a reduced row (a compatible). Then, every combination of state mappings for all of the specified transitions in this row are enumerated. For each combination, the required cubes of each specified transition are identified. A covering implicant is then formed for each required cube, if possible. Specifically, the procedure attempts to use the required cube to cover itself, and expands it as necessary to avoid illegal intersections with any privileged cubes. If at any point the product hits the OFF-set, it is no longer an implicant. When that happens, no hazard-free cover exists for that required cube. In that event, an incompatibility constraint is generated which outlaws that particular state mapping combination.

Algorithm 9 Identifying incompatible state mappings with respect to horizontal required cubes

```
identifyHorizontalIncompatibilities()         // Unoptimized version
{
   // Horizontal portion done separately since it only requires state
   // mapping source state transitions.
   for each compatible c/s' {                   // Select a reduced row
      for all state mappings of the transitions in c {
         for each transition t in c {       // Select a specified transition in c
            for each horizontal required cube r of t {
               // Try to cover r
               p' := r;                        // Start by using r itself
               // N.B. The only non-trivial privileged cubes in s' are for
               // s' itself: the horizontal portions of t are always stable
               while p' illegally intersects a priv cube of t' in s' {
                  p' := p' expanded to contain start(t');
                  // OFF-set(s') is defined wrt state mapping of s'
                  if p' intersects OFF-set of s' {
                     disallow this mapping set (generate a constraint);
                     continue with the next state mapping;
                  }
   }  }  }  }  }
}
```

6.7.5 Vertical Required Cubes for the Source State

This section examines the covering requirements for the source state in the vertical portion of the specified transition.

Vertical required cubes can never exist for the source state of any specified transition. In particular, when the transition exit point is stably mapped, no state change takes place, and hence there is no vertical transition. The only required cube in that case is a single horizontal cube. If, on the other hand, the exit point is unstably mapped, the vertical portion of the transition specifies the destination state as the next-state throughout. In other words, the source state undergoes a $0 \rightarrow 0$ transition in the vertical portion, and thus has no required cubes.

6.7.6 Vertical Required Cubes for the Destination State

Finally, we examine the covering requirements for the destination state in the vertical portion of a specified transition.

Figure 6.10 shows that any implicant covering these vertical required cubes must avoid illegal intersections with privileged cubes arising from unstable transitions in the destination state s'_j alone. This is true in part because there can *never* be non-trivial privileged cubes for s'_j in any state other than s'_j.

At the same time, an implicant of s'_j must by definition not intersect the OFF-set of s'_j. Now, the OFF-set of s'_j is simply defined as the union of the ON-sets of all other states. In particular, this OFF-set has three components:

1. the vertical portion of unstable transitions leaving s'_j (part of the ON-set of the corresponding destination state. [The existence of such an unstable transition in turn depends on the state-mapping of its exit point.]

2. stable transitions in s'_i, and

3. unstable transitions leaving s'_i which do not target s'_j.

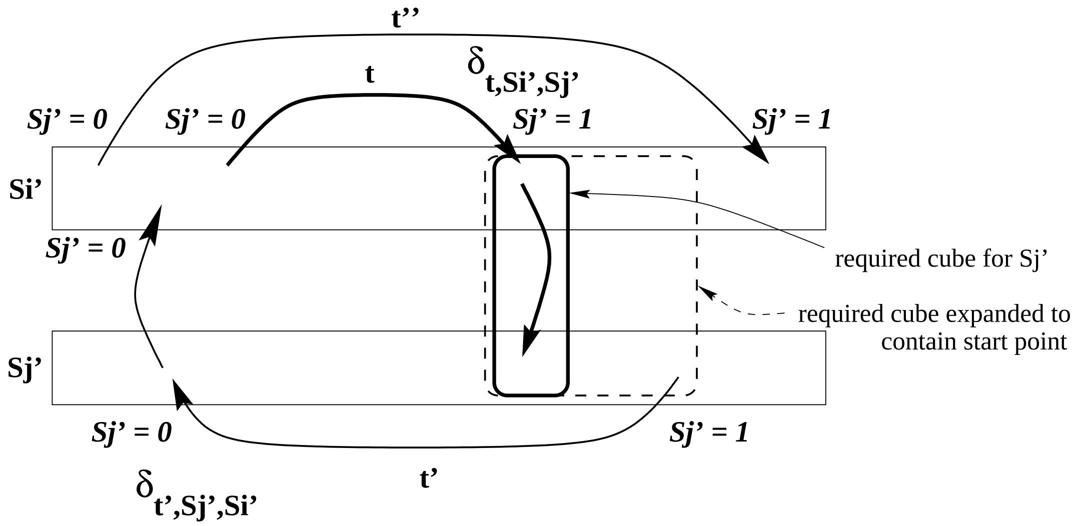


Figure 6.11: Required cubes for the reduced destination state in the vertical portion of a specified transition

The above conditions constitute the OFF-set intersection described as "`hits OFF-set of` s_d'" in the algorithm below.

6.7.7 Generating Vertical State Mapping Incompatibility Constraints

The above analysis suggests a "naive" algorithm for generating state mapping incompatibility constraints on behalf of required cubes in the vertical portion of a specified transition.

Like the horizontal case, all "relevant" transitions must be state mapped in order to determine the location of the various privileged cubes. Hence, the following algorithm enumerates each complete combination of state mappings (for all transitions originating in the source *and destination* states), and generates a constraint for each one that has no hazard-free cover for some required cube.

Again, the set of relevant transitions can be pruned by even more careful case analysis. An optimized algorithm that embodies such an analysis is presented in Appendix B as Algorithm 13.

The algorithm proceeds as follows. First, the outermost loop selects a reduced row (a compatible). Then, every combination of state mappings for all specified transitions in this row are enumerated. For each combination, the required cubes of each specified transition are identified. Unlike the horizontal case, a non-trivial required cube only exists if the transition is unstably mapped. A covering implicant is then formed for each required cube, if possible. Specifically, the procedure first attempts to use the required cube to cover itself, and then expands it as necessary to avoid illegal intersections with any privileged cubes. If at any point the product hits the OFF-set, it is no longer an implicant. Unlike the horizontal case, the OFF-set is defined in part by mappings of transitions in the destination state, s'_d. If an OFF-set intersection occurs, no hazard-free cover exists for that required cube. In that event, an incompatibility constraint is generated which outlaws that particular state mapping combination.

6.8 Binate Constraint Solution

As with OPTIMISTO, once all constraints are generated, the constraints are solved by a standard binate solver. The solution to the constraints results in a selection of state compatibles, a selection of state mappings for each specified transition in the reduced machine, and a minimum-cardinality selection of DHF output prime implicants. The current implementation uses SCHERZO, a highly-tuned binate solving package developed by Coudert [30].

6.9 Instantiation

This section describes the instantiation process, the means by which a unique reduced machine and hazard-free output cover are derived from the original specification, using the selected set of state compatibles, state mappings, and DHF output primes.

Algorithm 10 Identifying incompatible state mappings with respect to vertical required cubes

```
identifyVerticalIncompatibilities()
{
   for each compatible c/s' {
      for each combination of state mappings of the transitions in c {
         M := {⟨t, dest_t⟩, ∀t in c}; // M records the chosen state mappings
         for each transition t in c {
            s'_d := M(t); // Find chosen destination state s'_d for t under M.
            // If s'_d ≠ s', t is unstable. Therefore, there is a non-trivial
            // vertical required cube r to cover for s'_d.
            if s'_d ≠ s' {
               // Note that r spans both s' and s'_d.  Note also that
               // priv cubes and OFF-set of s'_d are defined only wrt
               // the set of state mappings.
               Implicant p' := r;
               while p' illegally xsects priv cube of t' in s' or s'_d {
                  p' := supercube(p', start(t')); // Make p' contain start point
                  if p' hits OFF-set of s'_d { // Must consider mappings in s'_d
                     disallow this mapping set (generate a constraint);
                     continue with next state mapping set for c;
                  }
   } } } } }
}
```

First, we establish that dynamic hazard-freedom is preserved under the Natural Mapping of Section 4.3.5. This result allows OPTIMISTA to use ordinary DHF output prime implicants that are formed *on the unreduced machine* as the candidate symbolic output implicants for covering the reduced machine.

OPTIMISTA's goal is a minimum-cardinality *hazard-free* SOP output logic cover; hence, any implicant used must clearly be DHF for the selected reduced machine. The following theorem proves that all DHF output primes on the unreduced machine remain DHF on *any* reduced machine, under the Natural Mapping of Section 4.3.5.

Theorem 6.11 *A DHF output prime implicant p is DHF for* any *reduced machine, after instantiation under the Natural Mapping.*

DHF prime p does not illegally intersect any dynamic transition in any unreduced state which it spans, by definition.[2] Every output transition t' of the reduced machine M' is the image of some unique specified output transition t of M, and is the same kind ($0 \rightarrow 0$, $0 \rightarrow 1$, etc.) as t. The Natural Mapping instantiates p's present-state field only with compatibles it completely spans. So, p' illegally intersects the privileged cube of some transition t' of M' iff p also illegally intersects t.

We now describe the instantiation procedure for the burst-mode machine. This is a deterministic process, since both the output and next-state behaviour of the reduced machine are dependent solely on the selected compatibles and state mappings.

Given the unreduced machine as a two-part representation, consisting of a functional specification of output and state behaviour (identical in form to a synchronous flow table) along with a set of specified input transitions, the reduced machine is derived in two parts, as follows. First, the functional specification is transformed using the selected set of state compatibles and state mappings, just as a synchronous machine would be instantiated. Next, the specified input transitions are instantiated, in a one-to-many transformation. Specifically, each input transition t in an unreduced state s is trans-

[2] considering privileged cubes of outputs to which p does not contribute as never intersecting p

formed to a set of one or more specified transitions $\{t'_c\}$, one for each selected compatible c containing s. The destination state for each transition is defined by the chosen state mapping ($\delta_{t,c,c'}$) for that particular transition/compatible pair.

Technically, the resulting machine is not a valid burst-machine *specification*, because state merging can result in intra-state transitions having non-empty output bursts, violating the unique entry point criterion. However, the machine conforms to proper burst-mode behaviour in all respects.

The selected DHF output prime implicants are instantiated using the Natural Mapping defined in Section 4.3.5, to produce a hazard-free two-level output logic cover for the given reduced machine.

6.10 Theoretical Results

This section demonstrates the soundness and optimality of OPTIMISTA. First, we show that any solution to the above binate constraints yields a valid reduced machine and a valid output logic cover. Then, we show that our method yields a (possibly) reduced machine with minimum hazard-free output logic cardinality over all possible state minimizations and encodings.

In the process, we prove in Section 6.10.2 an interesting and useful theoretical result. A minimum-cardinality hazard-free output logic cover for the unreduced machine itself has minimum cardinality *across all possible state minimizations and encodings.* A consequence of this is that state reduction can *never* reduce output logic complexity.

Although the unminimized machine always gives the optimum solution for output logic cardinality, it does not necessarily do so under other cost functions. For example, it is well-known that state reduction often helps simplify the next-state logic.[3] Thus, obtaining an optimum solution under a cost function incorporating, e.g., both output logic and state cardinality, may require that state reduction be performed.

[3]This is of course the classic motivation for state minimization.

6.10.1 Correctness of OPTIMISTA

OPTIMISTA's soundness is demonstrated in several steps. First, it is shown that any solution to the binate constraints of Section 6.6 yields a valid reduced machine. Next, we establish a relationship between the minterms of $\mathcal{M}$ and the minterms of $\mathcal{M}'$, and use that to relate functional covering on $\mathcal{M}$ to that on $\mathcal{M}'$. Finally, we prove that a solution to the binate constraints also produces a valid hazard-free logic cover for the outputs.

Theorem 6.12 *Any solution to the binate constraints of Section 6.6 identifies a valid reduced machine $\mathcal{M}'$.*

Proof: The constraints of part 1 are precisely the state covering constraints of Grasselli [60]. Since any solution to the constraints satisfies every constraint, the selected compatibles form a state cover. Next, as observed earlier, only the transition exit points in a burst-mode machine have non-trivial state implications. Now, each transition in each reduced state has a corresponding state mapping constraint (for its exit point) in part 2, which is "enabled" when (is conditional on) selection of that reduced state. Hence, the exit point of each transition in each selected reduced state is state mapped. The constraints of part 3 ensure that only a single state mapping is selected, and that burst-mode behaviour is maintained. Finally, given the selected state mapping for any given transition, the constraints of part 5 ensure that the corresponding next-state is also selected. Thus, state closure is guaranteed. □

Lemma 6.13 *Each output required cube r' of reduced machine $\mathcal{M}'$ is the image of some required cube r of $\mathcal{M}$.*

Proof: Each required cube r' of $\mathcal{M}'$ belongs to some specified transition t' originating in some reduced state s'. Now, each transition t' of $\mathcal{M}'$ is the image in s' of some unique specified transition t of $\mathcal{M}$ originating in an unminimized state s. s is contained by the compatible c to which s' corresponds. r' belongs to either the horizontal or the vertical portion of t'. If horizontal, r' spans s' and will correspond to a unique horizontal required cube of t spanning s. If vertical, r' spans s' and some s'_2, where s'_2 is a state covering the destination state of t. Thus, r' corresponds to a unique vertical required cube of t. $\Box$

Lemma 6.14 *If DHF output prime p on $\mathcal{M}$ appears in a functional covering constraint for a required cube r' of $\mathcal{M}'$, p covers r' when instantiated under the Natural Mapping.*

Proof: r' belongs to either the horizontal or the vertical portion of some transition t'. If r' is horizontal, p appears in the given part 4 constraint iff its present-state field contains the compatible (row) c in which r' resides. Now, the existence of r' is dependent upon c's selection. Further, c's selection implies that p's present-state field will be instantiated so as to span the reduced row corresponding to c. Thus, p' contains r'. If on the other hand r' is vertical, then p appears in the part 4 constraint iff its present-state field contains the compatibles (rows) in which the source and destination states of t' reside. In this case, the existence of r' depends on the selection of the source state and the state mapping of the exit point of t' to the given destination state of t'. Thus, under the Natural Mapping, p contains both the source and destination states. So, p covers r'. $\Box$

Theorem 6.15 *Any solution to the binate constraints of Section 6.6 identifies a valid hazard-free output logic cover for the reduced machine $\mathcal{M}'$.*

Proof: This follows from the above two Lemmas, and from the fact that every output required cube for a given reduced specified transition has a covering constraint in part 4 that must be satisfied once the corresponding reduced source state has been selected and the exit point state mapped. Also, note that because the set of implicants used is the set of DHF output prime implicants, every output required cube r in $\mathcal{M}$ is covered by some p (see also [102]). As a result, every functional covering constraint has at least one DHF output prime appearing on the right-hand side, and is thus satisfiable. □

6.10.2 Optimality of the Unminimized Machine

This section demonstrates that the unminimized machine $\mathcal{M}$ itself has the best possible two-level hazard-free output logic cover, over all possible state minimizations and encodings. The burden of the proof lies in finding for every prime hazard-free cover of any given reduced machine $\mathcal{M}'$, a corresponding cover for $\mathcal{M}$ of identical cardinality that maps onto it.

Lemma 6.16 *To every DHF output prime implicant p' on some reduced machine $\mathcal{M}'$ there corresponds at least one DHF output prime implicant p on $\mathcal{M}$ which Naturally Maps onto p'.*

Proof: By construction. For $p' = \langle I\ PS'\ O \rangle$, construct the product $p = \langle I\ PS\ O \rangle$, where PS is the union of all the compatibles corresponding to states s' in PS'. The analysis used in Lemma 5.7 for the synchronous case applies here, proving both that p' is an implicant of $\mathcal{M}'$ and that p Naturally Maps onto p'. However, it must also be shown that p is dynamic hazard-free for $\mathcal{M}$.

The specified transitions t in $\mathcal{M}$ map one-to-many onto specified transitions t' in $\mathcal{M}'$. Further, if output o undergoes a transition of a certain type (static-0, static-1, $0 \rightarrow 1$, $1 \rightarrow 0$) for t in $\mathcal{M}$, o undergoes the same type of transition for all corresponding t' in $\mathcal{M}'$. So, a dynamic transition t (for output o) in $\mathcal{M}$ induces one or more dynamic transitions t' (for output o) in $\mathcal{M}'$. Now, recall that all dynamic output transitions are horizontal for a plain burst-mode specification with early output changes.

Consider a dynamic transition t' and the corresponding transition t in $\mathcal{M}$. t' originates in some reduced state s'; t originates in some unreduced state s. t' has a privileged cube pc' spanning only s', while t has a privileged cube pc spanning only s. Note that s lies in the compatible corresponding to s'.

Now, p' is DHF for $\mathcal{M}'$ (given) ; hence p' does not illegally intersect pc'. I.e., either p' does not intersect pc' at all, or p' contains the start point of pc'. More specifically, there are three cases:

1. p' contains start(pc') $\Rightarrow$ p contains start(pc)

2. p' does not intersect $IN(pc')$ $\Rightarrow$ p does not intersect $IN(pc)$

3. p' does not span s' $\Rightarrow$??? p does not span s ???

In the first two cases, we see that p is DHF with respect to transition t.

In the last case, we cannot immediately conclude that p does not span s even though p' does not span s'. In particular, if state s was split, p' may span another state s'_2 which corresponds to a compatible containing s. However, we can make the following claim: If cases 1 or 2 above apply to one t'/s', then they apply to *all* t'/s' corresponding to t/s, and therefore p is DHF for t. Otherwise, for p' to be DHF, it must be DHF for all transitions in $\mathcal{M}'$, and case 3 must apply to every t'/s'. If that is true, then p' does not span *any* s' corresponding to s, so that p does not span s, and again p is DHF for t. $\Box$

From the above, given any output logic cover Π' for some reduced machine $\mathcal{M}'$, we can construct, member-wise, a set of implicants (call it Π) on $\mathcal{M}$.

Lemma 6.17 Π *is a hazard-free output cover for* $\mathcal{M}$*.*

Proof: Π consists solely of DHF output implicants, and hence is dynamic hazard-free. We need only prove that Π covers all required cubes of all transitions in $\mathcal{M}$.

For every specified transition t' of $\mathcal{M}'$ there is a corresponding transition t of $\mathcal{M}$. Conversely, for every t there is at least one t'.

The required cubes of t' fall into two categories: horizontal and vertical. Horizontal required cubes span only a single state s'; vertical required cubes span both s' and some destination state s'_2. For each, there is a corresponding horizontal (resp. vertical) required cube r of t spanning s (resp. s and some s_2). Note that s' (resp. s'_2) corresponds to a compatible containing s (resp. s_2).[4]

Now, for any horizontal required cube r' of t', there is a $p' \in \Pi'$ that contains it. It is easy to see that the corresponding $p \in \Pi$ contains r.

For any vertical required cube r' of t', there is a $p' \in \Pi'$ containing it. r' spans s' and some s'_2, while r spans s and some s_2. Since state s' covers s and state s'_2 covers s_2 (given a valid state mapping), p contains both s and s_2. Clearly, then, p contains r.

In short, if Π' covers the required cubes of $t' \in \mathcal{M}'$, then Π covers the required cubes of the corresponding t. Since every t has a corresponding t', and Π' covers all t', then Π covers all $t \in \mathcal{M}$ as well. $\Box$

Theorem 6.18 *The unminimized machine $\mathcal{M}$ possesses a two-level hazard-free output logic cover which has minimum cardinality across all possible state minimizations and encodings.*

Proof: This follows from Lemma 6.17, since, for every hazard-free output logic cover Π' for any reduced machine $\mathcal{M}'$, we can construct a hazard-free output cover for $\mathcal{M}$ of identical cardinality. $\Box$

[4] For s_2/s'_2, this is true because $\mathcal{M}'$ must embody a valid state mapping as well as a valid state cover.

6.10.3 Optimality of OPTIMISTA

The proof here is straightforward. Because any solution to the binate constraints represents a valid reduced machine and hazard-free output logic cover, we need only establish that $\mathcal{M}$ itself is a minimum-cost solution to our constraints.

Lemma 6.19 *The unminimized machine $\mathcal{M}$, along with any minimum-cardinality hazard-free output logic cover $\mathcal{C}$ for it, represent a minimum-cost solution to the binate constraints of Section 6.6.*

Proof: Clearly $\mathcal{M}$ satisfies the state covering constraints of part 1. $\mathcal{M}$ also satisfies the state mapping constraints of part 2. In particular, the clauses corresponding to unselected compatibles are trivially satisfied. Now, the remaining part 2 clauses correspond to the original specified transitions in each state (the singleton compatibles). A clause for transition t from state s_{src} to s_{dst} always has s_{dst} as a candidate state mapping. Hence, the clause is satisfied by selecting $\delta_{t,s_{\mathrm{src}},s_{\mathrm{dst}}}$; that is, by mapping t's exit point to s_{dst} (which is precisely what $\mathcal{M}$ does). Next, $\mathcal{M}$ makes a single choice for each state mapping, and so satisfies the constraints of part 3. Also, having selected $\delta_{t,s_{\mathrm{src}},s_{\mathrm{dst}}}$ and s_{dst}, $\mathcal{M}$ also satisfies the closure constraints of part 5. Finally, the functional covering constraints of part 4 are monotonic, so that there is always a solution, as long as there exists an implicant to cover any given output minterm. Because we make use of *all* DHF output primes formed on $\mathcal{M}$, there is always a prime to cover any given required cube (see [102]). Likewise, all possible DHF prime covers are accessible as solutions to the constraints of part 4. Hence, $\mathcal{M}$ and the given logic cover $\mathcal{C}$ are a solution to the binate constraints. That $\mathcal{M}$ and $\mathcal{C}$ constitute a minimum-cost solution follows directly from the cost function (logic cover cardinality) in use. □

Theorem 6.20 *A minimum-cost solution to the constraints of Section 6.6 identifies a reduced machine having an exactly minimum-cardinality two-level hazard-free logic cover* *for its outputs.*

Proof: This result follows from Theorem 6.18 (the optimality of the unminimized machine), along with the Lemma 6.19 above. □

6.11 Efficient Generation and Pruning of State Mapping Incompatibility Constraints

It is possible to form recursive versions of Algorithms 9 and 10 which state map transitions "on demand". Such algorithms incrementally form a covering DHF implicant (if possible) for a given required cube by expanding the required cube to form a candidate covering implicant, and stopping once the implicant is determined to be dynamic hazard-free. As they do so, they state-map transitions on-the-fly *only in the region of interest*, i.e., in the input columns spanned by the candidate implicant. As a result, they avoid examining state mapping alternatives for transitions that are disjoint from the region of interest, which can have no impact on hazard-free covering of the given required cube. Appendix B presents a pair of such algorithms.

6.12 Experimental Results

This section presents experimental results obtained using the current implementation of OPTIMISTA. First, the experimental set-up is described, and then the results are shown and analyzed.

Results were obtained for a suite of industrial benchmark burst-mode specifications. All circuits come from the same suite used in Chapter 7.6 to evaluate MINIMALIST.

Each experiment was run as follows. First, the burst-mode specification is translated to a form that OPTIMISTA can read (a Berkeley PLA file and a transition specification file). Next, STAMINA is used to generate prime compatibles, using a PLA-to-KISS2 translator. Finally, OPTIMISTA is run, taking the PLA, transition files, and prime compatibles as input. OPTIMISTA internally uses HFMIN to generate the DHF output prime implicants, and calls SCHERZO to solve the underlying binate covering problem. For comparison, MINIMALIST is also run on the same designs, using no fed-back outputs, the output-disjoint logic implementation style, and a cost function of product count.

Table 6.1 summarizes the experimental results. First, the number of inputs, unminimized states, and outputs are shown. Next, the number of prime and all compatibles are given, along with the number of DHF output prime implicants. Then, two sets of columns describe the results from two pairs of trial runs of OPTIMISTA. The first set of the pair shows the results of minimization using only pruned prime compatibles; the second, the results of minimization using all compatibles. In each case, the number of binate constraints, minimized (chosen) states, output products, and total run time are shown.

A limited set of trial runs was performed. OPTIMISTA does as well as MINIMALIST in each case, but no better. This suggests that either there is no room for improvement in these circuits, or that MINIMALIST's state minimization happens to be doing particularly well. As seen in the MINIMALIST benchmarks of Chapter 7, however, some of the most impressive gains come from the larger designs, which offer more latitude for optimization. We expect that runs on larger designs for OPTIMISTA will turn in equally good results.

Again, as in Chapters 4 and 5, some increase in the number of minimized states is observed, even though no benefit is seen in logic cardinality. Thus, OPTIMISTA results would clearly benefit from the use of a two-tiered cost function.

6.13 Conclusions and Future Work

This chapter extends the output-targetted framework of OPTIMISTO to address optimal state minimization for burst-mode asynchronous state machines. Like the previous chap-

		compats			OPTIMISTA (prime comps)				OPTIMISTA (all comps)				MINIMALIST	
design	**i/s/o**	prime	all	DHF-PI	cons	ms	**prod**	sec	cons	ms	**prod**	sec	ms	**prod**
dme-e	3/8/3	5	28	9	87	5	4	1	2122	†	-	-	3	4
dme-fast-e	3/8/3	4	20	11	58	4	8	1	694	9	6	8	4	6
pscsi-ircv	4/6/3	3	11	10	43	3	8	1	201	7	8	1	4	8
pscsi-trcv	4/6/3	3	11	8	43	3	6	1	180	10	6	1	3	6
pscsi-tsend	4/10/3	6	15	16	77	6	10	2	187	9	10	1	7	10
pscsi-tsend-bm	4/10/4	6	14	11	72	6	9	1	171	13	9	2	7	9
sbuf-read-ctl	3/7/3	3	19	11	54	3	5	1	807	†	-	-	5	3
sbuf-send-ctl	3/8/3	4	14	11	62	-	-	-	365	6	7	1	7	4

‡ Constraint solution failed to produce a solution in a reasonable amount of time

Table 6.1: Experimental results for OPTIMISTA on industrial benchmark specifications

ter, it makes two contributions. First, it offers a binate constraint satisfaction method, which is the first optimal state minimization method for asynchronous state machines of any form. This new method precisely targets output logic, producing *exactly minimum* cardinality two-level hazard-free output logic over *all possible state minimizations, state encodings, and logic minimizations.* This is particularly advantageous for burst-mode machines, since it addresses the key performance parameter for systems of such machines: output latency. The second contribution is an interesting theoretical result, analogous to one from the previous chapter: the unminimized machine always possesses minimum-cardinality two-level hazard-free output logic.

It offers two fundamental choices for state minimization. First, it proposes the startling choice of performing *no reduction at all.* Surprisingly, this yields exactly minimum-cardinality output logic over *all* state reductions, state encodings, and two-level logic minimizations. Second, it provides a novel binate constraint-satisfaction framework, based on the framework used by OPTIMIST in the previous chapter. Interestingly, this second method has far lower computational complexity than base OPTIMIST. At the same time, it supports a set of four useful cost functions, unlike the first choice (that of performing no reduction). As a result, the constraint-satisfaction method also suits applications requiring the best possible output logic and desiring reduction in next-state logic complexity.

The constraint satisfaction algorithm was implemented, and experimental results obtained for both OPTIMISTA and MINIMALIST. While the experimental results do not demonstrate a reduction in logic complexity over MINIMALIST, we expect to see reductions in an expanded set of experiments. This is an area for future work.

The present method does not handle fed-back outputs; such an extension would be of considerable interest. In particular, this might provide better reduction in next-state complexity, while retaining an optimum output logic implementation.

Like the previous chapter, a binate solver capable of handling two-tiered cost functions would allow Optimista to accomodate other cost functions. Exploring the trade-offs in these choices is another interesting area for future work.

Chapter 7

MINIMALIST: An Extensible Toolkit and Framework for Asynchronous Burst-Mode Synthesis

Minimalist is a new extensible environment for the synthesis and verification of burst-mode asynchronous finite-state machines. Minimalist embodies a complete technology-independent synthesis path, with state-of-the-art exact and heuristic asynchronous synthesis algorithms, e.g. optimal state assignment (Chasm), two-level hazard-free logic minimization (Hfmin, Espresso-HF, and Impymin), and synthesis-for-testability.

Unlike other asynchronous synthesis packages, Minimalist also offers many options: literal vs. product optimization, single- vs. multi-output logic minimization, using vs. not using fed-back outputs as state variables, and exploring varied code lengths during state assignment, thus allowing the designer to explore trade-offs and select the implementation style which best suits the application.

Minimalist benchmark results demonstrate its ability to produce implementations with an average of 34% and up to 48% less area, and an average of 11% and up to 37% better performance, than the best existing package [150]. Our synthesis-for-testability method guarantees 100% testability under both stuck-at and robust path delay fault models, requiring little or no overhead. MINIMALIST also features both command-line and graphic user interfaces, and supports extension via well-defined interfaces for adding new tools. As such, it is easily augmented to form a complete path to technology-dependent logic.

7.1 Introduction

While asynchronous circuits have undergone a renaissance by significant renewed interest in the last decade, their promises — reduced power, increased performance, and robustness — have only begun to be fully realized [150][29][98][74][120][73][78][126][80]. Although several of these methods have been effective, several synthesis steps still lack optimal solutions or practical tools. Likewise, a lack of well-integrated and extensible environments within which to embed these tools leaves designers without a smooth synthesis path. By contrast, the synchronous community possesses a wealth of such tools and environments, both commercial (e.g., Synopsis', Cadence's and ViewLogic's) and academic [51], which benefits both researchers and end-users.

Thus, MINIMALIST makes contributions on several fronts:

- An integrated synthesis path consisting of state-of-the-art asynchronous synthesis algorithms:
  - CHASM, the first general optimal state encoding tool for asynchronous machines, providing both exact and fixed-length modes, and which can produce exactly-minimum output logic, a key parameter in asynchronous system performance
  - HFMIN, the only exact hazard-free *symbolic* two-level logic minimizer, supporting both single- and multi-output implementations

  - IMPYMIN, a new *implicit* exact hazard-free two-level logic minimizer, capable of solving all available benchmark problems in under 15 minutes, including several previously unsolvable problems
  - ESPRESSO-HF, a very fast new heuristic hazard-free two-level logic minimizer, which typically produces optimal or near-optimal results in under 3 seconds
  - Synthesis for testability, yielding 100%-testable multi-level implementations under either stuck-at or robust path delay fault models, with little or no area overhead

- In contrast to existing synthesis paths, MINIMALIST provides a single synthesis path able to produce implementations in a variety of styles (e.g., single-output vs. multi-output, using vs. not using fed-back outputs as state variables, exploring various state code lengths) under various cost functions, allowing the exploration of design trade-offs

- The first complete and practical technology-independent synthesis path for burst-mode circuits using fast optimal algorithms

- An easily-usable environment with a software framework which can readily incorporate new tools

MINIMALIST currently supports widely-used plain *burst-mode* [101][132] specifications. Extended burst-mode specifications [155] will be supported in a forthcoming release. Also, OPTIMISTA, described in Chapter 6, has not yet been integrated into the MINIMALIST synthesis path.

7.2 Background and Overview

7.2.1 Technical Comparison: Burst-Mode Synthesis Toolkits

This section briefly reviews two previous burst-mode asynchronous synthesis systems, and compares them to MINIMALIST.

The UCLOCK [102] system is a nearly complete path from plain burst-mode specifications to two-level logic. It incorporates a safe, exact state minimization algorithm, and the first exact hazard-free single-output logic minimization algorithm [105]. Unlike MINIMALIST, however, it offers no automated method for state encoding or multi-output logic minimization.[1] Further, its Lisp implementation and slow algorithms for state minimization and logic minimization severely limit its usefulness. Finally, it does not allow fed-back outputs, missing an opportunity to significantly reduce implementation complexity.

The 3D system, presented in [150][153][152], also synthesizes two-level implementations, but accepts extended burst-mode specifications — a larger class of specifications than either UCLOCK or MINIMALIST (at present) handle. Unlike UCLOCK, 3D uses fed-back outputs; unlike MINIMALIST, their use is not an option: it is required. In contrast to MINIMALIST, it uses heuristic greedy state minimization and encoding algorithms. It also always performs exact single-output logic minimization (using HFMIN [55]), to produce reasonably high-performance implementations. Even so, none of its methods (save HFMIN) offers any guarantee of optimality; benchmarks show that MINIMALIST's algorithms give better results.

Finally, whereas both UCLOCK and 3D support only a single implementation style and one cost function, MINIMALIST supports *multiple* implementation styles and cost functions. MINIMALIST thus allows designers to explore various trade-offs and choose the implementation which best suits their application.

[1] In practice, critical race-free codes for UCLOCK were produced manually, possibly using auxiliary programs.

7.2.2 Comparative Summary: MINIMALIST vs. Previous Tools

The following table provides an overview of the choices available in the most important dimensions of the solution space for MINIMALIST and the two competing burst-mode synthesis toolkits, 3D and UCLOCK. Each dimension is correlated with the relevant operating mode or tool option, which will be defined later in this chapter.

synthesis package	fed-back outputs (`machine impl. style`)	state min	code length (`constr. sat. mode`)	type of logic (`logic impl. style`)	cost function
UCLOCK	no-fed-back only	exact	one solution	single-output only	products
3D	fed-back only	heuristic	one solution	single-output only	literals†
MINIMALIST	both	both	many solutions (variable code length)	single-output, multi-output, or output-disjoint	both

† In the original 3D implementation, the sole cost function was product count.

7.3 MINIMALIST Framework

The MINIMALIST framework consists of several key pieces: core data structures, a class and algorithm library, and an extensible interpreter, implemented in roughly 45,000 lines of C++ (including several of the core tools, e.g. HFMIN and CHASM).

The MINIMALIST framework incorporates a simple set of C++ classes to represent the original burst-mode specification. Early synthesis steps such as state minimization and state encoding simply transform or place annotations on these structures. As a result, additional steps or transformations are easily accommodated.

To assist in implementing new synthesis algorithms, MINIMALIST offers class libraries for manipulating both asynchronous burst-mode specifications, two-level logic (hazard- and non-hazard-free), dichotomies, unate and binate covering problem instances, arbitrary-length bitstrings, and the like. To facilitate interfacing to external programs, a small number of basic translators to common formats (e.g. Berkeley PLA or BLIF) is incorporated.

Finally, MINIMALIST provides a shell-like interpreter, extensible with commands written in C or C++. The interpreter supports user-defined shell functions, on-line help, command- and filename-completion, variables, control constructs (loops, conditionals), arithmetic, external process invocation, input/output redirection, and the like. Also, functionality can be augmented by dynamically-loaded code, without having to re-link the executable.

The result is a uniquely flexible, potent context for integrating synthesis tools, that we find lacking in existing packages [102][150].

7.4 MINIMALIST Tools

7.4.1 State Minimization

MINIMALIST includes two new and very efficient algorithms for exact state minimization. In contrast, the 3D method uses a heuristic greedy minimization algorithm. Therefore, in this section, we will focus on a more direct comparison: the MINIMALIST exact algorithms and an earlier exact state minimization algorithm found in UCLOCK.

MINIMALIST improves significantly on UCLOCK's state minimization approach in two ways — (i) by allowing outputs to be fed back as inputs, and (ii) by dramatically reducing run-time complexity.

MINIMALIST offers two new exact state minimization algorithms: 1) for implementations *without* fed-back outputs (loosely based on UCLOCK's method), and 2) for implementations *with* fed-back outputs. The latter is the first exact algorithm for state minimization that handles fed-back outputs. Thus, it supports two **machine implementation styles**. As mentioned earlier, fed-back outputs can dramatically reduce the implementation's logic complexity.[2] In particular, their use allows merging certain states which would otherwise be incompatible. MINIMALIST makes use of this fact by relaxing UCLOCK's compatibility relation. In fact, several benchmark specifications collapse into a single state using the new relation, whereas UCLOCK's relation results in 2 or more states.

Second, MINIMALIST improves run-time by several orders of magnitude over the UCLOCK method. UCLOCK uses an expensive algorithm to generate maximal compatibles. In contrast, MINIMALIST uses a simple transformation which allows it to generate maximal compatibles using a fast unate prime generation algorithm instead. In addition, both UCLOCK and MINIMALIST (currently) approximate the binate covering step by a unate one followed by a closure check.[3] UCLOCK, however, employs Petrick's method to solve the unate problem, while MINIMALIST employs a state-of-the-art tool, MINCOV [117]. The combination of these two algorithmic enhancements reduce the run-time of state minimization by two or more orders of magnitude. To date we have encountered only one specification for which state minimization requires more than a few seconds, whereas run-times of many minutes were common for UCLOCK. For such large specifications, both of MINIMALIST's algorithms also feature an *approximate mode* which further reduces run-time.

7.4.2 CHASM

For state encoding, MINIMALIST uses CHASM, the first exact method for input encoding of multiple-input change asynchronous machines. CHASM has many operating modes. One highlight is that its "exact mode" can be used to produce *exactly optimum two-level output logic*, over *all* critical race-free encodings, thus optimizing the key performance parameter for asynchronous networks (output latency). Its approximate mode also gives

[2] See the benchmark results.

[3] We have yet to see the closure check fail, due in part to the manner in which the set of state compatibles is refined to a partition.

significant reductions in overall implementation cost.

CHASM, as described in Chapter 3, loosely follows the flow of the KISS [40] method for *input encoding* of synchronous machines. Several significant modifications are required to handle asynchronous machines. There are three steps. First, *symbolic hazard-free logic minimization* is performed. Second, a set of *encoding constraints* is generated, which properly subsumes both classic synchronous KISS ("face embedding") optimality constraints and asynchronous critical race-free [137] constraints. The constraints are in the form of generalized dichotomies [137] (not face embedding constraints). Finally, the *constraints are solved*. For constraint solution, CHASM has two modes: (i) an **exact mode**, which uses DICHOT [122], and (ii) an **approximate mode**, using NOVA's [146] simulated annealing engine. The approximate mode, as in NOVA, attempts to solve as many constraints as possible, under the restriction of a *fixed code-length*; it has the advantage that it may reduce next-state logic complexity.

For MINIMALIST, CHASM has been extended in several new ways.

First, CHASM is now being applied to implementations with fed-back outputs. Specifically, we have proven that CHASM requires no modification to properly encode implementations with fed-back outputs. A modified functional specification is provided as input, which simply identifies the primary outputs as fed-back inputs. The symbolic two-level logic minimizer then forms a symbolic cover on this new function.

Second, CHASM can now target three **logic implementation styles**: (i) *multi-output* (where outputs and next-state may share products), (ii) *output-disjoint* (where products are shared among outputs, but not between outputs and next-state), and (iii) *single-output* (where *no* product terms are shared across any outputs). The motivation is that the "single-output style" is most suitable for performance-optimal designs: each output is individually optimized. Note that, in asynchronous machines (unlike synchronous), output latency is often the key parameter to overall system performance in a network of interacting machines. The "multi-output style" is most suitable for area-optimal designs, since it uses maximal sharing of logic. Finally, the "output-disjoint style" is a balanced compromise.

For modes (ii) and (iii), a novel feature of CHASM is that it produces *exactly optimum two-level output logic*, over all critical race-free encodings. This result holds, because the optimal output-only state encoding problem is a true "input encoding problem", unlike the more general optimal state encoding problem (which is an approximation).

Finally, CHASM now targets two distinct **cost functions:** (i) *product cardinality*, and (ii) *literal count*, at the symbolic level. It does the latter by performing weighted unate covering during symbolic logic minimization. This technique is only a heuristic, however, because the literal count of the final binary cover can only be estimated. In practice, the heuristic nonetheless yields significant reduction in literal count.

7.4.3 HFMIN

After state minimization and encoding, MINIMALIST performs two-level hazard-free logic minimization. This step is normally performed using HFMIN, the first exact multi-output symbolic hazard-free two-level logic minimization tool.

HFMIN, as reported in [55], uses ESPRESSO to generate ordinary prime implicants, then refines them as needed to dynamic hazard-free (DHF) primes [105], and finally, performs a unate covering step using MINCOV [118], covering *required cubes*[105] in lieu of minterms. HFMIN's use of such highly-optimized algorithms for sub-steps allows it to readily handle most minimization problems we have encountered.

HFMIN has also been enhanced for MINIMALIST with the ability to produce output-disjoint[4] and single-output covers, and the ability to minimize literal count. Output-disjoint and single-output covers are formed by generating a suitable set of prime implicants before DHF refinement takes place. The rest of the algorithm proceeds unchanged.

To minimize literal count, HFMIN performs a weighted unate covering step where

[4]products are shared among outputs, but not between outputs and next-state

prime implicants are assigned weights according to their literal count. In addition, HFMIN now supports a limited post-processing step that further reduces literal count. This step is similar in spirit to the *make-sparse* operation of ESPRESSO [118]. A single pass is made over each selected prime implicant, removing output literals as long as the result remains a valid (hazard-free) cover. The input portion is then expanded, if possible. Unlike *make-sparse*, our current method makes no guarantee that the resulting product is maximally expanded. Nonetheless, despite its simplicity, the operation often yields significant reductions in literal count.

HFMIN is now widely used in several other burst-mode CAD packages, including the 3D method [150] and ACK [74]. It has also been used as part of the asynchronous tool suite at Intel Corporation, where it has been applied in the design of a high-speed experimental asynchronous Instruction-Length Decoder (see [21]).

7.4.4 ESPRESSO-HF

For very large problems which HFMIN is unable to solve in reasonable time, MINIMALIST offers ESPRESSO-HF [134], a new fast heuristic two-level logic minimizer. ESPRESSO-HF uses an algorithm loosely based on ESPRESSO (but substantially different from it), to solve problems with up to 32 inputs and 33 outputs. On benchmark examples, ESPRESSO-HF can find very good covers — at most 3% larger than a minimum-size cover — in less than 105 seconds. For typical examples, ESPRESSO-HF obtains an exact or near-exact result in under 3 seconds.

Currently, ESPRESSO-HF only implements multi-output minimization targetting product cardinality, and so it is normally used only for designs which exceed HFMIN's capacity. However, output-disjoint or single-output minimization can easily be implemented by a trivial modification of the code, and will be available in future releases of MINIMALIST.

7.4.5 IMPYMIN

Impymin [135] is a new state-of-the-art fully-implicit exact two-level hazard-free logic minimizer. It greatly exceeds the capacity of previous exact tools (e.g. Hfmin), minimizing very large multi-output functions, including some for which no exact result had previously been obtained. Run-times are typically under 16 seconds. The most difficult problem available, with 32 inputs and 33 outputs, had never before been solved exactly, but required only 813 seconds using Impymin.

Both Espresso-HF and Impymin can solve all currently-available benchmark examples, including several which have not been previously solved. For larger examples that can be solved by Hfmin, these two minimizers are typically several orders of magnitude faster.

7.4.6 Synthesis-for-Testability

Minimalist incorporates a recent method [107] for synthesis-for-testability targetting multi-level logic. The method produces circuits that are both hazard-free and 100% testable under either stuck-at or robust path delay fault models, with little or no overhead. First, it uses a novel two-level hazard-free logic minimization algorithm which minimizes the number of redundant cubes, as well as the number of non-prime cubes. (The tool currently operates only in single-output mode.) This helps maximize testability without using additional inputs. If not yet completely testable, the circuit is converted to a multi-level form which is completely testable (if possible). If still not completely testable (rarely the case), controllable inputs are added, yielding 100% testable logic. Finally, hazard- and testability-preserving multi-level transformations are used to reduce the area of the resulting circuit. The area overhead is typically zero, and in all cases is less than 10% [107].

7.4.7 Verifier

MINIMALIST features a simple tool to verify that a given logic implementation (produced by any method) realizes the specified burst-mode behaviour, independent of the particular state merges or encoding which have been performed. The verifier simulates the implementation's response to each specified input burst, and compares it to the specification; any mismatches of output or state at the end of the burst and any logic hazards are reported. Although each burst is considered only once, the analysis that is performed accounts for all possible interleavings of individual input changes. Since the verifier needs to traverse each edge in the specification's state graph only once, this tool is eminently practical even for very large specifications — the time required is never more than a few seconds. Currently, verification of only two-level AND/OR implementations is provided. However, the framework allows for verification of multi-level implementations as well (using the 9-valued algebra developed by Kung [75]), which will be completely supported in the near future. Also, the current version of the verifier does not detect output changes made after *partial* input bursts. Finally, being a purely combinational verifier, it does not verify the one-sided timing constraints also needed to ensure correct operation [19]. Such a capability may be added in a future release.

7.5 A Synthesis Session

A transcript of a typical synthesis session follows. For more details on the MINIMALIST command set, see Appendix C.

```
$ MinShell                                   # Start the MINIMALIST shell
minimalist> help                                # Show list of commands
    assign-states          break         call             cd
         continue         define         echo           expr
              for           help           if  impymin-logic
    make-testable      min-logic   min-states            pwd
```

```
           quit        read-spec              set     set-encoding
  set-state-cover   show-encoding      show-logic           source
     verify-logic           while      write-flow    write-instant
       write-spec   write-symbolic     write-trans
minimalist> pwd                                 # Show current directory
/u/minimalist/demo
minimalist> ls                                                 # Run 'ls'
bin  ex   lib
minimalist> ls ex
dme-e        dram-ctrl    pe-send-ifc  scripts      stetson
dme-fast-e   hp-ir        pscsi        scsi-iccd92
minimalist> cd ex/dme-e                    # Move to another directory
minimalist> ls
dme-e.bms
minimalist> help read-spec                 # Show syntax of 'read-spec'
        read-spec [-v] <file> [<spec-var>]
Read the Burst-Mode specification in <file> and store it in <spec-var>,
or, if <spec-var> is not specified, 'theSpec'.
minimalist> read-spec dme-e.bms      # Read the Burst-Mode specification
Specification passed validity check.
Specification has 3 inputs, 3 outputs, and 8 states.
minimalist> plot_graph -b dme-e.bms
[... a window displays the burst-mode graph; press Ctrl-Q to dismiss ...]
minimalist> min-states                       # Perform state minimization
*** Performing state minimization... ***
State cover: { { S0 S1 S4 }, { S2 S3 }, { S5 S6 S7 } }
Machine has 3 states after minimization.
minimalist> assign-states -F           # Assign states with a CRF encoding
No encoding style specified; defaulting to critical race-free
*** Performing state assignment... ***
Invoking 'chasm' as 'chasm -C DME_E-F.func DME_E-F.trans'
*** Machine encoded by CHASM ***
State S0': 11
```

```
State S1': 10
State S2': 00
minimalist> min-logic -F -L                  # Produce the logic implementation
*** Performing logic minimization... ***
Invoking 'hfmin -P -C -l -S -o DME_E-FL.sol DME_E-FL.pla DME_E-FL.btrans'
*** Final PLA ***
# PLA file for machine DME_E-FL
.i 8
.o 5
.ilb LIN RIN UIN LOUT_i ROUT_i UOUT_i Y0 Y1
.ob y0 y1 LOUT ROUT UOUT
.type fr
.p 7
10-----1 00010
-01----1 00010
-0-----1 01000
10-----0 10100
00----1- 11000
--0---1- 10000
-01----0 00001
.e
Number of products:       7                   Total lits:               28
Number of products implementing outputs:   4   Lits in output products:  17
Lits in products implementing next-state: 16   Avg lits per output: 5.66667
Result stored in 'DME_E-FL.sol'.
minimalist> verify-logic -F -L                       # Verify the implementation
Using logic implementation stored in 'DME_E-FL.sol'.
*** Starting Burst-Mode machine simulation. ***
*** Implementation verified successfully! ***
minimalist> plot_graph -p DME_E-FL.sol       # Display the implementation
[... a window displays the 2-level logic; press Ctrl-Q to dismiss ...]
                                   # Run the above synthesis steps using a script
minimalist> source ../scripts/script-FBO.MOL-CRF dme-e.bms
```

```
[... same results as above, without interaction ...]
minimalist> quit
```

7.6 Experimental Results

This section compares synthesis results using MINIMALIST to those using the preeminent burst-mode asynchronous synthesis paths, namely 3D [150] and UCLOCK [102]. We highlight MINIMALIST's unique ability to support various cost metrics and implementation styles by showing several different experiments. For each, we indicate the cost function which we target, and the corresponding settings of MINIMALIST's modes.

7.6.1 Experimental Set-up

The benchmark suite consists of 23 burst-mode circuits, including several industrial designs, as well as a number of large asynchronous machines (e.g., see `sc-control` and `oscsi`). The circuits `it-control`, `rf-control`, `sc-control`, and `sd-control` are part of a low-power infrared controller designed at HP Labs as part of the Stetson project [86]. `pe-send-ifc` and `sbuf-xxx-ctl`, also from HP Labs, are part of a high-performance adaptive routing chip, used in the Mayfly parallel processing system [132]. Several others (e.g. the `scsi-xxx` and `pscsi-xxx` suites) come from a high-performance asynchronous SCSI controller designed by Yun while at AMD [154]. A DRAM controller circuit for Motorola 68K processors [150], `dram-ctrl`, completes the suite.

All MINIMALIST results are the best of a very small number of trials using fixed-length encodings. Generally, near-minimum code lengths are used. Here, minimum length refers to the smallest length sufficient for a critical-race free encoding, which is necessary to ensure correct operation. However, as demonstrated below, a trade-off exists, whereby significantly wider encodings sometimes offer better output logic at the expense of added next-state logic complexity. Hence, the results below occasionally make use of somewhat longer codes.

Run-times for the complete synthesis path are comparable for all tools (MINIMALIST, 3D, and UCLOCK), ranging from under 1 second to several minutes for the largest designs.

7.6.2 Performance-Oriented Comparison with 3D

The first experiment (shown in Table 7.1) shows synthesis results using both MINIMALIST and 3D for a **performance-oriented implementation**.

For asynchronous burst-mode machines, the metric that best approximates performance is *output latency*. In an asynchronous system, the input-to-output latency typically determines a machine's performance, as well as overall system performance. State changes are not bound to a clock period, and in practice are usually non-critical (see [86]).

Based on the above observation, we now indicate the settings of the various modes of MINIMALIST for this experiment. In the context of technology-independent two-level logic, the *cost function* that most reasonably approximates output latency is the average number of literals per output. When comparing two results for the same machine (so that the number of outputs is fixed), this cost is equivalent to **total output literal count**, so this is used instead.

Roughly half of the MINIMALIST results in this set of runs make use of the **fed-back output** machine implementation style. Table 7.1 identifies the particular style chosen for each design in the column labelled '`FB`'. MINIMALIST is directed to use the **single-output** logic implementation style, and the **literal count** cost function. This combination of modes best minimizes average output literal count. This cost function also allows for a fair comparison to 3D, which produces single-output logic with minimal literal count. Finally, the encoding step uses **fixed-length constraint satisfaction** mode, attempted under several code lengths.

The MINIMALIST script in Figure 7.1 summarizes the selected modes. The script

```
define syn_FBO_so_lit { specFile codeLen } { # Single-output, optimize lit count
   read-spec $specFile
   min-states -F
   assign-states -F -s -L -O -l $codeLen
   min-logic -F -s -L
   verify-logic -F -s -L
}
call syn_FBO_so_lit dram-ctrl.bms $codelength
```

Figure 7.1: Performance-oriented synthesis script for MINIMALIST

is parameterized by code length, using the variable `$codeLen`, and proceeds as follows. First, the specification is read from a file and checked for validity. Next, the machine is subjected to exact state minimization using fed-back outputs.[5] The states are encoded using CHASM, with the fed-back output ('`-F`'), single-output ('`-s`'), literal-count ('`-L`'), and fixed-length ('`-l`') flags. The final two-level logic is then synthesized using HFMIN, again passing the single-output and literal-count flags. Finally, the resulting logic is verified using the algorithm sketched in Section 7.4. The script was run in batch mode several times. The run having the lowest output literal count over 1-3 code lengths near the minimum is reported.

The 3D results were obtained using the 3D tool on the Unix platform. Unlike MINIMALIST, 3D embodies a hard-wired synthesis path, and produces a single deterministic result. Specifically, 3D first performs heuristic state minimization, followed by heuristic state encoding, and finally, exact two-level single-output logic synthesis using HFMIN, targetting total literal count. Thus, a single run for each design gives the reported (and the only possible) result.

Table 7.1 summarizes the comparison. MINIMALIST synthesis results demonstrate an average reduction of 11% in *output literals*, with the best being a 37% reduction for `sd-control`. This savings is frequently accompanied by a reduction in *total literal count*

[5]A nearly identical script exists in which all steps do not make use of fed-back outputs.

as well, with larger designs such as `stetson-p1` and `oscsi` among the most impressive gains (38% and 25%, respectively). However, in exchange for MINIMALIST's simplification in output logic, an increase in total literal count is occasionally observed (see for example `pe-send-ifc` and `pscsi-tsend`).

Clearly, MINIMALIST's gains come in part from its ability to explore wider encodings. In fact, in 12 of the 23 designs, the best result is seen at a longer code length than 3D uses. The greatest improvement overall, in `sd-control`, is seen at a *significantly* longer code length — 7 bits vs. 3D's 3 bits. For two designs, MINIMALIST chooses a shorter encoding than 3D: `dram-ctrl` (whose 0-bit encoding is enabled by the use of fed-back outputs), and `oscsi` (where 3D curiously uses 7 bits, despite no gain in output literals and a considerable increase in overall logic complexity). For the remaining examples, MINIMALIST achieved its best result at the same code length as 3D.

7.6.3 Area-Oriented Comparison with 3D

Our second experiment (also shown in Table 7.1) shows the results of an **area-oriented comparison** of MINIMALIST and 3D.

The cost metric that best approximates area for technology-independent two-level logic is total literal count; hence, *total literal count* is used in this comparison.

Based on the above observation, we now indicate the settings of the various modes of MINIMALIST for this experiment. The vast majority of the MINIMALIST results in this set of runs use the **fed-back output** machine implementation style. Again, the table identifies the particular style chosen for each design. Throughout, MINIMALIST is directed to use the **multi-output** logic implementation style, and the **literal count** cost function, which best minimizes total literal count. Finally, the encoding step uses **fixed-length constraint satisfaction** mode.

These runs were obtained using a script identical to that of Figure 7.1, but using the multi-output logic implementation style. In particular, the single-output (`'-s'`) flag

Table 7.1: A comparison of MINIMALIST and 3D synthesis results

		3D				MINIMALIST								
						performance (single-output)					area (multi-output)			
design	i/s/o	sbit	prod	**lit**	**outlit**	FB	sbit	prod	lit	**outlit**	FB	sbit	prod	**lit**
dram-ctrl	7/12/6	1	21	71	57	√	0	17	57	57	√	0	14	51
pscsi-ircv	4/6/3	2	13	44	30		3	14	46	26*		3	10	38*
pscsi-trcv	4/6/3	1	10	35	30		2	13	43	27	√	1	7	30
pscsi-isend	4/9/3	4	31	105	44		6	31	111	38	√	3	18	67
pscsi-tsend	4/10/3	4	22	77	41		7	31	116	35	√	3	18	66
pscsi-trcv-bm	4/7/4	2	18	58	45	√	2	17	58	43	√	2	11	45
pscsi-tsend-bm	4/10/4	3	24	86	43		7	31	118	43	√	3	16	63
sbuf-read-ctl	3/7/3	1	7	22	17		2	9	26	15	√	1	6	21
sbuf-send-ctl	3/8/3	2	17	51	32		3	14	41	24		2	11	36
pe-send-ifc	5/11/3	2	24	89	56		5	32	136	55	√	3	18	88
scsi-isend-bm	5/10/4	2	26	87	59	√	2	23	76	56	√	2	17	63
scsi-isend-csm	5/8/4	2	22	66	44		2	18	53	41		2	12	49
scsi-trcv-bm	5/10/4	2	24	78	50	√	2	22	71	50	√	2	17	64
scsi-trcv-csm	5/8/4	2	21	64	42		2	18	52	40		2	12	46
scsi-tsend-bm	5/11/4	2	28	92	67		4	24	86	48		3	19	75
scsi-tsend-csm	5/10/4	2	20	57	41		6	24	75	34	√	2	14	52
it-control	5/10/7	1	21	73	56	√	1	19	68	56	√	1	13	54
rf-control	6/12/5	2	12	44	34		5	15	67	30	√	2	11	43
sc-control	13/33/14	3	122	512	301	√	3	93	359	275	√	3	53	267
sd-control	8/27/12	4	50	217	155	√	7	43	160	98	√	4	28	125
stetson-p1	13/33/14	3	87	371	209	√	3	61	232	179	√	3	45	199
stetson-p2	8/25/12	4	46	194	148	√	4	45	177	130	√	4	30	140
oscsi	10/45/5	7	129	529	187	√	4	89	395	185	√	4	65	334
Total		58	795	3022	1788		82	703	2623	1585		55	465	2016
Diff wrt 3D		-	-	-	-		+41%	-12%	-13%	-11%		-5%	-42%	-33%

was removed from the state encoding and logic minimization steps. Again, the cost function used was total literal count.

As shown in Table 7.1, MINIMALIST's term-sharing across outputs and next-state provides for significant reductions in total area. MINIMALIST's results for the area-targetted run show an average reduction of 33% in total literal count over 3D, the best being 48% for `sc-control`. For all designs, MINIMALIST achieved strictly better results than 3D. Although these runs did not target product count directly, they offer similarly dramatic reductions by that metric as well. An average of 42% improvement is observed, the best being 57% for `sc-control`. Again, MINIMALIST's results are strictly better than 3D in every case.

Unlike the performance-targetted runs, the code length used by MINIMALIST rarely exceeded that of 3D (only 3 times out of 23 designs), and never by more than 1 bit. In fact, MINIMALIST uses slightly *fewer* total state bits over the entire benchmark suite than does 3D, by roughly 5%.

7.6.4 Area-Oriented Comparison with UCLOCK

The final comparison, in Table 7.2, shows synthesis results for UCLOCK (as reported in [55]) and MINIMALIST.

For a fair and interesting comparison, we plugged some of the MINIMALIST tools into the UCLOCK path, to isolate and highlight two differences: (i) *machine implementation style* (choice of fed-back vs. no fed-back outputs), and (ii) *state minimization algorithms*. Even though UCLOCK does not use any optimal state assignment algorithms, we attached CHASM and HFMIN as a back end, to isolate these front-end differences. We also limited MINIMALIST to the only logic minimization modes that are available in UCLOCK: the cost function is **product cardinality**, and the logic implementation style is **multi-output**.

Table 7.2 shows the experimental results. In both MINIMALIST and the "improved"

UCLOCK, reported results are the best of several fixed-length trials at or near the minimum code length. The majority of the MINIMALIST results use the **fed-back output** machine implementation style.

Not surprisingly, many MINIMALIST and UCLOCK results are nearly identical, since the operating modes are very similar. However, MINIMALIST's use of fed-back outputs achieves significant gains in several cases (e.g., `dram-ctrl` and `scsi-isend-bm`). In addition, MINIMALIST obtains synthesis results in several cases where UCLOCK failed to complete, again due in part to MINIMALIST's more capable state minimization method.

A performance-oriented comparison to UCLOCK (like the above comparison to 3D) is possible, but is omitted, due to space considerations.

7.6.5 Optimal Encoding for Output-Targetted Synthesis

The experiment shown in Table 7.3 explores the effect on MINIMALIST results of ignoring the optimality of next-state. In particular, CHASM is instructed to form an encoding so as to ensure a critical race-free implementation, while imposing no further constraint on the next-state implementation. This has the benefit of allowing the constraint satisfaction engine to concentrate attention on the optimality of the output logic. In this mode, the input encoding model, which precisely models output logic, is used to best advantage.

For comparison, a "base" set of runs was also performed, in which the only difference from the aforementioned runs is that CHASM is instructed to generate optimality constraints for next-state as usual. In both sets of runs, the cost metric used is total literal count, which most nearly approximates output literal count.

The settings of the various MINIMALIST modes for this experiment are as follows. Most of the MINIMALIST results in this set of runs use the **fed-back output** machine implementation style. Once again, the table identifies the particular style chosen for each design. Throughout, MINIMALIST is directed to use the **output-disjoint** logic implementation style, and the **literal count** cost function, which best minimizes total literal

		UCLOCK			MINIMALIST			
design	in/state/out	sbit	**prod**	lit	FBO	sbit	**prod**	lit
dram-ctrl	7/12/6	2	22	-	$\surd$	0	14	71
pscsi-ircv	4/6/3	2	9	-		2	9	41
pscsi-trcv	4/6/3	1	10	-	$\surd$	1	7	32
pscsi-isend	4/9/3	3	17	-	$\surd$	3	17	80
pscsi-tsend	4/10/3	3	18	-	$\surd$	3	16	86
pscsi-trcv-bm	4/7/4	2	12	-	$\surd$	2	12	53
pscsi-tsend-bm	4/10/4	†	†	-	$\surd$	3	16	84
sbuf-read-ctl	3/7/3	2	7	-	$\surd$	1	6	23
sbuf-send-ctl	3/8/3	2	11	-		2	11	47
pe-send-ifc	5/11/3	3	18	-	$\surd$	3	18	118
scsi-isend-bm	5/10/4	2	21	-	$\surd$	2	15	92
scsi-isend-csm	5/8/4	2	12	-	$\surd$	2	12	62
scsi-trcv-bm	5/10/4	2	18	-		2	18	99
scsi-trcv-csm	5/8/4	2	12	-		2	12	61
scsi-tsend-bm	5/11/4	3	17	-		3	17	101
scsi-tsend-csm	5/10/4	2	14	-	$\surd$	2	13	77
it-control	5/10/7	3	15	-	$\surd$	1	13	61
rf-control	6/12/5	3	13	-	$\surd$	2	11	45
sc-control	13/33/14	†	†	-	$\surd$	4	56	458
sd-control	8/27/12	5	29	-	$\surd$	4	28	182
stetson-p1	13/33/14	4	53	-	$\surd$	3	42	317
stetson-p2	8/25/12	4	31	-		4	28	173
oscsi	10/45/5	†	†	-	$\surd$	4	64	487
Total		52	359+???	-		55	325+136	200
Diff wrt UCLOCK		-	-	-		+5.8%	-9.5%	-

† Failed to complete within a reasonable time

Table 7.2: An area-oriented comparison of Minimalist and Uclock

count. For the "base runs" (the middle columns of Table 7.3), the **complete** constraint generation mode was used, while for the experimental runs (the right-hand columns of Table 7.3), the **output-only** constraint generation mode was used. Finally, for those machines for which **exact constraint satisfaction** requires a longer code length than that required for a critical race-free implementation, the encoding step uses **fixed-length constraint satisfaction** mode. For all other machines, the exact constraint satisfaction mode was used.

These runs were obtained using a script identical to that of Figure 7.1, but using the output-disjoint logic implementation style. In particular, the single-output (`'-s'`) flag was replaced by the output-disjoint (`'-d'`) flag in the state encoding and logic minimization steps. Also, in the experimental runs, CHASM was passed the no-next-state (`'-S'`) flag. Again, the cost function used was total literal count.

As shown in Table 7.3, neither method has a clear advantage. Although bypassing next-state optimality constraints achieves a noticeable average gain in next-state literal count, the distribution is varied. For some designs, the base case wins (see `pscsi-trcv-bm`), while for others, the trials with no next-state optimality constraints wins (see `pscsi-isend` or `pscsi-tsend`). However, in those cases where NONSOPT wins by a significant margin, the key difference was in fact the use of fed-back outputs.

7.6.6 Exploring Varying Code Lengths

This section briefly shows the effect on the synthesis results for a single design when varying code length in MINIMALIST.

Because a simple cost metric often fails to capture an application's cost completely, MINIMALIST better assists the designer in finding the point which best fits the application, by providing the opportunity to explore trade-offs. For example, Table 7.4 shows an interesting trade-off involving code length, arising from two competing tendencies. Output logic tends to improve, while next-state logic tends to grow, with increasing code

Table 7.3: Effect of focusing optimality constraints on output logic in MINIMALIST

		MINIMALIST "base"						**MINIMALIST "NONSOPT"**					
				next-state		outputs				next-state		outputs	
design	in/state/out	FBO	sbit	prod	**lit**	prod	**lit**	FBO	sbit	prod	**lit**	prod	**lit**
pscsi-ircv	4/6/3		3	6	20	8	26		3	8	24	8	26
pscsi-trcv	4/6/3		3	7	24	6	21		3	7	24	6	21
pscsi-isend	4/9/3		7	18	70	11	38	√	4	10	35	11	39
pscsi-tsend	4/10/3		6	17	68	10	35	√	4	7	26	10	37
pscsi-trcv-bm	4/7/4	√	2	6	19	9	35		3	7	25	9	34
pscsi-tsend-bm	4/10/4		6	16	62	9	36	√	3	10	36	9	36
sbuf-read-ctl	3/7/3		2	4	11	5	15		2	4	11	5	15
sbuf-send-ctl	3/8/3		3	6	17	7	23		3	6	17	7	23
pe-send-ifc	5/11/3		3	14	60	11	50		4	14	59	11	49
scsi-isend-bm	5/10/4	√	2	6	20	15	51	√	2	6	20	15	51
scsi-isend-csm	5/8/4		3	6	19	12	39		2	6	16	12	41
scsi-trcv-bm	5/10/4	√	2	7	24	13	48	√	2	7	21	13	48
scsi-trcv-csm	5/8/4		2	4	12	11	38		2	4	12	11	38
scsi-tsend-bm	5/11/4		4	10	40	13	46		4	9	45	13	46
scsi-tsend-csm	5/10/4		6	10	46	10	31		5	11	39	10	31
it-control	5/10/7	√	1	3	12	11	45	√	1	3	12	11	45
rf-control	6/12/5		5	10	37	7	30		3	9	31	7	32
sd-control	8/27/12	√	6	16	60	21	92	√	7	14	56	21	86
stetson-p2	8/25/12	√	4	14	47	25	118	√	5	16	54	21	88
Total			70	180	668	214	817		62	158	563	210	786
Diff wrt base		-	-	-	-	-	-		-14%	-12%	-16%	-2%	-4%

design	i/s/o	**sbits**	outprods	**outlits**	nsprods	**nslits**	totprods	**totlits**
scsi-tsend-csm	5/10/4	**3**	14	37	9	24	23	61
"	"	**4**	14	36	7	26	21	62
"	"	**5**	13	35	10	34	23	69
"	"	**6**	13	34	11	41	24	75

Table 7.4: Effect of varying code length on synthesis results for a single design

length.

For these runs, the design `scsi-tsend-csm` is synthesized using the performance-oriented script, but targetting the *fed-back output* machine implementation style. Code length is varied from 3 (the minimum needed to ensure a critical race-free encoding) to 6 (one less than the code length resulting when CHASM uses its exact, rather than its fixed-length, mode).

As the results show, the output logic complexity decreases as the code length increases, in exchange for a more expensive next-state implementation. This reflects the fact that CHASM'S input encoding model is exact for outputs, but is only approximate for next-state. Specifically, the fixed-length constraint satisfaction method favors neither output nor next-state constraints [55]. Thus, longer codes tend to satisfy a greater number of *both* kinds of constraints. So, output logic complexity decreases, because the corresponding constraints precisely model logic optimality. However, the next-state, whose constraints are less accurate, experiences an increase in logic complexity.

Without the ability to explore such trade-offs, a designer is forced to choose whatever single point in the solution space the synthesis path chooses. For example, suppose a synthesis path always chose to minimize output literal count. Given the results in Table 7.4, this would force the designer to accept an 8% decrease in output logic complexity, in exchange for a 71% increase in next-state logic complexity, which might be intolerable. MINIMALIST's ability to explore such trade-offs is unique among burst-mode synthesis toolkits.

7.7 Conclusion

Minimalist distinguishes itself in several respects. First, it integrates a suite of state-of-the-art algorithms for asynchronous burst-mode synthesis. Second, benchmark results demonstrate the effectiveness of the synthesis path on a large number of examples. Third, its support for multiple implementation styles and cost functions allows it to accommodate a variety of applications. In particular, the Minimalist tool chain provides well-optimized implementations using fed-back outputs which are guaranteed correct. Finally, its software framework provides both a potent end-user environment and the extensibility to allow it to encompass technology-dependent synthesis and other down-stream tasks. In short, Minimalist represents a first-of-a-kind environment for asynchronous synthesis, with significant contributions in algorithms, quality of results, and extensibility.

Chapter 8

Conclusions

We now give a synopsis of the main contributions of the thesis, and describe some promising avenues for future research and CAD tool development.

This thesis makes two fundamental contributions to the synthesis of asynchronous burst-mode controllers. First, it offers a trio of synthesis and optimization algorithms for asynchronous machines: HFMIN, CHASM, and OPTIMISTA. Each of these is a first-of-a-kind algorithm, embodied in a useful CAD tool that is applicable to a wide range of practical burst-mode designs. Experimental results indicate a significant improvement over existing algorithms. In their output-only modes, all three also guarantee exactly optimum two-level output logic. In all of its modes, CHASM is exact under the input encoding model.

Second, the thesis offers an extensible CAD package, called MINIMALIST, which includes a complete technology-independent asynchronous synthesis path. Coupled with HFMIN, CHASM and tools for state minimization and verification, MINIMALIST constitutes a uniquely flexible and potent environment for burst-mode synthesis and verification. MINIMALIST also incorporates new tools such as IMPYMIN (a fast implicit two-level hazard-free logic minimizer), ESPRESSO-HF (a fast heuristic two-level hazard-free logic minimizer) and synthesis-for-testability. Experimental results demonstrate MINIMAL-

IST's strength as a synthesis system.

Additionally, this thesis contributes new algorithms which address two previously-unsolved problems in *synchronous* sequential synthesis. Both algorithms solve the *optimal state minimization problem* for synchronous finite state machines. The first algorithm, OPTIMIST, represents the first exact solution to optimal state minimization under an input encoding model. The second algorithm, OPTIMISTO, which focuses on output logic, achieves exactly minimum-cardinality output logic over all state minimizations, encodings, and two-level logic minimizations. These results are truly exact, independent of any encoding model. Although developed as a foundation for the asynchronous method, both algorithms provide significant reductions in logic cardinality for certain machines. An interesting theoretical result was also introduced, which proves that the unminimized machine itself possesses exactly minimum-cardinality output logic. This result answers a previously open research question. As such, the unminimized machine can be used directly in applications where output logic complexity is the primary cost metric. (A similar result for asynchronous burst-mode machines is derived in the presentation on OPTIMISTA.)

Several areas remain for future work.

First, state encoding and state minimization (as embodied in CHASM and OPTIMISTA, respectively) would both benefit from more powerful symbolic hazard-free logic minimization methods. Specifically, the symbolic minimization methods should be made to target technology-dependent realizations (e.g. complex-gate logic), as well as more general multi-level logic implementations. Such methods would produce logic that targets certain cost functions more accurately. For example, multi-level logic typically has an advantage in area over two-level logic, while complex gate implementations often exhibit better performance or lower power than two-level logic.

Second, the MINIMALIST package as a whole would be even more widely applicable if it accepted a larger class of asynchronous specifications. One particularly useful class of specifications is known as *extended burst-mode* (XBM) [153]. XBM specifications are more expressive than the "plain" burst-mode specifications addressed by this thesis. In particular, XBM provides two additional features: (i) sampling signal *levels* (cf. waiting for input events), and (ii) *concurrent* input and output changes. XBM's additional features necessitate modular changes to each of MINIMALIST's algorithms. For instance, MINIMALIST's state minimization method would need to be modified to ensure that state merges do not introduce hazards for XBM's unique transition types. CHASM's symbolic hazard-free two-level logic minimization and constrained encoding steps must likewise be modified to handle these more general XBM transitions. However, HFMIN requires no modification; it is already in use in the 3D synthesis path for XBM designs.

Third, MINIMALIST's value would be significantly increased by incorporating timing analysis and more stringent functional verification methods. Timing analysis is particularly important for asynchronous systems using burst-mode, because they operate in *fundamental mode*. That is, they make assumptions regarding the environment's response time. Thus any timing constraints imposed on the environment by the given burst-mode controller must be calculated, and surrounding circuitry must be checked for compliance. Again, these additions are easily incorporated in modular fashion into MINIMALIST's extensible framework.

Finally, most of the algorithms presented herein (most notably OPTIMIST and OPTIMISTA) would be significantly improved by faster implementations based on implicit methods. Implicit methods use binary decision diagrams (BDD's) [16] to efficiently represent and manipulate very large sets of objects. The term "implicit" here derives from the use of a characteristic function of a set as an implicit representation of that set. Implicit methods have been very successfully applied to classic state minimization [70] and two-level logic minimization [30], among other key CAD problems. The use of implicit algorithms has been shown to greatly reduce run-time, while simultaneously enlarging the class of problems that can be solved practically.

Appendix A

Multiple-Valued Hazard-free Logic Minimization

A.1 Multiple-Valued Functions and Hazards

This section presents a theoretical formulation for the exact minimization of two-level hazard-free logic with symbolic inputs.

The following material assumes basic familiarity with the terminology of multiple-valued logic minimization (refer to [118]).

A.2 Circuit Model

This formulation considers combinational circuits having arbitrary finite gate and wire delays (*unbounded wire delay model* [106]). A pure delay model is assumed (see [139]).

A.3 Multiple-Valued Multiple-Input Changes

This section generalizes the notions of multiple-input changes and transition cubes from the binary domain [106] to the multiple-valued domain.

Definition A.1 (Multiple-valued transition cube) *A* ***multiple-valued transition cube*** *is a cube with a* ***start point*** *and an* ***end point****. Let A and B be two minterms in a multiple-valued domain D. The multiple-valued transition cube, denoted as $[A, B]$, from A to B has start point A and end point B and contains all minterms that can be reached during a transition from A to B. More formally, if A and B are described by products, with i-th literals $A_i^{S_{A_i}}$ and $B_i^{S_{B_i}}$, respectively, then the i-th literal for the product of $T = [A, B]$ is the literal $T_i^{S_{A_i} \cup S_{B_i}}$.*

Definition A.2 (Multiple-valued open transition cube) *The* ***multiple-valued open transition cube*** *$[A, B)$ from A to B is defined as: $[A, B] - \{B\}$.*

Definition A.3 (Multiple-valued input transition) *A* ***multiple-valued input transition*** *or* ***multiple-valued multiple-input change*** *from input state A to B is described by transition cube $[A, B]$.*

An *intermediate state*$X \in [A, B]$ is potentially reachable during the input transition from A to B if for all variables X_i, the corresponding literal X_i is either equal to A_i or B_i. A multiple-input change specifies what variables are permitted to change value and what the corresponding *starting* and *ending* values are. Input variables are assumed to change simultaneously. (Equivalently, since inputs may be skewed arbitrarily by wire delays, inputs can be assumed to change monotonically in any order and at any time.) Once a multiple-input change occurs, no further input changes may occur until the circuit has stabilized. An input transition occurs during a *transition interval*, $t_I \le t \le t_F$, where inputs change at time t_I and the circuit returns to a steady state at time t_F.

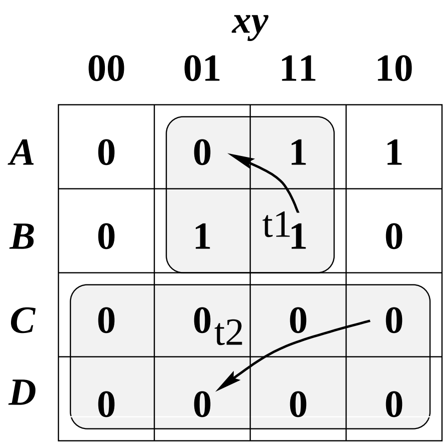


Figure A.1: A multiple-valued input function and two multiple-valued transitions

Definition A.4 (Static and dynamic transitions) *An input transition from input state A to B for a multiple-valued function f is a* ***static transition*** *if $f(A) = f(B)$; it is a* ***dynamic transition*** *if $f(A) \neq f(B)$.*

Example A.5 Figure A.1 depicts an mvi function with a symbolic variable over the set $S = \{A, B, C, D\}$ and two mvi transitions. One, t_1, is dynamic; the other, t_2, is static. The corresponding multiple-valued transition cubes are also shown.

In this framework, we consider only static and dynamic transitions where f is fully defined; that is, for every $X \in [A, B]$, $f(X) \in \{0, 1\}$.

A.4 Multiple-Valued Function Hazards

A function f which does not change monotonically during an input transition is said to have a *function hazard* in the transition.

Definition A.6 (Static function hazard) *A multiple-valued function f contains a* ***static function hazard*** *for the input transition from A to C if and only if: (1) $f(A) = f(C)$, and (2) there exists some input state $B \in [A, C]$ such that $f(A) \neq f(B)$.*

	xy			
	00	**01**	**11**	**10**
A	**0**	**0**	***1***	**1**
B	**0**	**0**	t1 **0**	**0**
C	**0**	**0** t2	***1***	**0**
D	**0**	**1**	**0**	**0**

Figure A.2: A multiple-valued function exhibiting both static and dynamic function hazards

Definition A.7 (Dynamic function hazard) *A multiple-valued function f contains a* ***dynamic function hazard*** *for the input transition from A to D if and only if: (1) $f(A) \neq f(D)$; and (2) there exist a pair of input states, B and C, such that (a) $B \in [A, D]$ and $C \in [B, D]$, and (b)$f(B) = f(D)$ and $f(A) = f(C)$.*

Example A.8 Figure A.2 illustrates both static and dynamic function hazards for a multiple-valued function. Static-0 transition t_1 passes through a transient 1 value at total state $\langle A, 11\rangle$, while dynamic transition t_2 passes through a transient 1 value at total state $\langle C, 11\rangle$.

If a transition has a function hazard, no multiple-valued implementation of the function is guaranteed to avoid a glitch during the transition, assuming arbitrary gate and wire delays. Therefore, we consider only transitions which are free of function hazards (cf. [106]).

A.5 Multiple-Valued Logic Hazards

If f is free of function hazards for a transition from input A to B, an implementation may still have hazards due to possible delays in the logic realization. Here, we extend notions of static and dynamic logic hazards to multiple-valued functions. To do so, we will provide these definitions in terms of an abstract **multiple-valued sum-of-products implementation**. That is, each multiple-valued product term in the multiple-valued cover is implemented as a single *multiple-valued AND gate*. The circuit output is implemented as a *Boolean OR gate* that combines the AND gates.

Definition A.9 (Static (Dynamic) logic hazard) *A multiple-valued cover circuit implementing multiple-valued function f contains a **static (dynamic) logic hazard** for the input transition from minterm A to minterm B if and only if: (1) $f(A) = f(B)$ ($f(A) \neq f(B)$), and (2) for some assignment of delays, the circuit's output is not monotonic during the transition interval.*

That is, a static logic hazard occurs if $f(A) = f(B) = 1$ (0), but the circuit's output makes an unexpected $1 \rightarrow 0 \rightarrow 1$ ($0 \rightarrow 1 \rightarrow 0$) transition. A dynamic logic hazard occurs if $f(A) = 1$ and $f(B) = 0$ ($f(A) = 0$ and $f(B) = 1$), but the circuit's output makes an unexpected $1 \rightarrow 0 \rightarrow 1 \rightarrow 0$ ($0 \rightarrow 1 \rightarrow 0 \rightarrow 1$) transition.

A.6 Problem Abstraction

The hazard-free multiple-valued minimization problem can now be stated as follows. Given a multiple-valued function f, and a set, T, of *specified* function-hazard-free multiple-valued (static and dynamic) input transitions of f, find a minimal-cost *multiple-valued cover* of f that is free of logic hazards for every specified input transition $t \in T$.

A.7 Symbolic Hazard-Free Minimization

In this section, we present the exact minimization algorithm for producing a hazard-free multiple-valued cover used by HFMIN. While the standard multiple-valued minimization problem *without* considerations for hazards has been adequately addressed before [118], the corresponding problem in the context of asynchronous synthesis and hazard-free synthesis has not yet been addressed. We first state the conditions that the cover must satisfy in order to ensure hazard-freeness. These conditions will lead to a notion of **multiple-valued dynamic-hazard-free (DHF-) prime implicants**. Using these *prime implicants*, a *constrained* covering step must be solved to select a hazard-free cover. These issues are addressed in the sequel.

A.7.1 Conditions for a Hazard-Free Transition

We now describe conditions to ensure that a sum-of-products implementation is hazard-free for a given input transition. Assume that $[A, B]$ is the transition cube corresponding to a *function-hazard-free* transition from input state A to B for a multi-valued combinational function f. In the following discussion, we assume that C is any multi-valued cover of f. The following lemmas present necessary and sufficient conditions to ensure that a multi-valued AND-OR implementation of f has *no logic hazards* for the given transition. The following results are extensions from the binary case [106].

Lemma A.10 *If f has a $0 \rightarrow 0$ transition in cube $[A, B]$, then the implementation is free of logic hazards for the input change from A to B.*

Lemma A.11 *If f has a $1 \rightarrow 1$ transition in cube $[A, B]$, then the implementation is free of logic hazards for the input change from A to B*if and only if*$[A, B]$ is contained in some cube of cover C.*

The conditions for the $0 \rightarrow 1$ and $1 \rightarrow 0$ cases are symmetric. Without loss of generality, we consider only a dynamic $1 \rightarrow 0$ transition, where f(A)=1 and f(B)=0. (A $0 \rightarrow 1$ transition from A to B has the same hazards as a $1 \rightarrow 0$ transition from B to A.)

Lemma A.12 *If f has a $1 \rightarrow 0$ transition in cube $[A, B]$, then the implementation is free of logic hazards for the input change from A to B*if and only if *every cube $c \in C$ intersecting $[A, B]$ also contains A.*

Lemma A.11 requires that in a $1 \rightarrow 1$ transition, *some* product holds its value at 1 throughout the transition. Lemma A.12 ensures that no product will glitch *in the middle* of a $1 \rightarrow 0$ transition: all products change value monotonically during the transition. In each case, the implementation will be free of hazards for the given transition.

An immediate consequence of Lemma A.12 is that, if a dynamic transition is free of logic hazards, then every static sub-transition will be free of logic hazards as well:

Lemma A.13 *If f has a $1 \rightarrow 0$ transition from input state A to B which is hazard-free in the implementation, then, for every input state $X \in [A, B)$ where $f(X) = 1$, the transition subcube $[A, X]$ is contained in some cube of cover C.*

Lemma A.14 *If f has a $1 \rightarrow 0$ transition from input state A to B which is hazard-free in the implementation, then for every input state $X \in [A, B)$ where $f(X) = 1$, the static $1 \rightarrow 1$ transition from input state A to X is free of logic hazards.*

Lemmas A.11 and A.13 are used to define the covering requirement for a hazard-free transition. The cube $[A, B]$ in Lemma A.11 and the *maximal* subcubes $[A, X]$ in Lemma A.13 are called *required cubes*. These cubes define the ON-set of the function in a transition. Each required cube *must* be contained in some cube of cover C to ensure a hazard-free implementation. This property can be more formally stated as follows.

Definition A.15 (Required cube) *Given a multiple-valued function f, and a set, T, of specified function-hazard-free multiple-valued input transitions of f, every cube $[A, B] \in$*

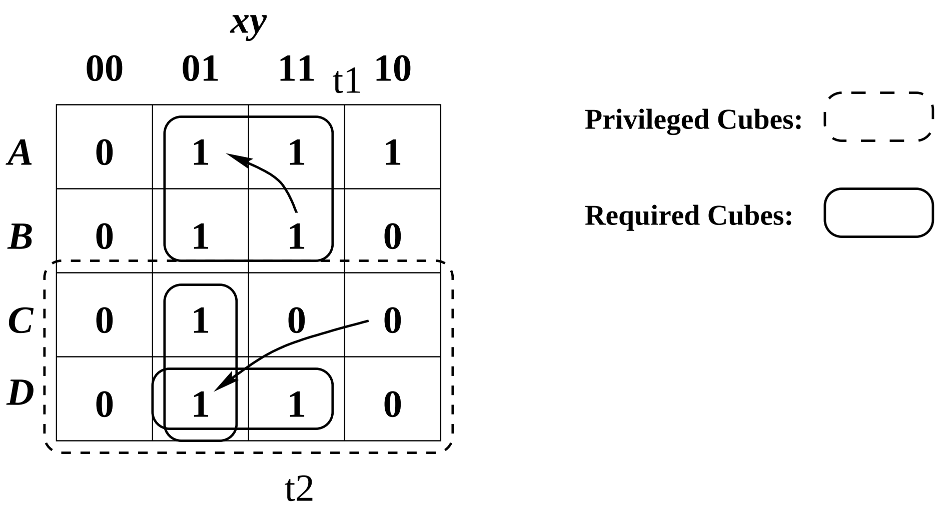


Figure A.3: Transitions in a multiple-valued function and their privileged and required cubes

*T corresponding to a static $1 \rightarrow 1$ transition, and every maximal subcube $[A, X] \subset [A, B]$ where f is 1 and $[A, B] \in T$ is a dynamic $1 \rightarrow 0$ transition, is called a **required cube**.*

Lemma A.12 constrains the cubes which may be included in a cover C. Each $1 \rightarrow 0$ transition cube is called a *privileged cube*, since no cube c in the cover may intersect it unless c contains its *start point*. If a cube intersects a privileged cube but does not contain its start point, it *illegally intersects* the privileged cube and may not be included in the cover. This property can be more formally stated as follows.

Definition A.16 (Privileged cube) *Given a multiple-valued function f, and a set, T, of specified function-hazard-free multiple-valued input transitions of f, every cube $[A, B] \in T$ corresponding to a dynamic $1 \rightarrow 0$ transition is called a **privileged cube**.*

Example A.17 Figure A.3 shows a multiple-valued function and two transitions, along with their required and privileged cubes.

A.7.2 Hazard-Free Covers

A *hazard-free cover* of function f is a cover of f whose multi-valued AND-OR implementation is hazard-free for a *given set* of specified input transitions. The following theorem describes all hazard-free covers for function f for a set of multiple-input transitions. (It is assumed below that the function is defined for all specified transitions; the function is undefined for all other input states.)

Theorem A.18 [Hazard-free covering]*A sum-of-products C is a hazard-free cover for function f for all specified input transitions if and only if:*
(a.) No cube of C intersects the OFF-set of f;
(b.) Each required cube *of f is contained in some cube of C; and*
(c.) No cube of C intersects any privileged cube *illegally.*

Conditions (a) and (c) in Theorem A.18 determine the implicants which may appear in a hazard-free cover of a Boolean function f. Condition (b) determines the covering requirement for these implicants in a hazard-free cover. Therefore, Theorem A.18 precisely characterizes the covering problem for hazard-free two-level logic.

In general, the covering conditions of Theorem A.18 may not be satisfiable for an arbitrary Boolean function and set of transitions (cf. [139, 106]). This case occurs if conditions (b) and (c) cannot be satisfied simultaneously.

A.7.3 Exact Hazard-Free Multiple-Valued Minimization

Many exact logic minimization algorithms, such as Espresso-MV-Exact [118], are based on the Quine-McCluskey algorithm [91]. The Espresso-MV-Exact algorithm solves the two-level multiple-valued minimization problem in three steps:

1. Generate multiple-valued prime implicants;

2. Construct prime implicant table;

3. Generate minimum cover of this table.

Here, we extend an existing exact *hazard-free* two-level minimizer [106] to multi-valued functions. Theorem A.18(a) and (c) determine the implicants which may appear in a hazard-free cover of a multiple-valued function f. Such implicants will be called *multiple-valued dynamic-hazard-free (DHF-) implicants*. They are defined as follows:

Definition A.19 (Multiple-valued DHF-implicants) *A* ***multiple-valued DHF-implicant*** *is an implicant which does not intersect any privileged cube of f illegally. A* ***multiple-valued DHF-prime implicant*** *is a multiple-valued DHF-implicant contained in no other multiple-valued DHF-implicant. An* ***essential multiple-valued DHF-prime implicant*** *is a multiple-valued DHF-prime implicant which contains a required cube contained in no other multiple-valued DHF-prime implicant.*

By Theorem A.18(c), **only multiple-valued DHF-implicants may appear in a hazard-free cover**. Theorem A.18(b) determines the covering requirement for a hazard-free cover of f: **every required cube of f must be covered**, that is, contained in some cube of the cover. Thus, the *two-level hazard-free logic minimization problem* is *to find a minimum cost cover of a function using only multiple-valued DHF-prime implicants where every required cube is covered.*

The modified hazard-free multiple-valued minimization algorithm is as follows:

1. Generate multiple-valued DHF-prime implicants;
2. Construct multiple-valued DHF-prime implicant table;
3. Generate a minimum cover of this table.

These steps are detailed below.

A.7.4 Generation of Multiple-Valued DHF-Prime Implicants

Multiple-valued DHF-prime implicants for function f are generated in two steps. The new algorithm follows the approach described in [106], but extended to multiple-valued functions. First, ordinary multiple-valued prime implicants of f are generated from the required cubes (which defines the ON-set) and the OFF-set, using existing algorithms [118]. Second, these prime implicants are transformed into multiple-valued DHF-prime implicants by iterative refinement. The new algorithm, *mvi-PI-to-DHF-PI*, checks each implicant p for illegal intersection with any multiple-valued privileged cube, q. If such an intersection occurs, the implicant is reduced in all possible ways to avoid intersection. In particular, p is replaced by the set $\{p_1, \ldots, p_n\}$ of maximal subcubes of p which do not intersect q (*i.e.*, $\forall i \in \{1, \ldots, n\}$, $p_i \cap q = \phi$). Note that, in the multi-valued framework, reduction is uniformly performed across both input and output spaces. Such reduced implicants may have remaining, or new, illegal intersections with other privileged cubes. The process continues until only DHF-implicants remain. Non-prime DHF-implicants are removed by a check for single-cube containment. Likewise, empty products, a possibility after certain reductions, are removed as well.

In addition to HFMIN's basic multi-output operating mode, HFMIN also supports several modes that provide somewhat different logic structures. For example, *single-output minimization* produces logic covers in which no product contributes to more than one output. Likewise, *output-disjoint minimization* produces covers in which each product may implement either one or more binary outputs, or the symbolic next-state, but not both. This variety of modes permits exploration of trade-offs between area and performance.

These additional modes are implemented by simply restricting the set of multiple-valued prime implicants fed to the iterative refinement step. In particular, for single-output minimization, mv-primes are generated for each output in isolation, and the sets of primes are combined in the obvious way afterward. Likewise, for output-disjoint minimization, primes for the binary outputs are generated in one batch, and symbolic next-state primes in another, and results are again combined. In all modes, the iterative refinement process itself, as well as all subsequent steps, remain unchanged.

A.7.5 Generation of the DHF-Prime Implicant Table

A *multiple-valued DHF-prime implicant table* is constructed for the given function. The rows of the table are labelled with the *multiple-valued DHF-prime implicants*. The columns are labelled with the *required cubes*, which must be covered. The table sets up the two-level hazard-free logic minimization problem.

To perform literal optimization, the rows of the table (corresponding to the DHF primes) are assigned integral weights. The cost of a given set of DHF primes is simply the sum of the weights of the individual primes.

The weight assigned to each DHF prime is as follows:

$$\mathrm{cost}(p) = \#\text{ of non}-\text{full input literals in } p + \#\text{ of } 1'\text{s in output field of } p$$

This formula captures the number of inputs to the AND gate corresponding to the DHF prime p, along with the number of inputs to OR gates to which p is connected.

Example A.20 Implicant $\langle 01--,\ 101\rangle$ for a 3-output function on B^4 has cost $2{+}2 = 4$.

For multiple-valued variables, the cost used approximates the number of non-full binary literals after encoding that mvi variable. Specifically, the weight of a multiple-valued literal $X_i^{S_{X_i}}$ is $(|S_{X_i}| - S_i)$, where S_i is the set describing the domain of X_i. This expresses the fact that the more values X_i spans, the closer to a don't-care it is, and the fewer binary literals in the encoded representation will be required. E.g., an mvi field containing all 1's corresponds to a don't-care, and is assigned a 0 weight. Conversely, a field containing a single 1 will require non-full literals in all N bits of the encoded form.

Example A.21 Implicant $\langle 01--, 101101, 101\rangle$ for a 3-output function on $B^4 \times S$, where $S = \{0, \ldots, 5\}$, has cost $2 + (6 - 4) + 2 = 6$.

A.7.6 Generation of a Minimum Cover

The multiple-valued DHF-prime implicant table describes a standard unate covering problem. It is solved using an existing algorithm, MINCOV [118].

For a simple heuristic covering, HFMIN exposes the *quasi-exact* mode of MINCOV, which returns the first solution found, rather than iterating until a minimum-cost cover is found. This mode has proven useful in certain cases.

A.7.7 Multiple-Output Minimization

As in ESPRESSO-MV-EXACT [118], *multiple-output functions* are handled by making the output parts into a single *N-valued MV variable*, where N is the number of outputs. The transformation is straightforward and is described in [118]. Using this transformation, the symbolic hazard-free multiple-valued minimization procedure can be used to minimize multiple-output functions.

Appendix B

Efficient Generation of State Mapping Incompatibility Constraints

This appendix presents a pair of optimized recursive algorithms used by OPTIMISTA to generate state mapping incompatiblity constraints more efficiently than the naive algorithms given in Chapter 6.

It is possible to form recursive versions of Algorithms 9 and 10 which state map transitions "on demand". Such algorithms incrementally form a covering DHF implicant (if possible) for a given required cube by expanding the required cube to form a candidate covering implicant, and stopping once the implicant is determined to be dynamic hazard-free. As they do so, they state-map transitions on-the-fly *only in the region of interest*, i.e., in the input columns spanned by the candidate implicant. As a result, they avoid examining state mapping alternatives for transitions that are disjoint from the region of interest, which can have no impact on hazard-free covering of the given required cube. Appendix B presents a pair of such algorithms.

The recursive algorithms have several advantages over their naive counterparts. First, the brute-force enumeration of state mapping combinations is largely avoided, so that generation is computationally less expensive. Second, the number of constraints generated is significantly smaller, which eases constraint solution. Finally, each constraint identifies a (related) class of invalid state mapping combinations. Consequently, each constraint has fewer disjunctive terms (i.e., it state maps fewer specified transitions). By comparison, the naive algorithm generates constraints each of which identifies a single invalid combination (which state maps all transitions in one or more reduced states).

It is important to realize that the set of constraints generated by the recursive algorithms is exactly equivalent to the set generated by the previous "naive" algorithms, however, it is more compact. In effect, each constraint generated by the recursive algorithms may correspond to *several* constraints as generated by the naive algorithms.

As in the "naive" case, the optimized recursive generation of these constraints is accomplished by a pair of algorithms. The first algorithm examines hazard-free covers for required cubes belonging to the *horizontal* portion of a specified transition. The second algorithm does the same, but for required cubes in the *vertical* portion of a specified transition. The following sections present the two algorithms in turn.

B.1 Horizontal Required Cubes

A recursive version of Algorithm 9 appears in two parts as Algorithms 11 and 12. As described above, this version maps transitions "on demand", so as to limit the binding of state mappings to the region of interest.

The outermost two loops of this algorithm are identical, as they identify the specified transitions in the various reduced machine rows.

In addition to on-demand state mapping, this algorithm makes use of another key observation: the location and shape of the required and privileged cubes for each unstable mapping of a given specified transition are identical. Thus, the algorithm can determine the suitability of *all* unstable mappings in a single analysis. When the unstable mappings of a given transition are determined to be invalid (in combination with other mappings), a single constraint is generated that effectively disallows each unstable mapping.

Once a specified transition in the reduced machine is identified, its required cubes are determined, and for each, a call to `horizontalRecur()` is made. This recursive routine is the heart of the algorithm, and determines whether the given required cube can be covered. In essence, it maintains a candidate covering implicant, which is expanded as needed to avoid illegal intersections with privileged cubes for transitions in the source state. The initial candidate implicant is the required cube itself.

Algorithm 11 Optimized identification of incompatible state mappings with respect to horizontal required cubes

```
identifyHorizontalIncompatibilities()      // Recursive optimized version
{
   // N.B.: The only required cubes in the horizontal portion of a transition
   // belong to the source state itself.
   for each compatible c/s' {
      TransitionList T := {transitions t | PS(t) ∈ c};
      for each transition t in T {
         // Map t to define the required cubes that must be covered.
         // We must consider both stable and unstable mappings of t.

         // In the recursion, we examine only transitions that can be unstably
         // mapped, since they are the only transitions which can interfere
         // with HF covering. T' contains exactly this set.
         TransitionList T' := T − {t} − {transitions t' | t' can be unstably mapped};

         if t can be stably mapped
            horizontalRecur(s', r, φ, T');// Cover the sole required cube

         if t can be unstably mapped {
            for each horizontal required cube r of t { // Maximal sub-cubes
               // Start off the recursion by trying to use r itself to cover r,
               // and expand as necessary (and possible) to avoid illegal
               // intersections. Note that r, {t}, and T' satisfy
               // horizontalRecur()'s entry conditions.
               horizontalRecur(s', r, {t}, T');
            }
   } } }
}
```

Algorithm 12 Part II of Algorithm 11

```
// This version is optimized knowing that req'd/priv cubes in the horizontal
// portion of specified transitions are completely defined given whether the
// transition is un/stably mapped. All unstable mappings are equivalent.
//
horizontalRecur(State s', Cube p, TransitionList T~, TransitionList T)
   //   s': the reduced state having some horizontal required cube r
   //   p: an implicant covering the required cube r of state s'
   //   T~: a set of transitions originating in s' which are unstably mapped
   //   T: unmapped transitions originating in s' which are not yet mapped
{
   // Entry conditions:
   // - all transitions in T~ are unstably mapped
   // - p does not intersect the OFF-set of any transition in T~
   // - p contains start pt of all t ∈ T~ (hence no illegal xsections wrt T~)
   // - no transition in T has been mapped
   for each transition t ∈ T intersecting p { // p illegally xsects t?
      // When t is stably mapped, it contributes neither priv cube nor OFF-set
      // pts for s', and hence can't interfere with HF covering. So, consider
      // only unstably mapping t. All unstable mappings of t have the same
      // OFF-set/priv cubes.  N.B.: Every t ∈ T  can be unstably mapped.

      TransitionList T' := T − {t}; // We handle t, one way or another.
      TransitionList T~' := T~ ∪ {t}; // Add t to unstably-mapped transitions.
      Cube p' := p;

      // N.B.: t is mapped unstably, so it has a privileged cube for s'.
      if p' does not contain start(t) {  // An illegal intersection
         p' := supercube(p', start(t));  // Make p' contain the start pt of t.
      } // else p' contains start(t), hence no illegal intersection.

      // Two things may have happened:
      //   - p' was expanded, and only now hits OFF-set of a mapped transition.
      //   - t is freshly mapped (unstably), and p' intersects the OFF-set of t.
      // Either way, we must now check for an OFF-set intersection.
      // N.B.: Optimize OFF-set test knowing whether p was actually expanded:
      //    Yes: check intersection with t's exit pt. No: check all t~ ∈ T~'.
      if hitsOFFset(p', s', T~') {
         // Generate constraint to disallow any set of mappings that unstably
         // maps all t~ ∈ T~'. Mapping any single t~ stably does the trick.
         generate the constraint c̄ + { Σ_{t∈T~'} δ_{t,c,c} }
         // N.B.: Mapping sets more specific than this can't be HF; don't recur.
      } else {
         // p' doesn't hit the OFF-set of, nor illegally intersect the priv
         // cube of, any mapped transition. So, the entry conditions hold.
         horizontalRecur(s', p', T~', T');
}  }  }
```

B.2 Vertical Required Cubes

A recursive version of Algorithm 10 appears as Algorithm 13, Algorithm 14, and Algorithm 16. Like its horizontal counterpart, it maps transitions "on demand", so as to limit the binding of state mappings to the region of interest.

This algorithm is driven by first mapping transitions in the destination state. This is based in part on the observation that there is no need to examine combinations of source-state transition mappings. In particular, since source-state transitions contribute only OFF-set points for s'_d, they never cause the expansion of the candidate implicant p. Moreover, if the candidate implicant p intersects the OFF-set of s'_d, then it must intersect OFF-set points of at least one mapped transition, say, t. Clearly, then, it would hit t's OFF-set points regardless of the state mappings of any other source-state transition.

The algorithm thus expands p as needed to avoid illegal intersections, and incrementally state-maps destination-state transitions that intersect p. As it goes, the algorithm performs OFF-set intersection tests with respect to the mapped transitions in the destination state. When there are no remaining illegal intersections, a final OFF-set intersection test is performed by state mapping each intersecting transition in the source state.

Like the optimized horizontal algorithm, this algorithm also makes use of the equivalence of all unstable state mappings with respect to hazard-free covering.

Algorithm 13 Optimized identification of incompatible state mappings with respect to vertical required cubes

```
identifyVerticalIncompatibilities()      // Top-level "driver" routine
{
    for each compatible c/s' {
        TransitionList T_s := { transitions t | PS(t) ∈ c};
        for each transition t in T_s {
            TransitionList T'_s := T_s − {t};
            // Only unstable mappings result in vertical required cubes.
            // Each mapping yields 1 required cube for a unique next-state.
            for each unstable mapping δ_{t,c,c'} of the exit point of t {
                MappingList M_s := {δ_{t,c,c'}};
                s'_d := the reduced (next-) state corresponding to c';

                // In the recursion, stably-mapped transitions originating
                // s'_d never interfere with HF covering.  Hence, we examine
                // only transitions in s'_d which can be unstably mapped.  T_d
                // contains exactly this set.
                TransitionList T_d :=
    {t | PS(t) ∈ c' and t has an unstable mapping};

                // There is a single vertical required cube r, for s'_d,
                // spanning both s' and s'_d. Start off the recursion by
                // trying to use r itself to cover r, and expand as
                // necessary (and possible) to avoid illegal intersections.
                checkDestMappings(s'_d, r, M_s, T'_s, T_d);
            }
    } } }
```

Algorithm 14 Part II of Algorithm 13

```
checkDestMappings(State s'_d, Implicant p, MappingList M_s, TransitionList T_s,
                  MappingList M_d, TransitionList T_d)
   //   s'_d: the reduced state having some horizontal required cube r
   //   p: an implicant covering the required cube r of state s'_d
   //   M_s: the state mapping choice made for the root transition in s'
   //   T_s: unmapped transitions originating in s'
   //   M_d: state mapping choices made for transitions originating in s'_d
   //   T_d: unmapped transitions originating in s'_d
{
   // Entry conditions:
   //   - There exists a set of mappings of the transitions in T_s for which
   //     p does not hit any transition's OFF-set.
   //   - p does not intersect the OFF-set of any mapped transition in M_d
   //   - p does not illegally intersect any mapped transition
   //   - no transition in T_s (resp. T_d) has been mapped

   // The following examines transitions in T_d and mappings in M_d.  It
   // has been optimized knowing the kinds of transitions for next-state
   // function s_d that are possible when starting in present-state s_d.
   pick some t ∈ T_d which intersects p;

   // Does p illegally intersect the privileged cube of t?
   TransitionList T'_d := T_d − {t}; // Take care of t, one way or another.

   // Do a recursion using T'_d without mapping t, which explores mappings of
   // subsets of T_d which do not constrain t.
   checkDestMappings(s'_d, p, M_s, T_s, M_d, T'_d);

   // If t is stably mapped, it will contribute neither a priv cube nor OFF-set
   // pts for s'_d, and hence can not interfere with hazard-free covering.  We
   // thus examine only unstable mappings of t.  Now, all unstable mappings of t
   // are equivalent wrt hazard-free covering, because they all share the same
   // privileged cubes and start points.  Hence, we need not examine each one
   // individually, but rather perform a single analysis which effectively
   // considers them all.
   //
   // N.B.: t̂ is "root" transition in s' whose required cube we're covering.

   MappingList M'_d := M_d ∪ {δ_{t,c,c'}}; // c' represents any unstable mapping of t

   // ...continued...
```

Algorithm 15 Part II of Algorithm 13, completed

```
    // checkDestMappings(), continued...
    //
    // N.B.: M'_d maps t unstably, so t has a privileged cube for s'_d.
    if p does not contain start(t) {        // Must expand p
       Cube p' := supercube(p, start(t));    // Make p' contain start pt of t.

       // At this point, p' does not illegally intersect any transition in M'_d

       if hitsOFFset(p', s'_d, M'_d) {   // Full-blown xsection test with all of M'_d
          if t cannot be mapped stably // Always a problem with t
```

$$\text{generate constr. } \overline{\delta_{\widehat{t},s',s'_d}} \vee \left\{ \sum_{\delta_{t,s'_d,*} \in M_d} \delta_{t,s'_d,s'_d} \right\} \text{ // Don't constrain } t$$

```
          else                               // Must map t to s'_d to avoid problems
```

$$\text{generate constr. } \overline{\delta_{\widehat{t},s',s'_d}} \vee \left\{ \sum_{\delta_{t,s'_d,*} \in M_d} \delta_{t,s'_d,s'_d} \right\} \vee \delta_{t,s'_d,s'_d}$$

```
       } else {                  // p does not hit the OFF-set of anything in M'_d
          if not checkSrcMappings(s'_d, p', M_s, T_s, M'_d)
             // checkSrcMappings() indicates a problem with M'_d itself, so bypass
             // the recursive call.
          else
             checkDestMappings(s'_d, p', M_s, T_s, M'_d, T'_d);
       }
    } else {                  // p contained start(t), hence no illegal intersection.
       if hitsOFFset(p, s'_d, t) { // Simpler test: M_d still ok since p unexpanded
          if t cannot be mapped stably      // Always a problem with t
```

$$\text{generate constr. } \overline{\delta_{\widehat{t},s',s'_d}} \vee \left\{ \sum_{\delta_{t,s'_d,*} \in M_d} \delta_{t,s'_d,s'_d} \right\} \text{ // Don't constrain } t$$

```
          else                                   // Must map t to s'_d to avoid problems
```

$$\text{generate constr. } \overline{\delta_{\widehat{t},s',s'_d}} \vee \left\{ \sum_{\delta_{t,s'_d,*} \in M_d} \delta_{t,s'_d,s'_d} \right\} \vee \delta_{t,s'_d,s'_d}$$

```
       } else {                  // p does not hit the OFF-set of anything in M'_d
          // Unlike the above, p was not expanded to avoid illegal intersection,
          // so there is no need to check for OFF-set intersections with source
          // state transitions. All is ok, so proceed to check other t ∈ T'_d.
          checkDestMappings(s'_d, p, M_s, T_s, M'_d, T'_d);
       }
 }  }
```

Algorithm 16 Part III of Algorithm 13

```
// N.B.: The OFF-set of s'_d consists of the stable pts of xsitions in s', (the
// maximal sub-cubes of transitions unstably mapped by M_s and the entire
// supercube of transitions stably mapped by M_s), along with the exit points of
// transitions unstably mapped by M_d.

checkSrcMappings(State s'_d, Implicant p, MappingList M_s, TransitionList T_s,
                 MappingList M_d)
   //   s'_d: the reduced state having some horizontal required cube r
   //   p: an implicant covering the required cube r of state s'_d
   //   M_s: state mapping choice made for the root transition in s'
   //   T_s: unmapped transitions originating in s'
   //   M_d: state mapping choices made for transitions originating in s'_d
{
   // Entry conditions:
   //  - p does not intersect the OFF-set of any mapped transition in M_s or M_d
   //  - p does not illegally intersect any transition
   //  - no transition in T_s has been mapped

   TransitionList T_p := {transitions t ∈ T_s | t intersects p};

   // The transitions T_p can be treated singly.  p hits the OFF-set of T_p iff p
   // hits the OFF-set of some transition t_p ∈ T_p. Further, we never expand
   // p on behalf of a transition in T_s: we only expand p in order to avoid
   // illegal intersections, and transitions in T_s never contribute non-trivial
   // privileged cubes.  Hence, p's shape is dependent solely on T_d and M_d,
   // which are not modified by this routine. As a result, we need not examine
   // combina-
tions of transitions in T_p. Rather, we examine mappings of each
   // transition in turn.

   // ...continued...
```

Algorithm 17 Part III of Algorithm 13, continued

```
// checkSrcMappings(), continued...
//
for each transition t ∈ T_p { // p intersects t; does it hit OFF-set of t?
   // Whether t is stably/unstably mapped, it has OFF-set pts for s'_d, which
   // can interfere with HF covering. We thus examine t's stable and unstable
   // mappings. But, t starts in s', and never has non-trivial priv cubes for s'_d.
   //
   // As in the horizontal algorithm, we optimize the checking using a case
   // analysis. There are 3 possible cases:
   //
   //   1) t is mapped stably (to s'). A single OFF-set cube spans all of t.
   //   2) t is mapped unstably, to a destination state other than s'_d.
   //      Again, a single OFF-set cube results, which spans all of t.
   //   3) t is mapped unstably to s'_d. The maximal sub-cubes of t (that exclude
   //      the end point) are all OFF-set cubes.
   //
   // Hence, there are only two cases: {1,2} and 3. Using this observation, M_s
   // becomes a set of binary variables, one per t ∈ T_s. Each says whether
   // t was mapped to s'_d or not.  Note that the only case which allows a
   // HF cover is case 3, iff p only contains the exit point and no other
   // minterm in t.  So, we simply check for that.  Thus:
   //
   //   a) If case 3 can't occur (if t can't be mapped to s'_d), then there is
  //      always a problem with t. In this case, we generate a constraint
   //      disallowing M_d but placing no constraint on the mapping of t.
   //   b) If case 3 can occur but p contains some point other than the exit
   //      pt of t (i.e. a stable pt of t), there is always a problem
   //      with t. We then generate the same constraint as in a).
   //   c) If case 3 can occur but p contains only the exit point of t, then
   //      a problem only exists if t is not mapped to s'_d.  We nominally
   //      generate a set of constraints disallowing M_d ∪ {δ_{t,c,c'}} for each
   //      c' ≠ s'_d. In practice, we generate a single constraint saying
   //      that if M_d is used, then t must be mapped to s'_d.
   //   d) If s'_d (case 3 above) is the only possible mapping, a HF cover
   //      always exists; simply proceed with the next transition in T_p.
   //
   // N.B.: t̂ is "root" transition in s' whose required cube we're covering.

   if t cannot be mapped to s'_d or p contains a stable point of t { // a) and b)
      generate constr. ¬δ_{t̂,s',s'_d} ∨ { Σ_{δ_{t,s'_d,*} ∈ M_d} δ_{t,s'_d,s'_d} }
      return false; // Always a problem with t -- forget examining rest of T_p
   } else if t can be mapped to some state other than s'_d {   // c)
      generate constr. ¬δ_{t̂,s',s'_d} ∨ { Σ_{δ_{t,s'_d,*} ∈ M_d} δ_{t,s'_d,s'_d} } ∨ δ_{t,s',s'_d} // Must map to s'_d
   } // else case d) applies -- no constraint, continue with next t ∈ T_p.
   // ...continued...
```

Algorithm 18 Part III of Algorithm 13, completed

```
        // ...continued...
        //
        // The above short-circuit that returns before scanning all of T_p prevents
        // the generation of some unnecessarily "large" constraints; however, it
        // still allows some such constraints to slip by.  Specifically, if we only
        // discover that M_d (cf. M_d ∪ {δ_{t,c,c'}}) is bad while examining some t
        // after the first in T_p, we may generate constraints on behalf of
        // transitions preceding t like M_d ∪ {δ_{t,c,c'}} ⊃ M_d.  We could prevent
        // this by deferring the generation of constraints until we have scanned
        // all of T_p or found one t which has no valid mapping.  Doing so would
        // entail keeping track of those t for which we would have generated a
        // constraint. As it is now, the short-circuit will prevent at least some
        // redundant constraints from being generated.
    }
    return true; // Every transition in T_p had some way to map and still cover r.
}
```

It would be possible to construct another version which maps transitions in the source state incrementally, rather than re-mapping all the source state transitions each time p has been expanded to contain the start point of some intersecting destination state privileged cube. To avoid such redundant mapping/checking of source state transitions, we would record the source state transition mapping and make a recursive call to continue examining destination state transition mappings. As a result, the source/dest mapping routines would be mutually recursive. It is not clear that such a version would be enough faster to warrant the additional complexity.

Appendix C

MINIMALIST Shell and Command Set

The following lists show the current set of MINIMALIST commands. First, a set of generic commands (not related to burst-mode synthesis), such as control-flow constructs, are shown. The set of burst-mode synthesis-related commands are given next. For brevity's sake, options flags are omitted.

Users can add their own commands by either defining functions (using `define`), or by dynamically loading a C/C++ library that binds their commands directly into MINIMALIST at run-time. Additionally, any external program (e.g. `ls`, `espresso`, `vi`) can be invoked from within the MINIMALIST shell.

The MINIMALIST command interpreter also supports variable substitution (e.g. `$foo`) and command substitution (e.g. `set foo ‘pwd‘`, or "back-quoting" in shell parlance). Other common shell features such as input/output redirection, piping, and so on are expected in a future release.

Generic Commands

break	`Break out of innermost loop`
call `<func-name> [<arg> ...]`	`Call a named function`
cd `[<directory>]`	`Change directory`

continue `Continue with next iteration of innermost loop`

define `[ <func-name> [ { <argName> ... } { <body> } ] ]`

`Define a named function`

echo `<arg> ...` `Echo words to stdout`

expr `<expression>` `Evaluate a numeric expression`

for `<init-expr> <cond-expr> <update-expr> <body>` `For loop`

help `[<command> ...]` `Print help on commands`

if `<cond> then <then-part> [ else <else-part> ]` `If/then/else`

pwd `Print current working directory`

quit `Quit MINIMALIST`

set `[<varname> [<value>]]` `Set or query a variable's value`

source `<source-file> [<arg> ...]` `Execute a MINIMALIST script`

while `<cond-expr> <body>` `While loop`

Synthesis Commands

assign-states `Assign an encoding (e.g. using CHASM)`

impymin-logic `Minimize logic using IMPYMIN`

make-testable `Derive testable logic`

min-logic `Perform HF logic minimization using HFMIN`

min-states `Perform state reduction`

plot_qt `Plot a burst-mode spec or PLA file`

read-spec `<file>` `Read a burst-mode specification`

set-encoding `<encoding1> ... <encodingN>` `Manually specify an encoding`

set-state-cover `<compatible> ... [<new-spec-var>]`

`Manually specify a state cover`

show-encoding `Show the current encoding`

show-logic `[<PLA-file>]` `Show the binary logic implementation`

verify-logic `[<start-Code>] [<PLA-file>]`

`Verify the current logic implementation`

write-flow `[<file>]` `Write a nicely-formatted flow table`

write-instant `[<PLA-file>]`

`Write out a PLA file for the instantiated machine`

write-spec `[<file>]` `Write out the burst-mode specification`

write-symbolic `[<file>]` `Write a symbolic PLA or KISS2 file`

write-trans `[<trans-file>]`

`Write a description of the specified transitions`

Many of the above synthesis commands allow the following option flags:

-F: Use outputs as fed-back state variables

-L: Minimize total literal count (default is to minimize product count)

-d: Produce logic in which outputs are implemented disjointly from next-state

-s: Produce logic in which each output and state bit is implemented separately (no shared products)

Bibliography

[1] P. Ashar, S. Devadas, and A.R. Newton. *Sequential Logic Synthesis.* Kluwer Academic, 1992.

[2] M.J. Avedillo, J.M. Quintana, and J.L. Huertas. New approach to the state reduction in imcompletely specified sequential machines. *IEEE ISCAS*, pages 440–443, 1990.

[3] M.J. Avedillo, J.M. Quintana, and J.L. Huertas. A new method for the state reduction of imcompletely specified finite sequential machines. *EDAC*, pages 552–556, 1990.

[4] M.J. Avedillo, J.M. Quintana, and J.L. Huertas. Smas: A program for the concurrent state reduction and state assignment of finite state machines. In *ISCAS*, pages 1781–1784, 1991.

[5] R. Bartlett, K. A. Brayton, G. D. Hachtel, R. M. Jacoby, C. R. Morrison, R. L. Rudell, and A. Sangiovanni-Vincentelli. Multilevel logic minimization using implicit don't cares. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, CAD-7(6):723–740, June 1988.

[6] J. Beister. A unified approach to combinational hazards. *IEEE Trans. on Computers*, C-23(6), 1974.

[7] M. Benes, A. Wolfe, and S. M. Nowick. A high-speed asynchronous decompression circuit for embedded processors. In *Advanced Research in VLSI*. IEEE Computer Society Press, September 1997.

[8] L. Benini and G. De Micheli. State assignment for low power dissipation. *IEEE Journal of Solid-State Circuits*, 30(3):258–268, March 1995.

[9] Kees van Berkel. *Handshake Circuits: an Asynchronous Architecture for VLSI Programming*, volume 5 of *International Series on Parallel Computation*. Cambridge University Press, 1993.

[10] Kees van Berkel, Ronan Burgess, Joep Kessels, Ad Peeters, Marly Roncken, and Frits Schalij. Asynchronous circuits for low power: A DCC error corrector. *IEEE Design & Test of Computers*, 11(2):22–32, Summer 1994.

[11] M. T. Bohr. Interconnect scaling — the real limiter to high performance ulsi. In *Proceedings of IEEE Electron Devices Meeting*, pages 241–242. IEEE Computer Society Press, 1995.

[12] D. Brand and V. Iyengar. Timing analysis using a functional relationship. In *Proc. International Conf. Computer-Aided Design (ICCAD)*, pages 126–129, 1986.

[13] R. Brayton and C. McMullen. The decomposition and factorization of boolean expressions. In *Proc. International Symposium on Circuits and Systems*, pages 49–54, 1982.

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

[15] Erik Brunvand. *Translating Concurrent Communicating Programs into Asynchronous Circuits*. PhD thesis, Carnegie Mellon University, 1991.

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

[17] Steven M. Burns. *Performance Analysis and Optimization of Asynchronous Circuits.* PhD thesis, California Institute of Technology, 1991.

[18] N.L.V. Calazans. Boolean constrained encoding: A new formulation and a case study. In *ICCAD-1994*, pages 702–706, 1994.

[19] Supratik Chakraborty, David L. Dill, Kenneth Y. Yun, and Kun-Yung Chang. Timing analysis for extended burst-mode circuits. In *Proc. International Symposium on Advanced Research in Asynchronous Circuits and Systems.* IEEE Computer Society Press, April 1997.

[20] V. L. Chi. Salphasic distribution of clock signals for synchronous systems. *IEEE Transactions on Computers*, C-43(5):597–602, 1994.

[21] W.-C. Chou, P.A. Beerel, R. Ginosar, R. Kol, C.J. Myers, S. Rotem, K. Stevens, and K.Y. Yun. Average-case optimized technology mapping of one-hot domino circuits. In *Proc. Int. Symp. Adv. Research in Async. Ckts. and Sys.*, pages 80–91, March 1998.

[22] T.-A. Chu, N. Mani, and C. K. C. Leung. An efficient critical race-free state assignment technique for asynchronous finite state machines. In *Proc. ACM/IEEE Design Automation Conference*, pages 2–6. IEEE Computer Society Press, 1993.

[23] Tam-Anh Chu. *Synthesis of Self-Timed VLSI Circuits from Graph-Theoretic Specifications.* PhD thesis, MIT Laboratory for Computer Science, June 1987.

[24] Tam-Anh Chu. Automatic synthesis and verification of hazard-free control circuits from asynchronous finite state machine specifications. In *Proc. International Conf.*

Computer Design (ICCD), pages 407–413. IEEE Computer Society Press, October 1992.

[25] M.J. Ciesielski, J.J. Shen, and M. Davio. A unified approach to input-output encoding for fsm state assignment. In *DAC*, 1991.

[26] Wesley A. Clark. Macromodular computer systems. In *AFIPS Conference Proceedings: 1967 Spring Joint Computer Conference*, volume 30, pages 335–336, Atlantic City, NJ, 1967. Academic Press.

[27] J. Cong and L. He. An efficient approach to simultaneous transistor and interconnect sizing. In *ICCAD*, 1996.

[28] J. Cortadella, M. Kishinevsky, A. Kondratyev, L. Lavagno, and A. Yakovlev. Complete state encoding based on the theory of regions. In *Proc. International Symposium on Advanced Research in Asynchronous Circuits and Systems*, pages 36–47. IEEE Computer Society Press, 1996.

[29] J. Cortadella, M. Kishinevsky, L. Lavagno, and A. Yakovlev. Synthesizing petri nets from state-based models. In *ICCAD*, pages 164–171, 1995.

[30] O. Coudert and J.C. Madre. New ideas for solving covering problems. In *DAC*, pages 641–646, 1995.

[31] O. Coudert and J.C. Madre. New ideas for solving covering problems. In *DAC*, pages 641–646, 1995.

[32] M. Damiani and G. De Micheli. Don't care specifications in combinational and synchronous logic circuits. *IEEE Transactions on Computer-Aided Design*, CAD-12(3):365–388, March 1993.

[33] J. Darringer, W. Joyner, L. Berman, and L. Trevillyan. LSS: A logic synthesis through local transformations. *IBM J. Res. Develop.*, 25(4):272–280, 1981.

[34] P.K. Datta, S.K. Bandyopadhyay, and A.K. Choudhury. A graph theoretic approach for state assignment of asynchronous sequential machines. *International Journal of Electronics*, 65(6):1067–1075, 1988.

[35] A. Davis, B. Coates, and K. Stevens. Automatic synthesis of fast compact self-timed control circuits. In *1993 IFIP Working Conference on Asynchronous Design Methodologies, Manchester, England*, March 1993.

[36] A. Davis and S. M. Nowick. An introduction to asynchronous circuit design. In A. Kent and J. G. Williams, editors, *Encyclopedia of Computer Science and Technology*, volume 38, pages 231–286. Marcel Dekker, Inc., 1997.

[37] A. L. Davis. A data-driven machine architecture suitable for VLSI implementation. In *Proceedings of the Caltech Conference on Very Large Scale Integration*, pages 479–494, January 1979.

[38] G. De Micheli. Symbolic design of combinational and sequential logic circuits implemented by two-level logic macros. *IEEE Trans. on CAD*, CAD-5(4):597–616, October 1986.

[39] G. De Micheli. *Synthesis and Optimization of Digital Circuits.* McGraw-Hill, 1994.

[40] G. De Micheli, R.K. Brayton, and A. Sangiovanni-Vincentelli. Optimal state assignment for finite state machines. *IEEE Trans. on CAD*, CAD-4(3):269–285, July 1985.

[41] Mark E. Dean. *STRiP: A Self-Timed RISC Processor Architecture.* PhD thesis, Stanford University, 1992.

[42] S. Devadas and K. Keutzer. Validatable nonrobust delay-fault testable circuits via logic synthesis. *IEEE Transactions on Computer-Aided Design*, CAD-11(12):1559–1573, December 1992.

[43] S. Devadas, K. Keutzer, S. Malik, and A. Wang. Verification of asynchronous interface circuits with bounded wire delays. In *Proc. International Conf. Computer-Aided Design (ICCAD)*, pages 188–195. IEEE Computer Society Press, November 1992.

[44] 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 Trans. on CAD*, CAD-7(12):1290–1300, December 1988.

[45] S. Devadas and A.R. Newton. Exact algorithms for output encoding, state assignment, and four- level boolean minimization. *IEEE Trans. on CAD*, CAD-10(1):13–27, January 1991.

[46] David L. Dill. *Trace Theory for Automatic Hierarchical Verification of Speed-Independent Circuits.* ACM Distinguished Dissertations. MIT Press, 1989.

[47] Jo Ebergen and Sylvain Gingras. A verifier for network decompositions of command-based specifications. In *Proc. Hawaii International Conf. System Sciences*, volume I. IEEE Computer Society Press, January 1993.

[48] Jo C. Ebergen. A formal approach to designing delay-insensitive circuits. *Distributed Computing*, 5(3):107–119, 1991.

[49] E.B. Eichelberger. Hazard detection in combinational and sequential switching circuits. *IBM J. Res. Develop.*, 9(2):90–99, 1965.

[50] D. W. Dobberpuhl et al. A 200-mhz 64-bit dual-issue cmos microprocessor. *Digital Technical Journal*, 4(4):35–50, 1993.

[51] E.M. Sentovich et al. Sequential circuit design using synthesis and optimization. In *Proc. Int. Conf. Computer Design*, October 1992.

[52] J. Hartmanis et al. Some dangers in state reduction of sequential machines. *Information and Control*, pages 252–260, September 1962.

[53] P.D. Fisher and S.-F. Wu. Race-free state assignments for synthesizing large-scale asynchronous sequential logic circuits. *IEEE Trans. on Computers*, 42(9):1025–1034, September 1993.

[54] R.M. Fuhrer, B. Lin, and S.M. Nowick. Algorithms for the optimal state assignment of asynchronous state machines. In *1995 Conference on Advanced Research in VLSI*, pages 59–75, 1995.

[55] R.M. Fuhrer, B. Lin, and S.M. Nowick. Symbolic hazard-free minimization and encoding of asynchronous finite state machines. In *ICCAD*, 1995.

[56] R.M. Fuhrer and S.M. Nowick. Optimist: State minimization for optimal 2-level logic implementation. In *ICCAD-1997*, 1997.

[57] S. B. Furber, J. D. Garside, P. Riocreux, and S. Temple. Amulet2e: An asynchronous embedded controller. *Proceedings of the IEEE*, February 1999.

[58] M. Garey and D. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W.H. Freeman, 1979.

[59] M. K. Gowan, L. L. Biro, and D. B. Jackson. Power considerations in the design of the alpha 21624 microprocessor. In *Proc. ACM/IEEE Design Automation Conference*, pages 726–731. IEEE Computer Society Press, 1998.

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

[61] G. Hachtel, R. Jacoby, K. Keutzer, and C. Morrison. On properties of algebraic transformations and the synthesis of multifault-irredundant circuits. *IEEE*

Transactions on Computer-Aided Design of Integrated Circuits and Systems, CAD-11(3):313–321, March 1992.

[62] G. Hachtel, J.K. Rho, F. Somenzi, and R. Jacoby. Exact and heuristic algorithms for the minimization of incompletely specified state machines. *IEEE Trans. on CAD*, CAD-13(2):167–177, February 1994.

[63] G. D. Hachtel, M. Hermida, A. Pardo, M. Poncino, and F. Somenzi. Re-encoding sequential circuits to reduce power dissipation. In *Proc. International Conf. Computer-Aided Design (ICCAD)*, pages 70–73. IEEE Computer Society Press, 1994.

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

[65] C. A. R. Hoare. *Communicating Sequential Processes.* Prentice-Hall, 1985.

[66] S. Hong, R. Cain, and D. Ostapko. MINI: A heuristic approach for logic minimization. *IBM J. Res. Develop.*, 18:443–458, 1974.

[67] D. A. Huffman. The synthesis of sequential switching circuits. In E. F. Moore, editor, *Sequential Machines: Selected Papers.* Addison-Wesley, 1964.

[68] D. A. Huffman. The synthesis of sequential switching circuits. In E. F. Moore, editor, *Sequential Machines: Selected Papers.* Addison-Wesley, 1964.

[69] Sasan Iman and Massoud Pedram. Two level logic minimization for low power. In *Proceedings of the 1995 IEEE/ACM International Conference on Computer-Aided Design*, 1995.

[70] T. Kam, T. Villa, R.K. Brayton, and A. Sangiovanni-Vincentelli. A fully implicit algorithm for exact state minimization. In *DAC*, 1994.

[71] K. Keutzer. Dagon: Technology mapping and local optimization. In *Proc. ACM/IEEE Design Automation Conference*, pages 341–347, 1987.

[72] Kurt Keutzer, Luciano Lavagno, and Alberto Sangiovanni-Vincentelli. Synthesis for testability techniques for asynchronous circuits. In *Proc. International Conf. Computer-Aided Design (ICCAD)*, pages 326–329. IEEE Computer Society Press, November 1991.

[73] Michael Kishinevsky, Alex Kondratyev, Alexander Taubin, and Victor Varshavsky. *Concurrent Hardware: The Theory and Practice of Self-Timed Design*. Series in Parallel Computing. John Wiley & Sons, 1994.

[74] Prabhakar Kudva, Ganesh Gopalakrishnan, and Hans Jacobson. A technique for synthesizing distributed burst-mode circuits. In *DAC*, 1996.

[75] D.S. Kung. Hazard-non-increasing gate-level optimization algorithms. In *ICCAD*, 1992.

[76] P.N. Lam, H.F. Li, and S.C. Leung. Optimization of state encoding in distributed circuits. *IEEE Trans. on CAD*, 13(5):581–588, May 1994.

[77] L. Lavagno, C. Moon, R. Brayton, and A. Sangiovanni-Vincentelli. Solving the state assignment problem for signal transition graphs. In *Proc. ACM/IEEE Design Automation Conference*, pages 568–572. IEEE Computer Society Press, June 1992.

[78] Luciano Lavagno and Alberto Sangiovanni-Vincentelli. *Algorithms for Synthesis and Testing of Asynchronous Circuits*. Kluwer Academic Publishers, 1993.

[79] C. Leiserson and J. Saxe. Retiming synchronous circuitry. *Algorithmica*, 6:5–35, 1991.

[80] B. Lin and S. Devadas. Synthesis of hazard-free multi-level logic under multiple-input changes from binary decision diagrams. *IEEE Transactions on CAD*, 14(8):974–985, August 1995.

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

[82] B. Lin and F. Somenzi. Minimization of symbolic relations. In *ICCAD-1990*, pages 88–91, 1990.

[83] C.N. Liu. A state variable assignment method for asynchronous sequential switching circuits. *JACM*, 10:209–216, April 1963.

[84] J. Lou, A.H. Salek, and M. Pedram. An exact solution to simultaneous technology mapping and linear placement problem. In *ICCAD*, 1997.

[85] S. Malik, E.M. Sentovitch, R.K. Brayton, and A. Sangiovanni-Vincentelli. Retiming and resynthesis: Optimizing sequential networks with combinational techniques. In *Proc. Hawaii International Conf. System Sciences*, pages 397–406, 1990.

[86] A. Marshall, B. Coates, and P. Siegel. The design of an asynchronous communications chip. *Design and Test*, June 1994.

[87] A.J. Martin. Programming in VLSI: From communicating processes to delay-insensitive circuits. In C.A.R. Hoare, editor, *Developments in Concurrency and Communication*, UT Year of Programming Institute on Concurrent Programming, pages 1–64. Addison-Wesley, Reading, MA, 1990.

[88] Alain J. Martin. Programming in VLSI: From communicating processes to delay-insensitive circuits. In C. A. R. Hoare, editor, *Developments in Concurrency and Communication*, UT Year of Programming Series, pages 1–64. Addison-Wesley, 1990.

[89] Alain J. Martin, Steven M. Burns, T. K. Lee, Drazen Borkovic, and Pieter J. Hazewindus. The design of an asynchronous microprocessor. In Charles L. Seitz, editor, *Advanced Research in VLSI: Proceedings of the Decennial Caltech Conference on VLSI*, pages 351–373. MIT Press, 1989.

[90] E. McCluskey. Minimization of boolean functions. *Bell System Technical Journal*, 35:1417–1444, 1956.

[91] E.J. McCluskey. *Logic Design Principles*. Prentice-Hall, 1986.

[92] P. McGeer and R. Brayton. *Integrating Functional and Temporal Domains in Logic Design*. Kluwer Academic Publishers, 1991.

[93] P. McGeer, J. Sanghavi, R. Brayton, and A. Sangiovanni-Vincentelli. ESPRESSO-SIGNATURES: A new exact minimizer for logic functions. In *Proc. ACM/IEEE Design Automation Conference*, pages 618–621. IEEE Computer Society Press, June 1993.

[94] C. E. Molnar, T.-P. Fang, and F. U. Rosenberger. Synthesis of delay-insensitive modules. In Henry Fuchs, editor, *1985 Chapel Hill Conference on Very Large Scale Integration*, pages 67–86. CSP, Inc., 1985.

[95] J. Monteiro, J. Rinderknecht, S. Devadas, and A. Ghosh. Optimization of combinational and sequential logic circuits for low power using precomputation. In *Advanced Research in VLSI*, pages 430–444, 1995.

[96] David E. Muller and W. S. Bartky. A theory of asynchronous circuits. In *Proceedings of an International Symposium on the Theory of Switching*, pages 204–243. Harvard University Press, April 1959.

[97] David E. Muller and W. S. Bartky. A theory of asynchronous circuits. In *Proceedings of an International Symposium on the Theory of Switching*, pages 204–243. Harvard University Press, April 1959.

[98] C. J. Myers, T. G. Rokicki, and T. H.-Y. Meng. Automatic synthesis and verification of gate-level timed circuits. Technical Report CSL-TR-94-652, Stanford University, January 1995.

[99] M. Nemani and F.N. Najm. High-level area and power estimation for vlsi circuits. In *ICCAD*, 1997.

[100] L. S. Nielsen and J. Sparso. Designing asynchronous circuits for low power: An ifir filter bank for a digital hearing aid. *Proceedings of the IEEE*, February 1999.

[101] S.M. Nowick. Automatic synthesis of burst-mode asynchronous controllers. Technical report, Stanford University, 1993. Ph.D. Thesis.

[102] S.M. Nowick and B. Coates. Uclock: Automated design of high-performance unclocked state machines. In *ICCD*, 1994.

[103] S.M. Nowick and D.L. Dill. Automatic synthesis of locally-clocked asynchronous state machines. In *ICCAD*, 1991.

[104] S.M. Nowick and D.L. Dill. Synthesis of asynchronous state machines using a local clock. In *ICCD*, 1991.

[105] S.M. Nowick and D.L. Dill. Exact two-level minimization of hazard-free logic with multiple-input changes. In *ICCAD*, 1992.

[106] S.M. Nowick and D.L. Dill. Exact two-level minimization of hazard-free logic with multiple-input changes. *IEEE Transactions on CAD*, 14(8):986–997, August 1995.

[107] S.M. Nowick, N.K. Jha, and F. Cheng. Synthesis of asynchronous circuits for stuck-at and robust path delay fault testability. In *VLSI-Design-1995*, January 1995.

[108] S.M. Nowick, K.Y. Yun, and D.L. Dill. Practical asynchronous controller design. In *ICCD*, 1992.

[109] M. Paull and S. Unger. Minimizing the number of states in incompletely specified sequential switching functions. *IRE Trans. on Elec. Comp.*, EC-8:356–367, Sept. 1959.

[110] J. L. Peterson. *Petri Net Theory and the Modeling of Systems*. Prentice-Hall, 1981.

[111] R. Puri, 1995. Private communication.

[112] R. Puri and J. Gu. Area efficient synthesis of asynchronous interface circuits. In *Proc. International Conf. Computer Design (ICCD)*, pages 212–216. IEEE Computer Society Press, 1994.

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

[114] A. Raghunathan, S. Dey, and N.K. Jha. Register-transfer level estimation techniques for switching activity and power consumption. In *ICCAD*, 1996.

[115] M. Rem, J. L. A. van de Snepscheut, and J. T. Udding. Trace theory and the definition of hierarchical components. In *Proceedings of the Third Caltech Conference on VLSI*, pages 225–239. CSP Inc., 1983.

[116] T. Rokicki and C. Myers. Automatic verification of timed circuits. In *Proc. International Workshop on Computer Aided Verification*, pages 468–480. Springer-Verlag, 1994.

[117] R. Rudell. Logic synthesis for VLSI design. Technical Report UCB/ERL M89/49, Berkeley, 1989.

[118] R. Rudell and A. Sangiovanni-Vincentelli. Multiple valued minimization for PLA optimization. *IEEE Trans. on CAD*, CAD-6(5):727–750, Sept. 1987.

[119] R. Rudell and A. Sangiovanni-Vincentelli. Multiple-valued optimization for PLA optimization. *IEEE Trans. on CAD*, 6(5):727–750, September 1987.

[120] J. Rutten and M. Berkelaar. Improved state assignments for burst mode finite state machines. In *Proc. International Symposium on Advanced Research in Asynchronous Circuits and Systems*. IEEE Computer Society Press, April 1997.

[121] A. Saldanha, T. Villa, R. Brayton, and A. Sangiovanni-Vincentelli. Satisfaction of input and output encoding constraints. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 13(5):589–602, May 1994.

[122] A. Saldanha, T. Villa, R.K. Brayton, and A. Sangiovanni-Vincentelli. A framework for satisfying input and output encoding constraints. In *DAC*, 1991.

[123] T. Sasao. Input-variable assignment and output phase optimization of programmable logic arrays. *IEEE Transactions on Computers*, C-33:879–894, October 1984.

[124] G. Saucier. Next-state equations of asychronous sequential machines. *IEEE Trans. on Computers*, EC-21(4):397–399, April 1972.

[125] G. Saucier. State assignment of asynchronous sequential machines using graph techniques. *IEEE Trans. on Computers*, C-21(3):282–288, March 1972.

[126] M. Sawasaki, C. Ykman, and B. Lin. Externally hazard-free implementations of asynchronous control circuits. *IEEE Trans. on CAD*, CAD-16(6), August 1997.

[127] C.L Seitz. System timing. In Carver A. Mead and Lynn A. Conway, editors, *Introduction to VLSI Systems*, chapter 7. Addison-Wesley, 1980.

[128] Chuan-Jin Shi and Janusz A. Brzozowski. An efficient algorithm for constrained encoding and its applications. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 12(12):1813–1826, December 1993.

[129] P. Siegel, G. De Micheli, and D. Dill. Automatic technology mapping for generalized fundamental-mode asynchronous designs. In *Proc. ACM/IEEE Design Automation Conference*, pages 61–67, June 1993.

[130] K. Singh, A. Wang, R. Brayton, and A. Sangiovanni-Vincentelli. Timing optimization of combinational logic. In *Proc. International Conf. Computer-Aided Design (ICCAD)*, pages 282–285, 1988.

[131] Jan L. A. van de Snepscheut. *Trace Theory and VLSI Design*, volume 200 of *Lecture Notes in Computer Science*. Springer-Verlag, 1985.

[132] K.S. Stevens, S.V. Robison, and A.L. Davis. The post office - communication support for distributed ensemble architectures. In *Sixth International Conference on Distributed Computing Systems*, 1986.

[133] C.-J. Tan. State assignments for asynchronous sequential machines. *IEEE Trans. on Computers*, C-20(4):382–391, April 1971.

[134] M. Theobald and S.M. Nowick. Espresso-hf: A heuristic hazard-free minimizer for two-level logic. In *DAC*, June 1996.

[135] M. Theobald and S.M. Nowick. An implicit method for hazard-free two-level logic minimization. In *Proc. Int. Symp. Adv. Research in Async. Ckts. and Sys.*, March 1998.

[136] H. Touati, H. Savoy, B. Lin, R. Brayton, and A. Sangiovanni-Vincentelli. Implicit state enumeration of finite state machines using bdds. In *Proc. International Conf. Computer-Aided Design (ICCAD)*, pages 130–133, 1990.

[137] J.H. Tracey. Internal state assignments for asynchronous sequential machines. *IEEE Trans. on Elec. Comp.*, EC-15:551–560, August 1966.

[138] Chi-Ying Tsui, Massoud Pedram, Chih-Ang Chen, and Alvin M. Despain. Low power state assignment targeting two- and multi-level logic implementations. In *Proceedings of the 1994 IEEE/ACM International Conference on Computer-Aided Design*, pages 82–87, 1994.

[139] S.H. Unger. *Asynchronous Sequential Switching Circuits.* New York: Wiley-Interscience, 1969.

[140] C.H. van Berkel and R.W.J.J. Saeijs. Compilation of communicating processes into delay-insensitive circuits. In *ICCD*. IEEE Computer Society Press, 1988.

[141] K. van Berkel, M. Josephs, and S. M. Nowick. Scanning the technology: Applications of asynchronous circuits. *Proceedings of the IEEE*, February 1999.

[142] H. van Gageldonk, D. Baumann, K. van Berkel, D. Gloor, A. Peeters, and G. Stegmann. An asynchronous low-power 80c51 microcontroller. *Proceedings of the IEEE*, February 1999.

[143] P. Vanbekbergen, F. Catthoor, G. Goossens, and H. De Man. Optimized synthesis of asynchronous control circuits from graph-theoretic specifications. In *ICCAD*, 1990.

[144] P. Vanbekbergen, B. Lin, G. Goossens, and H. de Man. A generalized state assignment theory for transformations on signal transition graphs. In *Proc. International*

Conf. Computer-Aided Design (ICCAD), pages 112–117. IEEE Computer Society Press, November 1992.

[145] T. Villa, T. Kam, R.K. Brayton, and A. Sanviovanni-Vincentelli. *Synthesis of Finite State Machines: Logic Optimization.* Kluwer Academic Publishers, 1997.

[146] T. Villa and A. Sangiovanni-Vincentelli. NOVA: State assignment of finite state machines for optimal two-level logic implementations. In *DAC*, pages 327–332, 1989.

[147] T. Villa and A. Sangiovanni-Vincentelli. NOVA: state assignment of finite state machines for optimal two-level logic implementation. *IEEE Trans. on CAD*, 9(9):905–924, Sept. 1990.

[148] Ted E. Williams and Mark A. Horowitz. A zero-overhead self-timed 160ns 54b CMOS divider. *IEEE Journal of Solid-State Circuits*, 26(11):1651–1661, November 1991.

[149] S. Yang and M. Cieselski. Optimum and suboptimum algorithms for input encoding and its relationship to logic minimization. *IEEE Transactions on Computer-Aided Design*, CAD-10(1):4–12, January 1991.

[150] K. Yun, D. Dill, and S.M. Nowick. Synthesis of 3D asynchronous state machines. In *ICCD*, 1992.

[151] K. Y. Yun, A. E. Dooply, J. Arceo, P. A. Beerel, and V. Vakilotojar. The design and verification of a high-performance low-control-overhead asynchronous differential equation solver. In *Proc. International Symposium on Advanced Research in Asynchronous Circuits and Systems.* IEEE Computer Society Press, April 1997.

[152] Kenneth Y. Yun. *Synthesis of Asynchronous Controllers for Heterogeneous Systems.* PhD thesis, Stanford University, August 1994.

[153] K.Y. Yun and D.L. Dill. Automatic synthesis of 3D asynchronous finite-state machines. In *ICCAD*, 1992.

[154] K.Y. Yun and D.L. Dill. A high-performance asynchronous scsi controller. In *ICCD*, 1995.

[155] K.Y. Yun, D.L. Dill, and S.M. Nowick. Practical generalizations of asynchronous state machines. In *EDAC*, 1993.