COMPLETE STATE CODING OF TIMED ASYNCHRONOUS CIRCUITS

by

Christopher D. Krieger

A thesis submitted to the faculty of
The University of Utah
in partial fulfillment of the requirements for the degree of

Master of Science
in
Electrical Engineering

Department of Electrical Engineering

The University of Utah

December 2002

Copyright © Christopher D. Krieger 2002

All Rights Reserved

THE UNIVERSITY OF UTAH GRADUATE SCHOOL

SUPERVISORY COMMITTEE APPROVAL

of a thesis submitted by

Christopher D. Krieger

This thesis has been read by each member of the following supervisory committee and by majority vote has been found to be satisfactory.

________________ ________________________________

Chair: Chris J. Myers

________________ ________________________________

Erik Brunvand

________________ ________________________________

Christian Schlegel

THE UNIVERSITY OF UTAH GRADUATE SCHOOL

FINAL READING APPROVAL

To the Graduate Council of the University of Utah:

I have read the thesis of Christopher D. Krieger in its final form and have found that (1) its format, citations, and bibliographic style are consistent and acceptable; (2) its illustrative materials including figures, tables, and charts are in place; and (3) the final manuscript is satisfactory to the Supervisory Committee and is ready for submission to The Graduate School.

Date

Chris J. Myers
Chair, Supervisory Committee

Approved for the Major Department

V.J. Mathews
Chair/Dean

Approved for the Graduate Council

David S. Chapman
Dean of The Graduate School

ABSTRACT

This thesis describes a method for solving the complete state coding problem for timed asynchronous systems in an efficient manner. Timed asynchronous systems differ from untimed, speed independent systems in that any change to the system or its timing may dramatically affect the reachable state space. Because frequent state space exploration is time consuming, timing information is used in a variety of ways to postpone or eliminate state space explorations. First, timing information is used to predict the impact of a state signal on the overall system. Second, concurrency information is used to narrow the search space to timing-unique solutions. Third, timing information allows state signal insertion in a timing-sequential, yet non-causal, manner. This permits insertion before input events, an option not readily available in speed-inpedendent systems. Finally, by considering timing, state signal insertion points can be chosen which minimally increase circuit latency. The method has been implemented in the automated design tool ATACS, and correctly and efficiently completes the state code for a variety of established state coding benchmark systems.

To Lockey, who ensured that this work had a finite upper bound

CONTENTS

ABSTRACT . . . . . . **iv**

LIST OF FIGURES . . . . . . **viii**

ACKNOWLEDGMENTS . . . . . . **ix**

CHAPTERS

1. INTRODUCTION **1**

1.1 Related Work 2
1.2 Contributions of This Work 3
1.3 Overview 5

2. SYSTEM MODELS **6**

2.1 Timed Petri Nets 6
2.2 Signal Transition Graphs 8
2.3 Reachability Graphs and State Graphs 9

3. COMPLETE STATE CODING **13**

3.1 Unique State Codes 13
3.2 Complete State Codes 14
3.3 Insertion Point Formalism 14

4. METHODOLOGY **18**

4.1 Investigation of Direct Method 18
4.2 Overview of the State Coding Method 19
4.3 Allowed Transition Points 21
4.4 Allowed Insertion Points 25
4.5 State Space Partitioning 26
4.6 IP Comparison Function 28
4.7 State Signal Insertion 29
4.8 Solution Merit Function 30
4.9 General State Coding Algorithm 31

5. CASE STUDIES **35**

5.1 Simple Oscillator 35
5.2 VME Bus Controller 36
5.3 HP Post Office Controllers 39
5.4 Summary of Results 39

6. CONCLUSION **41**
6.1 This Work's Major Contributions 41
6.2 Improvements to Multivariate Solutions 42
6.3 Enhancements to Avoid Non-persistent Solutions 42
6.4 Other Areas for Future Work 42

REFERENCES **44**

LIST OF FIGURES

2.1 Two representations of the *gradlife* system. 9
2.2 State graphs of the *gradlife* system. 11
3.1 A concurrent Petri net. 16
4.1 Two representations of a system with witness transitions. 22
4.2 The system of Figure 4.1 after complete state coding. 23
4.3 Overall algorithm for completely state coding a system. 32
4.4 The algorithm for $find_best_IPs(\ \mathcal{S}_{best}\ )$. 33
4.5 The algorithm for $explore_IPs(\ IP_{best}\ )$. 34
5.1 Two representations of a simple oscillator. 36
5.2 Two representations of a simple oscillator after complete state coding. 37
5.3 Petri nets of the *vme* system before and after state coding. 38
5.4 Petri nets of *sbuf-ram-write* system before and after state coding. 40

ACKNOWLEDGMENTS

I thank my graduate advisor, Dr. Chris J. Myers, for the teaching, support, and direction he has given during this work. I appreciate Dr. Christian Schlegel and Dr. Erik Brunvand for serving on my supervisory committee and for their assistance and comments on this research.

I also express appreciation to Brandon Bachman, Wendy Belluomini, Eric Mercer, Robert Thacker, and Hao Zheng, who together formed an asynchronous research team of the highest caliber. Their knowledge of systems administration, algorithm theory, and discrete mathematics has been of incalculable worth.

This research has been partially funded by a grant from Intel Corporation, NSF CAREER award MIP-9625014, and the Center for Asynchronous Circuit and System Design.

CHAPTER 1

INTRODUCTION

So many men, so many opinions;
every one his own way.
— *Publius Terentius Afer (Terence)*

Timed circuits are a class of asynchronous circuits whose specification includes explicit timing information for signal transitions. Although precise timing relationships are often unknown before synthesis and technology mapping, circuit designers usually know enough to produce some reasonable, relative estimates. Applying even rough estimates can lead to the removal of large amounts of circuitry that would be required by a speed-independent design. These timing assumptions can then be verified after synthesis when the actual timing values are better known. This design style can lead to significant gains in circuit performance over asynchronous circuits designed without timing assumptions [1].

Timed circuits are often specified using state graphs or Petri nets. The *complete state coding* (CSC) problem is inherently present in systems specified by these methods. This is because states in a state graph or markings in a Petri net may hold only partial, or incomplete, state. While they work well for modeling concurrency, causality, choice, and other temporal and graph properties, this lack of self-contained complete state information means that many abstract asynchronous systems cannot be mapped unambiguously to a present state-next state relation. Consequently, these systems also cannot be directly synthesized. Thus, a methodology for adding the missing state code information necessary for synthesizability is highly desirable. For this reason, a great deal of research has been invested in exploring efficient solutions to this so-called complete state coding problem.

1.1 Related Work

Early work by Vanbekbergen [2] focuses on removing CSC violations from signal transition graphs (STGs). This method restricts the STG to those without choice and which include only a single rising and a single falling transition on each signal. The algorithm modifies the STG until a connected lock graph can be found for the STG. It does make an effort to incorporate timing information in the form of a mean switching time for selected transitions. However, this information is only used in computing the longest simple cycle as a performance estimate and does not drive the CSC problem resolution process. The algorithm has complexity $O(n^5)$, where n is the number of transitions, and is not guaranteed to find a solution, even when one exists.

Lavagno et al. [3] initially investigated automated methods for solving the state coding problem at the state graph level. They map an initial state graph into a flow table synthesis problem. The state coding problem is then solved by using flow table minimization and state assignment methods. However, solutions obtained with their method correspond only to a restricted class of STG transformations.

Vanbekbergen et al.also work with complete state coding of state graphs [4]. By working with a state graph, rather than an STG specification directly, they are able to remove several graph restrictions. The algorithm they propose, while functional, has several drawbacks. It requires knowing in advance how many state signals are to be inserted. More importantly, the signal insertion point selection algorithm casts the problem as one of boolean satisfiability, which has double exponential complexity. The largest example published involves only 10 signals. Due to the algorithm's complexity, it is not easily extensible to large, real-world systems with many signals.

Work on state graph methods by Ykman-Couvreur et al. [5] restricts the potential signal insertion points to *excitation regions* and *switching regions*. These are small subsets of the whole state space in which a given signal is enabled to transition or has just transitioned. This method is very successful at removing CSC violations in a computationally efficient manner. Concurrency reduction as a means of CSC

violation removal is also performed. A framework for first reducing the state space using concurrency reduction and then expanding it using signal insertion has been presented [6]. Their algorithms are implemented in the tool ASSASSIN [7].

Work has also been done by Cortadella et al. in the area of state signal insertion. Their work concentrates on specifications expressed as Petri nets. They have developed a tool, petrify [8], which transforms between Petri nets and transition systems. Transition systems are manipulated to introduce or enforce properties such as pure choice, free choice, unique choice, place irredundancy, and complete state coding. The transition system is then converted back into a Petri net which now also retains these desired properties. Their algorithm inserts state signals into *regions*, which correspond to places in the initial Petri net specification, intersections of regions, or unions of intersections of regions. Since the excitation regions and switching regions of Ykman-Couvreur are a subset of Cortadella's regions, this algorithm is more general and finds a larger number of valid insertion points. The cost function used to select a solution does not consider performance directly, but seeks to reduce an estimate of circuit area. Some provisions exist for specifying general timing properties, such as "lazy transitions," or "slow environment" assumptions, but explicit timing information is not used [9].

1.2 Contributions of This Work

To date, most work done in this field has been limited to untimed circuits. This thesis focuses on the unique aspects of *timed asynchronous circuits*. Timed circuits use explicit timing information throughout the design and synthesis process to produce better performing circuits. There are several differences between timed and untimed systems which impact the complete state coding process. The major complication in the design of timed systems is that any change to the system results in different timing which affects the whole system, including altering the reachable state space. Thus, inserting a state signal transition can have implications which are difficult to identify without completely reexploring the timed state space. Because state space exploration is often a lengthy endeavor, the method presented

here resorts to various examinations of available timing information in an effort to predict the impact of a given signal transition insertion. Though reexploration of the state space is eventually necessary to validate any predictions made during the state signal insertion process, it is done infrequently.

On the other hand, the availability of timing information provides for analysis and optimizations which would otherwise be extremely difficult. First, with timing information, it is possible to restrict the causal set of transitions for a state signal transition to those transitions which are timed-concurrent. Any causal transitions which are not timed-concurrent are redundant. Removing redundancy greatly reduces the portion of the solution space that needs to be searched, thereby reducing algorithm run-time. It also produces solutions with smaller support sets.

Second, intimate knowledge of a system's timing enables the insertion of state signal transitions which do not explicitly delay any other transitions in the system. These so-called *hanging insertions* have minimal impact on the system and do not increase system latency. They can also be inserted to occur before input events, because, while they have a timing-determined *sequential relationship*, they possess no *causal* relationship with respect to the input event. This type of relationship is difficult to determine in untimed systems. The availability of hanging insertion points increases the size of the solution space and often leads to solutions with minimal performance impact.

Third, most methods for completing the state code of a system explored to date have not considered performance of the resulting circuit as a parameter of the state signal insertion cost function. Rather, they have placed emphasis on the estimated circuit area of the state signal implementation logic. As fabrication process technologies improve, area is becoming less and less a critical metric, while circuit performance is rising in importance. Since complete state coding is a necessary precursor to synthesis, aggressive designers cannot afford to ignore performance issues during this step in the design process. Performance issues can be partially addressed by determining a critical cycle through an asynchronous system, conceptually related to the critical paths of synchronous designs. Using

this information to identify points where state signals transition only infrequently and where they delay only non-critical transitions allows the overall performance impact of a state signal to be minimized.

1.3 Overview

The remainder of this thesis describes a performance-driven state signal insertion method for timed asynchronous circuits. First, a brief overview of Petri nets describes some necessary terminology and introduces the Petri net extension used to include timing bound information. Properties and terms of particular concern to this work are then defined. With this groundwork laid, the state signal insertion method itself is presented, with a particular focus on how this timed method varies from previous work on untimed systems. A presentation of results obtained from this CSC solution method, as implementated within the design automation tool ATACS, and some conclusions comprise the final chapters.

CHAPTER 2

SYSTEM MODELS

If we spoke a different language,
we would perceive a somewhat different world.
— *Wittgenstein*

In order to reason about timed systems, several graph-based models are commonly used. One of these, a timed Petri net, can be used to specify and to model concurrency, causality, and choice. However, many complete state coding problems are more apparent when examined using a reachability graph or state graph. For the sake of clarity, all of these graph types are defined below.

2.1 Timed Petri Nets

Timed Event-Rule (ER) or *Timed Event/Level* (TEL) structures are used by the `ATACS` tool to model and to analyze timed circuit specifications and are able to efficiently represent complex timed concurrent systems [1, 10]. Closely related to timed ER structures, though distinctly different, are Petri nets. Both Petri nets and timed ER structures can be used to describe the state variable insertion method developed in this work. This thesis uses Petri nets to describe the theoretical aspects of complete state coding. Note, however, that the internal data structure used by the `ATACS` implementation is a timed ER structure. A detailed presentation of ER structures is given in [1]. When the distinction between ER structures and Petri nets is significant, it is noted in this text.

Formally, a timed Petri net (TPN) system is a modified one-safe Petri net represented with the tuple $N = (P, T, F, M_o, \Delta)$ where

- P is the set of *places*;
- T is the set of *transitions*, $T \cap P = \emptyset$;
- $F \subseteq (P \times T) \cup (T \times P)$ is the *flow relation*;
- $M_o \subseteq P$ is the *initial marking*;
- $\Delta : P \to \mathcal{N} \times \{\mathcal{N} \cup \infty\}$, maps each place to a *bounded time constraint.*

Conventional Petri nets lack any notion of, and therefore notation for, timing information, which obviously is core to specifying timed asynchronous systems. Therefore, an extended Petri net model, known as a *timed Petri net* (TPN) [11], is used. This extended model essentially appends a lower and an upper timing bound, $[l, u]$, to each place in the Petri net, as specified by the labeling function Δ. For example, in Figure 2.1(a) the timing bounds on the place labeled 4 are $[30, 60]$. When a token arrives at a place, a theoretical timer begins to count from zero upward. The place becomes *marked* immediately, but does not become *satisfied* until the counter reaches the place's lower timing bound. If the counter reaches the upper timing bound, the place is said to be *expired.* The lower bound can be zero, and the upper bound infinite, allowing untimed behavior to be expressed withing this framework.

Given a transition $t \in T$, the set of all places which immediately precede t is called the preset of t and is given by $\bullet t = \{p \in P \mid (p, t) \in F\}$. Similarly, the set of all places immediately following a transition t is known as the postset of t and is defined as $t\bullet = \{p \in P \mid (t, p) \in F\}$. Likewise, for a place $p \in P$, the set of all transitions which immediately precede p is the preset of p, denoted $\bullet p = \{t \in T \mid (t, p) \in F\}$. The set of all transitions immediately following p, $p\bullet = \{t \in T \mid (p, t) \in F\}$, is its postset. A transition can fire when all places in its preset are satisfied. It must fire before all places in its preset are expired.

A marking or a state M in a TPN is $M \subseteq P$, where each $p \in M$ is a marked place. With a marking M, the *untimed enabled* function returns the set of transitions T_e that have all places in their presets marked in M, or $T_e(M) = \{t \in T \mid \forall p \in \bullet t, p \in M\}$. A marking is graphically depicted by filling each circle representing a marked place with a solid dot representing a token. In Figure 2.1(a), the initial marking $M_o = \{1\}$ consists only of place 1, so therefore only the uppermost place is initially filled with a token.

One additional important aspect of Petri nets is that they allow the modeling of choice. This is achieved through the use of *choice places* and *merge places*. A choice place is a place which has a postset containing more than one transition. A merge place is a place with a preset containing more than one transition. Since the Petri nets discussed in this thesis are exclusively *one safe*, only one of the transitions in the postset of the choice place can fire for each time the choice place is satisfied. A merge place requires only one transition in its preset to fire in order to be satisfied. In Figure 2.1(a), place 3 is a both a choice place and a merge place. Transitions $\{hungry, undress, bored\}$ are in its postset. The firing of any one of these transitions mutually excludes the firing of any other because it consumes the token in place 3. Therefore, only one of these three transitions is allowed to fire for each time a token arrives in place 3. Note that this method of modeling choice requires all transitions $t \in p\bullet$ to share the timing bounds affixed to the choice place. For example, the timing bounds between transitions *dress* and *hungry* must equal the timing bounds between transitions *dress* and *undress*, and likewise *dress* and *bored*. Occasionally, it is useful to have different timing bounds for each transition following a choice place. The timed ER structure used in the implementation of this work allows for these different timing bounds, as explained in [1].

2.2 Signal Transition Graphs

In a circuit, only signal transitions are possible, rather than the Petri net's generalized notion of transitions as arbitrary actions. If a Petri net is to model a circuit, the Petri net must be converted to a *signal transition graph* (STG). A

signal transition graph is essentially a Petri net in which all system transitions correspond directly to actual transitions on signals. This can be accomplished with a mapping $T \rightarrow S \times \{+, -\}$, where S is the set of all signals and $+$ and $-$ are used to indicate whether the transition is rising from low to high, or falling from high to low, respectively. Figure 2.1(b) shows an STG derived from the original Petri net shown in Figure 2.1(a). All transitions in the Petri net have been mapped to rising and falling transitions on four signals: dsd, hun, con, and duh. When depicted graphically, STGs omit drawing all places except choice places, merge places, and those in the initial marking. All other places, while not drawn, are considered to be implicitly present, and their timing bounds are placed next to the edge containing the implicit place.

2.3 Reachability Graphs and State Graphs

The reachable untimed state space for a TPN can be represented as a *reachability graph* (RG). An RG is a graph with vertices which are untimed states (i.e. markings) and edges which are possible state transitions. An RG is modeled by the tuple (Φ, Γ), where Φ is the set of markings and $\Gamma \subseteq \Phi \times T \times \Phi$ is the set of edges.

A *state graph* (SG) is an RG in which the states have been labeled with bit vec-

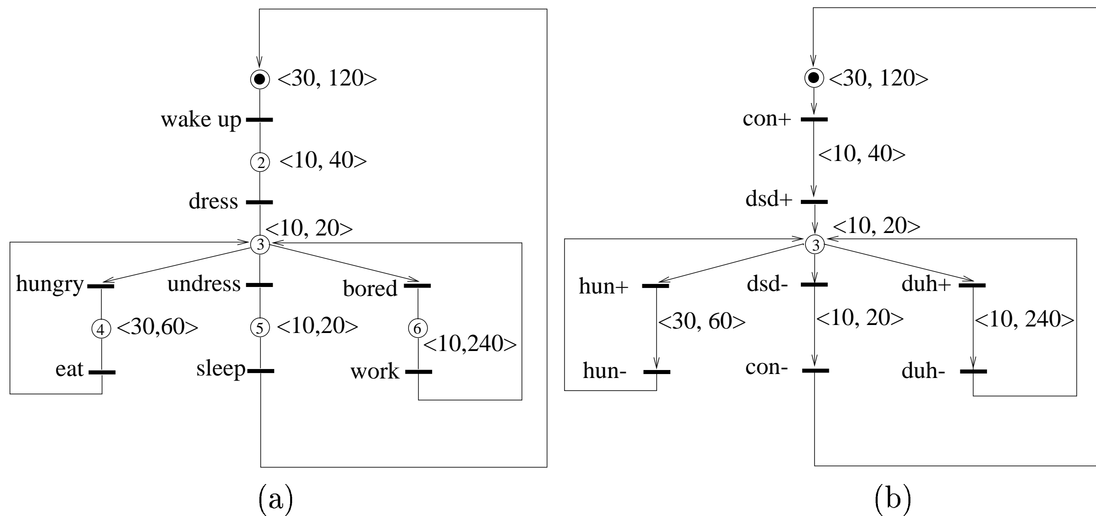


Fig. 2.1. Two representations of the *gradlife* system. (a) The timed Petri net representation. (b) The signal transition graph representation.

tors corresponding to the binary values of each of the system's signals in that state, and the transitions are labeled with the signal whose transition causes the system to move between the two states. An SG is modeled by the tuple $(S, \Phi, \Gamma, \lambda_\Phi, \lambda_\Gamma)$ where S is the set of signals, $\lambda_\Phi : \Phi \rightarrow (S \rightarrow \{0,1\})$ is the state labeling function, and $\lambda_\Gamma : T \rightarrow S$ is the transition labeling function. S can be further partitioned into the set of input signals, I, and output signals, O. The set of transitions on output signals is defined as $T_o = \{t \in T \mid \lambda_\Gamma(t) \in O\}$, and the set of transitions on input signals can be similarly defined. A *state vector* can be written as $\langle v_0 v_1 ... v_n \rangle$ where each v_i is either 0 or 1, indicating the value returned by the λ_Φ labeling function for the ith signal for the given state. Figure 2.2(a) shows a state graph for the STG shown in Figure 2.1(b). Each state is labeled with the value of λ_Φ for that state.

There exists an SG for a corresponding RG if and only if there is a *consistent state assignment*. A consistent state assignment exists if each state and transition can be labeled such that between two states in a state transition the only signal to change value is the one which experienced the transition. More formally, for an RG, (Φ, Γ), and a set of signals, S, an SG exists if there exists a λ_Φ and λ_Γ such that:

$$\begin{array}{c} \forall (M, t, M') \in \Gamma \ . \ \forall u \in S \ . \ (\lambda_\Gamma(t) \neq u \wedge \lambda_\Phi(M)(u) = \lambda_\Phi(M')(u)) \\ \vee \ (\lambda_\Gamma(t) = u \wedge \lambda_\Phi(M)(u) \neq \lambda_\Phi(M')(u)) \end{array}$$

When state coding a system, it is useful to be able to determine in which states a signal is untimed enabled to rise or fall. The sets *rise*(u) and *fall*(u) provide this information and are defined as follows:

$$\begin{array}{rcl} \mathit{rise}(u) & = & \{M \in \Phi \mid \lambda_\Phi(M)(u) = 0 \wedge \exists t \in T_e(M).\lambda_\Gamma(t) = u\} \\ \mathit{fall}(u) & = & \{M \in \Phi \mid \lambda_\Phi(M)(u) = 1 \wedge \exists t \in T_e(M).\lambda_\Gamma(t) = u\} \end{array}$$

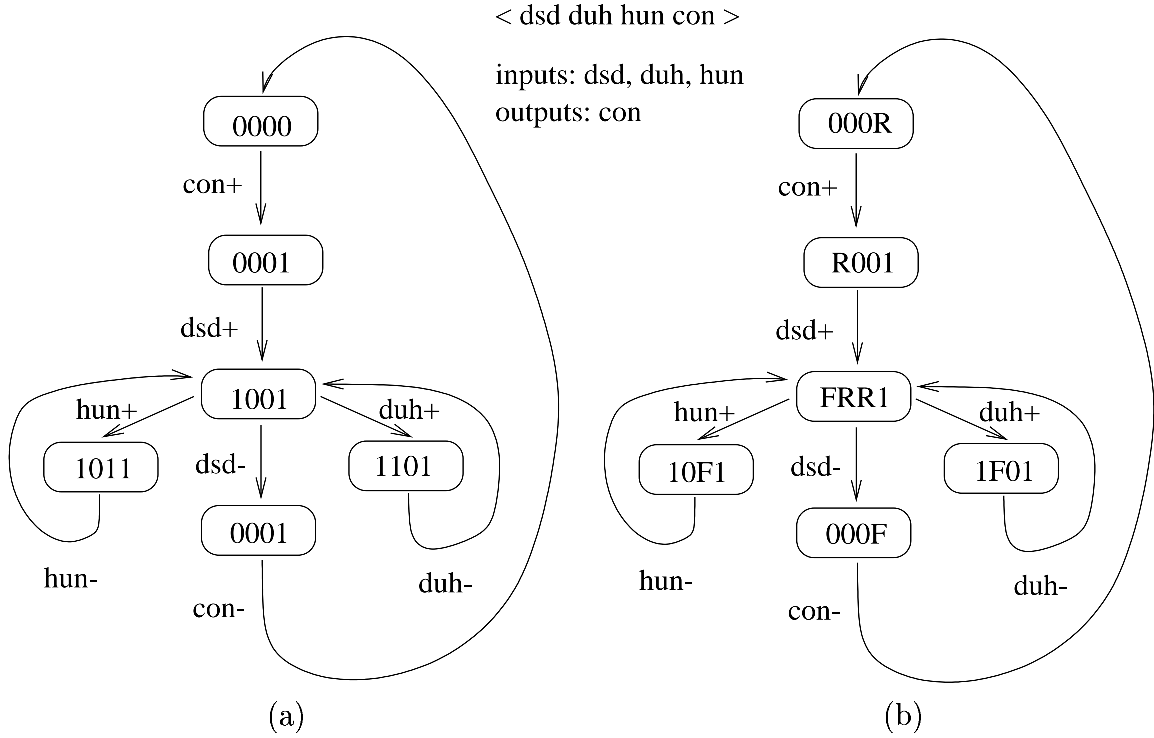


Fig. 2.2. State graphs of the *gradlife* system. (a) The state graph with state labels. (b) The state graph with enabling labels.

Occasionally, it is useful to present the rising and falling information in the form of an *enabling vector*, where, for each signal u in marking M, the corresponding vector entry is:

$$\begin{array}{ll} 0 & \text{if } \lambda_\Phi(M)(u) = 0 \wedge M \notin rise(u) \\ R & \text{if } \lambda_\Phi(M)(u) = 0 \wedge M \in rise(u) \\ 1 & \text{if } \lambda_\Phi(M)(u) = 1 \wedge M \notin fall(u) \\ F & \text{if } \lambda_\Phi(M)(u) = 1 \wedge M \in fall(u) \end{array}$$

In Figure 2.2(b), each state is labeled with its enabling vector. In the state labeled $\langle FRR1 \rangle$, signal *dsd* is high and enabled to fall, signals *duh* and *huh* are low and enabled to rise, and signal *con* is stable high.

The *switching region* for a transition t, $SR(t)$, is the set of states directly reachable through transition t:

$$SR(t) \quad = \quad \{M' \in \Phi \mid (M, t, M') \in \Gamma\}$$

In related literature, a switching region is often defined in terms of a rise or a fall on a given signal, rather than on a given transition as it is herein defined. The

distinction, while subtle, is important. Switching regions on transitions are not necessarily maximally connected, and there is only one SR for any given transition.

These system models form the theoretical foundation upon which the complete state coding algorithm is built. Extensive reference to these definitions and concepts is made in the following chapters.

CHAPTER 3

COMPLETE STATE CODING

Of all men's miseries, the bitterest is this:
to know so much and have control over nothing.
— Herodotus

Circuits and Petri nets are fundamentally dissimilar. A circuit is a physical device usually constructed from metal and silicon. A Petri net is an abstract mathematical tool. For this reason, synthesizing a circuit which implements the behaviors of a given Petri net is a rather unnatural process. It begins by generating an SG from the consistent TPN or STG. From the SG, a present state – next state relation is extracted for all output signals. Regrettably, this relation may be ambiguous, and, if so, cannot be implemented directly by circuitry. Therein lies the heart of the complete state coding problem. The rest of this chapter explicitly defines this problem and provides the necessary formalism for understanding the proposed solution.

3.1 Unique State Codes

Two markings are said to have a *unique state code* (USC) if they map to different sets of signal values, and an STG is said to have a unique state code if all markings have unique state codes. Therefore,

$$\begin{aligned} USC(M, M') &\equiv \lambda_\Phi(M) \neq \lambda_\Phi(M') \\ USC(\Phi) &\equiv \forall (M, M') \in \Phi \times \Phi \;.\; USC(M, M') \end{aligned}$$

Consider Figure 2.2(a). There are two states, each of which is labeled $\langle 0001 \rangle$. Therefore, this state graph does not have a unique state code.

3.2 Complete State Codes

Two markings are said to have a *complete state code* if they either have unique state codes or if they share a state code but have the same output signal transitions enabled in each marking. An STG is said to have a complete state code if all its markings have a complete state code:

$$CSC(M, M') \equiv USC(M, M') \vee (T_e(M) \cap T_o = T_e(M') \cap T_o)$$
$$CSC(\Phi) \equiv \forall (M, M') \in \Phi \times \Phi \, . \, CSC(M, M')$$

A set containing all marking pairs which violate the CSC property is defined as:

$$CSCV(\Phi) \equiv \{(M, M') \in \Phi \times \Phi \mid \neg \, CSC(M, M')\}$$

In Figure 2.2(b), there are two states with enabling vectors $\langle R001 \rangle$ and $\langle 000F \rangle$, which each have the same state label, $\langle 0001 \rangle$. In the first state, signal *dsd* is enabled to rise, while in the second state, signal *con* is enabled to fall. Because signal *con* is an output, these two states differ in their output enablings and therefore lack a complete state code. Consequently, the system as a whole does not have a complete state code.

A circuit lacking a unique state code, but having a complete state code, has a present state – next state relationship for input signals that is not uniquely defined. Because control circuitry is being synthesized only for output signals, the burden of distinguishing between two markings which map to the same signal values in this case lies with the environment. In contrast, for the case in which a system lacks a complete state code, the present state – next state relationship for output signals cannot be determined unambiguously. The circuit must be augmented in some way so that, using only signal values, the present state – next state relationship for all output signals can be defined uniquely and unambiguously. The method described herein accomplishes this task by adding state holding signals to make this possible.

3.3 Insertion Point Formalism

The insertion of a state signal into a system involves the addition of a new signal and at least one rising and one falling transition on this new signal. *Transition*

points (TPs) are one useful way of specifying where these transitions occur. A transition point is an ordered pair of sets of transitions, TP = *(st, et)*. At the TPN level, each transition point represents a transition with incoming edges from each of the transitions in the start transition set, or *st*, and with outgoing edges to the transitions in the end transition set, or *et*. An *insertion point* consists of an ordered pair of transition points, $IP = (TP_R,\ TP_F)$, where TP_R is the rising transition and TP_F is the falling transition.

To define to which markings in an SG a TP corresponds, the following preliminary definitions are in order. First, to facilitate the comparison of transition firing times, the function $\tau(t)$ is defined to return the amount of time elapsed between the absolute start of the system and the firing time of transition t. The minimal and maximal worst-case time differences between two transitions,

$$\begin{aligned} min(t_1, t_2) &= min(\tau(t_2) - \tau(t_1)) \\ max(t_1, t_2) &= max(\tau(t_2) - \tau(t_1)) \end{aligned}$$

are the lower and upper bound on the time which can elapse between the firing of the two transitions [12, 13].

As an example, suppose transition $f+$ in Figure 3.1 fires at time 0. Then $min(f+, e-) = (6 + 4) - 0 = 10$. If the upper bound on a state signal transition is less than 10 time units, the transition can be inserted after the $f+$ transition and it always occurs before the $e-$ transition. In this case, $e-$ is called a "witness transition," as its occurance testifies that the state signal transition has already occured. Likewise, since $min(f+, d-) = 4 + 5 - 0 = 9$, a state signal transition inserted after $f+$ with an upper bound lower than 9 always occurs before $d-$. In this case, $d-$ is also a witness transition.

Witness transitions are by their nature conservative, in that they indicate that the state signal transition occured some amount of time previous to their transitioning. Suppose in this second example that the state signal transition has an upper bound of 3. Then the witness transition $d-$ actually indicates that the state signal transitioned at least 6 time units previous to the witness transition. Each

witness transition may occur the same or a different amount of time after the state signal transition without affecting correctness. The presence of witness transitions is what makes "hanging insertions" possible, as is discussed in Section 4.3.

Formally, the set of witness transitions for a transition inserted in a given TP is defined,

$$T_w(st) \equiv \{t_w \in T \mid \forall_{t \in st}\ min(t, t_w) > U\},$$

where U is the greatest upper bound on a place in the preset of the new state signal to be inserted into the given transition point. A transitive closure function $transitioning_path(M_1, M_n)$ returns true if there exists a sequence of markings $(M_1, M_2, \ldots, M_n)$ connected by transitions, such that

$$\forall i < n \ \ . \ \ M_i \xrightarrow{t} M_{i+1} \wedge t \notin (et \ \cup \ T_w(st))$$

is true. The intersection of switching regions for the start transitions forms the set of seed markings, $seed(st) = \cap_{t \in st}\ SR(t)$. The markings in a state graph to which

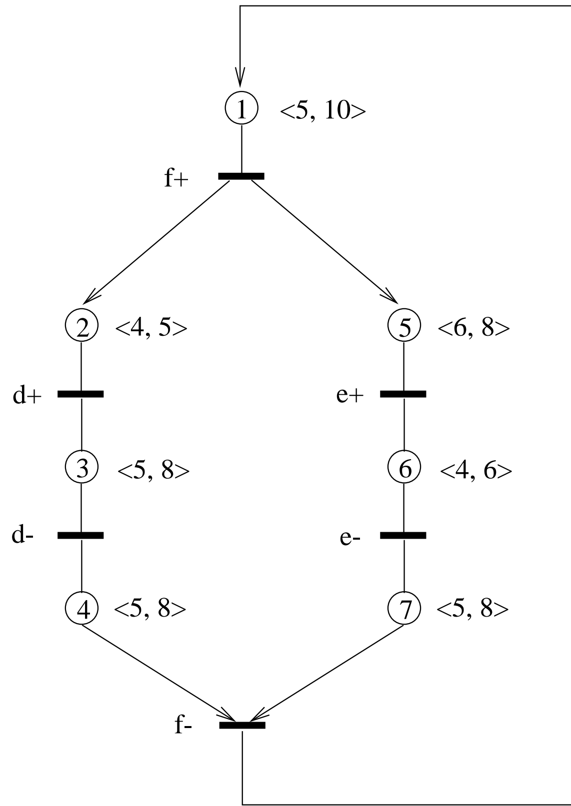


Fig. 3.1. A concurrent Petri net.

a given TP_k corresponds can now be succinctly defined as

$$M(TP_k) \quad = \quad seed(st) \quad \cup \quad \{M' \mid M \in seed(st) \ \wedge \ transitioning_path(M, M')\}$$

The correspondence between state graph markings and Petri net transitions is critical to the state variable algorithm's efficiency, as it enables verification of various properties of the transition point using whichever is more efficient, the TPN or the SG.

A transition point bears a great similarity to a *region*, as described in [14]. However, while regions correspond to places in a Petri net, transition points map to transitions. In many cases, the distinction is insignificant. For example, where $|st| = |et| = 1$, a transition point corresponds to a single edge in the Petri net, and uniquely identifies the implicit place on that edge. In this case, the TP corresponds to a region, with the incoming signals being those in st, the outgoing signals those in et, and all other events not crossing the border of the region. This relationship holds when $|st|$, $|et| > 1$, if intersections and unions of intersections of regions are considered. The two methods diverge in their respective handling of choice. Both transition points and regions allow for the handling of free choice. However, transition points can only describe points between transitions. If a choice place lies between two transitions, the transition point only describes one of the possible directions the token can go when leaving the choice place. Regions can describe insertion into the choice place itself. For this reason, transition points describe a strict subset of the insertion locations described by regions. Typically choice places are in the preset of input transitions, which are often excluded from being insertion points, so the utility of such insertion points remains an open research question.

CHAPTER 4

METHODOLOGY

The first rule of intelligent tinkering
is to save all the parts.
— Paul Erlich

An algorithm for completing the state code of timed asynchronous circuits is presented. Throughout, timing information is used in an effort to produce a correct and realizable circuit with high performance. First, all unique and allowed transition points are generated. Then an all-pairs combination algorithm is used to combine the transition points into unique insertion points. Each insertion point is subsequently examined, and those which are predicted to be the most useful in solving the complete state coding problem are kept. Finally, a state signal actually is inserted into the best insertion points, as dictated by a user-adjustable merit function. If a complete state code has not been achieved, the algorithm continues to explore various partial solutions, adding state variables to each of them until complete solutions are found. The best possible complete solutions are then chosen, according to a solution selection function. One or more of these lowest-cost complete solutions is presented to the user for further analysis and customization.

4.1 Investigation of Direct Method

Conventional wisdom and Occam's razor dictate that the simplest solution to a problem is the best one. Perhaps the simplest way to solve a complete state coding violation is to add a state signal to the system such that it holds a different value in each of the two states forming the violation. This is true and is the essence of all state signal insertion procedures. The methods diverge in where the signal

transitions on this state signal should occur. Examining the state graph, it seems simplest that one transition should occur just before the system transitions into a previously ambiguous, violating state. The other should occur just after the system leaves the violating state. These points are easily identified on either an STG or an SG.

However, while this approach is the simplest, it usually does not make appreciable progress toward reducing the number of complete state code violations. To understand this, consider a system consisting of two signals where both are outputs. Assume that there is a state $\langle OO \rangle$ and a state $\langle OR \rangle$ in the state graph. These two states form a complete state coding violation. Let us assume that we add a state signal transition according to the method just described around state $\langle 0R \rangle$. Upon state space exploration, we find that the state is split into two states, $\langle 00R \rangle$ and $\langle 0R1 \rangle$, one of which still is ambiguous. This phenomenon of *state splitting* usually renders the direct, simple approach to selecting state signal transition points useless. In some cases, it can actually cause the number of state code violations to increase. This occurs when uniform state code violations become complete state code violations because the state variable itself is an output. In the cases where the simple approach does not cause state splitting and actually removes state code violations, it removes the minimal number, making it inefficient for solving systems with many state code violations. Clearly, a more sophisticated approach is necessary.

4.2 Overview of the State Coding Method

Consideration of previous work in the field and an examination of the state splitting problem led to the following algorithm. The overall algorithm consists of three phases: exploration, insertion and verification, and solution selection. The algorithm passes through these phases systematically on the way to an optimal solution. Several different possible solution paths are examined in parallel, thereby avoiding the high costs associated with misprediction and local minima. Detailed explanations of each step of the algorithm follow this brief overview.

The exploration phase essentially generates a ranked list of valid insertion points. First, a list of systems, $\mathcal{S}_{best}$ is created. Each system in such a list of systems is represented as an ordered pair (N, SG_N), where N is the Petri net and SG_N is the corresponding state graph. $\mathcal{S}_{best}$ holds the best systems from the previous iteration and initially is set to contain only the original, incompletely state coded system. Next, an insertion point list, IP_{best}, is created and initially set to empty. The first system in $\mathcal{S}_{best}$ is chosen and loaded. The exploration loop then begins, in which an insertion point is generated and tested for validity through graph partitioning. If valid, the relative merit of the IP is determined using a comparitor function described in Section 4.6. If the merit is greater than that of the least useful IP kept in IP_{best}, or if IP_{best} does not yet contain the user-specified number of IPs to keep, this IP is inserted into IP_{best} in a ranked fashion. This continues until all valid insertion points have been considered. The algorithm then loads the next system in $\mathcal{S}_{best}$ and likewise explores all valid insertion points for this system, keeping those IPs which are better than the least useful IPs already in IP_{best}. This continues until all valid IPs of all systems in $\mathcal{S}_{best}$ have been considered.

The algorithm transitions into the verified insertion phase at this point. This phase begins by clearing $\mathcal{S}_{best}$. A state signal is inserted into the system corresponding to the highest ranked IP in IP_{best} using the process described in Section 4.7. At this point, a new timed-reachable graph is found through timed state space generation. This is an extremely expensive process in terms of run time, but is necessary to verify that estimates made during the exploration phase are valid. The number of CSC violations remaining in the system after the insertion of the state variable is then computed. If no violations remain, the modified system is added to the set of completely state coded solution systems, $\mathcal{S}_{sol}$. If violations remain after the addition of the state variable, this system is added to $\mathcal{S}_{best}$ in a ranked fashion which keeps only the *num_paths* best systems. The user sets *num_paths* to the maximum number of exploration branches to be considered in parallel. If none of the state signal insertions results in a system with fewer CSC violations, a non-progress counter is incremented. Otherwise, the counter is reset to zero.

If the non-progress counter reaches the $max_non_progress_levels$ value set by the user, the algorithm terminates, returning an unsuccessful status. Otherwise, if no solutions are found, the algorithm loops back to the exploration phase and proceeds to add additional variables from there. However, if solutions are found, control shifts to the selection phase. Each solution is evaluated according to $merit_{sol}$, the solution evaluation function described in detail in Section 4.8. Up to num_sols of the best solutions, as specified by the user, are then returned.

4.3 Allowed Transition Points

For the overall state coding algorithm to be both correct and efficient, transition points must be generated which are valid and irredundant. A transition point x is valid and irredundant if and only if:

$$\begin{gathered} st \cap et = \emptyset \\ \forall t \in et \;.\; t \notin \text{input transitions} \\ \forall t_1, t_2 \in st \;.\;\; t_1 \parallel_t t_2 \\ \forall t_1, t_2 \in et \;.\;\; t_1 \parallel_t t_2 \\ \forall t \in et \;.\; min(\tau(t) - \tau(x)) < 0 \end{gathered}$$

The first requirement, that st and et be disjoint, simply prevents transition points from having unnecessary loops. This is ensured by choosing a transition and adding it to st and et in a mutually exclusive manner. Likewise, the second requirement is met by choosing only outputs as elements of the set et. Note that prohibiting an input transition from being a member of et does not prevent the insertion of a state signal transition sequentially before an input transition, but rather prevents creating a *causal* relationship between the state signal transition and the input transition. A causal relationship can alter the environment's specified behavior and is therefore forbidden. However, inserting a transition with causality and timing such that it always occurs shortly before an input signal transition does not constrain the environment and is therefore allowable. This distinction can only be made in a timed system and calls for a simple example.

Consider the Petri net shown in Figure 4.1. With the proper timing bounds, transition $e-$ can serve as a witness transition for a state signal transition starting after transition $e+$. This insertion is shown in Figure 4.2(a). As the postset of

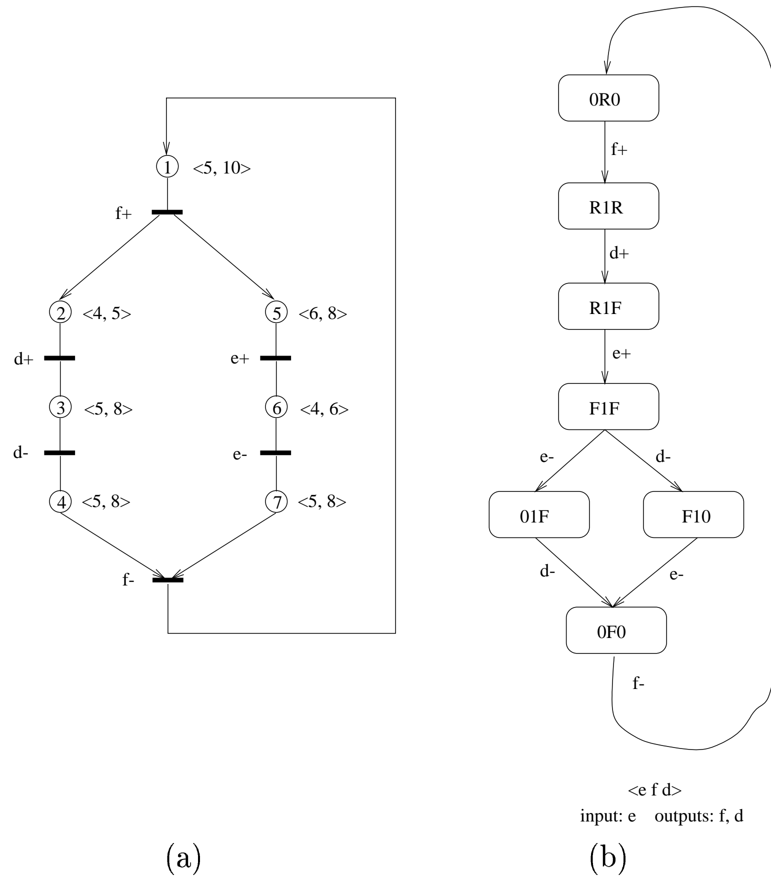


Fig. 4.1. Two representations of a system with witness transitions. (a) The timed Petri net representation. (b) The state graph representation.

transition $csc+$ is empty, this is a so-called *hanging insertion*. The firing time of the witness transition is used as an approximate upper bound on the firing time of $csc+$. The state graph, Figure 4.2(b), shows that, while $csc+$ is concurrent with $d-$, it does in fact always preceded the witness transition $e-$. Being able to estimate what markings are impacted by a hanging insertion makes it possible to sequence transitions before input transitions, and thereby opens up insertion points not usually considered available using speed-independent methods.

The requirement that all elements of st and all elements of et be mutually *timed-concurrent* (denoted $\|_t$), also requires additional explanation. Timed-concurrent transitions are those transitions which are untimed enabled concurrently and, even when timing is considered, can fire in either order. A transition point for which these conditions do not hold describes the same behavior as a simpler transition

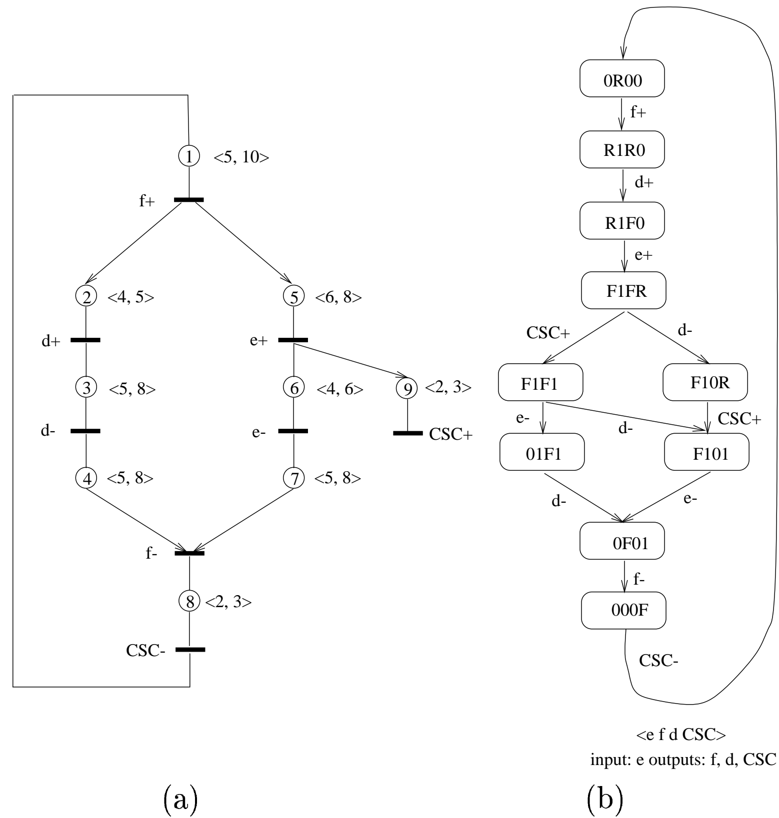


Fig. 4.2. The system of Figure 4.1 after complete state coding. (a) The Petri net showing a hanging transition for $CSC+$. (b) The state graph labeled with enablings.

point for which these conditions do hold. By eliminating redundancy at this early point in the solution space exploration, the algorithm's run time is tremendously improved.

Again consider the Petri net shown in Figure 4.2(a). Transitions $d-$ and $f-$ are sequential, not concurrent. Adding an edge from transition $d-$ to $csc-$ with an upper timing bound less than 11 time units does not change the behavior of the system. Therefore, $TP_1 = (\{d-, f-\}, \{f+\})$ is a redundant and more complex transition point than $TP_2 = (\{f-\}, \{f+\})$. When timed concurrency is considered, even transitions which are untimed enabled in the same marking, but which are forced by timing to fire in a sequential fashion, are redundant. Consider transitions $d+$ and $e+$. These two transitions are untimed concurrent, meaning that if timing is not considered, they could occur concurrently. However, because the lower timing

bound on $e+$ is greater than the upper timing bound on $d+$, $d+$ always precedes $e+$. These two transitions are not timed-concurrent. In other words, only those transitions

$$t_1 \parallel_t t_2 \equiv \{t_1, t_2 \in T \mid \exists M, M_1, M_2 \in \Phi \,.\, M \xrightarrow{t_1} M_1 \in \Gamma \;\wedge\; M \xrightarrow{t_2} M_2 \in \Gamma\}$$

are considered timed-concurrent.

The last redundancy-eliminating condition simply decrees that no transition shall be made an explicit member of et if it can be used as a witness transition instead. As a witness, the transition provides nearly the same infomation without having to add the state variable to its support set and switching logic.

By way of example, the valid, irredundant transition points for the system represented by the Petri net shown in Figure 4.1(a) are listed in Table 4.1.These transition points assume a delay bounded by $[5, 10]$ on the arcs leading to the state signal transition. For each transition point, the start transitions are listed, then the witness transitions, finally followed by the end transitions. If any set is empty, the null set is listed.

The algorithm generates all valid and irredundant transition points, as defined by the immediately aforementioned criteria, and stores them in memory. During subsequent stages, these TPs are combined into insertion points, which must also be checked for validity. While storing the transition points consume some amount of

TABLE 4.1
ALL VALID TRANSITION POINTS FOR THE SYSTEM OF FIGURE 4.1

({e-} {e+} ∅)	({e+} {f+} ∅)	({e-} ∅ {f+})	({f-} {e+} ∅)
({f-} {e-} ∅)	({f-} ∅ {f+})	({e+} ∅ {f-})	({e-} ∅ {f-})
({f+} {f-} ∅)	({d+} {e+} ∅)	({d+} {e-} ∅)	({d+} {f+} ∅)
({d+} ∅ {f-})	({e+} {d+} ∅)	({e-} {d+} ∅)	({f+} ∅ {d+})
({f-} ∅ {d+})	({d-} {e+} ∅)	({d- e-} {e+} ∅)	({d-} {e-} ∅)
({d-} ∅ {f+})	({d- e-} ∅ {f+})	({d-} ∅ {f-})	({d- e-} ∅ {f-})
({d-} {d+} ∅)	({d- e-} {d+} ∅)	({e+} {d-} ∅)	({e-} {d-} ∅)
({f+} ∅ {d-})	({f-} {d-} ∅)	({f-} {d- e-} ∅)	({d+} ∅ {d-})
({d+} {e-} {d-})			

memory, the memory usage is not prohibitive. Tests show that this provides a very significant decrease in runtime over implementations which require regeneration of the transition points.

4.4 Allowed Insertion Points

The space of all possible insertion points theoretically includes all combinations of transitions in *st* and *et* for the rising transition and all combinations of transitions in *st* and *et* for the falling transition. Thus, the upper bound on the number of possible insertion points is $|2^E|^4$, the cardinality of the power set of distinct transitions raised to the fourth power. Each of these insertion points may or may not be consistent. Of those that are consistent, only a small number are useful in actually reducing the number of complete state coding violations. The algorithm attempts to categorize a potential insertion point as soon as possible, so that needless consideration is avoided.

First, a computationally efficient check is made on the IP for a necessary, but not sufficient, consistency condition. This weak form of consistency is termed *compatibility.* If the compatibility test is passed, then a more rigorous check for true consistency is performed. Because this second test is computationally expensive, it is incorporated into the graph partitioning scheme, as described in Section 4.5. If an insertion point is found to be consistent, it is then evaluated to determine its utility in solving complete state coding problems. A user-specified number of useful insertion points is stored, with an insertion point only being kept if its worth is greater than that of at least one of the insertion points already being stored, or the maximum number of useful insertion points has not yet been found. When the maximum number has been reached, only if this IP's value is greater than a stored IP will the less useful IP be eliminated and the new IP added to the set of highest value insertion points.

The process begins by drawing two TPs from the pool of valid TPs previously generated. These two TPs are selected such that each pair is unique in an unordered sense. For example, if a pair (TP_1, TP_2) is selected, then the pair (TP_2, TP_1)

will never be drawn from the TP pool. This prevents the generation of "mirror solutions." These mirror solutions differ only in the logic sense or phase of the state signal, and do not represent truly unique solutions. A solution and its mirror are equally valid and equally useful, and can be interconverted easily. Avoiding mirrors exactly halves the overall size of the solution space.

The pair of TPs is then combined into an insertion point $IP = (TP_R, TP_F)$ and checked for compatibility. Incompatible insertion points are those which contain a TP_R which is incompatible with TP_F, where compatibility is defined as:

$$(st_{TP_R} \cap st_{TP_F} = \emptyset) \wedge (et_{TP_R} \cap et_{TP_F} = \emptyset)$$

Consistent state assignment dictates that any transition on a signal is followed by an opposite transition before another transition of the same type occurs. Incompatible insertion points always lead to inconsistent state assignment, because at least one transition causes both the rising and falling transition on the state signal. Since the compatibility check can be performed much more quickly than a full inconsistency check, it is worthwhile to eliminate these insertion points at this stage of the algorithm. The ability to check for consistency using either the SG or the TPN is one of the benefits of the insertion point formalism duality.

To make more concrete the concept of insertion points, the valid insertion points for the system represented by the Petri net shown in Figure 4.1(a) are listed in Table 4.2. These insertion points are composed from the transition points shown in Table 4.1.

4.5 State Space Partitioning

The purpose behind state state partitioning is twofold. First, it determines the consistency of the given IP. Second, it determines the efficacy of the proposed IP, by revealing the number of CSC violations which are disambiguated by a state signal inserted into the given insertion point. This could be determined by simply inserting the state variable and counting the number of violations which exist in the new system. Unfortunately, this method would require a complete reexploration of the timed state space and generation of a new SG. It also would permit insertions of

TABLE 4.2
ALL VALID INSERTIONS POINTS FOR THE SYSTEM OF FIGURE 4.1

(({f-} ∅ {f+}), ({d+} {e-} ∅))	(({f-} ∅ {f+}), ({e+} {d-} ∅))
(({f-} ∅ {f+}), ({d+} ∅ {d-}))	(({f-} ∅ {f+}), ({d+} {e-} {d-}))
(({e+} {f+} ∅), ({d+} {e+} ∅))	(({d+} {e-} ∅), ({e-} {d+} ∅))
(({d-} {d+} ∅), ({e+} {d-} ∅))	(({d+} {e-} ∅), ({d- e-} {d+} ∅))
(({d- e-} {d+} ∅), ({e+} {d-} ∅))	(({e+} {f+} ∅), ({f+} ∅ {d+}))
(({e-} ∅ {f+}), ({d+} {e+} ∅))	(({e-} ∅ {f+}), ({d+} {e-} ∅))
(({f-} {e+} ∅), ({e+} ∅ {f-}))	(({f-} {e+} ∅), ({e-} ∅ {f-}))
(({f-} {e+} ∅), ({d-} ∅ {f-}))	(({f-} {e-} ∅), ({e-} ∅ {f-}))
(({f-} ∅ {f+}), ({f+} {f-} ∅))	(({f-} ∅ {f+}), ({d+} {e+} ∅))
(({e+} ∅ {f-}), ({d+} {e+} ∅))	(({e-} ∅ {f-}), ({d+} {e+} ∅))
(({e-} ∅ {f-}), ({d+} {e-} ∅))	(({d+} {e+} ∅), ({d-} ∅ {f+}))
(({d+} {e+} ∅), ({d-} ∅ {f-}))	(({d+} {e-} ∅), ({f+} ∅ {d+}))
(({d+} {e-} ∅), ({f-} ∅ {d+}))	(({d+} {f+} ∅), ({f+} ∅ {d+}))
(({f+} ∅ {d+}), ({e+} {d-} ∅))	(({f-} ∅ {d+}), ({e+} {d-} ∅))
(({d-} ∅ {f+}), ({e+} {d-} ∅))	(({d-} ∅ {f-}), ({e+} {d-} ∅))

signals with inconsistent state assignment. To avoid both of these problems, the SG is partitioned into four subpartitions. Each state of the state graph is categorized and annotated based on the excitation that a state signal inserted into the given IP would have in that state, either rising, falling, stable high, or stable low.

First the rising transition is considered. All markings in $M(TP_R)$ are annotated as *rising*, indicating that the state signal would be untimed enabled to rise in these markings. This procedure is repeated for $M(TP_F)$, this time annotating the states as *falling*. Once both the rising and falling markings have been annotated, all markings between the rising markings and the falling markings are annotated as being *stable high*. All markings between those annotated as falling and those annotated as rising are annoated as being*stable low*. If at any time during the partitioning process a marking is found to belong to more than one subpartition, the partition, by definition, is invalid. A signal inserted in this insertion point would not have consistent state assignment, so this insertion point is rejected.

If an IP is found to be consistent, then the usefulness of a state signal inserted into the IP must be determined. A state signal is deemed useful if it resolves state

coding violations. This determination is also made using the partitioned state graph, as part of the IP comparator function described below.

4.6 IP Comparison Function

A simple boolean function is used to compare the relative worth of two insertion points. The $betterof_{IP}(IP_1, IP_2)$ function is used only to assign a relative ordering to two insertion points. Thus, it is structured as a boolean comparator rather than as a traditional cost function returning a numeric value. As one would expect, the primary component of the insertion point comparison function is the number of CSC violations which would exist in a system if a state signal were to be inserted into the given IP. The number of CSC violations which would be disambiguated by this state signal insertion is found by removing from CSCV(Φ) any pair of violations in which one violation marking is in the *stable low* partition and the other violation marking is in the *stable high* partition. This is found by examining the partitions of the two markings in the partitioned state graph, and does not require any additional exploration.

When calculating the cost of CSC violations which exist in the system, some consideration must be given not only to the number of violations, but also to the nature of the violations. A CSC violation can either be a violation which existed in the initial system, or it can be a unique state coding violation which is converted into a CSC violation by the insertion of the state signal. This happens when two markings map to the same signal values, and all output enablings match between the two markings, yet the enabling of the proposed state signal would not match between the two markings. In other words, if two states comprising a unique state coding violation pair are partitioned such that one state is annotated *rising* and the other is annotated *stable low*, or one is in the *falling* partition and the other is in the *stable high* partition, the state signal itself will not have a complete state code. For example, state $\langle 00 \rangle$ and state $\langle R0 \rangle$ form a unique state coding violation, where the left signal is an input and the right signal is an output. If, however, a new state signal were to be added such that the state vectors were now $\langle 00R \rangle$

and $\langle R00 \rangle$, this USC violation would be escalated to a CSC violation. Note that a state signal, like the other outputs of the system, is generated by the circuit and must have complete state coding itself. It is therefore possible to increase the number of CSC violations in a system through the naïve insertion of state signals. If a state signal inserted into an IP removes one CSC violation and converts one USC violation into a CSC violation, it is prefered over a partial solution which does not remove any CSC violations, even though both have an equal number of CSC violations. This heuristic is founded on the assumption that USC to CSC conversions are easier to solve with additional state variables than those violations which occur "naturally" in the system. This is not always true and occasionally misguides the search. However, experience has shown that this rule of thumb is generally helpful.

The $betterof_{IP}$ function greedily assigns the lowest cost to the IP in which a state signal would remove more CSC violations and escalate fewer USC violations to CSC violations. If the number and nature of CSC violations do not indicate a clear preference, the performance impact of the state signal rises to the forefront. In this case, the IP with the smallest sum $|et_{rising}| + |et_{falling}|$ is selected. This solution delays a smaller number of other transitions and leads to a circuit with lower latency. Next, the IP with the smallest sum $|st_{rising}| + |st_{falling}|$ is selected. Transitions enabled by fewer signals tend to be easier and smaller to implement. In cases where both insertion points are equal in every aspect, one IP is given deterministic yet arbitrary preference.

4.7 State Signal Insertion

Once an insertion point has been selected, the task remains of actually inserting a state signal into it. This process begins by adding a transition to the TPN which represents the system. Arcs are then added from each transition $t \in st$ to the new state. Similarly, arcs are added from the new transition to each of the transitions $t \in et$. Each of these arcs defines an implicit place. The same steps are followed for the complementary transition of the signal. After both transitions have been

added to the Petri net, the state signal is assigned an initial value based on the partition into which the initial marking in the original TPN was placed. If the initial marking is in the *stable high* partition or the *falling* partition, the initial value is high. Otherwise, it is initially set to low.

The initial marking of each of the places on the newly added arcs must also be determined. For places in the postset of a new transition, the place is initially marked only if the transition $t_e \in et$ can be reached from every $t_s \in st$ and an initial token lies on every one of these paths. If this is the case, the system deadlocks without the addition of an initial token. If any place in the postset of a transition is initially marked in this manner, the initial value of the state signal is set to the value it would assume by the firing of this transition. Initial marking of places in the preset of the new transition is determined through simple token flow analysis. If this analysis indicates that deadlock occurs in the system, a token is added to one of the places in the preset. This continues until sufficient tokens have been added to resolve the deadlock. Finding a better method than this crude, yet effective, approach is an area of ongoing research.

4.8 Solution Merit Function

Often, there are several signal insertions which independently resolve the incomplete state code of a given system. In this case, the algorithm must choose from among them and return only the best solutions. The function $merit_{sol}$ evaluates complete solutions based on an estimate of the area and delay of their implementations. The area is approximated using the number of literals needed. The delay calculation is more complicated. The system delay is scored by computing a stochastic cycle period metric, as described at length in [15]. In brief, the stochastic cycle period is a weighted sum of possible delays, where each delay corresponds to the amount of time a transition takes to fire when triggered by a signal in its support set, weighted by the probability that this signal actually causes the given transition to fire. Using sums of these delays, a value for the average cycle period of the timed circuit is found and used as a measure of the delay of the system. Solutions which

greatly impact the performance of the circuit will have greater system delays than higher performance solutions. For example, suppose that a solution required the insertion of a state signal transition such that it would lengthen the most critical path, while another solution placed the state variable transition such that it delayed a less common path. While the transition time of the state signal is the same in either case, in the second solution it has a smaller impact on the average-case performance of the system.

The return value of the $merit_{sol}$ function is computed using

$$merit_{sol}(solution) = (1 - P) \times area + P \times delay$$

where P is a user-specified real value between 0 and 1, indicating the emphasis that should be placed on performance. As P increases, delay becomes more important, at the cost of greater area. As P decreases, minimizing circuit area takes increasing priority over minimizing delay. This allows the user to choose a solution in keeping with the objectives of the overall design.

4.9 General State Coding Algorithm

An algorithmic presentation of the described method for completing the state code of timed asynchronous circuits is presented in Figure 4.3 through Figure 4.5. At each step, timing information is used in an effort to produce a correct and realizable circuit with high performance. First, all unique and allowed transition points are generated. Then an all-pairs combination algorithm is used to combine the transition points into unique insertion points. Each insertion point is subsequently explored, and those which are most useful in solving the complete state coding problem are kept. Finally, a state signal is inserted into the best insertion points, as dictated by a user-adjustable merit function. If a complete state code has not been achieved, the algorithm continues to explore various partial solutions, adding state variables to each of them until complete solutions are found. The best possible complete solutions are then chosen, according to a solution selection function. These lowest-cost complete solutions are then returned to the user for further analysis and customization.

```
complete_state_code( (N_initial, SG_{N_initial}), max_non_progress_levels )
{
      all_solutions = ∅;
      S_best = {(N_initial, SG_{N_initial})};
      non_progress_levels_remaining = max_non_progress_levels;

      while ( (all_solutions == ∅) and (non_progress_levels_remaining > 0) )
      {
            IP_best = find_best_IPs( S_best );
            (S_best, all_solutions) = explore_IPs( IP_best, S_best );
      }

      if ( all_solutions ≠ ∅ )
      {
            best_solution = pop( all_solutions );
            foreach ( solution ∈ all_solutions )
            {
                if ( merit_sol( solution ) > merit_sol( best_solution ) )
                      best_solution = solution;
            }
            return best_solution;
      }

      return unsolvable;
}
```

Fig. 4.3. Overall algorithm for completely state coding a system.

```
find_best_IPs( S_best )
{
    foreach ( (N, SG_N) ∈ S_best )
    {
        unique_TPs = all_irredundant_TPs( (N, SG_N) );
        foreach ( TP_A ∈ unique_TPs )
        {
            remove( TP_A, unique_TPs );
            foreach ( TP_B ∈ unique_TPs )
            {
                IP = (TP_A, TP_B);
                if ( compatible( IP ) )
                {
                    if ( consistent( IP ) )
                        ranked_insert( IP, betterof_IP(IP), IP_best );
                }
            }
        }
    }

    return IP_best;
}
```

Fig. 4.4. The algorithm for $find_best_IPs(\mathcal{S}_{best})$.

```
explore_IPs( IP_best )
{
        S_next = ∅
        foreach ( IP ∈ IP_best )
        {
                N_new = insert_state_variable( IP, system(IP) );
                SG_N_new = state_space_explore( N_new );
                if ( violations( SG_new ) < violations( system( IP ) ) )
                {
                        non_progress_levels_remaining = max_non_progress_levels;
                        ranked_insert( (N_new, SG_N_new), violations( SG_N_new ), S_next );
                        if ( violations( SG_N_new ) == 0 )
                        {
                                insert( N_new, all_solutions );
                        }
                }
                else non_progress_levels_remaining--;
        }

        return (S_next, all_solutions);
}
```

Fig. 4.5. The algorithm for $explore_IPs(IP_{best})$.

CHAPTER 5

CASE STUDIES

Today's weirdness is tomorrow's reason why.
— Hunter S. Thompson

The complete state coding algorithm herein described is implemented as part of the CAD tool `ATACS` and has been used to complete the state code of a variety of timed and untimed systems. In every case, the goal was to produce a circuit with optimal performance, regardless of the complexity of the logic needed to implement the circuit. Results obtained from the state coding method implemented in `petrify` are also presented. It should be noted that `petrify`'s cost function attempts to minimize area without regard for performance, thereby favoring very different solutions than the cost function used in this work. Also, the timed circuit design method makes extensive use of explicit timing information. Therefore, a direct comparison with speed-independent state coding methods is somewhat misguided. However, since there is no other published method which specifically deals with timed circuits, the speed independent results form a useful reference point.

In the figures, dashed lines between transitions on Petri nets or explicitly filled places indicate that a place is initially marked. For the sake of uniformity, all delays have been specified as $[6, 14]$ to represent a gate delay of 10 time units with a $\pm 40\%$ tolerence. Environment delays are also $[6, 14]$.

5.1 Simple Oscillator

The first system to be considered is the trivial example given in Figure 5.1. In this system, signals a and b are inputs and c is an output. The only complete state code violation pair is $(R00, 00R)$. In spite of the seeming simplicity of the system,

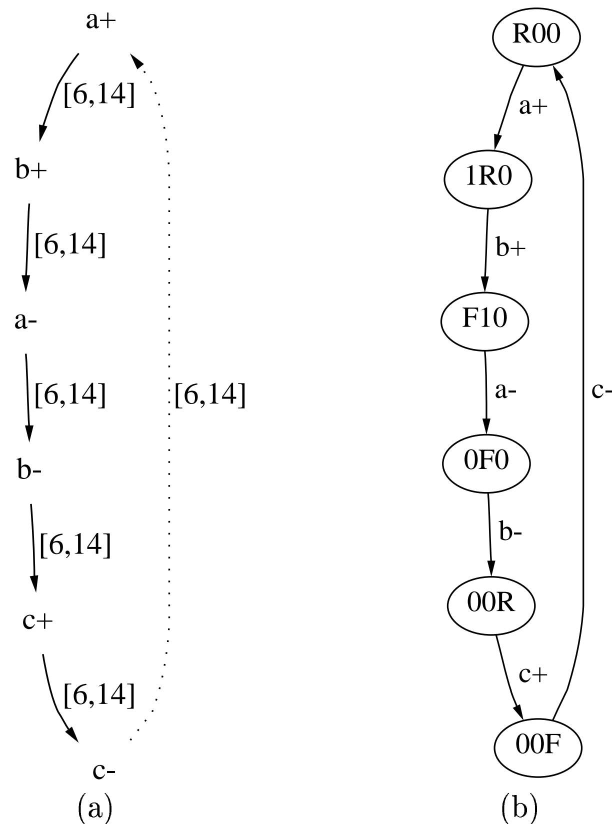


Fig. 5.1. Two representations of a simple oscillator. (a) The timed Petri net representation. (b) The state graph representation.

it cannot be completely state coded using speed-independent methods, because it requires inserting a state transition before an input transition. Using a hanging insertion point made available by the witness transition $b-$, it can be state coded with the addition of one state variable. The hanging insertion of transition $csc0+$ is timed concurrent with $b+$, as is easily seen in the state graph shown in Figure 5.2(b).

5.2 VME Bus Controller

The *vme* example is a simple controller which partially implements the VME bus protocol. This system has two concurrent execution paths, making it marginally more complex than the previous system. Signals *dsr* and *ldtack* are inputs and signals *d* and *lds* are outputs. Both `ATACS` and `petrify` found a complete coding

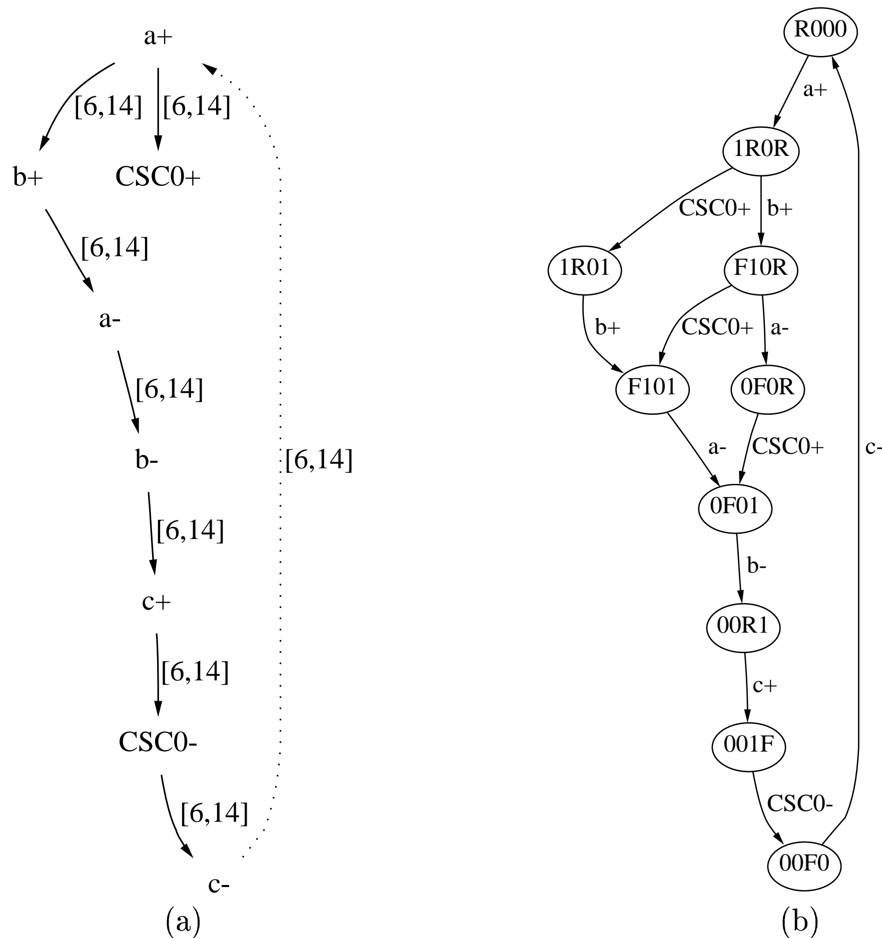
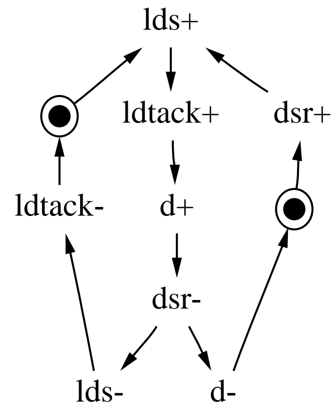
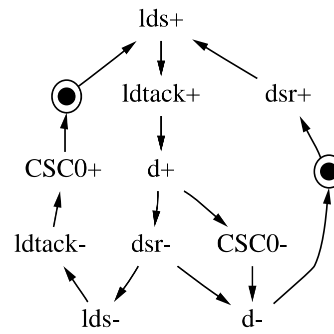


Fig. 5.2. Two representations of a simple oscillator after complete state coding. (a) The timed Petri net representation. (b) The state graph represenation.

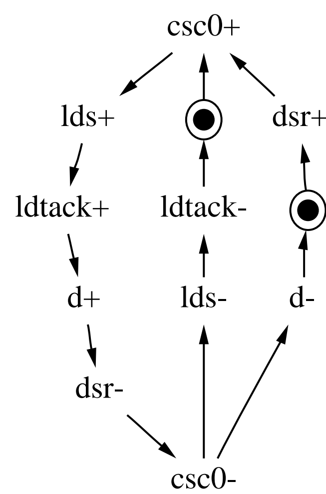
using a single state signal. The original system, as shown in Figure 5.3(a), had a cycle period of 62.3 time units. `ATACS` found a solution which increased the cycle period by 6%, while `petrify`'s solution increased the cycle period by 28%. The difference in added latency can be see by examining the two different solutions shown in Figure 5.3(b) and Figure 5.3(c). Figure 5.3(b) shows that, using the method presented in this thesis, the $CSC0-$ transition is placed such that it is concurrent with the $dsr-$ transition.



(a)



(b)



(c)

Fig. 5.3. Petri nets of the *vme* system before and after state coding. (a) The original Petri net. (b) The system as state coded by `ATACS`. (c) The system as state coded by `petrify`.

5.3 HP Post Office Controllers

The *sbuf-ram-write* and *sbuf-send-pkt2* specifications come from the Post Office chip designed at Hewlett-Packard Labs [17]. In figure 5.4(a), the original specification for *sbuf-ram-write* is given. Note that in the solution using timing, given in Figure 5.4(b), only one state variable is needed, rather than the two required by `petrify`'s solution, shown in Figure 5.4(c). The rising transition is a hanging transition, and the falling transition is concurrent with $precharged+$. This solution did not impact the critical path delay. The `petrify` solution impacted the cycle time by 33%. The improved cycle time, however, has a cost in the controller's area.

The *sbuf-send-pkt2* specification can also be solved by both tools. However, the *petrify* solution is too complex to be meaningfully displayed, and its performance could not be calculated. For this reason, the *sbuf-send-pkt2* example is not presented in greater detail here.

5.4 Summary of Results

In summary, results generated by `ATACS` using the complete state coding methodology of this thesis tend to be larger than those generated by `petrify`, but with much better performance. The results are summarized in Table 5.1. Listed are the number of states, signals needed to achieve CSC, area in terms of literals, and delay calculated using the stochastic cycle period metric. The *etlatch* example is an edge triggered latch controller taken from [16]. The other examples were examined individually in the corresponding sections of this chapter.

TABLE 5.1
SUMMARY OF RESULTS OBTAINED BY DIFFERENT STATE CODING METHODS

	`ATACS`				`petrify`			
circuit	states	signals	area	delay	states	signals	area	delay
etlatch	67	1	31	54	84	1	14	68.8
vme	12	1	11	66	12	1	6	79.8
sbuf-ram-write	52	1	37	119	58	2	19	160
sbuf-send-pkt2	24	1	16	171	24	1	21	-

(a)

(b)

(c)

Fig. 5.4. Petri nets of *sbuf-ram-write* system before and after state coding. (a) The original system. (b) The system as state coded by `ATACS`. (c) The system as state coded by `petrify`.

CHAPTER 6

CONCLUSION

Time is a great teacher,
but unfortunately it kills all its pupils.
— Hector Berlioz

In the area of complete state coding, as in most worthwhile areas of research, a complete understanding and mastery of the problem at hand cannot be achieved in a single leap. Rather, understanding is reached through many small steps, each building upon the last. This thesis presents new ideas and a detailed method for completing the state code of timed asynchronous systems. Yet, at the same time, it raises additional questions and leaves them to be answered by those who follow. If timed asynchronous systems are to be synthesized into functioning circuits, the complete state coding issue must be resolved. This work is a significant stride toward that goal.

6.1 This Work's Major Contributions

The method herein presented correctly completes the state code for timed asynchronous systems. It handles systems with causality, concurrency, and choice. Unlike other existing methods, it takes full advantage of timing information. This allows state signal insertion in a sequential, yet non-causal, manner. It also enables the insertion of state signal transitions such that they preceed input transitions. This work is somewhat unique in that it also capitalizes on timing information by using stochastic performance estimates during solution selection.

6.2 Improvements to Multivariate Solutions

While systems requiring more than one state variable are properly state coded, some optimization opportunities are currently untapped in the area of multivariate insertion. At present, each state signal is inserted one at a time, and only two transitions are allowed for each state signal. If multiple transition pairs were allowed per state signal, it could decrease the number of variables needed in cases currently requiring many state variables. Improved results could be obtained if the total number of state variables necessary were estimated, then used to guide the overall state coding process. This type of estimate would also be useful for bounding partial solutions in a "branch and bound" manner.

6.3 Enhancements to Avoid Non-persistent Solutions

Both timed and speed-independent specifications have been successfully state coded using the current algorithm. The results obtained from this method compare favorably with those from established speed-independent algorithms in terms of final circuit performance. In terms of circuit area, there is clearly room for improvement. However, the large discrepancy in area is somewhat misleading. For many cases, it is because the state variable insertion has introduced non-persistency into the system. To correctly synthesize the now non-persistent system, additional logic is necessary. This non-persistency is not intrinsic to the method, but merely a side effect which manifests itself in some solutions. If the algorithm were modified to identify non-persistency as it occurred, and reject or deprecate these solutions, the resulting simpler implementations would have a synthesized area more in line with values reported by other methods. A method to identify non-persistency at an early stage in the algorithm has been identified, but additional work is necessary to refine and to validate it.

6.4 Other Areas for Future Work

The cost function currently in use is effective at choosing a solution by considering the relative importance of performance and area. However, there is no provision for establishing bounds on the area or performance. If a design with minimal area,

yet no more than some specified delay, is the target, the cost function's P parameter must be manually adjusted until such a solution is generated. Switching the solution selection method to a Pareto point based system would increase its flexibility.

The current deadlock resolution method is generously considered to be crude. A method with stronger theoretical underpinnings would do a better and more efficient job than the current brute force technique. This area should certainly be examined and resolved in the near future.

REFERENCES

[1] C. J. Myers, *Computer-Aided Synthesis and Verification of Gate-Level Timed Circuits.* PhD thesis, Dept. of Elec. Eng., Stanford University, Oct. 1995.

[2] P. Vanbekbergen, F. Catthoor, G. Goossens, and H. D. Man, "Optimized synthesis of asynchronous control circuits form graph-theoretic specifications," in *Proc. International Conf. Computer-Aided Design (ICCAD)*, pp. 184–187, IEEE Computer Society Press, 1990.

[3] 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*, pp. 568–572, IEEE Computer Society Press, June 1992.

[4] 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)*, pp. 112–117, IEEE Computer Society Press, Nov. 1992.

[5] C. Ykman-Couvreur and B. Lin, "Optimised state assignment for asynchronous circuit synthesis," in *Asynchronous Design Methodologies*, pp. 118–127, IEEE Computer Society Press, May 1995.

[6] C. Ykman-Couvreur and B. Lin, "Efficient state assignment framework for asynchronous state graphs," in *Proc. International Conf. Computer Design (ICCD)*, pp. 692–697, IEEE Computer Society Press, 1995.

[7] C. Ykman-Couvreur, B. Lin, and H. de Man, "Assassin: A synthesis system for asynchronous control circuits," tech. rep., IMEC, Sept. 1994. User and Tutorial manual.

[8] J. Cortadella, M. Kishinevsky, A. Kondratyev, L. Lavagno, and A. Yakovlev, "Petrify: a tool for manipulating concurrent specifications and synthesis of asynchronous controllers," *IEICE Transactions on Information and Systems*, vol. E80-D, pp. 315–325, Mar. 1997.

[9] A. Kondratyev, J. Cortadella, M. Kishinevsky, A. Taubin, L. Lavagno, and A. Yakovlev, "Lazy transition systems: Application to timing optimization of asynchronous circuits," in *IEEE/ACM International Conference on Computer-Aided Design*, 1998.

[10] W. Belluomini and C. J. Myers, "Timed event-level structures," in *Proc. International Workshop on Timing Issues in the Specification and Synthesis of Digital Systems (TAU)*, (Austin, Texas, USA), Dec. 1997.

[11] P. Merlin and D. J. Faber, "Recoverability of communication protocols," *IEEE Transactions on Communications*, vol. 24, no. 9, 1976.

[12] H. Hulgaard and S. Burns, "Bounded delay timing analysis of a class of CSP programs with choice," in *Proc. International Symposium on Advanced Research in Asynchronous Circuits and Systems*, pp. 2–11, November 1994.

[13] C. J. Myers and T. H.-Y. Meng, "Synthesis of timed asynchronous circuits," *IEEE Transactions on VLSI Systems*, vol. 1, pp. 106–119, June 1993.

[14] J. Cortadella, M. Kishinevsky, A. Kondratyev, L. Lavagno, and A. Yakovlev, "A region-based theory for state assignment in speed-independent circuits," *IEEE Transactions on Computer-Aided Design*, vol. 16, pp. 793–812, Aug. 1997.

[15] E. G. Mercer, "Stochastic cycle period analysis of timed asynchronous circuits," Master's thesis, Dept. of Elec. Eng., University of Utah, Jan. 1999.

[16] S. B. Furber and J. Liu, "Dynamic logic in four-phase micropipelines," in *Proc. International Symposium on Advanced Research in Asynchronous Circuits and Systems*, IEEE Computer Society Press, Mar. 1996.

[17] B. Coates, A. Davis, and K. Stevens, "The Post Office experience: Designing a large asynchronous chip," *Integration, the VLSI journal*, vol. 15, pp. 341–366, Oct. 1993.