SYNTHESIS AND TESTING OF BOUNDED WIRE DELAY ASYNCHRONOUS CIRCUITS FROM SIGNAL TRANSITION GRAPHS

by

Luciano Lavagno

Memorandum No. UCB/ERL M92/140

21 December 1992

SYNTHESIS AND TESTING OF BOUNDED WIRE DELAY ASYNCHRONOUS CIRCUITS FROM SIGNAL TRANSITION GRAPHS

by

Luciano Lavagno

Memorandum No. UCB/ERL M92/140

21 December 1992

ELECTRONICS RESEARCH LABORATORY

College of Engineering
University of California, Berkeley
94720

SYNTHESIS AND TESTING OF BOUNDED WIRE DELAY ASYNCHRONOUS CIRCUITS FROM SIGNAL TRANSITION GRAPHS

by

Luciano Lavagno

Memorandum No. UCB/ERL M92/140

21 December 1992

ELECTRONICS RESEARCH LABORATORY

College of Engineering
University of California, Berkeley
94720

Synthesis and Testing of Bounded Wire Delay Asynchronous Circuits from Signal Transition Graphs

Luciano Lavagno

Abstract

Synthesis and Testing of Bounded Wire Delay Asynchronous Circuits from Signal Transition Graphs

by
Luciano Lavagno

Doctor of Philosophy in
Electrical Engineering and Computer Sciences
University of California at Berkeley
Professor Alberto Sangiovanni-Vincentelli, Chair

The design of asynchronous circuits is increasingly important in solving problems such as complexity management, modularity, power consumption and clock distribution in large digital integrated circuits. The task is difficult mainly for the possible presence of hazards, i.e. deviations from the expected circuit behavior due to gate and wire delays. Efficient synthesis tools, which take into account the need for testing manufactured circuits, are required. The problem has been studied extensively in the past, but no satisfactory automated solution using a realistic delay model has been presented. This thesis introduces the problem through an extensive literature review and then proposes a synthesis procedure based on the bounded wire delay model commonly used in synchronous circuit design.

The first aspect of the proposed methodology is the formal modeling of asynchronous circuits both at the specification and implementation level. At the specification level, we describe a generic asynchronous system using an extended version of the well-known Signal Transition Graph specification (based on an interpreted Petri net and hence built upon a large theoretical research). At the implementation level, we develop a general model that can describe all asynchronous computation paradigms. This model uses an abstraction of the gate-level model for a compact but formal representation of complex subsystems whose classical description in terms of gates would be impossible or impractical. We then relate specification-level properties with implementation-level ones, so that the designer can restrict the specification to satisfy system-level requirements such as behavior independence from wire delays.

The second aspect of the methodology is a complete and formal procedure to implement a Signal Transition Graph specification as a hazard-free asynchronous circuit. The procedure involves a state assignment step, an initial implementation step, and a hazard elimination step. The hazards can be eliminated knowing bounds on the delays both inside the circuit (gates and wires) and in the environment. We show that the problem of optimal delay padding is NP-complete, and give a range of efficient exact and heuristic procedures to solve it.

The manufactured circuit operates correctly without hazards only if the delay bounds assumed during the synthesis process are actually met. We introduce an automated synthesis for testability procedure that preserves hazard-free operation while ensuring the testability of the delay bounds. This provides the last element of a complete top-to-bottom automated design methodology for asynchronous circuits.

Prof. Alberto Sangiovanni-Vincentelli
Thesis Committee Chairman

Contents

List of Figures

List of Tables

List of Theorems and Procedures

Acknowledgments

This work would not have been possible at all without the invaluable help of many wonderful people. They are in fact so many that I will certainly forget to mention some: my sincere apologies if this happened at all.

A few people played a very key role in persuading me that an easy, quiet life in Italy was not all I was born for. Tiziano Villa first had this apparently crazy idea, prof. Alberto Sangiovanni-Vincentelli believed him, and Paola supported me in the difficult time when I had to choose. They all deserve a very special place (I let the reader decide whether it should be in Heaven or Hell). Of course, their support was not just limited to the origin of the Berkeley adventure. My advisor needs no introduction, as anyone who reads this page certainly knows him very well already. He taught me how to do serious research, told me what topics were most interesting and promising, and scolded me a few times for failing to write readable papers or give a clear presentation. Tiziano has been a faithful friend for almost a decade now, and his last (but not least) effort was to read and comment this thesis. Paola was a constant supportive presence either in front of my eyes or (too often) only in the back of my mind, when my eyes were occupied by flickering characters on a computer screen. I would never have undertaken this path alone, and I will never forget her infinite patience (running out only sometimes after midnight).

After the three main culprits, there is a huge crowd of colleagues to thank. First of all, some of them helped me directly in doing the research that constitutes the body of this thesis. Prof. Alex Yakovlev, who was my friend for a long time only through e-mail, is the co-author of Chapter 3. I would never have done such a theoretical work alone. I owe to Kurt Keutzer, whose hand can be recognized in part of Chapter 5 and Chapter 6, the ability to write down the formal statement of a theorem and its proof. Cho Moon and Narendra Shenoy helped me in the work that now is part of Chapters 4 and 5. Prof. Robert Brayton has been a constant reference point for all members of the CADgroup at Berkeley: his steadfastness in listening to my ramblings and in helping me clarify my thoughts is admirable. Ellen Sentovich deserves a special mention for her effort in reading and correcting the thesis (which often included translating my Italo-Saxon dialect into English). Then I would like to collectively thank Rick Rudell, Albert Wang, Sharad Malik, Alex Saldanha, KJ Singh, Rajeev Murgai, Paul Stephan, Timothy Kam (my patient cubicle mate), Rick McGeer, and all the other temporary or semi-permanent occupants of Room 550, Cory Hall. All the people in Room 550, together with staff members (special thanks to Flora Oviedo, Elise Mills, Brad Krebs and Mike Kiernan), contributed to the creation of a uniquely fruitful and cooperative work environment I shall not forget. I also thank prof. Maxwell Rosenlicht for serving on my thesis committee even if he was far away, teaching in Ivory Coast.

Berkeley has been a wonderful place to live, thanks to its unique mix of academic excellence and human diversity. Many friends (besides those whom I thanked already for their "professional" roles, such as Alex, Ellen, KJ, Narendra, Tiziano, . . .) contributed to make it even better: Roberto and Marinella Guerrieri, Cliff and Teresa Price, Lev Ruzer and his family, Tatsuya Fujiye, and many others.

Finally, the lively asynchronous community offered me endless opportunities for useful discussions (and sometimes heated arguments). By a fortunate case in history, a sizeable portion of the asynchronous community lives (or used to live until recently) in the Bay Area, allowing a very tight interaction: Ken Stevens, Tam-Anh Chu, Teresa Meng, Peter Beerel, Chris Myers, David Dill, Steve Nowick, and Jeremy Gunawardena. Other examples of cooperation were less direct, but no less fruitful. I would like to mention in particular my friends (once from St. Petersburg, now in diaspora) Leonid Rosenblum, Michael Kishinevsky and Alex Kondratyev. I also gratefully acknowledge, among others, Jerry Burch (who with his formal verification techniques contributed to give me some confidence that the synthesis methodology worked after all), Steve Burns, Gaetano Borriello and Peter Vanbekbergen.

Chapter 1

Introduction

This thesis addresses the problem of automated asynchronous circuit design. It describes a design methodology that is supported by synthesis algorithms and that takes into account testability issues. The problems encountered during the algorithm implementation, and the experimental results obtained from them are also detailed.

1.1 Motivation

An *asynchronous circuit*, informally, is an arbitrary interconnection of logic gates, with the only restriction that no two gate outputs can be tied together (note that wired-*or* or wired-*and* circuits can be modeled as a gate for most practical purposes).

A *synchronous circuit* satisfies the further restriction that all cycles in it *must* be broken by clocked memory elements. The clocking scheme must be such that no event (i.e. a change of value of some signal inside the circuit) can propagate freely along a cycle without being "stopped" by a memory element that is not actively clocked when the event reaches its input. Furthermore, no event should reach a memory element at a time smaller than a certain pre-specified amount (setup/hold time) from the clock event.

This restriction makes the analysis, synthesis and testing of synchronous circuits much easier than their asynchronous counterparts. The presence of the clock allows the decomposition of a synchronous circuit into a set of blocks of combinational logic, i.e., sub-circuits composed of combinational gates, such as *and*, *or*, *inverter*, etc., and a set of memory elements interconnecting those blocks. The techniques for analysis, synthesis and testing of such decomposed circuits are well developed, widely used, and subject of a large amount of excellent research.

The cost of this simplified handling of circuit behavior may be high. A synchronous system must have *only one clock signal* (or a set of clocks, if multi-phase clocking is used), and the circuits operates properly only if each clock event reaches all memory elements within a tightly controlled time bound (the *clock skew*). This requirement is not easily satisfied, because the system may be built out of components that are physically far apart, so that the finite speed of propagation of events becomes noticeable. Consequently, a key problem of modern circuit design is the distribution of clocks to a set of integrated circuits, and within the integrated circuits themselves, the minimization of the clock skew among the memory elements. Informally, the clock skew gives a hard limit on the maximum speed of operation of the system, so the larger the skew is, the slower the clock must be.

Furthermore, the clock event is usually sent to all memory elements, regardless of whether their inputs have changed since the last event. This means that *power* is wasted in charging and discharging capacitors in the circuit, along resistive wires.

There are implementation technologies with very high quiescent power dissipation (e.g. *NMOS*), in which the clock power dissipation is negligible with respect to the total. There are also technologies with a quiescent power dissipation that is almost zero (e.g. static *CMOS*). For these technologies, the clock power dissipation can be a substantial fraction of the total. It is possible in power-critical applications to "stop the clock" as long as nothing changes at the external inputs of the circuit, but these techniques are *asynchronous* in nature.

Finally, in a synchronous circuit the data input of a memory element must be stable near the clock event. So the clock period must be chosen to be long enough so that in the *worst case* the combinational logic has finished propagating events when the clock event occurs. It is well known that the worst case delay along a combinational circuit is a function of its input values. For example, the output of a ripple-carry adder stabilizes very quickly if there is no carry propagation, while it has a worst case propagation time proportional to the number of bits if the carry "ripples" from the least significant bit to the most significant bit. Similarly, a controller unit may have very simple logic to handle "common" operation cases, and much more complex logic to handle "uncommon" conditions such as interrupts. In both cases, a synchronous circuit must be clocked taking the worst case into account.

Asynchronous circuits handle all the problems above more gracefully. Namely, the problem of clock skew disappears completely or is replaced by much more "local" versions of it, in which the need for the same event to be received almost at the same time by various sub-circuits is not global to the system, but is a *local property* [106].

The elimination of the clock may improve the overall power consumption of the circuit. Alternately, an asynchronous design methodology can be used to design the "clock stopping" subsystem. A problem to be kept in mind when doing power consumption analysis, though, is that the different implementation requirements of asynchronous circuits may nullify the advantages. For example, the number and/or the size of gates that must switch per time unit may increase substantially with respect to the synchronous case, thus increasing the overall power consumption.

An example of this is the use of dual-rail encoding for completion detection. In this case, each abstract signal is implemented as *a pair of wires* (*dual-rail encoding*). The circuit signals that it is either still computing (both wires are at 0) or finished (exactly one wires goes to 1, encoding a data value of 0 as 01, and a data value of 1 as 10). This kind of completion detection can double the complexity of the circuit, and thereby increase its power consumption.

In addition to power consumption considerations, replacement of a single component of an asynchronous system with a faster one is much easier than in the synchronous case. The whole system adjusts itself to the new speed without the need for performing timing analysis again and replacing the clock generation circuitry.

Finally, an asynchronous design methodology is necessary in some cases, because there are pre-existing asynchronous specifications that the designer must satisfy: for example, a bus protocol. In this case, at least a part of the implementation must be done with asynchronous techniques.

However, asynchronous design is harder and more constrained than synchronous design, due to *hazard* and *meta-stability* problems. Asynchronous circuits are by definition sensitive to all signal changes, whether they are intentional (part of the specification) or not (then they are called hazards). An example of such unintentional changes is the oscillation of a signal that is supposed to have a single transition. Furthermore, if two transitions with conflicting effects (e.g. the datum and the clock transition on a latch) occur too close together, then the circuit may produce a meta-stable output that is neither 0 nor 1 and can last for an unbounded amount of time.

Asynchronous design was a popular design style when the complexity of the circuits permitted hand analysis of hazard and meta-stability problems. As designs became too complex to be done by hand, synchronous styles became more popular, since automated tools helped manage the increase in complexity. Asynchronous methodologies lagged behind until recently, when a new wave of interest was spurred by the advance both in automated circuit synthesis techniques and in new specification formalisms.

This thesis brings together a wide range of techniques from theoretical computer science and electrical engineering. These techniques give the asynchronous circuit designer automated tools to help in all aspects of design, from specification to implementation and testing. We hope that it will have practical applications as well as theoretical interest.

The remainder of this chapter is devoted to a complete example showing the implementation of a *VMEbus* master interface. It provides an overview of the main steps of the proposed design methodology. The rest of the thesis will give a more formal definition of the terminology and the algorithms.

1.2 Overview of the Design Methodology

A formal design procedure of any kind of physical device requires a *mathematical model* of the device operation. The closer this model is to reality, the better the results that can be obtained from the procedure. Historically, two main models of operation for an asynchronous circuit have been used (Chapter 2 describes the models and some synthesis methodologies in more detail):

- the Huffman model, in which the circuit is decomposed into a combinational circuit and a set of feedback wires ([53]), and a *bounded* delay is associated with each *interconnection* between gates (*bounded wire delay model*), and
- the Muller model, in which the circuit is decomposed into gates with an arbitrary interconnection ([88]), and an *unbounded* (but finite) delay is associated with each *gate output* (*unbounded gate delay model*).

The classical asynchronous circuit design methodology based on the Huffman model (see, e.g., [53], [115] and [117]) starts from a Finite State Machine-like specification, the Flow Table, and produces logic equations implementing it. Unfortunately, the resulting circuit is hazard-free only under the *Fundamental Mode* assumption, i.e., the designer has to make sure that the circuit inputs can change only when the circuit itself is stable and ready to accept them. This verification task is non-trivial, and requires the use of the *bounded wire* delay model to analyze the circuit. This delay model is both more expensive to analyze ([4, 72]) and less reliable in the presence of manufacturing process variations than models that do not make any assumption on delay bounds. Moreover, the methodology ensures hazard-free operation by *increasing the delay* of some state signals (to eliminate the so-called *essential hazards*).

Muller (see [88] and [83]) introduces an *analysis* technique that verifies that a gate-level circuit is *speed-independent*. Speed-independence ensures correct operation of a circuit modeled using the *unbounded gate* delay model. This model is used in the synthesis methods proposed, for example, by [5], [126], [87] and [8]. Unfortunately, the unbounded gate delay model is not realistic: it ignores technological limits on the delays, which is *pessimistic*, and ignores the wire delays, which is *optimistic* ([118]).

The *unbounded wire* delay model ([116]) can be, in principle, the delay model that is most realistic and robust with respect to manufacturing process and environmental variations. Unfortunately it cannot be used, as shown by [94] and [77], to build circuits out of "basic" gates (*and*, *or*, ...). Authors who proposed design methodologies for circuits that operate correctly with this model (*delay-insensitive* circuits) are forced either to assume that *some wire delays* are almost zero (*isochronic forks*, [23]) or to take delays into account within hand-designed logic modules ([101], [40], [17], [57], [113]).

The *Signal Transition Graph* (STG) model was introduced independently by Rosenblum *et al.* ([102]) and by Chu ([26, 27]), to model formally both the circuit and the environment in which it operates. The STG can be considered a formalization of the widely used *timing diagram*, because it describes the *causality relations* between transitions on input and output signals of the specified circuit. Unlike other proposed causality-based models, e.g. the *Change Diagram* ([62]), it allows the explicit description of data-dependent *choice* between various possible behaviors. The standard synthesis methodology from the STG specification ([27], [81]), though, still has the partial drawback of being based on the unrealistic Muller model of the circuit implementation.

For these reasons, we chose to use the STG specification and the bounded wire delay model, providing the necessary level of formality and a realistic model of the circuit behavior. A characterization of the formal properties of various classes of STGs is described in Chapter 3.

Our synthesis procedure offers a set of algorithms with proven properties that have been implemented in the sequential logic synthesis system SIS ([108, 107]). Namely, in Chapter 4 we will describe a state encoding method for STG specifications. In Chapter 5 we will show in more detail how an STG can be implemented in a hazard-free circuit using the realistic *bounded wire delay* model. Finally, in Chapter 6 we will show how the circuit implementation can be tested for all *delay faults* that may cause hazards and impair its operation. Our methodology is based on a *rigorous mathematical formulation* of the specification and synthesis tasks, and uses a *realistic delay model*, the bounded wire delay model.

The remainder of this chapter summarizes the complete design flow, from the *specification* to the *implementation*. After a brief informal introduction of the STG syntax and semantics, we describe how to translate a realistic and complex specification of a *VMEbus* master interface into an STG, and how to obtain a hazard-free implementation of the interface from the STG.

1.2.1 Signal Transition Graphs

A *Signal Transition Graph* can informally be considered as a collection of:

- *Causality relations* between signal transitions. For example a relation $x^+ \rightarrow y^-$ means that a rising transition on signal x is followed by a falling transition on signal y. It is interpreted either as a specification of the *circuit* behavior, if y is an *output* signal of the circuit, or as a specification of the *environment* behavior, if y is an *input* signal. Note that no information on the elapsed *time* between the two transitions is given: it must only be *positive* and *finite*.

- *Choice* and *merging* points between different computation flows. For example, an acknowledgment transition may be followed either by a write request or by a read request, each one following its own protocol. The choice whether to perform a read or a write operation must causally depend on some other operation, since digital logic circuits are in general *deterministic*[1]. Conversely, it is often convenient to use non-determinism as a form of *abstraction* to describe more compactly implementation details that are of no concern during a particular synthesis step. We allow only *input* signals of the circuit to have the kind of non-deterministic behavior outlined above.

In *timing diagram* terms, causality relations are represented as directed arrows between signal transitions, and choice/merging points are described by informal textual notes (even though there have been formalization attempts, such as [14] or [54]). For example, Figure 1.1.(a) describes the interaction between three signals, x, y and z. When x rises, y falls and z rises. No order or explicit timing is specified: y^- and z^+ are *concurrent* and *caused by* x^+. Subsequently y may rise again or stay at 0. No reason for this event is given, hence we assume that it depends on something else "hidden away" by abstraction. We cannot synthesize a logic implementation for y given only this timing diagram (while we can do so for z, because the set of events causing its transitions is completely specified). Finally x can fall, and if y is high, then z must fall. Note that this apparently "innocent" behavior hides a possible cause for meta-stability if y^+ and x^- occur too close together. We will return to this problem later.

This *non-deterministic* behavior can also be represented as in Figure 1.1.(b), where the first segment of the diagram non-deterministically *chooses* between two different, mutually exclusive

[1]*Arbitration* circuits are a different matter, since they have to be designed with analog techniques in order to alleviate the meta-stability problem. Our synthesis methodology does not deal with arbiters directly.

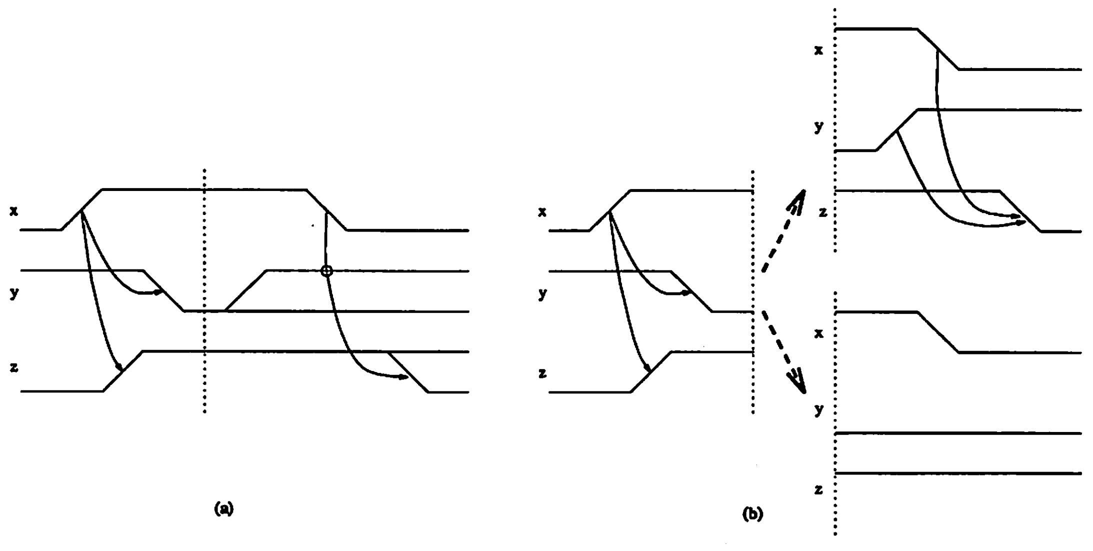


Figure 1.1: A timing diagram with non-deterministic behavior

behaviors. As in Figure 1.1.(b), the event that "witnesses" which choice is made (y^+ in this case) must be occurring on an *input* signal of the circuit that is being specified. Non-determinism is allowed only in the environment.

Petri nets and Signal Transition Graphs

Now we will examine in more detail the formal definition of a Signal Transition Graph, and describe its connection with timing diagrams. An STG is an *interpreted Petri net.* A Petri net ([97, 96, 89]) is composed of a set of *transitions* and a set of *places.* A *flow relation* connects each transition with a set of *predecessor* places (representing the *pre–conditions* of the transition) and a set of *successor* places (representing its *post–conditions*).

A *marking* of a net assigns a non-negative number of *tokens* to each place in the net. A transition is *enabled* if all its pre–conditions are marked with at least one token. If it is enabled, then it can *fire*, which means that one token is removed from each pre–condition and one is added to each post–condition. Unless otherwise stated we will assume that every net is given an *initial marking* (corresponding, as we will see, to the *initial state* of the circuit and its environment).

An example of a Petri net is shown Figure 1.2.(a). Squares represent transitions, circles represent places, directed edges represent the flow relation, and black dots represent the tokens in the initial marking. The *reachability graph* of the net, shown in Figure 1.2.(b), represents all the markings reachable from the initial one through sequences of *transition firings*. For example, only

t_0 can fire in the initial marking, corresponding to p_5 and p_6 being marked (top center vertex in Figure 1.2). When it fires, p_0 becomes marked, and so on.

A place in a Petri net can represent both a *causality relation* and a *choice/merging* point. For example:

- place p_1 specifies that transition t_3 *must* follow transition t_1.
- place p_0 specifies that *either* t_1 *or* t_2 must follow t_0. This mutual exclusion is due to the conjunction of two facts:
  - t_1 and t_2 share a predecessor place p_0, hence when p_0 is marked, the first firing transition removes one token from it.
  - no marking reachable from the initial marking has more than one token in place p_0. Hence the first firing transition among t_1 and t_2 *disables* the other one, because it leaves p_0 without tokens.

  p_0 describes a *choice* point in the operation of the system described by the Petri net.
- p_5 represents a *merging* point between two flows, because it can be marked by two different predecessors.

If we now *interpret* Petri net transitions as transitions of signals, we obtain the STG specification of asynchronous circuits. In order for the STG to have a consistent interpretation, we must be able to identify each *marking* with a unique *set of values* for the specified signals. We require that each marking m in the reachability graph can be *consistently labeled* with a vector of signal values $\lambda(m)$, where bit $\lambda(m)_i$ is the value of signal z_i in marking m, and for each edge in the reachability graph $m \rightarrow m'$:

- $\lambda(m)_i = 0$ and $\lambda(m')_i = 1$ if the edge is labeled z_i^+,
- $\lambda(m)_i = 1$ and $\lambda(m')_i = 0$ if the edge is labeled z_i^-,
- all other bits have the same value in $\lambda(m)$ and $\lambda(m')$.

Note that this labeling (which is unique, given an initial marking) associates the initial marking with the *initial state* of the circuit.

Let us examine the timing diagram in Figure 1.3.(a). Each signal, in this case, is assumed to be an *output* signal, i.e. one for which we must synthesize a circuit implementation. If we assume

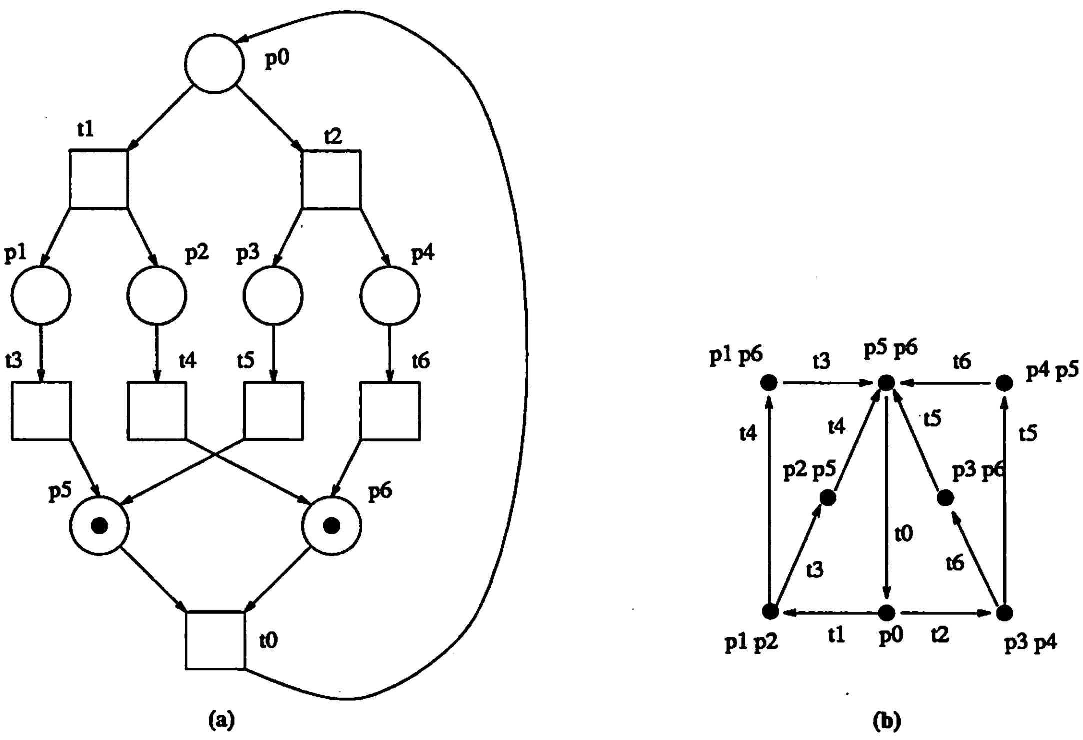


Figure 1.2: A Petri net and its reachability graph

that it represents a cyclic behavior, where the first and the last rising transitions of x coincide, then it can be translated into an equivalent STG, represented in Figure 1.3.(b). Note that in the standard representation of STGs places with only one predecessor and one successor (*implicit* or *trivial* places) are omitted, and transitions are denoted directly by the corresponding signal transition label. Thus there is an implicit place between transitions y^- and x^+, which is initially marked.

The labeled reachability graph, or *State Transition Diagram* (STD), of this STG appears in Figure 1.3.(c), for signal ordering xyz. The initial marking of Figure 1.3.(b) corresponds to the state labeled 0*00. The labels in Figure 1.3.(c) use four values for each signal: "1" means a stable 1, "1*" means a 1 that is about to change to 0, and similarly for "0" and "0*". The marking labeled 10*0* corresponds to the implicit places between x^+ and z^+ and between x^+ and y^+ being marked. Signal x has value 1, because x^+ has just fired. Signal y has value 0* because y^+ has not fired yet, but is *enabled* (similarly for signal z).

In this section, following [27], we will further restrict a valid STG to be

- *live*: from every reachable marking there exists a sequence of transition firings that will eventually fire every transition.

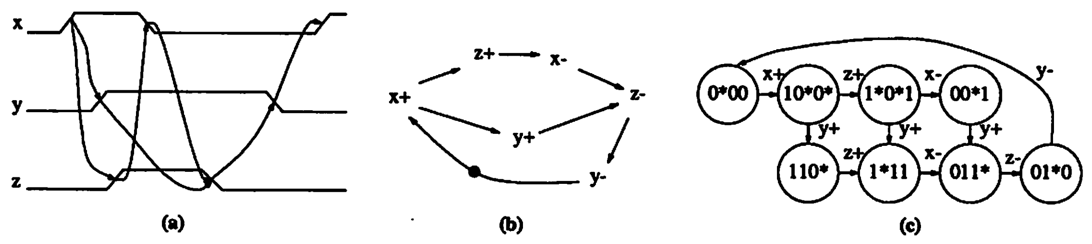


Figure 1.3: A timing diagram, its Signal Transition Graph and its State Transition Diagram

- *safe*: no reachable marking can have more than one token per place.
- *free-choice*: if a place has more than one successor transition, then it must be the ***only predecessor*** of those transitions. That is, a choice must not depend on the marking of other places

These restrictions are not inherent to the use of STGs to *specify* asynchronous circuits *per se*, but are only required to guarantee the success of some STG synthesis algorithms (for example, the state encoding algorithm described in Chapter 4). Chapter 3 contains a detailed analysis of the implications of such restrictions on the modeling capabilities of STGs.

1.2.2 Signal Transition Graph Synthesis

The STG specification can now be transformed into a logic circuit implementation that is hazard-free using the *bounded wire delay* model. Under this delay model, the circuit must operate correctly according to the specification when modeled as a connection of

- delay-free logic gates (in this section we will use a standard cell library).
- one *pure* delay element (i.e., a translation in time of the input waveform) for each wire connecting a circuit input or a gate output with a gate input. An upper and lower bound on the magnitude of this delay must be known before the synthesis begins.

The synthesis process, described in more detail in Chapters 4 and 5, consists of the following phases:

1. State encoding, in which the STG is transformed into a Flow Table (a Finite State Machine-like specification for asynchronous circuits, see e.g. Section 2.3.1), which is then minimized and encoded using a critical-race-free algorithm ([115]). The minimization and encoding information is used to insert state signal transitions back into the STG.

2. Initial synthesis, where a two-level logic implementation is derived from the STG. All the STG signals may be used as inputs to the logic implementing each output signal, and the logic must satisfy two conditions:

   (a) It must map every STD label into the corresponding *implied value* of the output signal. The implied value of an output signal in a marking is defined as follows:

      - the value of the signal in the marking label if no transition of that signal is enabled in the marking.
      - the complement of that value otherwise.

      Enabled signals are denoted by a trailing "*" in Figure 1.3.(c). In the marking labeled 0*00, signal x has an implied value of 1 because x has value 0* in that label (the first bit). In the marking labeled 10*0*, x has an implied value of 1, the same as the value of x in its label, because no transition for x is enabled in it.

   (b) Its output must not depend on the firing order of *concurrent transitions* whose firing does not enable it. For every reachable marking in the STG, there is a set of transitions *concurrently enabled* in it. For example, in the initial marking in Figure 1.3 the (singleton) set $\{x^+\}$ is enabled. In the next marking, the set $\{y^+, z^+\}$ is concurrently enabled and so on. The specification does not prescribe an order of occurrence for those transitions, so we want the output signal value to be *stable independently of the firing order* of those transitions.

      The implementation is two-level, hence for each reachable marking, for each set of concurrently enabled transitions that does not cause a change of value in the output signal:

      - if the marking has an implied value of 1, then there must be an *and* gate whose output is 1 under the corresponding input label, and that does not have any input in the enabled set, so its output is constant 1.
      - if the marking has an implied value of 0, then every *and* gate must have some input that forces its value to 0 and that is not in the enabled set, so their outputs are constant 0.

For example, we can derive the two-level implementation shown in Figure 1.4 for the output signals of Figure 1.3. The reader can verify that this implementation satisfies both condi-

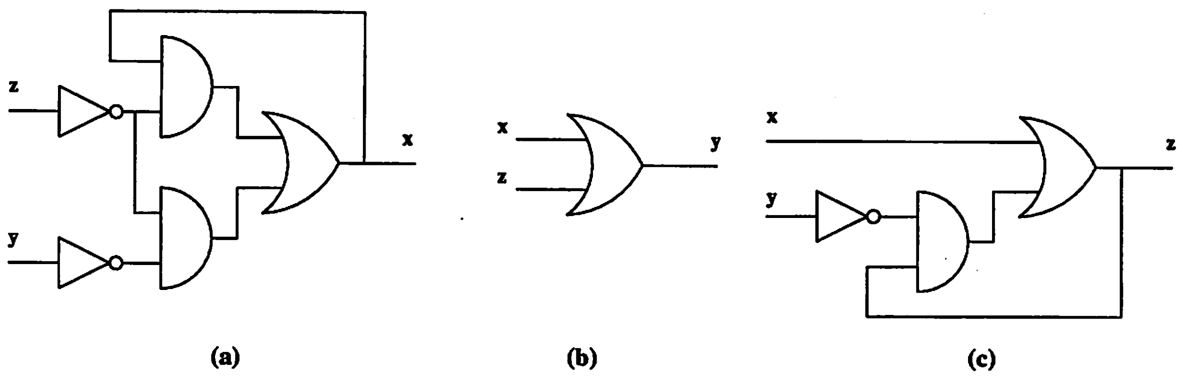


Figure 1.4: A two-level implementation of the Signal Transition Graph of Figure 1.3

tions 2a and 2b (i.e. it produces the correct implied value in each marking, and it has a stable *and* gate under every set of concurrent transitions that should not affect each signal).

3. Hazard analysis, in which the STG specification and the initial implementation are analyzed to detect the presence of a hazard condition under some *wire delay* assignment. We will show in Chapter 5 that a hazard can only occur in circuits synthesized according to our methodology if some *causality relation* specified by the STG is *reversed* due to delays inside the circuit. This notion is similar to the classical notion of *essential hazard*, as described, e.g., in [117].

   For example, transition y^+ enables transition $z-$ in Figure 1.3. If the circuit path between y and output x is very slow (e.g. due to the bottom *inverter* gate in Figure 1.4.(a)), then it is possible for $z-$ to propagate faster than y^+ to the circuit implementing x. At this point the *and* gate has both inputs at 1, so it may incorrectly be turned on (a *hazard*). As a result, x is set to 1 when a constant value of 0 was specified for it in the STG.

   The hazard analysis produces a set of triples (z_a^*, z_b^*, z_h) that represent potential hazards. z_a^* and z_b^* are transitions, z_a^* enables z_b^* (possibly through some intermediate transition) and z_h is an output signal. Each triple implies that a hazard will occur if the circuit delay from z_a^* to z_h is larger than the delay from z_a^* to z_b^* plus the delay from z_b^* to z_h.

4. Constrained logic synthesis, in which a set of transformations based on the distributive, associative and DeMorgan's laws are used to optimize and implement a multi-level circuit from the initial two-level one, using any available gate library. For example, the function for x in Figure 1.3 can be implemented as shown in Figure 1.5. We will show in Chapter 5 that this set of transformations produces a circuit in which the set of potential hazards generated in the hazard analysis step remains complete and correct.

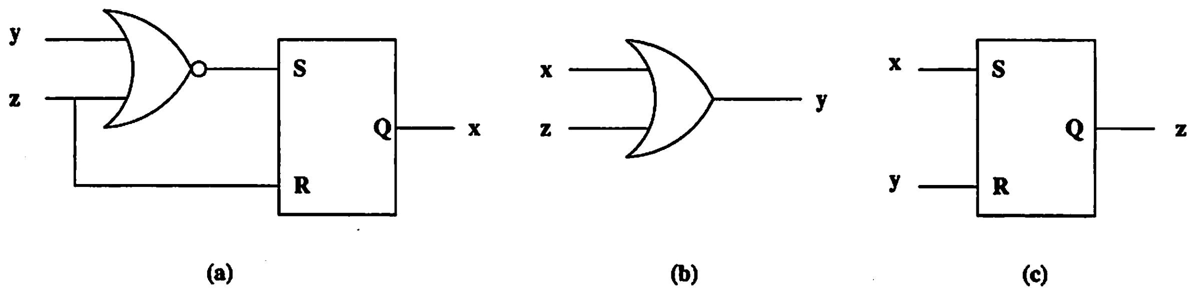


Figure 1.5: A standard cell implementation of Figure 1.4

5. Hazard elimination, in which the timing constraints imposed by the potential hazards derived during hazard analysis are analyzed. Whenever a violation of a constraint is detected, it can be eliminated either by using logic synthesis to balance the delays $z_a^* \rightarrow z_h$ and $z_b^* \rightarrow z_h$ or by *increasing the delay* at the output of z_b^*. This corresponds to ***slowing down each signal to allow all transitions that caused its change to finish propagating***. The operation is similar to slowing down the clock in synchronous circuits to allow all transitions to finish propagating. For hazard elimination, the slow-down is done locally for each signal. We will also show in Chapter 5 that this delay insertion procedure always produces a hazard-free circuit.

1.2.3 The *VMEbus* Master Interface Protocol

In this section we describe our main example, taken from [109]: a master interface designed according to the *VMEbus* asynchronous communication protocol (see also [49]). All the signals are active low, and bus lines are open-collector.

Our circuit receives as inputs:

- the access request from the master *bcsl*,
- the daisy chain grant from the previous board *bgninl*,
- the address synchronization from the master *basl*,
- the address synchronization from the bus *asl* (this signal is also an output),
- the write request from the master *bwrl*,
- the data transfer acknowledgment from the bus *dtackl*.

It produces as outputs:

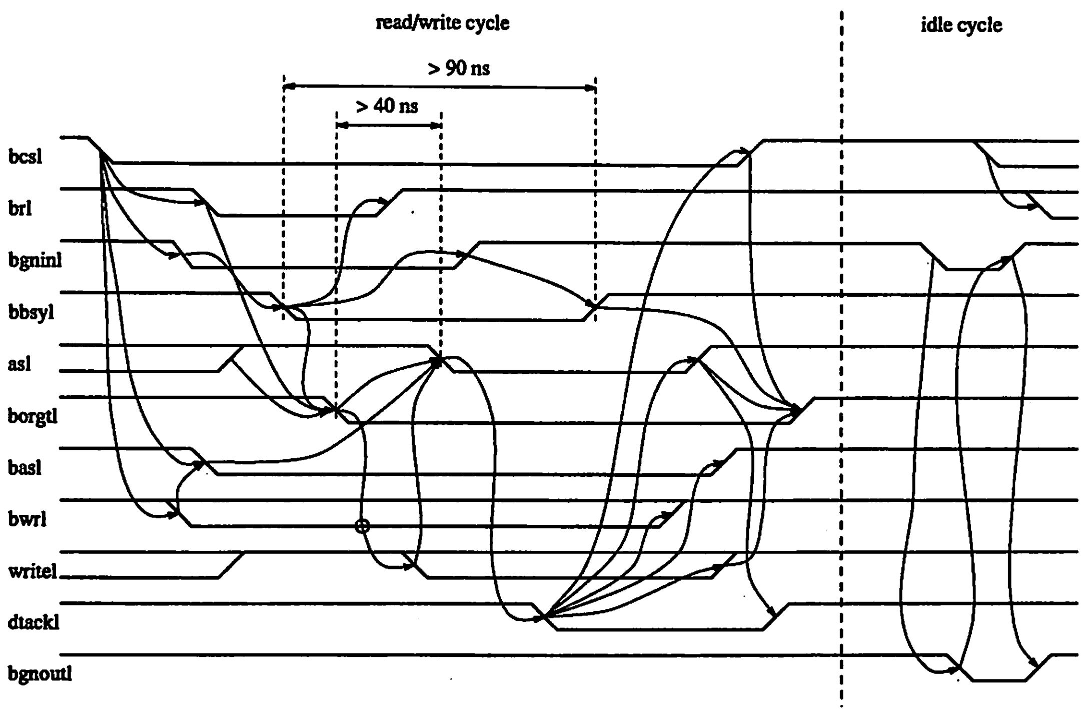


Figure 1.6: The *VMEbus* master interface timing diagram

- the bus request to the centralized arbiter *brl*,
- the daisy chain grant to the next board *bgnoutl*,
- the acknowledgment to the arbiter *bbsyl*,
- the access grant to the master *borgtl*,
- the write request to the bus *writel*.

The protocol proceeds as follows (see the timing diagram in Figure 1.6):

1. The master requests access to the interface by asserting *bcsl*. When the address is ready, it asserts *basl*. During a write cycle it must assert *bwrl* between *bcsl* and *basl*.
2. The interface requests access to the bus from the centralized arbiter by asserting *brl*.
3. The arbiter grants access to the bus to the first board on the grant daisy chain by asserting its *bgninl*. We now have two cases:

- each interface that did not receive *bcsl-* when it receives *bgninl-*, passes *bgnoutl-* down the chain (this is represented in the "idle cycle" section of the diagram). If it receives *bcsl* after *bgninl*, then it asserts *brl* and waits for the next grant.
- otherwise, the interface keeps *bgnoutl* high and starts its cycle, by asserting *bbsyl* for at least 90 ns to inform the arbiter that the bus grant was received, and it releases *brl*.

4. After asserting *bbsyl*, the interface waits for the previous transaction on the bus to complete; that is, it waits for *asl* to be high, then it asserts *borgtl* to inform the master that the bus is available. It also asserts *writel* if *bwrl* was asserted to indicate a write cycle.
5. After *bbsyl* falls, the arbiter causes *bgninl* to go high, because the grant is acknowledged. Also *brl* is released at this time.
6. The interface releases *bbsyl* after the rising edge of *bgninl*.
7. The interface now waits for the bus lines to settle (i.e., it waits for *asl* to be high for at least 40 ns) and for the master to assert *basl*, indicating that the address is ready. The interface then asserts *asl*.
8. The addressed slave asserts *dtackl*. This causes the master to release *bcsl*, *basl* and *bwrl* and the interface to release *asl* and *writel*.
9. The slave responds to the rising edge of *asl* by releasing *dtackl*.
10. The interface waits for *bcsl*, *bbsyl*, *writel* and *asl* to rise, and it signals to the master that its cycle is complete, by releasing *borgtl*.

We can easily see that there is a potential *synchronization* problem at step 3, because depending on whether *bcsl* or *bgninl* falls first, the interface responds by asserting *bbsyl* and *borgtl* in the former case, and *bgnoutl* in the latter case. The next section will show how this problem can be handled within the proposed design methodology.

1.2.4 A Signal Transition Graph Specification for the *VMEbus* Interface

We are now ready to model the *VMEbus* protocol described in the previous section with a Signal Transition Graph. The timing diagram is split into two parts, to keep the synchronization problem separated from the rest of the cycle. The STG that handles the bus arbitration appears

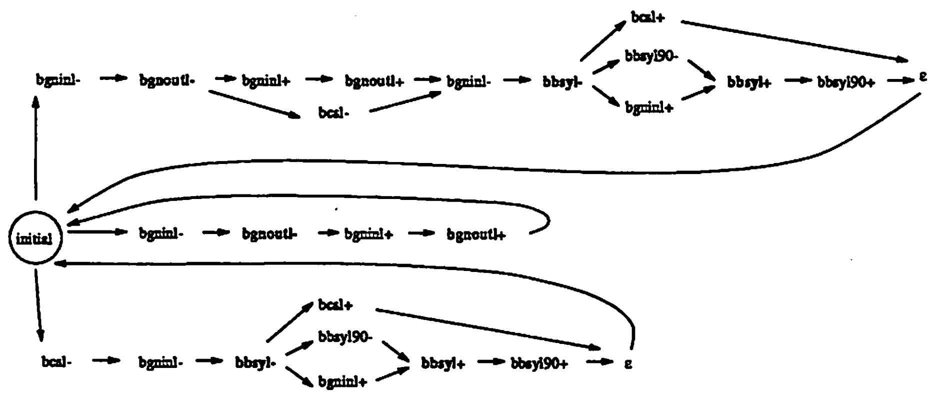


Figure 1.7: The *VMEbus* arbitration Signal Transition Graph

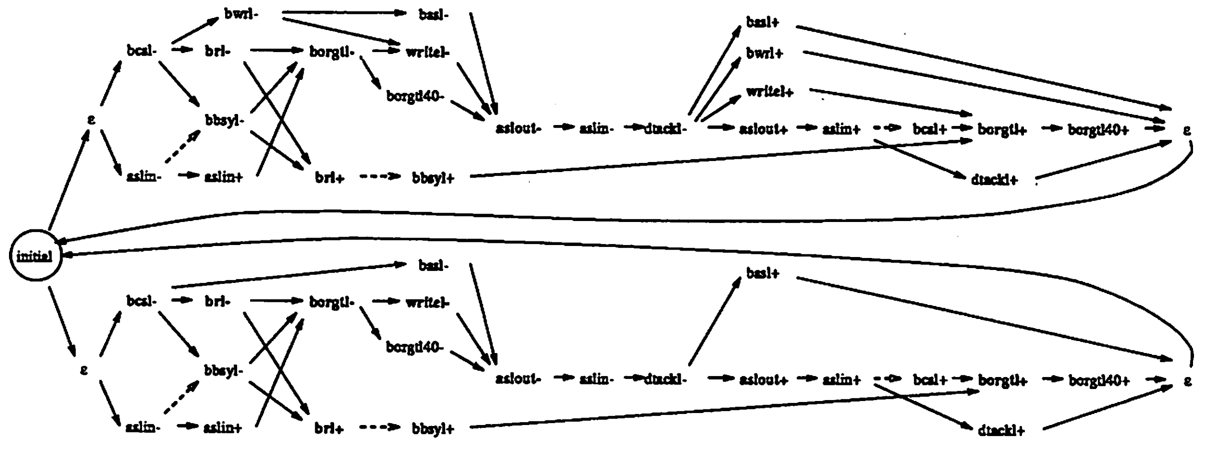


Figure 1.8: The *VMEbus* read/write Signal Transition Graph

in Figure 1.7. The STG that deals with the read/write transaction appears in Figure 1.8. The translation required to add some constraints, shown by dashed lines in Figure 1.8 and described more in detail below.

Output signals are shown in **bold** font. Note that we use *empty* transitions (i.e. transitions that do not correspond to the change of value of a signal), which are denoted by ϵ, as "placeholders" to ensure the STG safeness. For example, if in Figure 1.8 the place *initial* had been directly connected as a successor of *borgtl40+*, *basl+* and so on, then it would be marked with more than one token at the end of a cycle, and the STG would not be safe. Now let us examine each STG in more detail.

The Arbitration Signal Transition Graph

The arbitration STG, shown in Figure 1.7, is composed of three cycles, which are joined by a choice/merge place, *initial*, which corresponds to the initial state. The place has three successor transitions:

1. If *bgninl-* fires and then *bcsl-* fires, then we enter a *delayed grant* cycle. In this case, the interface must first pass the grant to the next board on the daisy chain by asserting *bgnoutl*. It must then wait for a second *bgninl-* before asserting *bbsyl*, which signals both the arbiter and the read/write section of the STG that the bus is granted.

2. If only *bgninl-* fires, then we enter an *idle* cycle. In this case, the interface simply passes the grant to the next board on the daisy chain.

3. If *bcsl-* fires before *bgninl-*, then we enter a *grant* cycle, by asserting *bbsyl*.

Signal Transition Graphs can describe only *causality* and *choice*, so they cannot handle *synchronization* problems directly. Nevertheless we can still model approximately the race between *bcsl* and *bgninl* as a *non-deterministic choice* between its possible outcomes, as shown above. This ensures that the logic produces the correct values, but it "hides away" from the synthesis system the fact that there is a potential synchronization problem.

In order to solve this problem we can, for example, use the circuit in Figure 1.9.(a), where a level-sensitive latch is used to synchronize *bcsl* to *bgninl*. *bgninl* and *bcsl* denote the signals received by the interface board, while *bgninl_del* and *bcsl_del* denote the signals that are used as inputs to the logic for *bgnoutl* and *bbsyl* (as described in Section 1.2.5).

The delay must be chosen large enough to satisfy two constraints:

1. It must allow the latch to leave *meta-stability*. The probability of a meta-stable condition decreases exponentially with its duration, so the delay can be chosen to make the probability of *bcsl* being in a meta-stable state when *bgninl-* is received arbitrarily small.

2. It must ensure that the STG constraint between *bgnoutl-* and *bcsl-* is satisfied, i.e., its delay must be greater than the delay between *bgninl-* and *bgnoutl-* (available after the synthesis procedure described in Section 1.2.5).

The waveforms for the cases when *bcsl-* arrives before, after and together with *bgninl-* appear in Figure 1.9.(b). The oscillation in *bcsl_del* causes no harm, because, as shown in Fig-

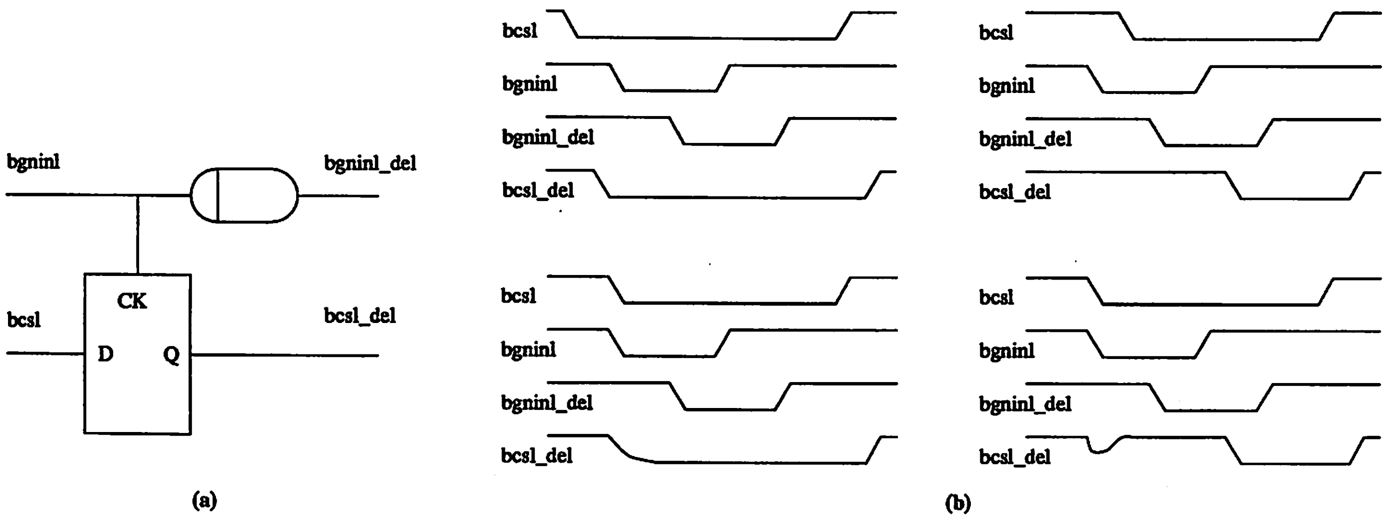


Figure 1.9: A synchronization circuit for *bcsl*

ure 1.11, neither the circuit implementing *bgnoutl* nor the circuit implementing *bbsyl* are sensitive to the value of *bcsl_del* when *bgninl_del* is high.

The Read/Write Signal Transition Graph

Signal *asl* is represented in this STG by two signals *aslin* and *aslout* because it is both an input (when it signals that the previous transaction is complete) and an output (during the current transaction). Due to the open-collector implementation, every transition of *aslout* causes a similar transition of *aslin* (the *asl* line is driven by one board at a time).

The read/write STG has two cycles, selected by the *bwrl-* transition. Each cycle proceeds after the initial choice exactly as in the timing diagram, except for the following additional constraints:

1. If *asl* goes low before the cycle begins due to some other master accessing the bus, then it must go low *before bbsyl-*. Otherwise, when the interface receives the grant, it cannot tell whether it can already access the bus or not.

2. The constraint between *aslin+* and *bcsl+* was added by the state encoding procedure. It means that we must constrain the master board design to wait for *asl* to go high to complete the transaction and release *bcsl*.

3. The constraint between *brl+* and *bbsyl+* was added because every transition in a well-formed STG must have at least one successor. It does not appear in the arbitration STG because *bbsyl*

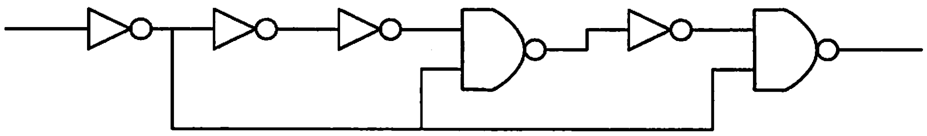


Figure 1.10: A delay line for the falling transition only

would then have *brl* among its inputs and we would lose the nice separation between *bbsyl* and the rest of the circuit. The constraint is certainly satisfied because the minimum duration of *bbsyl* low, 90 ns (Figure 1.6), is much larger than any realistic circuit delay between *bbsyl* and *brl*. Hence *brl* has time to rise before *bbsyl* rises.

The minimum delay of 40 ns between the time when *asl* is detected high and the next falling transition of *asl* is implemented with an external timer (e.g an *inverter* chain). The minimum duration of 90 ns for *bbsyl* low is implemented similarly. The STG has two input signals *bbsyl90* and *borgtl40*, corresponding to *bbsyl* and *borgtl* delayed by 40 ns and 90 ns respectively, which are used to delay *aslout-* and *bbsyl+*.

Note also that *borgtl40* and *bbsyl90* are both required to close the cycles. This means that the external delays should be quickly "reset" to 1, using, for example, the circuit of Figure 1.10 rather than an *inverter* chain. Moreover, the actual end-of-cycle acknowledgment sent back to the master must be delayed to wait for them to be high (e.g., with an *and* gate with inputs *borgtl*, *borgtl40* and *bbsyl90*).

1.2.5 The Circuit Implementation of the *VMEbus* Master Interface

The Signal Transition Graphs derived in the previous section can now be used as input to the synthesis algorithm outlined in Section 1.2.2 (and described in more detail in Chapters 4 and 5).

The first step involves state encoding. The arbitration STG does not require any state signal beside the internal feedback of its two output signals.

One state signal (called *is0*) is required to obtain a valid circuit implementation for the read/write STG. Furthermore, the state encoding algorithm required the insertion of a constraint between *aslin-* and *bcsl-* in order to complete successfully (see Section 4.4.1). The following transitions of *is0* are added to the STG, both in the read and write cycles:

- A rising transition *is0+* is inserted with predecessors *bbsyl-* and *brl-* and successors *borgtl-* and *brl+*.

- A falling transition *is0-* is inserted with predecessor *aslin-* and successor *dtackl-*. Strictly speaking, conditioning *dtackl-* to follow *is0-* is an illegal operation, because it amounts to constraining an *input* signal. However, we can assume that the off-chip delays are so large that *is0* can change before *dtackl* does.

After state encoding, an initial two-level logic implementation is derived for each output signal:

$$\begin{aligned}
bgnoutl &= \overline{bbsyl} + \overline{bcsl} \cdot bgnoutl + bgninl \\
bbsyl &= \overline{bbsyl90} \cdot bgninl + bbsyl \cdot bcsl + bbsyl \cdot bgninl + \overline{bgnoutl} \\
brl &= bcsl + \overline{borgtl} + \overline{is0} \\
borgtl &= \overline{aslin} \cdot borgtl + bbsyl \cdot bcsl \cdot writel + borgtl \cdot is0 \\
aslout &= aslout \cdot is0 + basl + borgtl40 + \overline{bwrl} \cdot writel + \overline{dtackl} \\
writel &= aslout \cdot is0 + borgtl + bwrl + \overline{dtackl} \\
is0 &= \overline{aslin} \cdot \overline{borgtl} + bbsyl \cdot is0 + \overline{borgtl} \cdot is0 + brl \cdot is0
\end{aligned}$$

The final implementation was obtained using a simple standard-cell library. The circuit diagram appears in Figure 1.11.

If we want to switch our implementation to a Field-programmable Gate Array for our physical design, we would have to change only the technology mapping step in the logic synthesis procedure to automatically derive the desired netlist (i.e. the gate array programming data). For example, using a *Xilinx* component which allows the implementation of an arbitrary combinational logic function of up to five inputs in each module, we would obtain the following partitioning of the logic (dashed boxes in Figure 1.11):

- each circuit for *bbsyl, bgnoutl, brl, writel* and *is0* fits directly in one combinational logic module.

- the circuit for *aslout* can be partitioned into one block implementing
  $t_0 = basl + borgtl40 + \overline{bwrl} \cdot writel + \overline{dtackl}$
  and one implementing
  $aslout = aslout \cdot is0 + t_0$.

- the circuit for *borgtl* can be partitioned into one block implementing
  $t_1 = bbsyl \cdot bcsl \cdot writel$

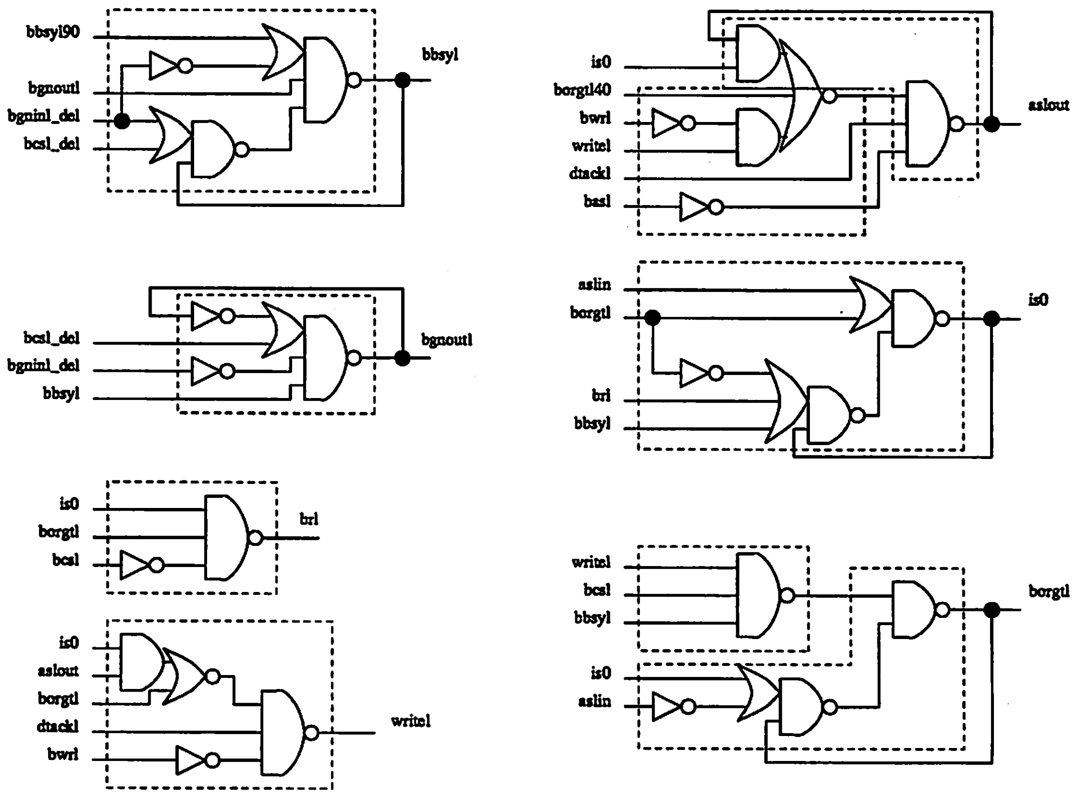


Figure 1.11: The *VMEbus* master interface circuit

and one implementing
$borgtl = \overline{aslin} \cdot borgtl + t_1 + borgtl \cdot is0.$

After obtaining the standard cell implementation, a delay analysis was performed to verify which potential hazards can actually occur. The delay model used in our logic synthesis system ([108, 107]) assigns a propagation delay from each input pin of a gate to its output pin. The value of this delay depends both on the intrinsic delay of the gate and on its fanout load.

The hazard elimination step discovered the following hazards in the implementation, under the assumption that the environment has zero delay (the worst case for hazards):

1. If the delay between *bwrl-* and *basl-* plus the delay from *basl* to *aslout* is less than the delay from *bwrl* to *aslout*, then a hazard can occur which incorrectly sets *aslout* to 0. This hazard can be eliminated by giving the synthesis system a more detailed knowledge of the master board delays. For example the designer can inform the synthesis system that the minimum delay between *bwrl-* and *basl-* is larger than the delay of an *aoi221* gate. The hazard can also be eliminated by delaying *basl*, e.g., with a pair of *inverters*.

2. If the delay (slave response) between *aslout+* and *dtackl+* plus the delay from *dtackl* to *aslout* is less than the feedback delay of *aslout*, then a hazard can occur which resets aslout to 0 when it should be set to 1. Again, this hazard can be eliminated either by assuming that the bus delay is larger than the *aoi221* delay, or by delaying *dtackl*.

3. If the delay between *bgninl_del+* and *bgninl_del-* is less than the feedback delay of *bbsyl*, then *bbsyl* can have a hazard. Again, the external delay is in general enough to eliminate the hazard.

4. The race between *bcsl_del-* and *bgninl_del-* can cause a hazard on *bgnoutl*. See Section 1.2.4 for a discussion of how *bcsl* must be latched in order to synchronize it with *bgninl*. Furthermore, since the hazard occurs when *bgninl_del-* occurs *too soon after bcsl_del-*, the magnitude of the delay of *bgninl_del* must be larger than the delay required by the hazard elimination algorithm.

5. If the delay between *aslout+* and *dtackl+* plus the delay from *dtackl* to *writel* is less than the delay from *aslout* to *writel*, then *writel* can have a dynamic hazard: it is set due to *dtackl-*, but then falls again because *dtackl* falls before *aslout* can keep it at 1. The same considerations as in case 2 apply.

In conclusion we have obtained, almost directly from a timing diagram specification, a hazard-free implementation of the specified *VMEbus* master interface protocol.

The design procedure is completely automatic, proceeding through state assignment, initial implementation, logic synthesis, and hazard elimination, and is completely supported by design automation tools. The final result is a hazard-free implementation of the given STG specification in a standard cell library. The only fundamental limitation is the inability of STGs to directly describe, and hence handle, synchronization and meta-stability. Nevertheless the STG specification immediately reveals where such problems lie, thus helping the designer to isolate and solve them separately.

The remainder of the thesis is devoted to a more formal and detailed explanation of the proposed methodology.

1.3 Thesis Overview

The thesis is organized as follows.

- In Chapter 2 the major results in the field of synthesis, analysis and testing of asynchronous circuits are reviewed, to provide a setting for the proposed methodology.
- In Chapter 3 the Signal Transition Graph specification formalism is described. Interesting *general* properties that are shared by *all possible* implementations of a given Signal Transition Graph, and hence must be taken into account by the designer at a very high level are also analyzed.
- In Chapter 4 and Chapter 5 the Signal Transition Graph synthesis procedure itself, which is divided into state encoding, initial implementation, hazard analysis, and hazard elimination is explained in detail.
- In Chapter 6 a design for testability methodology, that can be used to verify that the *delay bounds* assumed by the hazard elimination procedure are met by each circuit, is described. This guarantees hazard-free operation of each manufactured circuit.

To help the reader remember the numerous abbreviations used throughout the thesis, we have collected them in Table 1.3, together with a brief summary of their meaning.

As a matter of notation, a **bold** font is used both in the text and in the index to denote the *definition* of a term.

FSM	Finite State Machine	State-transition model for sequential system synthesis
FA	Finite Automaton	State-transition model for sequential system verification
FT	Flow Table	FSM model of asynchronous systems
ACS	Asynchronous Control Structure	Structural model of asynchronous systems
BACS	Binary Asynchronous Control Structure	Binary version of ACS
TS	Transition System	Uninterpreted state-transition behavioral model (not to be confused with the Trace Structure)
ALTS	Arc-Labeled Transition System	TS with transitions interpreted as signal value changes
STD	State Transition Diagram	ALTS with binary-labeled states
CD	Cumulative Diagram	Cumulative history of transitions in the system (not to be confused with the Change Diagram)
PD	Pure Delay	All input changes transmitted to the output
ID	Inertial Delay	Pulses shorter than the delay magnitude not transmitted
PN	Petri net	Uninterpreted event-based behavioral model
MG	Marked Graph	PN with concurrency and no choice
SM	State Machine	PN with choice and no concurrency
FCPN	Free-choice PN	PN with choice and concurrency
STG	Signal Transition Graph	PN with transitions interpreted as signal value changes (not to be confused with the State Transition Graph)
CSC	Complete State Coding	STG whose signals completely define the circuit state

Table 1.1: Principal abbreviations used in the thesis

Chapter 2

Previous Work

This chapter provides a brief account of some of the results in the field of analysis, synthesis and testing of asynchronous digital circuits that are directly related to our work. Most of the material is drawn from Unger's book ([117]) that summarizes the earlier efforts in the field, and from a book edited by Varshavsky ([126]) that is an invaluable source of information about the work done by researchers in Russia. A more recent account of the latter will also appear in [63].

Sections 2.2.3, 2.3.5, 2.3.6, 2.4, 2.5, 2.6, 2.7.2, 2.7.3 and 2.7.4 are not directly required to understand the rest of this work, and can be skipped on a first reading.

2.1 Circuit Model Taxonomy

The previous approaches to synthesis of asynchronous circuits can be distinguished by the classical taxonomy based on the **delay model** used for the circuit. Asynchronous circuits are generally modeled as an interconnection of components belonging to two major classes:

1. The *gate*, a component computing a set of *discrete* output variables (or, more often, a single output variable) as *delay-less discrete functions* of its input variables. These functions can be *logic functions* if the variables have two values, 0 and 1, or *ternary functions*, to take into account a third *undefined* value ([105]).

2. The *delay element*, a component producing a single output that is a delayed version of its single input.

Each *wire* connects a *single* gate or delay output to one or more gate or delay inputs. Primary inputs and outputs of the circuit can be considered *gates* computing the *identity function*.

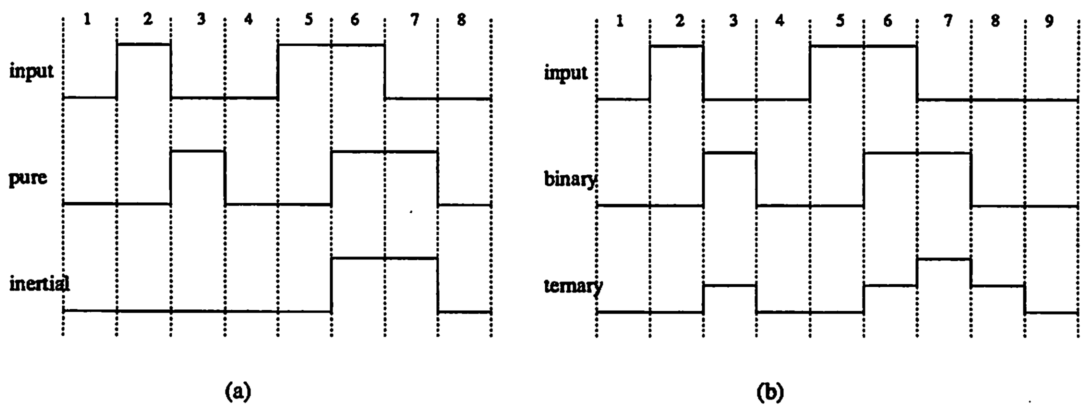


Figure 2.1: Various types of delay models

A delay element is:

- **pure** if it transmits each event on its input to its output, i.e., it corresponds to a pure translation in time of the input waveform.
- **inertial** if pulses shorter than or equal to the delay magnitude are not transmitted.

A delay element is *excited* if some transition is propagating through it, and *stable* otherwise. In the inertial case this is equivalent to saying that the element is excited if its input value is different from its output value, and stable otherwise. Figure 2.1.(a) represents the output waveforms of a pure delay element and an inertial delay element. Both delays have a magnitude of 1 time unit (hence the inertial delay only propagates pulses of more than 1 time unit). The pure delay is excited in intervals 2, 3, 5 and 7. The inertial delay is excited in intervals 2, 5 and 7.

The delay of each element is:

- **unbounded** if no bound on the magnitude is known a priori, except that it is *positive* and *finite*.
- **bounded** if an upper and lower bound on its magnitude are known before the synthesis or analysis of the circuit is begun. In this case, we have a further subdivision into:
  - **integer delay** (or **discrete delay**), in which the delay magnitude is constrained to be a multiple of some unit,
  - **real delay** (or **continuous delay**), in which the delay magnitude is not constrained (except by the bounds),

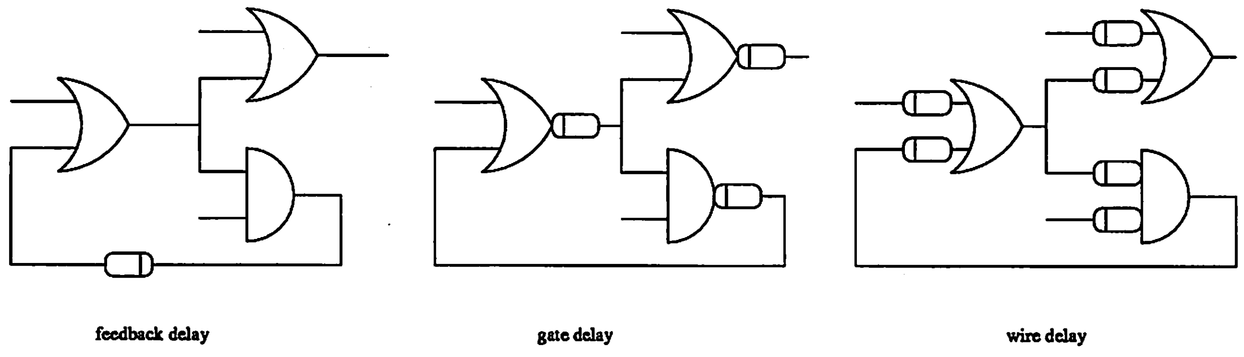


Figure 2.2: Feedback, Gate and Wire Delay

- **ternary delay**, in which the value of the delay between the lower and upper bound is *undefined*, i.e. a third logic value that is neither 0 nor 1 (shown by the "intermediate" level in Figure 2.1.(b)).

A circuit can be modeled (as shown in Figure 2.2) using:

- the **feedback delay** model if the set of delay elements is such that:
  1. every cycle contains at least one delay element and
  2. replacing any delay element with a wire produces a circuit where some cycle contains no delay element.
- the **gate delay** model if there is exactly one delay element per *gate output*.
- the **wire delay** model if there is exactly one delay element per *gate input*.

Hence the delay of a circuit is modeled by four parameters: the delay model for the circuit (feedback, gate, wire), the delay element type (inertial, pure), the delay type (bounded, unbounded), and the value of the delay (integer, real, ternary).

The remainder of the chapter is organized as follows. Section 2.2 briefly recalls some definitions from the theory of logic functions (see, e.g., [15]), Finite State Machines (see, e.g., [65] or [117]), Finite Automata (see, e.g., [43]) and Petri nets (see, e.g., [89] or [96]). Section 2.3 describes the classical synthesis techniques, derived from the work of Huffman ([53]). Section 2.4 outlines the design methodology proposed by Sutherland ([113]) to implement efficiently fast data processing asynchronous modules. Section 2.5 describes the synthesis techniques derived from the work of Muller ([88]) using the *inertial unbounded gate delay* model (*speed-independent* circuits).

Section 2.6 describes the synthesis techniques that use the *inertial unbounded wire delay* model (*delay-insensitive* circuits). Section 2.7 shows how a circuit implementation obtained with any of these techniques or by hand can be analyzed for correct operation using various models ranging from three-valued simulation to timed automata.

2.2 Definitions

2.2.1 Logic Functions

A **completely specified single-output logic function** g of n input variables (or Boolean function) is a mapping $g : \{0,1\}^n \rightarrow \{0,1\}$. Each input variable x_i corresponds to a coordinate of the domain of g. Each element of $\{0,1\}^n$ is a **vertex**. An **incompletely specified single-output logic function** f of n input variables (often called simply a **logic function** in the following) is a mapping $f : \{0,1\}^n \rightarrow \{0,1,-\}$. The set of vertices where f evaluates to 1 is the **on-set** of f, the set of vertices where f evaluates to 0 is its **off-set**, and the set of vertices where f evaluates to $-$ (i.e. it is not specified) is its **dc-set**. The **complement** of a logic function f, denoted by $\overline{f}$, is a logic function obtained by exchanging the on-set and off-set of f.

A **literal** is either a variable or its complement. The complement of variable x_i is denoted by $\overline{x_i}$. A **cube** c is a set of literals, such that if $x_i \in c$ then $\overline{x_i} \notin c$ and vice-versa. It is interpreted as the Boolean product of its elements. The cubes with n literals are in one-to-one correspondence with the vertices of $\{0,1\}^n$. A **cover** F is a set of cubes. It is interpreted as the Boolean sum of its elements. A cube c' **covers** another cube c'', denoted $c'' \sqsubseteq c'$, if $c' \subseteq c''$, for example $\{a,\overline{b}\} \subseteq \{a,\overline{b},c\}$, so $a\overline{b}c \sqsubseteq a\overline{b}$ (from now on we will drop braces and commas from a cube representation, and use the more familiar "product" notation).

The **Hamming distance** between two cubes c' and c'', denoted by $d(c',c'')$, is the cardinality of the set of variables x_i such that either $x_i \in c'$ and $\overline{x_i} \in c''$ or $\overline{x_i} \in c'$ and $x_i \in c''$.

The **intersection** of two **cubes** c' and c'' is defined only if $d(c',c'') = 0$ and is $c''' = c' \cup c''$. It is called "intersection" because it covers the intersection of the sets of vertices covered by c' and c''. For example, the intersection of $a\overline{b}$ and ac is $a\overline{b}c$. The **intersection of a cover** F with a cube c' is the set of all the intersections of $c_i \in F$ such that $d(c',c_i) = 0$ with c'.

A cube is an **implicant**, or **product term**, of a logic function f if it does not cover any off-set vertex of f. An **on-set cover** F of a logic function f is a set of cubes such that each cube of F is an implicant of f and each on-set vertex of f is covered by at least one cube of F. An **off-set**

cover R of a logic function f is a cover of the complement of f. In the sequel we shall use the generic term "cover" to denote on-set covers. Each cover F corresponds to a *unique completely specified* logic function, denoted by $B(F)$. A logic function can have *many* covers.

A cover is interpreted as the Boolean sum of its elements, so it can also be seen as a two-level sum-of-products implementation of the completely specified function $B(F)$. A **two-level** combinational circuit hence contains three classes of gates:

- *inverter* gates, with fanin primary inputs.
- *and* gates, with fanin *inverter* gates and primary inputs.
- *or* gates, with fanin *and* gates, *inverter* gates and primary inputs.

Primary outputs of the circuit can be connected to *inverter* gates, *and* gates or *or* gates.

An implicant of f is a **prime implicant** if it is not covered by any other single implicant of f. A cube c' in an on-set cover F of a logic function f can be *expanded against the off-set of f* by removing literals from it while it does not cover any off-set vertex. The result of the **expansion** is not unique (it depends on the order of removal), but it is always a *prime implicant* of f. A cover F is a **prime cover** of a function f if all its cubes are prime implicants of f. A **cube** c of a cover F of a function f is **redundant** if $F - \{c\}$ is still a cover of f (i.e., if c covers only vertices that are either covered by other cubes in F or belong to the dc-set). A **cover** F is **redundant** if some cube in it is redundant, otherwise it is **irredundant**. In a *prime* and *irredundant* cover F each cube has at least one **relatively essential vertex** which is an on-set vertex that is *not covered by any other cube* of F.

The **cofactor** f_{x_i} of a **logic function** f with respect to literal x_i is the logic function obtained by evaluating f at $x_i = 1$. Similarly $f_{\overline{x_i}}$ is obtained by evaluating f at $x_i = 0$. The cofactor of f has the following property (**Shannon decomposition**) for each variable x_i: $f = x_i \cdot f_{x_i} + \overline{x_i} \cdot f_{\overline{x_i}}$. The **cofactor** c_{x_i} of a **cube** c with respect to literal x_i is undefined if $\overline{x_i} \in c$ and $c - \{x_i\}$ otherwise. The **cofactor** F_{x_i} of a **cover** F with respect to literal x_i is the set of cubes of F for which the cofactor with respect to x_i is defined, cofactored with respect to x_i. Notice that the two definitions of cofactor for functions and covers are consistent: if F is a cover of a logic function f, then $B(F_{x_i}) = f_{x_i}$.

A **logic function** f is **monotone increasing** in a variable x_i if $f(x_i = 0, \beta) = 1 \Rightarrow f(x_i = 1, \beta) = 1$ for all $\beta \in \{0,1\}^{n-1}$, that is if increasing the value of the variable x_i from 0 to 1 never decreases the value of f from 1 to 0. Similarly a **logic function** f is **monotone decreasing** in a variable x_i if $f(x_i = 1, \beta) = 0 \Rightarrow f(x_i = 0, \beta) = 0$ for all $\beta \in \{0,1\}^{n-1}$. A **logic function** f is **unate** in variable x_i if it is either monotone increasing or monotone decreasing in x_i. Otherwise, f is **binate** in x_i. A **cover** F is **unate** in a variable x_i if variable x_i appears in only one phase (i.e., either x_i or $\overline{x_i}$) in its cubes (respectively **monotone increasing** and **monotone decreasing**). Otherwise it is **binate** in x_i. As shown in [15], a function that is unate may have binate covers, but prime covers of unate functions are unate. A unate function has a *unique* prime and irredundant cover, which is the set of all its prime implicants. Moreover, if F is a unate cover of f, then f is unate. Similar results exist for cover and function monotonicity.

A **logic expression** is a parenthesized expression using the *and*, *or* and *not* operators on a set of variables, called its **support.** The *and*, *or* and *not* operators are denoted by $\cdot$ (often omitted), $+$, and $-$ respectively. An expression represents a *unique* logic function; a given logic function can have many different expressions. A logic function f **essentially depends** on a variable x_i if $f_{x_i} \neq f_{\overline{x_i}}$, i.e., if there exists an assignment of values to the other variables such that x_i can change the value of f. An expression has **minimum support** if its support consists only of variables its associated function essentially depends on.

A **multi-valued** variable is a variable v_i that can take any value from a finite set S_i. A **multi-valued** logic function of a set of n multi-valued variables v_i is a mapping from the Cartesian product of the S_i into $\{0, 1\}$. The properties of ordinary logic functions (where $S_i = \{0, 1\}$ for all i) can be directly extended to logic functions of multi-valued variables ([75]).

2.2.2 Finite State Machines

An **incompletely specified non-deterministic Mealy Finite State Machine** (FSM) is a sextuple (S, X, Z, N, O, R):

- S is a finite set of **internal states** (often called simply states) of the FSM.
- X is a finite set of **input states** of the FSM, which either are symbolic or are represented as a binary vector of values of its **input signals.**
- Z is a finite set of **output states** of the FSM, which either are symbolic or are represented as a binary vector of values of its **output signals.**
- $N(i, s)$ is a relation from the (input state, present state) pairs, also called **total states,** to the next states (i.e., $N \subseteq X \times S \times S$).

- $O(i, s)$ is a relation from the (input state, present state) pairs to the output states (i.e., $O \subseteq X \times S \times Z$).

- $R \subseteq S$ is a set of **initial** (or reset) **states.**

An FSM is **deterministic** if R is a singleton set and both N and O are *functions*. This is the FSM class most often considered in this work. We will often call an incompletely specified deterministic FSM simply "an FSM".

A deterministic FSM is **completely specified** if both N and O are *total functions* (i.e., they are defined for all elements of their domains).

A deterministic incompletely specified FSM is a **Moore** FSM if the edges directed into a state s have a single output label. Hence, for Moore FSMs the output state is associated with the internal state rather than with the total state.

In the following we will only consider FSMs where each state is **reachable** from the initial state through at least one sequence of input states. Unreachable states can be deleted without changing the specified behavior.

An FSM can also be represented as a directed graph, called the **State Transition Graph,** where:

- each vertex is associated with an internal state

- each edge (also called a *transition*) is labeled with an input state/output state pair, and is directed from the *present state* vertex to the *next state* vertex.

A **stable total state** of an FSM is a pair (i, s) such that $N(i, s) = \{s\}$, i.e., a self-loop in the graph representation. An FSM is **normal** if every unstable state leads directly to a stable state (i.e., for every input state i, for every pair of states s, s' such that $N(i, s) = \{s'\}$, if $s \neq s'$ then $N(i, s') = \{s'\}$)

2.2.3 Finite Automata

A **non-deterministic Finite Automaton** (FA) is a quintuple (S, X, N, R, F):

- S is a finite set of **internal states** (often called simply states) of the FA.

- X is a finite set of **input states** of the FA.

- $N(x, s)$ is a relation from the (input state, present state) pairs to the next states (i.e., $N \subseteq X \times S \times S$).
- $R \subseteq S$ is a set of **initial states.**
- $F \subseteq S$ is a set of **final** (or accepting) **states.**

In the application of FAs to asynchronous circuit modeling, each input state represents a *transition* or *event*, that is, a change of value of a signal in the modeled circuit. Internal states can sometimes be labeled with a vector of *values* of all the circuit signals, for the sake of clarity (e.g., in [39]).

Finite Automata are used mainly as *analysis* tools in the context of asynchronous circuits. A sequence of *input states* is used to describe a sequence of events on a set of wires of the circuit under consideration. A sequence is *accepted* by the FA if and only if it corresponds to a "legal" operation of the circuit (e.g., it does not exhibit hazards on externally visible wires). The verification of correctness of the circuit can be transformed into a *state reachability analysis* problem on a corresponding automaton (see Sections 2.7.3 and 2.7.4 for examples of such transformations).

Acceptance conditions may be defined both on *finite strings* and on *infinite sequences* of input events. The latter case is more interesting, because it permits analysis of *liveness* properties of the specification ("something good will eventually happen") rather than just *safeness* properties ("nothing bad will ever happen"). An example of liveness property is that an arbiter circuit will eventually grant access to a resource after a request is received, while an example of safeness property is that the arbiter will never grant access to two requesters at the same time.

A **Büchi** FA ([114]) is a nondeterministic FA whose acceptance condition is defined on *infinite sequences* of input symbols. Given an infinite sequence of input symbols, a **run** of an FA is a sequence of states (beginning with one of the initial states) that the FA can reach under that input. A run of a Büchi FA is an **accepting run** if some member of F appears *infinitely often* in it.

The **product**[1] of two Büchi FAs (S, X, N, R, F) and (S', X', N', R', F') is a Büchi FA $(S'', X'', N'', R'', F'')$:

- $S'' = S \times S' \times \{0, 1, 2\}$,
- $X'' = X \cup X'$,
- the transition relation can be viewed, for a more compact description, as "decomposed" into a state component and into an integer component:

[1]The definition is not essential for the remainder of this work, and is referenced only for the sake of completeness.

- the state component advances both states only when the input state is common to both FAs, and leaves one component constant otherwise:
    1. if $x'' \in X \wedge x'' \in X' \wedge (x'', s_1, s_2) \in N \wedge (x'', s_1', s_2') \in N'$ then $(x'', (s_1, s_1', i), (s_2, s_2', j)) \in N''$
    2. if $x'' \in X \wedge x'' \notin X' \wedge (x'', s_1, s_2) \in N$ then $(x'', (s_1, s_1', i), (s_2, s_1', j)) \in N''$
    3. if $x'' \notin X \wedge x'' \in X' \wedge (x'', s_1', s_2') \in N'$ then $(x'', (s_1, s_1', i), (s_1, s_2', j)) \in N''$
- the integer component is set to 1 every time a state in F is seen, set to 2 every time a state in F' is seen, and reset to 0 immediately afterwards:
    1. if $s_2 \in F$ then $(x'', (s_1, s_1', 0), (s_2, s_2', 1)) \in N''$
    2. else if $s_2' \in F'$ then $(x'', (s_1, s_1', 1), (s_2, s_2', 2)) \in N''$
    3. else $(x'', (s_1, s_1', 2), (s_2, s_2', 0)) \in N''$
    4. otherwise $(x'', (s_1, s_1', i), (s_2, s_2', i)) \in N''$

- $R'' = \{(s, s', 0) \in S'' \, s.t. \, s \in R \wedge s' \in R'\}$
- $F'' = \{(s, s', 2) \in S''\}$

The product FA accepts a sequence σ if and only if its restriction to X (i.e. the sequence obtained removing from σ the input states not in X) is accepted by the first FA and its restriction to X' is accepted by the second FA.

2.2.4 Petri Nets

Petri nets (introduced in [97], see also [96] or [89]) are a widely used model for concurrent systems, because they have a very simple and intuitive semantics, that directly captures concepts like causality, concurrency and conflict between events.

A **Petri net** (PN) is a triple $\mathcal{P} = \langle T, P, F \rangle$:

- T is a non-empty finite set of **transitions,**
- P is a non-empty finite set of **places,** and
- $F \subseteq (T \times P) \cup (P \times T)$ is the **flow relation** between transitions and places.

A PN can be represented as a directed bipartite **graph,** where the arcs represent elements of the flow relation. We will only consider PNs with a *connected* underlying graph[2].

[2]A directed graph is **connected** if there exists an *undirected path* between every pair of nodes, i.e. the underlying undirected graph is connected.

A PN **marking** is a function $m : P \rightarrow \{0, 1, 2, \ldots\}$, where $m(p)$ is the number of **tokens** in p under marking m.

A **marked** PN is a quadruple $\mathcal{P} = \langle T, P, F, m_0 \rangle$, where m_0 denotes its **initial marking**.

A transition $t \in T$ is **enabled** at a marking m if all its predecessor places are marked. An enabled transition t *may* **fire**, producing a new marking m' with one less token in each predecessor place and one more in each successor place (denoted by $m[t > m'$).

A sequence of transitions and intermediate markings $m[t_1 > m_1[t_2 > \ldots m'$ is called a **firing sequence** from m. The set of markings m' reachable from a marking m through a firing sequence is denoted by $[m >$. The set $[m_0 >$ is called the **reachability set** of a marked PN with initial marking m_0, and a marking $m \in [m_0 >$ is called a **reachable** marking.

The directed graph $([m_0 >, E)$ where an edge joins two markings (m, m') if $m[t > m'$, is called the **reachability graph** of the marked PN.

A PN marking m is **live** if for each $m' \in [m >$ for each transition t there exists a marking $m'' \in [m' >$ that enables t. A marked PN is **live** if its initial marking is live.

A marked PN is k-**bounded** (or simply "**bounded**") if there exists an integer k such that for each place p, for each reachable marking m we have $m(p) \leq k$. A marked PN is **safe** if it is 1-bounded.

A transition t_1 **disables** another transition t_2 at a marking m if both t_1 and t_2 are enabled at m and t_2 is not enabled at m' where $m[t_1 > m'$. A marked PN is **persistent** if no transition can ever be disabled at any reachable marking.

Two transitions t_1 and t_2 of a marked PN are **concurrent** if there exists a reachable marking m where both t_1 and t_2 are enabled, and neither t_1 disables t_2 nor vice-versa. Two transitions t_1 and t_2 of a marked PN are in **direct conflict** if there exists a reachable marking m where both t_1 and t_2 are enabled, and either t_1 disables t_2, or vice-versa, or both.

Marked Graphs, State Machines and Free Choice Petri Nets

A PN is a **Marked Graph** (MG) if every *place* in it has exactly one predecessor and one successor. An MG is persistent for every initial marking m_0, furthermore every strongly connected MG has at least one live and safe initial marking ([30]).

A PN is a **State Machine** (SM) if every *transition* in it has exactly one predecessor and one successor. Also every strongly connected SM has at least one live and safe initial marking (a single token in any single place). The subclass of Petri nets called State Machines can be considered equivalent to classical Finite State Machines (Section 2.2.2), if we label each SM transition with an input/output state pair and we interpret each place as an internal state. The term "State Machine" is used in the Petri net literature, so we prefer it in this context.

A PN is **free-choice with respect to a subset of transitions** T' if for any two transitions t_1 and t_2 such that at least one of them is in T' and both share a predecessor place, then t_1 and t_2 have only one predecessor. The name "free-choice" then suggests that whenever a place is marked, it "freely chooses" which successor transition is enabled, without the need to check if some other predecessor of these transitions is also marked. A PN is **free-choice** (FCPN) if it is free-choice with respect to all its transitions (see, e.g., Figure 1.2.(a)). A PN is an **extended free-choice Petri net** (simply called *free-choice* in [12]) if any two transitions that share some predecessor places have exactly the same set of predecessor places. A PN is a **behaviorally free-choice Petri net** if any two transitions that share some predecessor places are enabled exactly in the same set of reachable markings. Any extended free-choice or behaviorally free-choice PN can be transformed into an equivalent FCPN ([10]), hence we will consider only FCPNs in the sequel.

Let $\mathcal{P} = \langle T, P, F \rangle$ and $\mathcal{P}' = \langle T', P', F' \rangle$ be two PNs. $\mathcal{P}'$ is a **subnet** of $\mathcal{P}$ if $T' \subseteq T$, $P' \subseteq P$ and $F' = F \cap ((T' \times P') \cup (P' \times T'))$.

A subnet $\mathcal{P}'$ is a **transition-generated subnet** of a PN $\mathcal{P}$ if P' is exactly the set of predecessors and successors in $\mathcal{P}$ of the elements of T'. A transition-generated subnet $\mathcal{P}'$ is a **Marked Graph component** of a PN $\mathcal{P}$ if it is a Marked Graph.

See, for example, the PN in Figure 2.3.(a), taken from [10] (circles denote places, boxes denote transitions). The subnet in Figure 2.3.(b) is not transition-generated, because there exists $t_1 \in T'$ whose predecessor (e.g. p_2) or successor (e.g. p_3) is not in P'. Moreover, the subnet in Figure 2.3.(c) is transition-generated, by $T' = \{t_1, t_2\}$, but it is not an MG component, because both p_1 and p_4 have more than one predecessor or successor. Finally, the subnet in Figure 2.3.(d) is an MG component.

A set of MG components **covers** a PN if each transition belongs to at least one component. A **Marked Graph allocation** over an FCPN $\mathcal{P} = \langle T, P, F \rangle$ is a function $f : P \rightarrow T$ that *chooses one* among the various *successors* of each place. A PN with $|P|$ places, with n_i successors each, has $\prod_{i=1,|P|} n_i$ distinct MG allocations. The **Marked Graph reduction** associated with an MG allocation over an FCPN is the unique MG component including all the transitions selected by the allocation function.

Figure 2.4 contains the MG reductions of the FCPN of Figure 1.2 associated with the MG allocations $f(p_0) = t_1$ and $f(p_0) = t_2$ (all other places have only one successor, so the value of f

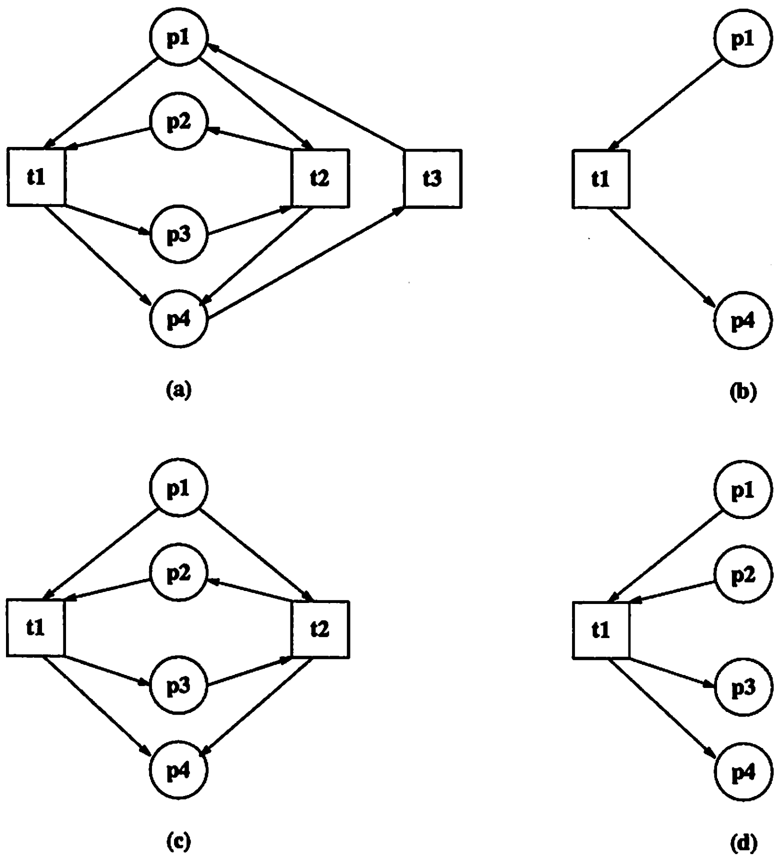
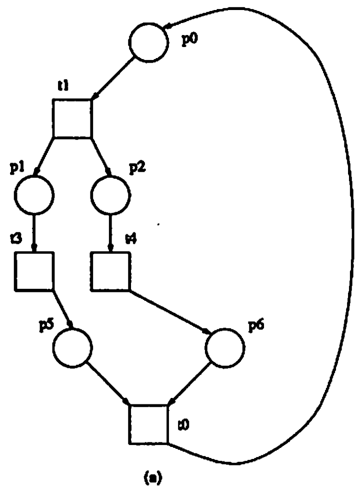


Figure 2.3: Examples of subnets and Marked Graph components



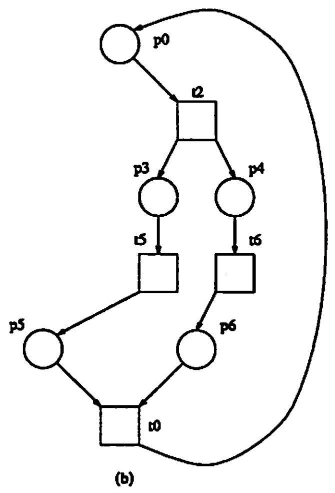


Figure 2.4: Marked Graph reductions of the Petri net of Figure 1.2

is uniquely determined). The reductions are obtained by extending the set of transitions selected by the allocation to yield an MG component.

Similarly (note the strong **duality** between places/transitions and SM/MG), a subnet $\mathcal{P}'$ is a **place-generated subnet** of a PN $\mathcal{P}$ if T' is exactly the set of predecessors and successors in $\mathcal{P}$ of the elements of P'. A place-generated subnet $\mathcal{P}'$ is a **State Machine component** of a PN $\mathcal{P}$ if it is a State Machine.

A set of SM components **covers** a PN if each place belongs to at least one component. A **State Machine allocation** over a FCPN $\mathcal{P} = \langle T, P, F \rangle$ is a function $f : T \rightarrow P$ that *chooses one* among the various *predecessors* of each transition. A PN with $|T|$ transitions, with m_i predecessors each, has $\prod_{i=1,|T|} m_i$ distinct SM allocations. The **State Machine reduction** associated with an SM allocation over a FCPN is the (unique) SM component including all the places selected by the allocation function.

Figure 2.5 contains the SM reductions of the FCPN of Figure 1.2 associated with the SM allocations $f(t_0) = p_5$ and $f(t_0) = p_6$ respectively.

The following Theorem is due to Hack ([48]), and is very useful to prove liveness and safeness results about FCPNs.

Theorem 2.2.1 *Let $\mathcal{P}$ be an* FCPN.

1. *If $\mathcal{P}$ has a* live *and* safe *marking then:*

   - MG *components cover the net.*

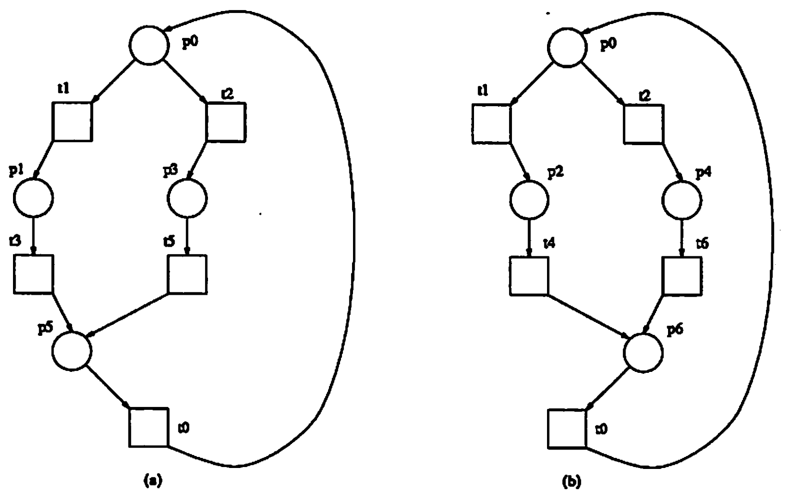


Figure 2.5: State Machine reductions of the Petri net of Figure 1.2

- SM *components cover the net.*

2. $\mathcal{P}$ *has a* live *and* safe *marking if and only if:*

   - every MG *reduction is* non-empty *and* strongly connected, *and the* MG *components* cover $\mathcal{P}$
   - every SM *reduction is* non-empty *and* strongly connected, *and the* SM *components* cover $\mathcal{P}$

3. *Let* m *be a* live *marking of* $\mathcal{P}$. *Then* m *is* safe *if and only if there exists an* SM *cover where each component is marked with* exactly one *token in* m.

In Figures 2.4 and 2.5 all reductions are non-empty, strongly connected and cover the net, so the net has a live and safe marking (e.g. the marking shown in Figure 1.2). Note that the initial marking in Figure 1.2 marks exactly one place in every SM of the (only) cover of the FCPN.

2.3 The Huffman Model for Asynchronous Circuits

This section describes a number of issues concerning the modeling, analysis and synthesis techniques for asynchronous circuits based on the work of Huffman ([53], see also [117]). Such techniques use an FSM-like specification called the Flow Table (Section 2.3.1). They are based on the **Huffman model** shown in Figure 2.6. This model decomposes the circuit into a combinational logic part and a feedback part. The *inertial unbounded delay* model is used for modeling delay on

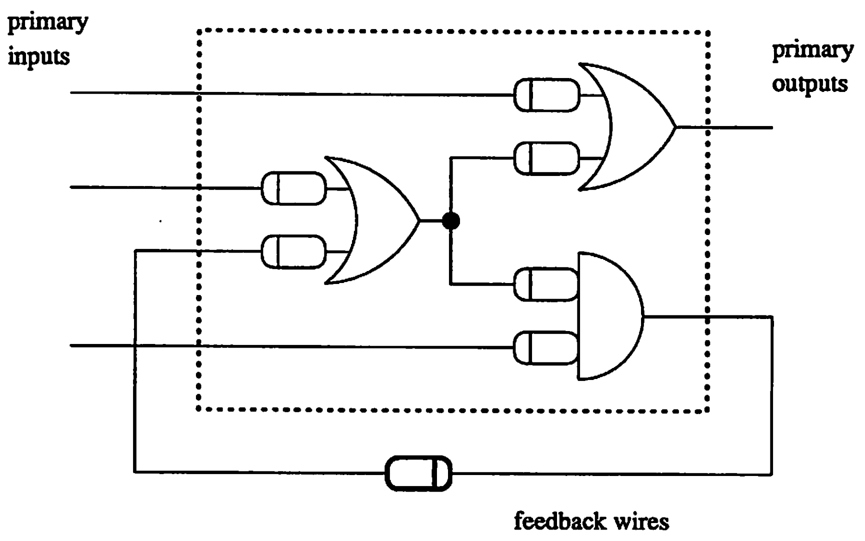


Figure 2.6: Huffman model of an asynchronous circuit

feedback wires (*race* analysis). The *inertial bounded* and *unbounded wire delay* models are used for modeling delay of non-ideal combinational logic blocks (*hazard* analysis).

The synthesis according to this model is performed by minimizing the Flow Table (Section 2.3.2), encoding it (Section 2.3.3), and realizing its combinational next state and output functions with hazard-free combinational logic (Section 2.3.4). An alternative technique uses *one-hot* encoding of the FSM states (Section 2.3.5). Other researchers proposed the use of *synchronous latches* to implement the feedback lines rather than just wires, and of a *local clock* to drive them (Section 2.3.6). Both one-hot and self-clocked techniques greatly simplify the synthesis and analysis at the expense of some increased implementation cost.

2.3.1 The Flow Table Specification

The **Flow Table** (FT) is a *Finite State Machine* specification of the behavior of an asynchronous circuit. The FT is generally described as a two-dimensional table, with the vertical axis indexed by the **present state** of the circuit, and the horizontal axis indexed by the **input state** (or simply *input*) of the circuit. Each entry of the table contains a **next state** and an **output state** (or simply *output*). The FT in Figure 2.7 (from [117]) describes the operation of an asynchronous circuit that controls a simple traffic light. Its input state is encoded using two variables, x_1 (a timer, whose value is 0 for 60 seconds and then 1 for 30 seconds) and x_2 (with value 1 if a car is waiting on the minor road). Its output state is encoded using only one variable, z (with value 1 if the major

	x_1x_2			
	00	01	11	10
1	$\boxed{1}$,0	3,0	6,1 / 7,0	2,0
2	1,0	3,0	7,0	$\boxed{2}$,0
3	4,0	$\boxed{3}$,0	6,1	5,1
4	$\boxed{4}$,0	3,0	6,1	5,1
5	1,0	3,0	6,1	$\boxed{5}$,1
6	1,0	3,0	$\boxed{6}$,1	5,1
7	4,0	3,0	$\boxed{7}$,0	8,0
8	4,0	3,0	7,0	$\boxed{8}$,0

Figure 2.7: A Flow Table specification of a traffic light controller

road sees a red light, 0 otherwise). A ***stable state*** is denoted by $\boxed{\text{state}}$ in the FT representation.

The circuit must produce $z = 1$ only at the beginning of an interval with $x_1 = 1$ and only if $x_2 = 1$, and maintain $z = 1$ for the full interval with $x_1 = 1$. The circuit starts in the internal state labeled 1, its **initial state**. The FT is built by defining the circuit behavior under all possible input transitions from this state, then from the states reached after each transition, and so on. When there is no traffic on the small road, the circuit cycles between entries (input state, internal state) (00,1), (10,1), (10,2), (00,2), (00,1) .... When a car arrives on the small road, the circuit goes to state 3 and waits for the timer to expire.

Note that if a car shows up and the timer expires *at the same time* we can choose either to produce $z = 1$ immediately, or wait for the next timer expiration, i.e., the FT is *non-deterministic*. As shown below, it must be made *deterministic* in order to synthesize a circuit (that is deterministic by nature) from it.

The definition of *"at the same time"* is a bit problematic, and this kind of *synchronization* problem has haunted asynchronous designers since the very beginning. In general circuits described with the Huffman model are assumed to operate in the so-called **Huffman Mode**, in order to associate meaningfully the FT behavior with the circuit behavior. This means that two time intervals δ_1 and δ_2 (with $\delta_1 < \delta_2$, and both intervals depending on the specific circuit delay characteristics) are defined. If input changes are separated by less then δ_1, then they are considered *simultaneous*. If they are separated by more then δ_2, then they are considered *distinct*. Otherwise the change is illegal, and the circuit behavior is *undefined*.

There are two *special cases* of Huffman Mode that deserve special mention and are attributed names of their own:

- **Fundamental Mode**: *inputs are constrained to change only when all the delay elements are stable* (i.e., they have the input value equal to the output value). In this case:
  - δ_1 is the *minimum* propagation delay inside the combinational logic and
  - δ_2 is the *maximum* settling time for the circuit, i.e., the maximum time that we must wait for all the delay elements to be stable once its inputs have become stable.

  Note that Fundamental Mode excludes FTs that have a *cycle of oscillations*, where a set of internal states does not have any stable entry under a particular input state.

- **Input-Output Mode**: the environment monitors the circuit outputs and *applies new inputs as soon as the effect of the current inputs appears on them.* This is the operation mode used, for example, by Muller (Section 2.5). In this case δ_1 coincides with δ_2 and is the *minimum* delay of any gate in the circuit.

The definition of Fundamental Mode may seem a bit strange because the feedback delays in the Huffman model are *unbounded* yet we require the circuit to stabilize before we can change the inputs. To be more precise, all the feedback delays are assumed to have some very large upper bound, but no lower bound. From an analysis standpoint this yields the same results as the unbounded delay assumption, but it allows a meaningful definition of the settling time of a circuit.

If we consider the number of inputs that can change for any given state transition, the circuit can operate in two modes:

1. **Single Input Change mode** implies that only one input can change at any given time (i.e., within one δ_1). Together with the Fundamental Mode of operation, this is known as **normal** Fundamental Mode.

2. **Multiple Input Change mode** implies that any number of inputs can change at any given time.

An FT is **Single Output Change** if it is either *normal* or any sequence of unstable states due to a single input state change causes at most one output state change. Otherwise, it is **Multiple Output Change**. A Single Output Change FT can always be made *normal* (without modifying the observable behavior) by replacing the first next state in a chain of unstable states with the last one.

2.3.2 Flow Table Minimization

The initial FT describing the desired circuit behavior can be minimized using appropriate techniques (see, for example, [53], [46], [65] or [117]). The purpose of this minimization is to reduce the number of states so that:

- the following synthesis steps may be simplified, since all known optimum state encoding and logic minimization algorithms have a worst-case behavior that is exponential in the number of states or state signals (heuristic algorithms take advantage of the reduced solution space also).
- the number of state bits, and hence possibly the complexity of the next state and output logic may be reduced.

A pair (s_1, s_2) of internal states of an FT is **output compatible** if, for each input state x, the output state under (x, s_1) is either the same as under (x, s_2) or is undefined. Otherwise it is output incompatible. A set of internal states is **output compatible** if all pairs of states in it are output compatible.

A set c_1 of internal states **directly implies** a set c_2 of internal states under input state x if c_2 is the set of all next states under x for all states in c_1. A set c_1 of internal states **implies** a set c_2 of internal states under a sequence of input states $x_1 x_2 \ldots x_n$ if there exists a chain of implications from c_1 through intermediate sets of states to c_2 under those input states.

A set c_1 of internal states is **a compatible** if it is output compatible and it does not imply an output incompatible set. A **maximal compatible** is a compatible that is not a subset of any other compatible. Compatibility is a relation of the form $C \subseteq (S \times S)$. It is *reflexive* and *symmetric*, but it is *not transitive* for an incompletely specified FT, so it is not an equivalence relation (transitivity holds only in the completely specified case).

The set I of all (unordered) pairs of incompatible states can be computed as follows:

Procedure 2.3.1

- *Let I be the set of output incompatible pairs*
- *Repeat until no new incompatible pairs can be added*
  - *For each pair of internal states (s_1, s_2), and for each input state x such that both $N(x, s_1)$ and $N(x, s_2)$ are specified*

  * *Let s'_1 denote $N(x, s_1)$ and s'_2 denote $N(x, s_2)$*
  * *If $(s'_1, s'_2) \in I$ then add (s_1, s_2) to I*

Note that incompatibility is defined only if the output and next state are uniquely specified or totally unspecified, i.e., if they are defined by a *function*. There is no state minimization procedure, to the best of our knowledge, that can minimize a non-deterministic FT, i.e., an FT where the next states and/or the outputs are defined by a *relation*.

From I we can compute the set of all maximal compatibles:

Procedure 2.3.2

- *Write the pairs of incompatibles as a product-of-sums logic expression, with one variable for each state and one sum term for each pair.*
- *Convert the product-of-sums expression into an* irredundant *sum-of-products expression (an irredundant cover)*[3].
- *Create one maximal compatible for each product term which contains all the states whose variables* do not appear *in the term.*

In the example of Figure 2.7, after resolving the non-determinism by choosing 7,0 for entry (11,1)[4], we have the following results:

1. The initial set of output incompatible pairs is $\{(1,3),(1,4),(1,5),(1,6),(2,3),(2,4),(2,5),(2,6),(3,7),(3,8),(4,7),(4,8),(5,7),(5,8),(6,7),(6,8)\}$.
   For example, 1 and 3 are incompatible because under input 11 state 1 produces output 0, while state 3 produces output 1.

2. The following pairs of incompatibles are added because under some input they lead to incompatible pairs: $\{(1,7),(1,8),(2,7),(2,8),(3,5),(3,6),(4,5),(4,6)\}$.
   For example, 1 and 7 are incompatible because under input 00 state 1 goes to itself, while state 7 goes to 4, and (1,4) are already known to be incompatible.

3. The product-of-sums expression describing the incompatibles is: $(1+3)(1+4)(1+5)(1+6)(2+3)(2+4)(2+5)(2+6)(3+7)(3+8)(4+7)(4+8)(5+7)(5+8)(6+7)(6+8)(1+7)(1+8)(2+7)(2+8)(3+5)(3+6)(4+5)(4+6)$.

[3]The cover is *unique* because the corresponding function is *unate*, i.e., all variables appear only in the positive phase.
[4]We could also have chosen 6,1 obtaining a minimized FT with 5 states.

4. The equivalent irredundant sum-of-products expression is: $123456 + 123478 + 125678 + 345678$.

5. The maximal compatibles are respectively: $\{7,8\}, \{5,6\}, \{3,4\}, \{1,2\}$.

A set C of compatibles is **closed** if for each compatible $c_1 \in C$, for each input state x, and for each pair of states $s_1, s_2 \in c_1$ such that the next state under x is defined, there exists a compatible c_2 to which both next states of s_1 and s_2 under x belong.

A sequence of input states is an **admissible sequence** from an internal state s of an FT T if no unspecified next state is encountered applying it to T from s (except possibly in the last step).

An **internal state** s of an FT T **covers** an internal state s' of another FT T' if

- all *finite* sequences of input states admissible for T' from s' are also admissible for T from s and
- for all such sequences of input states, the output states of T' from s', if specified, are the same as those of T from s.

A **Flow Table** T **covers** another FT T' if every state in T' is covered by *at least* one state of T. This is equivalent to requiring that the initial state of T cover the initial state of T', because we are considering only deterministic FTs where all states are reachable from the initial state. In more intuitive terms, T covers T' if T is a "more specific" description of the same basic behavior as T', i.e., the two coincide, except in sequences that would be illegal for T'.

Two FTs are **equivalent** if they cover each other. An FT is **minimized** if there exists no FT with fewer states that covers it.

A **set of compatibles** of an FT T **covers** T if each internal state of T appears in *at least one compatible*. Given a *closed set of compatibles* C that covers an FT T (a **closed cover** of T) we can build another FT T' that covers T as follows.

1. T' has one internal state for each compatible $c_1 \in C$.

2. The output state under each input state i for each compatible c_1 is uniquely defined by the corresponding output of some state $s \in c_1$, by the definition of *output compatibility*.

3. The set of next states under each input state i for all the states in a compatible c_1 must be a *subset* of some other compatible c_2, because the set is *closed*. This c_2 is chosen as the next state under c_1 and i, so the next state function of T' is uniquely defined.

Given these definitions, the **state minimization** problem can be stated as follows: given an FT T, find a closed cover of T of minimum cardinality. The set of all maximal compatibles of a *completely specified* FT (i.e., one in which the next state and output functions are defined for all total states) is the unique minimum closed cover. For an *incompletely specified* FT, a closed cover consisting only of maximal compatibles may have a larger cardinality than a closed cover in which some or all of the compatibles are proper subsets of maximal compatibles. The search space of all subsets of all maximal compatibles is too large, so we need to define the concept of *prime compatible*, that plays a role similar to the *prime implicant* in logic minimization.

The **class set** C_1 implied by a compatible set of internal states c_1 is the set of all sets c_2 such that:

1. there exists an input state x such that c_2 is implied by c_1 under x and
2. c_2 has more than one element and
3. $c_2 \not\subseteq c_1$ and
4. $c_2 \not\subseteq c_3$ for any other $c_3 \in C_1$

A compatible c_1, with class set C_1, **dominates** a compatible c_2, with class set C_2, if:

1. $c_1 \supset c_2$ and
2. $C_1 \subseteq C_2$

That is, c_1 dominates c_2 if c_1 is a *superset* of c_2 and the conditions on the *closure* of c_1 are a *subset* of the conditions on the closure of c_2. A compatible c that is not dominated by any other compatible is a **prime**.

The following basic theorem was proved in [46]:

Theorem 2.3.1 *For any FT T there is a minimum closed cover composed only of prime compatibles of T.*

The following procedure computes the set P of all prime compatibles of an FT T:

Procedure 2.3.3

- *Let P be the set of maximal compatibles of T (as computed by Procedure 2.3.2).*
- *Let l be the cardinality of the largest maximal compatible.*

```
• For i from l down to 2 do:
    – For each compatible $c_1 \in P$ such that $c_1$ has cardinality l do:
        ∗ For each internal state $s \in c_1$ do:
            1. Let $c_2 = c_1 - \{s\}$
            2. If $c_2$ is not dominated by any $c_3 \in P$, then add $c_2$ to $P$
```

The example in Figure 2.7 is completely specified, so the set of maximal compatibles is also the set of prime compatibles.

In order to find a minimum closed cover of prime compatibles, [46] proposed the construction of a product-of-sums logic expression, with:

- one variable for each prime compatible, which is true if the compatible is *chosen* as member of the cover,
- one sum term (covering term) for each internal state of the original FT, stating that at least one prime compatible covering it must be chosen,
- one sum term (closure term) for each compatible c_2 in the class set C_1 of each prime compatible c_1, stating that either c_1 is not chosen, or some prime compatible c_3 superset of c_2 must be chosen.

Each assignment of logic values to the variables that *satisfies* the expression corresponds to a closed cover, hence an assignment with the *minimum number* of true variables is a minimum closed cover.

The problem of finding (given a product-of-sums expression and a set of weights for the true and false value of each variable), a satisfying *partial* assignment (where an unassigned variable has a weight of 0) of minimum weight is called a **covering problem.** The problem is called **binate covering** ([103], [73]) if some variable appears both in positive and negative form, and **unate covering** ([45]) otherwise. Efficient heuristics to solve each problem have been reported in the cited papers. The problem of finding a minimum closed cover can be formulated and solved as a *binate covering problem*, with a weight of 1 for each true variable and a weight of 0 for each false variable.

For example, suppose that an FT has the following prime compatibles:

- $c_1 = \{s_1, s_2\}$ with class set $P_1 = \{\{s_4\}\}$

	x_1x_2			
	00	01	11	10
1	$\boxed{1}$,0	2,0	4,0	$\boxed{1}$,0
2	$\boxed{2}$,0	$\boxed{2}$,0	3,1	3,1
3	1,0	2,0	$\boxed{3}$,1	$\boxed{3}$,1
4	2,0	2,0	$\boxed{4}$,0	$\boxed{4}$,0

Figure 2.8: The minimized Flow Table of Figure 2.7

- $c_2 = \{s_1, s_2, s_3\}$ with class set $P_2 = \{\{s_1, s_2\}, \{s_4\}\}$ (note that neither c_1 dominates c_2 nor vice-versa)
- $c_3 = \{s_3, s_4, s_5\}$ with class set $P_3 = \{\{s_1, s_2\}\}$

Then we have the following covering terms (one for each state):
$(c_1 + c_2)(c_1 + c_2)(c_2 + c_3)(c_3)(c_3)$
and the following closure terms:

- c_1: $(\overline{c_1} + c_3)$
- c_2: $(\overline{c_2} + c_1 + c_2)(\overline{c_2} + c_3)$
- c_3: $(\overline{c_3} + c_1 + c_2)$

An assignment of minimum weight that satisfies all the clauses yields the closed cover $\{c_1, c_3\}$.

In the example of Figure 2.7, the cover (which is unique because the FT is completely specified) must include all maximal compatibles and we obtain the minimized FT shown in Figure 2.8.

2.3.3 Flow Table State Encoding

Using the Huffman model of an asynchronous circuit, each FT state is implemented as a bit vector of values for the *feedback wires* of the circuit. Each such wire is also called a **state signal**: the *output* of the corresponding feedback delay element is called a **present state signal**, while its *input* is called a **next state signal**. In this phase of the circuit synthesis only *unbounded feedback delays* are considered, and the combinational logic block is considered delay-free. We will see in Section 2.3.4 under what hypotheses this assumption can be satisfied and the circuit operates correctly according to the specification.

The delays on the state signals can be conservatively assumed to be *unbounded* because efficient methods of dealing with them on the feedback wires exist. On the other hand, there is no satisfactory approach based on the Huffman model that synthesizes circuits with unbounded delays *inside* the combinational logic block, so these delays must be assumed to be *bounded* (see Section 2.5 for a completely different approach to analysis and synthesis with the unbounded gate delay model). Hence the state encoding algorithm for asynchronous circuits using the Huffman model must satisfy a basic requirement: it must avoid *critical races* among state signals. A **race** among two or more state signals is a *simultaneous change of their value* due to a *single* state transition. A race is a **critical race** if the stable state reached by the circuit depends on the race winner, i.e., on the order in which the signals change.

For example, in Figure 2.8, we can encode the states using 2 state signals, y_1 and y_2, assigning the code 00 to state 1, 01 to 2, 10 to 3 and 11 to 4. The transition $(01, 3) \rightarrow 2$ involves a change of both y_1 and y_2. This race is *not critical*, because:

- if y_1 falls first, we go through state 1 (with code 00), that has the same next state 2 and output 0 under input 01.
- if y_2 rises first, we go through state 4 (with code 11), that similarly has the same next state 2 and output 0 under input 01.

On the other hand, the race produced by transition $(10, 2) \rightarrow 3$ is *critical*, because:

- if y_1 rises much faster than y_2 falls, the new stable state 4 under input 10 can be reached, resulting in incorrect behavior.
- if y_2 falls much faster than y_1 rises, the stable state 1 is reached, again resulting in incorrect behavior.

There are many techniques to solve this problem and produce a **critical race-free state encoding**. **Multiple Transition Time** encodings involve creating a path through many *unstable states* before reaching the final stable state. Multiple transition time encodings are interesting because they produce *general solutions*, valid for any FT, independent of its transition structure (of course these solutions are parameterized in terms of number of input and internal states).

In general *throughput* is a key point in asynchronous design, so it is not desirable to go through many intermediate states. Repeatedly rippling through the combinational logic section of the circuit may excessively slow down its operation. Hence researchers have devoted a lot of effort to the development of **Single Transition Time** state encoding algorithms, in which each state transition involves only *one* change of the state signals (even though many state signals can change at the same time). For this reason we chose an algorithm belonging to this class for our synthesis procedure (Chapter 4), and we describe it at some length here. The algorithm is due to Tracey ([115]), and is called a *Unicode* Single Transition Time encoding, because each symbolic state in the minimized FT is assigned a *single* code[5].

The basic idea of Tracey's state encoding is that whenever two pairs of state transitions occur under the same input (so that the input values cannot be used to distinguish among them), at least some state signal must remain *constant* during both transitions, and have a *different value* for each transition. This set of constant signals allows the circuit to "distinguish" among different transitions, which avoids critical races.

In Tracey's work, the concept of *dichotomy* corresponds informally to the idea of a *column* (bit) in the binary encoding of the internal FT states. It distinguishes one set of states from another by a single bit in the corresponding encodings. An **ordered dichotomy** is a 2-block partition of internal states. If an ordered dichotomy is used in generating an encoding, then one code bit of the states in the left block has the value *0* while the same code bit has the value *1* for the states in the right block. Conversely, a bit (state signal) of an encoding implies an **associated** ordered dichotomy: all the states in which the bit is 0 are in the left block and all the states in which it is 1 are in the right block.

For example, given the ordered dichotomies $(12;34)$ and $(14;23)$, the unique (up to permutation of bits) encoding derived from them is $00, 01, 11, 10$ (i.e., state 1 is assigned code 00 and so on).

Two ordered dichotomies d_1 and d_2 are **compatible** if the left block of d_1 is disjoint from the right block of d_2 and the right block of d_1 is disjoint from the left block of d_2. Otherwise, d_1 and d_2 are incompatible.

The **union** of two compatible ordered dichotomies, d_1 and d_2, is the ordered dichotomy whose left and right blocks are the union of the left and right blocks of d_1 and d_2, respectively. The union operation is not defined for incompatible ordered dichotomies.

An ordered dichotomy d_1 **covers** an ordered dichotomy d_2 if the left and right blocks of d_2 are subsets respectively either of the left and right blocks, or of the right and left blocks of d_1.

[5]Other state encoding methodologies may assign two or more codes to the same FT state in order to avoid critical races.

Note that the definition of covering relation views *ordered* dichotomies as *unordered* pairs. For example, (1; 23) is covered by (14; 235) and (234; 1), but not by (12; 3).

A member of a set of ordered dichotomies is **prime** if it is incompatible with all ordered dichotomies in the set that are not covered by it.

The following Theorem is due to Tracey:

Theorem 2.3.2 *A Unicode Single Transition Time state encoding for a Single Output Change* FT *does not have a* critical race *if and only if*

- *for every input state x for every pair of transitions $(x, s_1) \to s_2$ and $(x, s_3) \to s_4$ such that $s_2 \neq s_4$*
  - *there exists a state signal such that its associated dichotomy* covers either $(s_1 s_2; s_3 s_4)$ or $(s_3 s_4; s_1 s_2)$.

Note that s_1 may coincide with s_2 or s_3 may coincide with s_4.

The basic idea is that whenever two different transitions (possibly including stable states) occur under the same input, there must exist at least one state signal that *does not change* and *distinguishes* between the transitions.

An encoding algorithm based on this Theorem was first given in [115]. We describe a more efficient formulation, due to [104], that has a complexity *linear* in the number of *prime dichotomies* and produces an encoding with the *minimum* number of bits that satisfies the Theorem.

Procedure 2.3.4

1. *Let $D = \{(s_i; s_j) \forall i \neq j\}$ be the set of* uniqueness dichotomies[6]
2. *For each input state x*
   - *For each pair of transitions $s_1 \to s_2$ and $s_3 \to s_4$ under x such that $s_2 \neq s_4$ and either $s_1 \neq s_2$ or $s_3 \neq s_4$*
     - *Let $D = D \cup \{(s_1 s_2; s_3), (s_1 s_2; s_4), (s_1; s_3 s_4), (s_2; s_3 s_4), (s_3 s_4; s_1), (s_3 s_4; s_2), (s_3; s_1 s_2), (s_4; s_1 s_2)\}$*
3. *Derive the product-of-sums expression that describes the* pairwise incompatibilities *between dichotomies in D (i.e., one variable for each dichotomy in D, also called the set of* seed *dichotomies, and one sum for each incompatibility)*

[6]So called because they ensure that the final set of prime dichotomies assigns a unique code to each internal state.

4. *Convert the product-of-sums expression into an* irredundant *sum-of-products expression (an irredundant cover)*

5. *Create one* prime *dichotomy for each term, as the union of all the seed dichotomies corresponding to variables that* do not appear *in the term*

6. *Find a minimum cardinality set of prime dichotomies covering all the seed dichotomies*

The dichotomies due to pairs involving only *stable states* (excluded by the condition that either $s_1 \neq s_2$ or $s_3 \neq s_4$) are covered by *uniqueness* ones.

The conversion from product-of-sums to sum-of-products expression is exponential in the number of variables. However, the special structure of the clauses, each with exactly two variables, reduces the execution time so that it has linear complexity in the number of prime dichotomies. Unfortunately, the number of prime dichotomies can still be exponential in the number of states, so heuristic algorithms such as those described in [134] or [136] should be used when the number of FT states precludes the application of the *exact* algorithm above.

Encoding constraints that are used to minimize a two-level or multi-level *implementation* of the circuit can also be expressed as dichotomies ([134], [104], [71]). So this encoding algorithm also allows the *minimization of the implementation cost* within the *critical race-free* constraint.

Let us examine an example, the encoding of the FT in Figure 2.8, to clarify the above procedure.

1. The set of uniqueness dichotomies is
   $D = \{(1;2),(1;3),(1;4),(2;3),(2;4),(3;4)\}$

2. The set of dichotomies necessary and sufficient (due to Theorem 2.3.2) to avoid any critical race is:

   - input 00:
     - pair $3 \rightarrow 1$ and $4 \rightarrow 2$ produces
       $\{(31;4),(31;2),(3;42),(1;42),(42;3),(42;1),(4;31),(2;31)\}$
     - pair $3 \rightarrow 1$ and $2 \rightarrow 2$ produces only dichotomies that are *covered* by those above
     - pair $1 \rightarrow 1$ and $4 \rightarrow 2$ produces only dichotomies that are *covered* by those above
   - input 01:
     - no pair of transitions with different final state exists

- input 11:
  - pair $2 \to 3$ and $1 \to 4$ produces
    $\{(23;1),(23;4),(2;14),(3;14),(14;2),(14;3),(1;23),(4;23)\}$
  - pair $2 \to 3$ and $4 \to 4$ produces only dichotomies that are *covered* by those above
  - pair $3 \to 3$ and $1 \to 4$ produces only dichotomies that are *covered* by those above
- input 10:
  - pair $2 \to 3$ and $1 \to 1$ produces
    $\{(23;1),(1;23)\}$
  - pair $2 \to 3$ and $4 \to 4$ produces
    $\{(23;4),(4;23)\}$
- The set of seed dichotomies (excluding covered dichotomies) is:
  $D = \{(13;4),(13;2),(3;24),(1;24),(24;3),(24;1),(4;13),(2;13),(23;1),(23;4),$
  $(2;14),(3;14),(14;2),(14;3),(1;23),(4;23)\}$
- If now we arbitrarily set state 1 to receive the all-zero code, we can reduce the number of seed dichotomies without losing generality by dropping those with state 1 in the right hand side. We then obtain (together with the associated variables):
  $D = \{a = (1;24), b = (24;3), c = (3;24), d = (13;4), e = (13;2), f = (23;4),$
  $g = (1;23), h = (4;23), i = (14;2), j = (14;3)\}$

3. The product-of-sums expression describing incompatibilities is:
   $(a+b)\,(a+f)\,(a+h)\,(a+i)\,(a+j)\,(b+c)\,(b+d)\,(b+e)\,(b+f)\,(b+g)\,(b+h)\,(b+i)$
   $(c+f)\,(c+g)\,(c+h)\,(c+i)\,(c+j)\,(d+g)\,(d+h)\,(d+i)\,(d+j)\,(e+f)\,(e+g)\,(e+h)$
   $(e+j)\,(f+g)\,(f+h)\,(f+i)\,(f+j)$
   For example, $a + b$ describes the incompatibility between $(1;24)$ and $(24;3)$ due to 2 and 4 being in different blocks.

4. The sum-of-products equivalent expression is:
   $abcdef + abcdfghj + abceghij + acdefghi + bcdefhij + bfghij$

5. The prime dichotomies, obtained by the *union* of seed dichotomies corresponding to signals that *do not appear* in each term, are:

   - $abcdef$ (g, h, i, j missing) yields $(14;23)$
   - $abcdfghj$ (e, i missing) yields $(134;2)$

- $abceghij$ (d, f missing) yields $(123; 4)$
- $acdefghi$ (b, j missing) yields $(124; 3)$
- $bcdefhij$ (a, g missing) yields $(1; 234)$
- $bfghij$ (a, c, d, e missing) yields $(13; 24)$

6. All seed dichotomies can be covered by just two prime dichotomies, namely:
   $(13; 24)$ and $(14; 23)$
   This yields the following state encoding:
   00, 11, 01, 10
   The first dichotomy distinguishes state sets by the first state bit. The reader can now check that no critical race can occur with this encoding.

2.3.4 Hazard-free Flow Table Implementation

After the Flow Table has been minimized and encoded, it describes a set of logic functions that produce the next state and output of the circuit, which can be implemented by a combinational logic circuit. Connecting the next state output with the present state input of this circuit should yield the desired behavior as long as the circuit operates in Fundamental Mode or Huffman Mode.

There is still one problem, though: digital gates have a *finite, non-zero* delay. So the implementation of the specified logic function may not behave, as we assumed above, as a pure logic evaluator followed by a delay element. We must take into account the behavior of the *individual gates* that constitute the combinational logic part of the *Huffman model.*

A simple example will give an idea of the nature of the problems encountered during this final step of classical asynchronous circuit synthesis. Note that these problems (unlike FT minimization and encoding) *do not have a general satisfactory solution yet,* and are a major motivation of the introduction of the novel synthesis algorithms presented hereafter.

The logic expressions describing the next state signals (Y_1 and Y_2) and the output signal (z) of the FT described in Figure 2.8, using the state encoding derived in Section 2.3.3, are as follows:

$$\begin{aligned} z &= x_1 y_2 \\ Y_1 &= \overline{x_1} y_1 + y_1 \overline{y_2} + x_2 \overline{y_2} + \overline{x_1} x_2 \\ Y_2 &= x_1 y_2 + \overline{x_1} y_1 + \overline{x_1} x_2 \end{aligned}$$

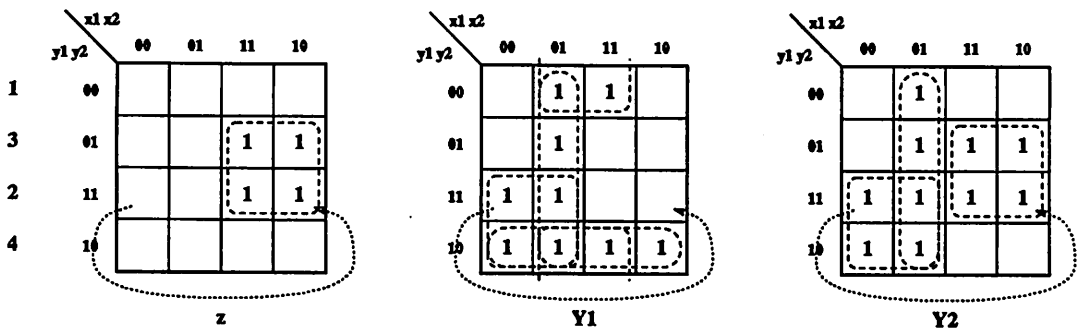


Figure 2.9: Karnaugh maps of the encoded Flow Table of Figure 2.8

where y_1 and y_2 are the present state signals and x_1 and x_2 are the input signals. The Karnaugh maps of the three logic functions, with the implicants implementing them, are drawn in Figure 2.9 (the leftmost column represents the symbolic present states, before encoding). So we obtain the two-level circuit given in Figure 2.10 (ignore the dashed gate for now).

Let us examine now the circuit operation in more detail. The discussion follows that in [117]. Consider the transition from total state (input state, internal state) (00,2) to (10,2) and then to the stable total state (10,3), due to a rising edge on x_1 (represented by the dotted arrows in Figure 2.9). Output z rises, next state Y_1 falls and next state Y_2 should stay at 1. But the output of one *and* gate, $\overline{x_1}y_1$, goes to 0 while another one, x_1y_2, goes to 1. If the gates have non-zero, different delays (due, e.g., to process variations or different rising and falling times), the output of the *or* gate implementing Y_2 can go briefly to 0. With a very large delay on gate x_1y_2 the circuit would stabilize in *another*, incorrect total state, (10,1).

The pulse on the output of the *or* gate is termed a *combinational logic hazard* because it depends on the magnitude of the delays (hence "combinational"), and it can be eliminated by changing the *implementation* of the function (hence "logic"). For example, if we add the gate y_1y_2 to the circuit implementing Y_2, this gate does not change value when x_1 rises, thus ensuring that no hazard occurs *assuming that the delays in the feedback lines are much larger than the delays in the combinational logic*. Moreover, the gate implements a *redundant* implicant of the original function, so its addition does not change the logic function of the circuit.

If the assumption that the feedback delays are much larger than the combinational logic delays is not met, there is still a problem. If y_1 falls too fast, the output of gate y_1y_2 may fall, and the circuit ends up in the (incorrect) total state (10,1). This second hazard is *sequential* (because it involves the feedback lines) and *essential* (because it is inherent in the FT specification and independent of the particular logic implementation).

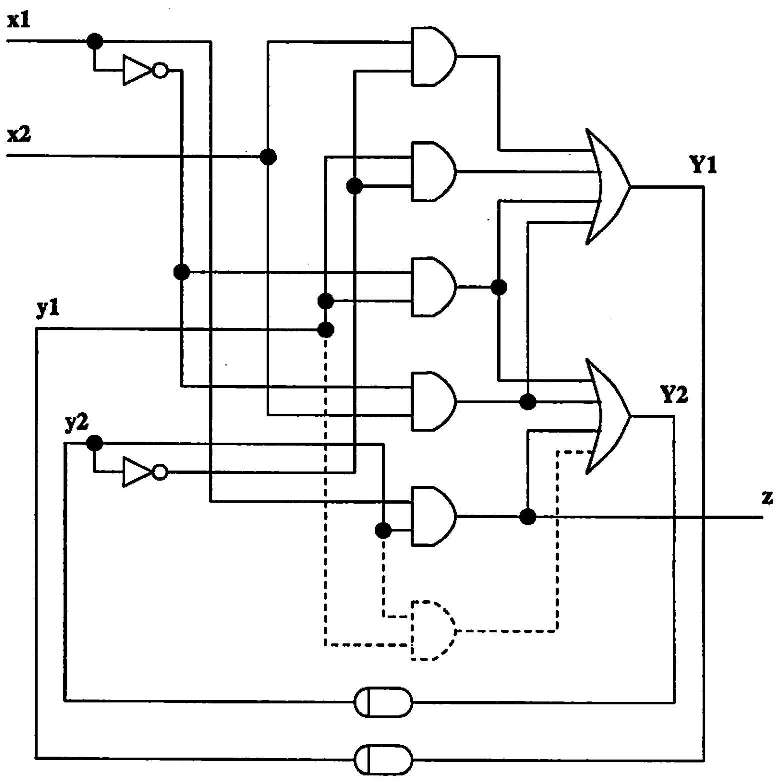


Figure 2.10: A Huffman circuit implementation of the Karnaugh maps of Figure 2.9

In this case the problem is due to some part of the circuit (gates $\overline{x_1}y_2$ and y_1y_2) perceiving the transition on x_1 that *caused* the transition on y_1 as occurring *after* the transition on y_1. The problem can be avoided by using *delay padding*: if the change of y_1 is slow enough, no hazard occurs. This observation, which has been used in classical Huffman model asynchronous circuit synthesis, will be the key to our own hazard-free synthesis algorithm (described in Chapter 5).

We can now more precisely define a **hazard** as any transition on an output signal at a time when no such transition is allowed by the specification. Using the Huffman model and a critical race-free state encoding, a hazard is *any change of value on some output or next state signal that would not occur if the delays were concentrated on the feedback lines* (rather than being distributed on gates and wires).

According to a common taxonomy from the literature, a hazard is classified as static or dynamic and as combinational or sequential:

- A **static hazard** exists if a signal that should remain constant changes twice (in opposite directions). It is a **0-hazard** if the signal should stay at 0, and a **1-hazard** if the signal should stay at 1.
- A **dynamic hazard** exists if a signal that should change once changes multiple times.
- A **combinational hazard** is one that may or may not occur due to the distribution of delays. The first example of a hazard discussed above is combinational because it occurs only if the difference between the delays of x_1y_2 and $\overline{x_1}y_1$ is larger than the delay of the *or* gate implementing Y_2 (recall that the Huffman model assumes *inertial delay*, hence short pulses are "killed").

  Combinational hazards are further classified as:

  - **logic** hazards if they depend on the *particular implementation* of the logic function (as in the first example above),
  - **function** hazards if they cannot be eliminated, regardless of the delays, by changing the logic.

  Several properties of combinational hazards have been established ([117]):

  1. A two-level implementation of any logic function can be made static 1-hazard-free under *Single Input Change* by including *every prime* implicant in the implementation (this is sometimes referred to as a Disjunctive Normal Form implementation [126]).

2. Transformations of the circuit using:
   - De Morgan's laws: $\overline{ab} = \overline{a} + \overline{b}$ and $\overline{a+b} = \overline{a}\,\overline{b}$
   - the associative law
   - the distributive law (factoring, but not multiplying out)
   - the absorption laws: $a + ab \rightarrow a$ and $a + \overline{a}b \rightarrow a + b$

   do not introduce any *new hazards*, under *Fundamental Mode* operation, besides those initially present. See Section 5.1.2 for a proof that this important result holds also under less restrictive circuit operating conditions. Also, see [68] for a longer list of *hazard-non-increasing* logic transformations.
3. Almost no logic function has a hazard-free implementation, under unrestricted *Multiple Input Change*. For example, even $f = a + b$ has a combinational hazard in any implementation if one input rises at the same time as the other one falls.

- A **sequential hazard** is one that inherently involves the feedback lines. Sequential hazards that are inherent in the FT specification and occur regardless of the actual implementation (as opposed, e.g., to hazards due to critical races, that can be eliminated with a proper state encoding) are **essential** hazards.

  Such essential hazards, as shown in [117], are identified by a simple property of a minimized FT: for some initial total state and input signal x_i, the total state reached after *three* consecutive transitions on x_i is *different* from the total state reached after *one* single transition on x_i. For example, in Figure 2.8, if we start from (00,2) and change x_1 once we reach (10,3), while if we change x_1 three times we reach (10,3), then (00,1), then (10,1). So the FT has an essential hazard that led to the problem described above.

  A sequential hazard is **transient** if its effect does not change the final total state of the circuit when it stabilizes, and **steady-state** if its presence changes the stable state with respect to the FT specification.

  Several properties of sequential hazards have been established ([117]):

  1. Any FT without essential hazards admits an implementation free of steady-state hazards without the need for *delay padding* if operated in *Single Input Change* mode. The procedure involves the analysis of the so-called **d-trio** conditions (i.e., conditions in which one and three changes of a signal lead to exactly the same final state of the FT) and uses a *dichotomy*-based formulation as in Tracey's state encoding.

2. No FT with essential hazards has an implementation without steady-state hazards independent of the feedback delays. As noted in [20], the Theorem of [117] is the first known proof that some behavior has no *delay-insensitive* implementation (see also Section 2.6).

3. Any FT has an implementation without steady-state hazards using *at most* one delay element if it is operated under *Single Input Change*, if *upper and lower bounds* on the delays of gates and wires are known, and if the *lower bound* on that delay can be arbitrarily set by the designer.

4. The situation under *Multiple Input Change* is more complicated, especially if different orderings of multiple transitions which occur simultaneously lead to different stable states. For example, in the following FT row:

	x_1x_2			
	00	01	11	10
1	$\boxed{1}$,0	2,0	3,0	4,0

   the feedback delays must be chosen to be larger than the maximum separation between transitions (δ_1 as defined by the *Huffman Mode* of operation).

We will not enter into further details about classical hazard analysis and elimination techniques, since the FT specification formalism does not provide a clean way of handling such problems. We only mention one last type of hazard that provided the motivation for the work described in Section 2.5 on speed-independent circuits: the **delay hazard**. It is defined as a condition in a circuit without logic hazards such that a sequence of input changes $x' \rightarrow x'' \rightarrow x'''$ produces either of the following sequences on some signal (described by a function of the inputs $f(x)$):

1. $f(x') \rightarrow f(x'') \rightarrow \overline{f(x'')} \rightarrow f(x''')$, where $f(x'') = f(x''')$. For example, the application of $0111 \rightarrow 1111 \rightarrow 1011$ to the two-level circuit $z = abd + acd$ may cause such a delay hazard, if *abd* is much faster turning on and then off than *acd* is turning on. In this case, the environment applies the second transition *before acd* turns on; *acd* is the gate that would have kept the output stable at $f(x'') = f(x''') = 1$.

2. $f(x') \rightarrow f(x'') \rightarrow f(x''') \rightarrow f(x'') \rightarrow f(x''')$, where $f(x'') \neq f(x''')$. For example, the application of $0111 \rightarrow 1111 \rightarrow 1110$ to the above circuit may cause such a delay hazard if *abd* turns on fast, the environment changes the input, *abd* turns off and then *acd* turns on and

off. Note that this latter case happens only if the *pure delay* model is used, rather than the *inertial* delay model that has been used thus far.

In summary, a delay hazard is caused by two transitions *too close together*. It occurs if the environment monitors the circuit output, sees an output transition in response to the input transition, and applies the second transition immediately. It occurs if the circuit is operated in *Input-Output Mode* (in which the environment applies a new change to a circuit that is still unstable) rather than in *Fundamental Mode*. A brief account of the analysis and synthesis techniques for circuits that must operate in Input-Output Mode is given in Sections 2.5 and 2.6.

The Huffman model, which decomposes the circuit into a combinational part and a set of feedback lines, is not the only model used for asynchronous circuit synthesis from a Flow Table specification. At least two other methods deserve mention in this very brief summary because they are somewhat related to our work. The first method, described in Section 2.3.5, uses a fixed architecture, which directly matches the Flow Table specification. It completely skips the state encoding step while making hazard analysis much simpler. The other method, described in Section 2.3.6, couples the advantages of synchronous and asynchronous logic with a *self-clocked* design methodology.

2.3.5 One-Hot Encoding-based Synthesis

The basic idea of **one-hot state encoding** is to assign one bit of memory to each minimized Flow Table state. The method was proposed by Hollaar ([52]) and is related to various direct methods for Petri net implementation, such as, for example, [93], [84] and [1].

One-hot encoding has some significant advantages over the standard FT synthesis method described in the previous sections.

1. The critical race problem disappears, since exactly two state signals change for each transition. The circuit can also be checked for stabilization in the final state by delaying the generation of the new outputs until exactly one of the state signals is at 1.

2. The logic implementation becomes highly structured: one *and* gate implements each state transition, two *or* gates and a Set-Reset flip-flop implement each state bit, and one *or* gate implements each output ([52] considered only Moore FTs, but the approach can be generalized to Mealy FTs). This methodology naturally lends itself to completely automated layout in a PLA-like style. Furthermore, hazard analysis is much simplified. Figure 2.11 contains a

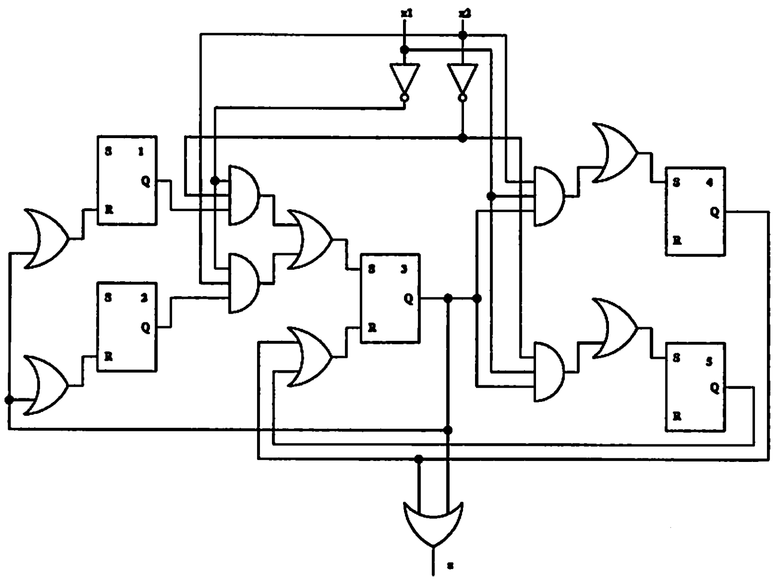


Figure 2.11: Example of one-hot-based Flow Table synthesis

circuit fragment implementing the following FT fragment (each *SR* flip-flop is labeled with the corresponding FT state):

	x_1x_2			
	00	01	11	10
1	3,1	$\boxed{1}$,0		
2		3,1	$\boxed{2}$,0	
3	$\boxed{3}$,1	$\boxed{3}$,1	4,1	5,0

The primary disadvantages of one-hot-based synthesis are:

1. Very little optimization is possible given the fixed architecture. The area of the implementation in terms of gate count, for example, *may* grow more rapidly as a function of the FT complexity than it would with methods based on more compact state encodings.

2. Some specific characteristics of the FT are poorly handled. For example, a so-called *scale-of-two-loop* between states, such as the case shown in the following FT fragment:

	x_1x_2			
	00	01	11	10
1	2,1	$\boxed{1}$,0		
2	$\boxed{2}$,1	1,0		

In this case, state 1 goes to state 2 under input 00 and state 2 goes to state 1 under input 01. This causes a problem to the circuit structure shown in Figure 2.11 because the logic that resets the flip-flop of state 1 needs the flip-flop of state 2 to be on (so that it goes through the transition $\ldots 1 \ldots 0 \ldots \rightarrow \ldots 1 \ldots 1 \ldots \rightarrow \ldots 0 \ldots 1 \ldots$). However if flip-flop of state 1 is on, it also resets the flip-flop of state 2, and the simple scheme fails.

This can be solved inserting unstable states between states 1 and 2 in the FT, but the solution further increases the already high cost of this methodology.

The hazard analysis with this methodology becomes greatly simplified. Even some deviations from Fundamental Mode become tolerable, and the circuit may be operated in a variation of Huffman Mode with a minimum pre-specified separation between different sets of input changes. Moreover the nature of the method allows some *concurrent* activity between states. *Fork* and *join* constructs can be used, and repeated subgraphs of the FT can be factored into subroutines.

As we have seen, the one-hot-based synthesis methodology has one significant advantage: *simplicity*, and one significant disadvantage: *cost*. A very attractive alternative way to achieve the same advantage is to combine the best of both the *synchronous world* (simplicity, efficiency) and the *asynchronous world* (speed, modularity) as outlined in the next section.

2.3.6 Self-Clocked Circuits

The idea of **self-clocked circuits** is quite simple and attractive. The circuit is decomposed into three parts (e.g. as in Figure 2.12, from [90]):

- a clock logic, which produces a clock pulse whenever the state and/or output signals must change. In general, there exists a set of one-sided inequalities that the delay of the clock signal must satisfy in order to ensure correct, hazard-free operation.

- a set of clocked memory elements.

- a next state/output combinational logic that does not have to be handled with special care to avoid races and hazards. The clock is chosen to be *slow enough* to allow the output to settle before it is allowed to propagate to the feedback wires and to the environment.

Various approaches to the design of self-clocked asynchronous circuits have been presented in the literature ([29], [135], [85, 101], [69], [2], [90] to name but a few).

Some distinctive points among them are:

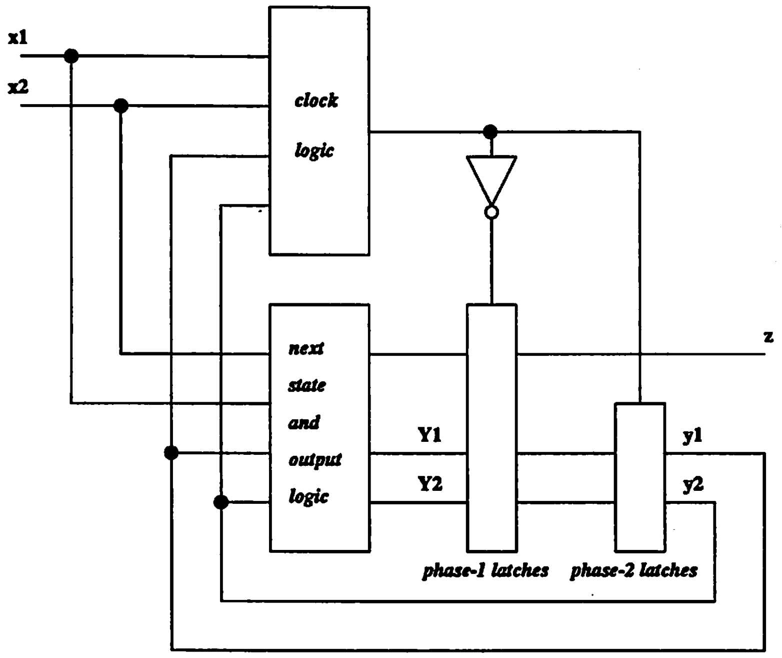


Figure 2.12: Example of self-clocked architecture (from [90])

- Nowick *et al.* ([90]) use a restricted form of FT specification, called *burst-mode* FSM (see also Section 2.5.3), in which:

  1. each state transition is restricted to occurring under a set of input changes (an *input burst*) such that no burst from some state can be a subset of another burst from the same state (e.g., if transitions on x_1 and x_2 move the FSM from s_1 to s_2, then a a single transition on x_1 cannot move the FSM from s_1 to any other state except itself),
  2. a given state must always be entered with the same set of input values.

  Apart from these two limitations, the clocking scheme is such that:

  1. hazard-free logic can always be produced from the FSM specification.
  2. the circuit can operate in *Input-Output Mode*, rather than in Fundamental Mode, because the clock delay constraints ensure that the inputs can be changed again whenever the output change (*output burst*) associated with the previous state transition has occurred.

- Ladd *et al.* ([69]) use a *bundled data* communication scheme, where input and output signal validity is defined in reference to a hazard-free local clock signal. Moreover, both inputs and outputs are latched, so the circuit may in general be slower than in [90].

- Rosenberger *et al.* ([85, 101]) use a special type of flip-flop with a *completion signal,* called **Q-flop**, to ensure that the circuit has stabilized when the clock pulse is issued. This limits the need for precise delay requirements to a single flip-flop, rather than a complete self-clocked FSM as in the other approaches. The design of the Q-flop itself is based on analog design techniques to ensure freedom from *meta-stability* problems.

  Unlike the other approaches in which the clock logic can be rather complex, (because it must detect input changes that require state/output changes), here the clock is an **n-input** *C* **gate**[7] whose inputs are the Q-flop ready signals. Only a delay is required to be sure that the combinational logic has computed the desired values correctly.

  This approach also provides a scan-based testing methodology (including the testing of the clock delay) and the self-clocked modules are used within a framework that ensures *delay-insensitive* operation among them.

2.4 Micropipelines

This section describes the work done by Sutherland on efficient implementation of asynchronous pipelined modules. The methodology does not fit precisely in the classical taxonomy, as it uses speed-independent signaling protocols coupled with delay matching for data communication. It deserves mentioning, though, because it can lead to very efficient and fast implementation of arithmetic units.

Sutherland summarized his work on self-timed circuits in his Turing Award Lecture ([113]), where he described an approach based on three basic ideas:

1. pipelined organization of the computation (hence the name **micropipeline**), ranging from a simple buffer, to a floating point divider ([128]).

2. **two-phase handshake**, (also called **transition signaling**) that permits a much faster operation than the classical **four-phase handshake** (due to the lack of a resetting phase), at the expense of slightly more complex circuitry.

3. **bundled data** communication protocol, where control signals define the validity period of data signals. Only the control part must be designed with asynchronous methodologies. The

[7]A *C* gate, or Muller *C* element, is a sequential gate whose output *c* rises when all its inputs rise, and falls when all its inputs fall. For example, a *C* gate with two inputs implements the function $c = ab + bc + ac$.

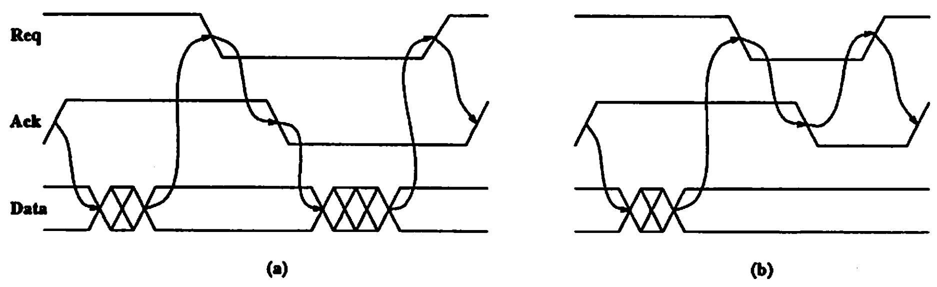


Figure 2.13: Four-phase and two-phase handshaking (from [113])

only drawback is that the delay of the control part must be guaranteed to be larger than the delay in the data part.

The bundled data protocol is illustrated in Figures 2.13.(a), where signal *Req* changes value after the data lines have stabilized. The delay of *Req* must be adjusted to allow the combinational logic output to settle and the latch setup time to be satisfied, as shown in Figure 2.15. Note that the *Cd* signal produced by stage i plays the role of *Ack* for stage $i - 1$ and of *Req* (after an appropriate delay) for stage $i + 1$.

Figure 2.13.(a) shows a bundled data two-phase handshake, as opposed to the four-phase handshake shown in Figure 2.13.(b). Figure 2.14 describes the **event-controlled latch** (i.e., a latch active on *both edges* of the clocking signal) that is needed to operate in a two-phase environment. The **toggle** gate transmits input transitions alternately to each output, beginning with the output marked with the dot. A transition on the C input puts the latch into "memory" mode, while one on the P input puts the latch into "transparent" mode. Cd and Pd acknowledge the respective input transitions, ensuring that hold time requirements are met.

Sutherland's approach does not currently provide an automated design methodology. However, it offers a very efficient *architectural framework* for control-oriented automated synthesis methods such as ours and those based on the unbounded gate delay model as described in the next section.

2.5 Speed-independent Circuits

The seminal work of Muller ([88], see also [83]) introduced the first *formal* model of an asynchronous circuit using the concept of a State Transition Diagram. A more complete and precise account of the relationship between the **Muller model** and the Signal Transition Graph and Change

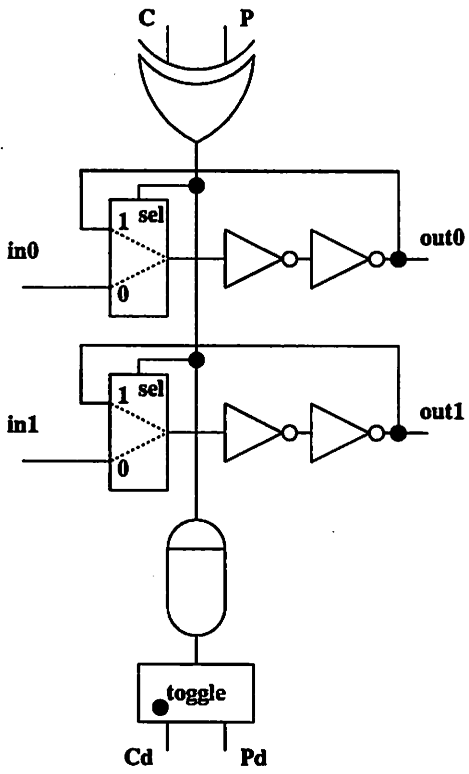


Figure 2.14: An event-controlled latch (from [113])

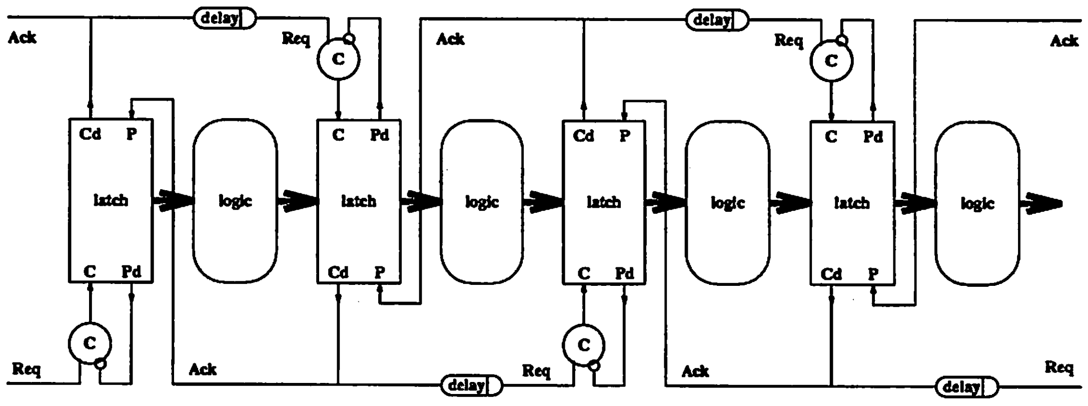


Figure 2.15: The organization of a micropipeline with data processing (from [113])

Diagram models is given in Chapter 3. Here we only summarize it, for the sake of comparing it with other asynchronous circuit modeling and synthesis methods.

Muller considered *autonomous* asynchronous circuits, i.e., circuits without *primary inputs* (also called *complete* circuits). He modeled a circuit as a set of logic gates, using *inertial unbounded gate delays*. Each gate is associated with a *logic expression* which describes a completely specified logic function of the form:

$$z_i = \phi_i(z_1, \ldots, z_n)$$

A gate is *combinational* if its expression does not depend on the output signal, while it is *sequential* if its expression depends on the output signal. For example, an *or* gate, with $z_1 = z_2 + z_5$ is combinational, while a *C* gate, with $z_2 = z_2(z_3 + z_4) + z_3 z_4$ is sequential. Note that many different expressions can represent the same function, so we will assume that the support of the expression is *minimum* (i.e., it contains only signals that the corresponding function essentially depends on).

Muller showed that the behavior of the circuit can be *equivalently* described with a Finite Automaton called a *State Transition Diagram* (STD, called State Diagram in [88] and [83]). This means that:

- given an autonomous circuit $\mathcal{A}$ modeled with *unbounded gate delays* there exists a unique STD $\mathcal{S}$ representing $\mathcal{A}$, and
- given an STD $\mathcal{S}$, if there exists an autonomous circuit $\mathcal{A}$ modeled with *unbounded gate delays* whose STD is isomorphic to $\mathcal{S}$, then this circuit is *unique* up to Boolean expression equivalence. Note that an STD may not have a corresponding circuit: see Section 3.1.1 for an example.

A *state* of the STD is labeled with a *vector of values* on gate outputs. An element of the vector can be:

- *stable* if it is equal to the value computed by the corresponding logic function under the input values given by the vector.
- *excited* otherwise.

For example, the output of an *or* gate whose inputs have value 0 is excited if it has value 1 (it will eventually go to 0 after a finite but unbounded amount of time), and stable if it has value 0.

The *State Transition Diagram* is a directed graph where each node corresponds to a state, and an edge joins a pair of states $s_1 \rightarrow s_2$ if a gate (or a set of gates) that is excited in s_1 has an output transition towards its stable value reached in s_2.

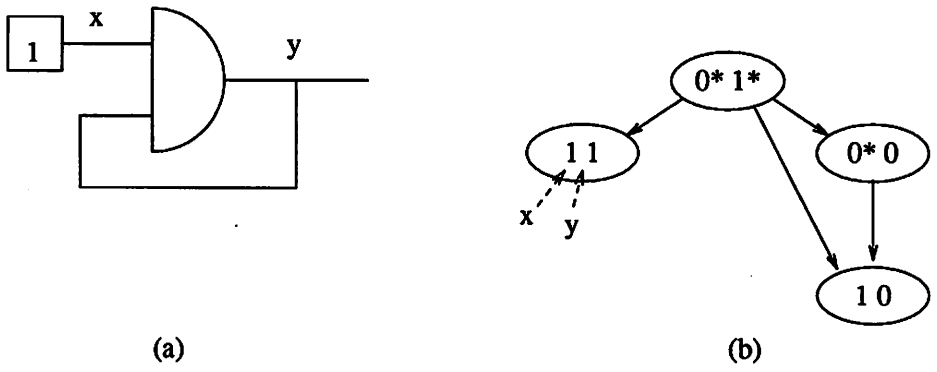


Figure 2.16: Muller model of a circuit.

The circuit described in Figure 2.16.(a) is built out of a constant value 1 and an *and* gate, and has the STD described in Figure 2.16.(b) ("1*" or "0*" denotes an excited value).

The set of states of the STD is partitioned into a set of *equivalence classes* of mutually reachable states (i.e., two states s_1 and s_2 belong to the same equivalence class if there exists a directed path from s_1 to s_2 and one from s_2 to s_1). An equivalence class of states is a *final class* if no other equivalence class can be reached from it.

A circuit is defined to be *speed-independent with respect to a state* if there is *only one final class* reachable from that state. A circuit is *speed-independent* if it is speed-independent with respect to all STD states. In Figure 2.16.b each state belongs to a distinct equivalence class, and the states labeled 11 and 10 are final. The circuit is speed-independent with respect to states 11, 0*0 and 10. It is not speed-independent with respect to state 0*1*. So the circuit is not speed-independent.

A circuit is *semi-modular* if every excited signal becomes stable only by changing its value (i.e., not because one of the gate inputs has changed value). Muller showed that a semi-modular circuit is speed-independent, but the converse is not true. Both circuits in Figures 2.17 and 2.18 are speed-independent, because their STDs have only one equivalence class, which is of course final. The circuit in Figure 2.17.(a) is not semi-modular, because, for example, in state 10*0* signal y can become stable without changing if signal z is "fast enough" to change.

In the next sections we will briefly examine methods to synthesize circuits under the unbounded gate delay model, using the self-synchronizing codes (Section 2.5.1) or a set of transformation techniques for the synthesis of semi-modular circuits (Section 2.5.2). We also outline some speed-independent circuit synthesis methods based on the Signal Transition Graph specification (Section 2.5.3).

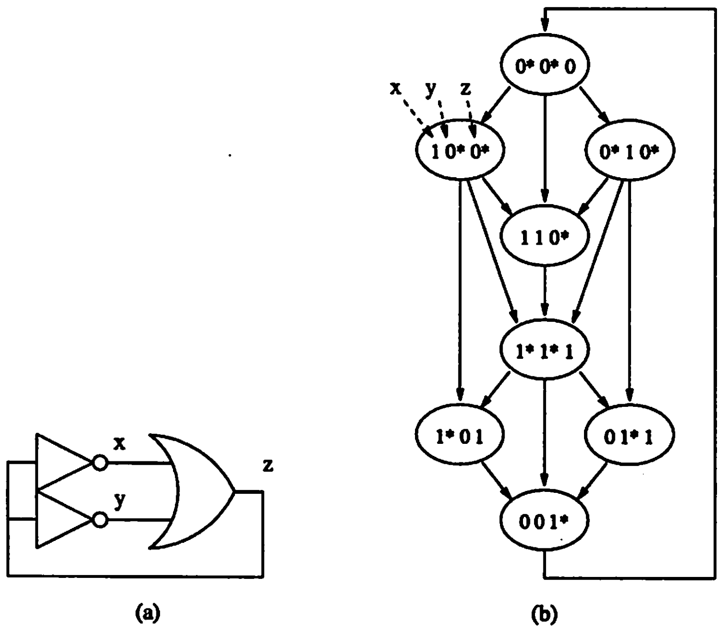


Figure 2.17: A speed-independent but non semi-modular circuit

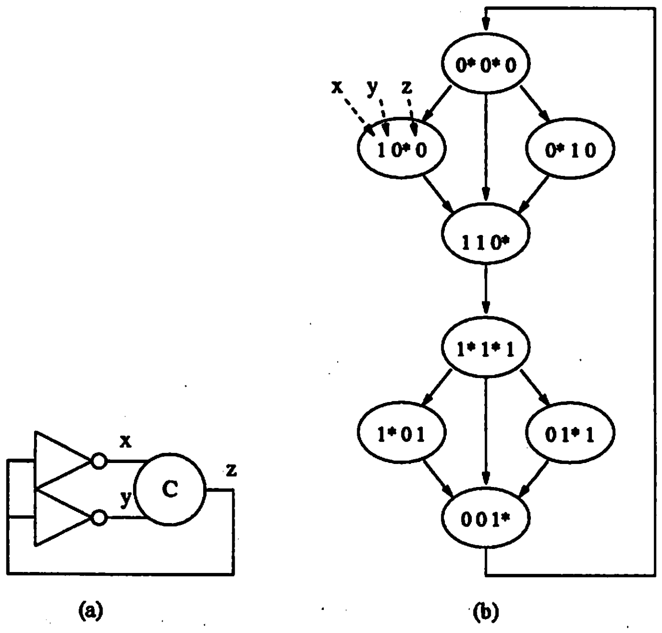


Figure 2.18: A speed-independent and semi-modular circuit

2.5.1 Synthesis with Self-Synchronizing Codes

Armstrong *et al.* ([5]), following the modeling work done by Muller, provided a formal design methodology for circuits using the unbounded gate delay model. The basic idea is to use a **self-synchronizing code** to encode the input and output states of the circuit (which can be combinational or sequential) and to use a request/acknowledge protocol to synchronize various cooperating circuits.

A self-synchronizing code (see [126] for a detailed analysis) uses two classes of binary vectors to encode input and output signals. It is assumed that the two classes of codes *alternate* in time in the operation of the circuit, both on its inputs and its outputs (so that two such circuits can communicate successfully). A basic property of such codes is that whenever any code belonging to one class starts changing to become any code of the other class, no *intermediate state* reached in between can be a *valid code* of the second class, independent of the order in which individual signals change. In this way it is possible to use a **code checker** circuit to detect the completion of the transition.

The **alternating data sets** method achieves this result by using *pairs* of bits, each of them representing one bit of "actual information". These bits assume the following configurations in the two classes:

information	class 1	class 2
0	01	00
1	10	11

A checker for class 1 is the *and* of the *ex-or* of each pair, while a checker for class 2 is the *and* of the *ex-nor* of each pair.

For example, suppose that a set of wires *representing* the information 0110 initially must change to 1010 and then 1001. Assuming that the operation begins in class 1, 0110 is represented as 01 10 10 01. This code changes to 11 00 11 00 (using class 2), and then to 10 01 01 10. The reader can check that (assuming *no hazards* on stable signals) the first transition can only go through intermediate states that are not members of class 2, and the second transition can only go through intermediate states that are not members of class 1.

In the **spacer** method one class contains all the "useful" *data codes*, and the other contains only a single *spacer code*. One example is the **dual-rail code**, where the spacer is the all-zero code, and information bits are encoded as in class 1 above.

The same set of transitions as above can be implemented with a dual-rail code by going from 01 10 10 01 to 00 00 00 00 (the spacer), then 10 01 10 01, then to 00 00 00 00 and finally

10 01 01 10. The transitions obviously may be slower in this case, due to the need to go through an idle phase between two active phases.

Alternating data sets may require four times as much logic to implement the function as would be required by a standard combinational logic implementation. In addition, they require a multiplexer circuit to generate the outputs for the currently active class. Each circuit *requests* a new set of data from the circuit feeding its inputs and *acknowledges* its absorption using a pair of wires driven by class 1 and class 2 detectors.

The synthesis method for combinational functions using spacer codes relies on a two-level sum-of-products representation of the function, where all input combinations that belong to transients are assigned to the off-set of the function. This ensures that the output is the *spacer* as long as a transient is present on the input. So spacer codes require less gates than alternating data sets (only about twice the size of a standard combinational logic implementation), but are *slower*, due to the idle phase ([5] and [31]).

A similar methodology can also be used to implement sequential circuits from an FT specification without the need for *Fundamental Mode* or *Huffman Mode* hypotheses.

Armstrong *et al.* observed that the nature of the methodology ensures a high degree of *self-checking*, because almost half of the faults will stop the circuit operation (see also [126] and [6] for more general results on self-checking properties of speed-independent circuits).

The dual-rail encoding is still the most common way of implementing *data-oriented* computations with unbounded delays, as opposed to *control-oriented* ones. It can be very advantageous if coupled with an implementation technology such as DCVS ([50]) that automatically provides glitch-free operation and dual-rail outputs.

Speed-independent control logic, on the other hand, is probably better implemented using techniques derived from the theoretical work of Varshavsky *et al.* on the realizability of various classes of speed-independent circuits using particular logic families, as described in the next section.

2.5.2 Synthesis of Semi-Modular Circuits

A very interesting body of research on asynchronous circuit synthesis in Russia was brought to the attention of the Western world only recently in a book by Varshavsky *et al.* ([126]). The most significant result from that work in the area of asynchronous circuit synthesis using the *unbounded gate delay* model is the constructive proof of the existence of an implementation of any semi-modular STD with *and-or-not* gates and with 2-input/2-fanout *nand* and *nor* gates.

Let us suppose we are given an STD that is known to be semi-modular (for example the STD obtained from an STG without choice, as shown in Section 3.4.1). We would like to know if there exists a semi-modular implementation of the *same* STD using a particular *family of logic gates*:

- *n-input* and-or-not gates, i.e., gates that can implement an arbitrary sum-of-products logic function with only one inversion, which must be at the gate output. This class of gates can be considered an acceptable approximation of the actual behavior of static *CMOS* gates, with some limit on the number of stacked transistors.
- a mixture of *2-input* nand and *2-input* nor gates, with fanout limited to two gates.

The implementability result is established for every class of gates in various steps. First of all we need some definitions.

A **decomposition** of an asynchronous circuit described by a set of gates each with a logic expression of the form:

$$z_i = \phi_i(z_1, \ldots, z_n), \;\; i = 1, \ldots n$$

is a set of logic expressions defined on a *superset* of the signals $z_1, \ldots z_n, \ldots, z_{m+n}$ as:

$$z_i = f_i(z_1, \ldots, z_{m+n}), \;\; i = 1, \ldots m+n$$

such that the function denoted by the expression

$$\phi_i(z_1, \ldots, z_n)$$

is the same as the function denoted by the expression

$$f_i(z_1, \ldots, z_n, \ldots f_{m+n}), \;\; i = 1, \ldots n$$

Note that the decomposition is *well-defined* only if none of the f_i for $i = n+1, \ldots m+n$ introduces a cycle in the dependencies, i.e., if pure *resubstitution* of the f_i's is sufficient to yield the original set of functions.

For example, consider the semi-modular circuit described by:

$$\begin{aligned} z_1 &= z_2 z_3 \\ z_2 &= z_3 z_4 + z_2(z_3 + z_4) \end{aligned}$$

$$\begin{aligned} z_3 &= \overline{z_1} \\ z_4 &= \overline{z_2} \end{aligned}$$

Then a well-defined decomposition of this circuit, introducing the new variables z_5, z_6 and z_7 is, for example:

$$\begin{aligned} z_1 &= z_2 z_3 \\ z_2 &= z_5 + z_6 + z_7 \\ z_3 &= \overline{z_1} \\ z_4 &= \overline{z_2} \\ z_5 &= z_3 z_4 \\ z_6 &= z_2 z_3 \\ z_7 &= z_2 z_4 \end{aligned}$$

This circuit is ***no longer*** semi-modular because in state 0*111*10*0* (for ordering $z_1 \ldots z_7$) variables z_4 and z_6 are both excited, and if z_4 falls, it ***disables*** z_6 without giving it the option to change.

A circuit $\mathcal{A}'$ is a **correct implementation** of a circuit $\mathcal{A}$ (or, equivalently, of the STD associated with $\mathcal{A}$) if

- the logic expressions of $\mathcal{A}'$ are a decomposition of those of $\mathcal{A}$ and
- $\mathcal{A}'$ belongs to the same class as $\mathcal{A}$ (we are primarily interested in the semi-modular class because of the connection between semi-modularity and hazards described in Section 2.7.2).

In the example above, the decomposition is valid but it is *not a correct implementation* because it loses semi-modularity.

The *excitation region* of an STD signal z_i is a *maximal set* of connected states in which z_i is excited to the same value. Excitation regions in which z_i is labeled 0* are denoted by S_{ij}, regions in which it is labeled 1* are denoted by R_{ij}[8]. The excitation region S_z of signal z in Figure 2.17 contains the states 10*0*, 110*, 0*10*, and R_z contains only the state 001*.

[8]The j index is used to identify multiple excitation regions of the same signal. We will often drop it if signal z_i has only two excitation regions.

Each excitation region of each signal z_i defines a unique **excitation function,** which is a completely specified logic function that does not depend on z_i and evaluates to 1 exactly on the vertices in the region (so we will often identify excitation regions and excitation functions in the sequel). A cover of an excitation function is in **Disjunctive Normal Form** if it includes all *prime implicants* of the function[9]. The set of prime implicants of a given logic function is unique, so its Disjunctive Normal Form is also unique (even though it may be much larger than a prime and irredundant cover).

An STD has a **term takeover** with respect to a signal z_i if there exists a path of states, all belonging to a single excitation region for z_i, such that all the terms of Disjunctive Normal Form for the function of that region that were at 1 at the beginning of the path are at 0 at the end of it. Note that some other term in the Disjunctive Normal Form must go from 0 to 1 along the path in order to keep the function at 1.

Consider signal y in Figure 1.3.(c). Its excitation region S_y contains the states 10*0*, 1*0*1, 00*1, and its Disjunctive Normal Form is $x + z$. The path 10*0* $\rightarrow$ 1*0*1 $\rightarrow$ 00*1 has only the (trivial) prime implicant x at 1 at the beginning, and the same implicant is at 0 at the end, while implicant z goes from 0 to 1 along the same path. So the STD has a term takeover with respect to signal y.

The first result of [126] is the proof that every term takeover can be eliminated from a circuit without losing semi-modularity. The proof itself contains a constructive method for adding at most three additional signals for each takeover. The idea is to add a signal that remains at 1 across the two transitions that turn off a term initially at 1 and turn on a term initially at 0. This creates a new set of prime implicants that remain at 1 along the path, and it can be done so that the new signals have excitation regions free from new term takeovers (called **secondary** takeovers).

Then Varshavsky *et al.* proceed to describe a method for implementing semi-modular STDs in each family of logic gates.

- n-input *and-or-not* gates are handled by proving that given a semi-modular STD, one can derive logic expressions for every signal z_i that can be factored as:
  $z_i = z_i\overline{R_i} + S_i$
  where S_i and R_i (Boolean sums of the excitation functions for z_i) represent **disjoint** functions (i.e., $R_iS_i = 0$) and do not depend on z_i.

[9]Some authors use the term Disjunctive Normal Form to denote any sum-of-products representation. Here we will use the more restrictive notion, as in [126].

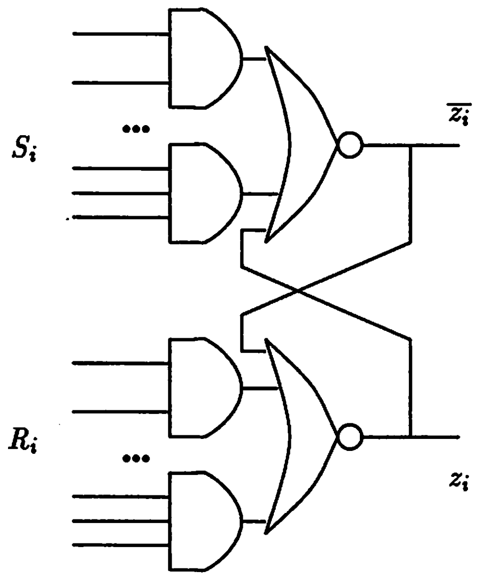


Figure 2.19: The "perfect implementation" structure with *and-or-not* gates

This **perfect implementation** of the STD is described in Figure 2.19, where S_i and R_i are the Disjunctive Normal Form of each excitation function of z_i. The authors then show that at least one prime of S_i or R_i is stable in every STD state where z_i is excited; the output of the corresponding *and-or-not* gate cannot be disabled, and the *perfect implementation* of the given STD is semi-modular. So any semi-modular STD (and hence circuit) has a correct implementation using *and-or-not* gates.

- The constructive proof of implementability for semi-modular STDs using **2-input** *nand* and **2-input** *nor* gates with fanout limited to two gates is much more complex and it is mainly of theoretical interest, due to the large area penalty incurred in the construction. The interested reader is referred to [126] for more details.

In the same direction as [126], recent work by Beerel *et al.* ([8, 7]) has shown that a more efficient speed-independent implementation of an STG specification can be obtained using n-input *not-nand* and *not-C* gates (i.e., *nand* and *C* gates where an arbitrary number of inputs can be complemented). The delay model is slightly less realistic than the previous approaches, as gates with mixed inverting/non-inverting inputs cannot be easily designed with today's *CMOS* technology. On the other hand the gains in area can be substantial, providing an implementation that is competing with efficient bounded delay designs. Moreover [8, 7] uses an extended definition of semi-modular circuits which allows a limited amount of non-determinism in the *environment*.

2.5.3 Synthesis from Signal Transition Graphs

This section briefly introduces the work done on the Signal Transition Graph specification. This specification is a compact, high-level model from which an STD can be automatically derived. The STG can be used as a more user-friendly input also for the synthesis methods described in the previous section, like the similar Change Diagram model ([125, 60, 62]), described in more detail in Section 3.5.

Two fundamental papers, [102] and [26], independently introduced rather similar, Petri net-based, specification formalisms for asynchronous circuits. Both defined the Signal Transition Graph (called Signal Graph in [102]) as an interpreted Petri net (PN), in which each transition is labeled with a signal value change. The idea of using a PN to describe asynchronous behavior was not new. For example Patil and Dennis ([95]) described a method for direct implementation of a PN as an asynchronous interconnection of combinational and sequential gates, corresponding to various local configurations of the PN transition and places. The novelty of the STG approach was to interpret each *transition* of the net as a *change of value* of some input or output signal of the modeled circuit. Moreover, the circuit was not derived directly from the PN, but rather by using an intermediate representation which served as a "bridge" between the two domains and admitted a very *efficient* implementation[10].

This bridge is the PN reachability graph labeled with a vector of signal values, called the *State Transition Diagram* ([102] used the term Transition Diagram and [26] used the term State Graph to denote the STD). An example of STG and of the corresponding STD appears in Figure 1.3. A *sufficient condition* for the STD to correspond to a logic circuit modeled with *unbounded gate delay*, as shown in [102] and [27], is that the STG is *normal*, i.e., that no subset of the STG variables can change value twice without any other variable changing value. If this condition is satisfied, no two reachable markings can be labeled with the same vector of values, and the STD corresponds to a logic circuit whose combinational part computes the *implied value* of each STG signal (Section 1.2.2).

The two approaches had some interesting differences. Rosenblum *et al.* ([102]) defined the STG simply as an interpreted *safe* Petri net. They concentrated on the analysis of the properties of the STD implied by the properties of the STG (as will be described in more detail in Chapter 3). For example, they showed that an STG with an underlying *persistent* PN[11] generates a *semi-*

[10]A similar trade-off appears between classical synthesis methods based on the Huffman model and the one-hot-based method, as discussed in Section 2.3.5.

[11]A PN is persistent if no enabled transition can ever be disabled without firing.

modular STD, and hence can be implemented with a circuit whose behavior does not depend on the delay of the signals modeled by the STG. On the other hand, Chu ([26, 25, 27]) restricted the domain of "acceptable" PNs to those where the powerful, syntactic characterization techniques developed by Hack ([48]) could be applied. Such techniques do not require an exhaustive exploration of the reachability graph of the PN, but are based on a decomposition of a free-choice PN into State Machine components and into Marked Graph components.

In order to apply Hack's result (Theorem 2.2.1), Chu defined an STG as **well-formed** if:

1. its underlying Petri net is free-choice, live, safe, and such that each SM component is marked with exactly one token, and
2. the successor transitions of each multi-successor place can be labeled with transitions of *input signals* only. Non-deterministic choice is limited to the *environment* of the circuit.

To contrast the approaches of Rosenblum *et al.* and Chu:

- [102] defined as *valid* any STG where, for all firing sequences, the transitions of each signal alternate ($x^+ \rightarrow x^- \rightarrow x^+ \ldots$).
- [27] defined the similar, but more restrictive, concept of **live STG** by requiring that for each signal there exist at least one SM to which all the transitions of this signal belong. This SM is marked with exactly one token both initially (by definition of well-formed STG), and after any firing sequence (by Hack's result), so it *syntactically* ensures that such transitions alternate.

Chu's approach allows a relatively easy formal characterization of concepts such as concurrent transitions. The price is a restriction of the class of specifiable behaviors, as pointed out recently by [60, 62] and by [130].

In this work, we try to combine the advantages of both approaches. Namely, in Chapter 3 we give a broad characterization of the circuit properties implied by various STG properties. On the other hand in Chapters 4 and (partially) 5 we use syntactic limitations in order to prove the *correctness* of our state encoding and synthesis algorithms.

The STG has been used as a specification formalism for circuits described using various delay models:

- Chu ([27]) and Meng ([81]) used the *unbounded gate* delay model with *arbitrary complex gates* (i.e., where each gate can implement an arbitrary logic function). Even though the (rather unrealistic) complex gate delay model was not explicitly described in any of those works, it is necessary (as we show in Section 5.3.3) to ensure that the synthesis result is hazard-free.

- Moon *et al.* ([87]) used the unbounded gate delay model as well, but with a two-level logic implementation.

  Strictly speaking, they also require *Fundamental Mode*, which is in contrast with the unbounded delay assumption since the environment cannot be guaranteed to wait long enough to change the inputs without knowing a bound on the circuit delays. The Fundamental Mode assumption is often satisfied provided that the circuit delays are much smaller than the environment delays.

  Hazard-free operation in this case is ensured by:

  1. using enough implicants in the two-level implementation to ensure that each signal remains stable when no transition for it is enabled[12], and
  2. modifying the implicants to remove potential glitches.

- Beerel *et al.* ([8, 7]) used, as outlined in Section 2.5.2, n-input *not-nand* and *not-C* gates with unbounded delays.

The next section summarizes an interesting recent work by Chu, which relates Finite State Machine-based specifications with Signal Transition Graphs in order to combine the advantages of both into a single, integrated design methodology. A brief account of a related work done by Borriello on synthesis from formalized timing diagrams is also given.

2.5.4 Asynchronous Finite State Machines and Signal Transition Graphs

STGs in general seem more suitable for interface, handshaking and timing diagram-like specifications. However, an asynchronous circuit specification is often given in terms of a set of asynchronous Finite State Machines. The latter model is more natural for expressing control-dominated protocols (e.g. [111]), so ideally a complete design methodology should be able to handle both types of specification. In this section we summarize an algorithm recently proposed in [28] for this purpose. It automatically translates an FSM, under reasonable hypotheses on the environment behavior, into an STG.

[12]This is in general less expensive than the classical technique described in Section 2.3.4 that required using *all the prime implicants* of the function.

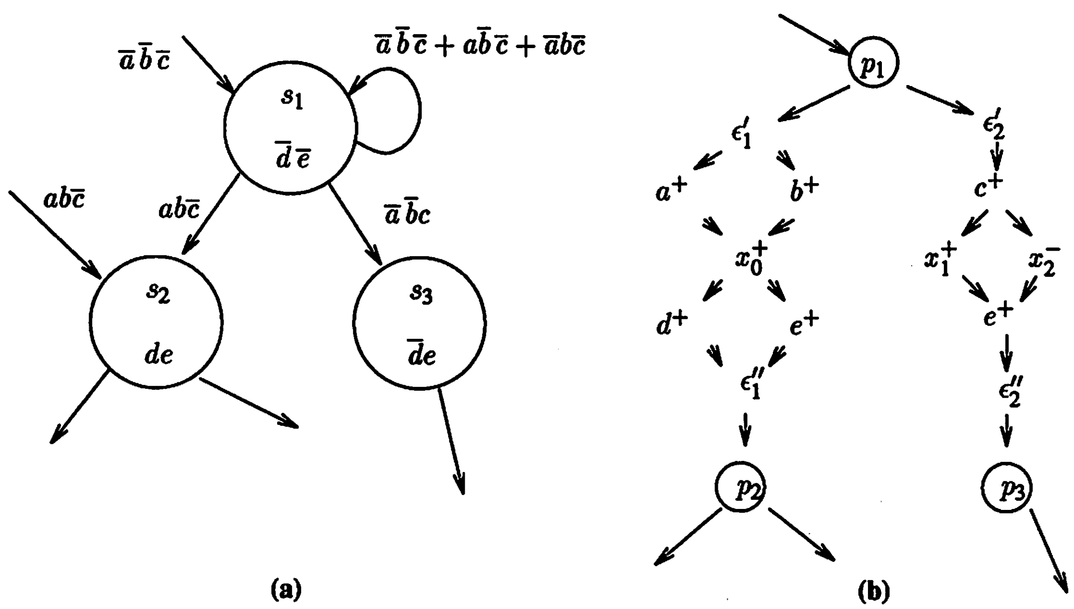


Figure 2.20: An asynchronous Finite State Machine and the corresponding Signal Transition Graph

The FSM is restricted to operating in *burst mode* ([111], [90], see also Section 2.3.6 for a synthesis methodology based on such a specification). That is, for each state, a collection of sets of inputs (*bursts*) can change. Each input in a burst can change at most once, and no burst can be a subset of another burst. When all the inputs in some burst have changed, the machine produces an output burst and moves to a new state. No new input burst can begin before the previous output burst is completed, i.e., the machine operates in *Input-Output Mode*, rather than in Fundamental Mode, as assumed by Huffman style FSMs. As a practical restriction (that can always be satisfied with some state duplication), every state must also have a *unique entry point*, i.e., it must always be entered with the same value of input and output signals.

For example, examine the fragment of a burst-mode FSM in Figure 2.20.(a). Suppose that in state s_1 all input and output signals have value 0. Two bursts are enabled: $\{a^+, b^+\}$ and $\{c^+\}$. If inputs a and b rise, then the FSM goes to state s_2 and produces the output burst $\{d^+, e^+\}$. Otherwise, if input c rises, it goes to state s_3 and produces the output burst $\{e^+\}$.

We can directly translate a burst-mode FSM into an equivalent STG as shown in Figure 2.20.(b), after applying a critical race-free state assignment algorithm to it. In this example, we assume that the states were encoded using three state signals, x_0, x_1 and x_2, as: $s_1 = \overline{x_0}\,\overline{x_1}x_2$, $s_2 = x_0\overline{x_1}x_2$ and $s_3 = \overline{x_0}x_1\overline{x_2}$.

Bold signal names denote *output* signals and $\epsilon_1', \epsilon_1'', \ldots$ denote *empty* transitions. Empty transitions just act as placeholders in order for the STG to satisfy free-choice and safeness constraints.

In the STG each FSM state is represented by a place. Each place is followed by an empty transition for each input burst. Each empty transition is followed by a set of input signal transitions, followed by a set of state signal transitions, followed by a set of output signal transitions. Each output burst is then followed by another empty transition, followed by the place corresponding to the next state. In Figure 2.20.(b) place p_1 corresponds to state s_1, place p_2 to state s_2, and place p_3 to state s_3.

2.5.5 Synthesis from Formalized Timing Diagrams

The synthesis algorithm *"Suture"* due to Borriello ([14]) can in some sense be considered an STG synthesis method. In this case, the specification is a *formalized timing diagram* that is used to produce an *Event Graph*. The method is especially suited for mixed synchronous/asynchronous interfaces since it permits the construction of a complete specification from a pair of timing diagrams describing the two sides of the interface[13]. The relationship between the two timing diagrams can be inferred automatically (in part) from *data dependencies* between wires with the same names that appear on both sides (see also [112] for a similar approach using STGs for the low-level synthesis task). The Event Graph can be considered an *acyclic* STG fragment which describes the *causality relations* extracted from the timing diagrams and can be augmented with some timing constraint information.

The synthesis method itself is based on *direct translation* of each signal into an *SR* flip-flop and a set of gates computing the pre-conditions for its rising and falling transitions respectively. Synchronizers and D-latches are similarly used to implement each signal when the circuit is synchronous or when it interfaces an asynchronous input to a synchronous subsystem.

The initial circuit can be rather large, so "Suture" optimizes it with a set of *combinational* and *sequential* logic transformations, based on [16] and on pattern replacement guided by waveform analysis respectively. In the second case, for example, a reset-dominant *SR* flip-flop where the S input is always high when R rises can be replaced by the simpler circuit $S \cdot \overline{R}$.

Similar local transformations are used in order to eliminate potential hazards, add state signals if necessary, and so on, yielding a procedure that has some similarities with the synthesis methodology for pseudo-delay-insensitive circuits described in the next section. The circuit and

[13]Quite appropriately, such procedure is called *"Janus"*, from the two-faced Roman god.

delay model used by Borriello, though, are not formal, and only the good designer intuition behind "Suture" guarantees that the final result satisfies the specification and is hazard-free.

2.6 Delay-insensitive Circuits

Informally, a *delay-insensitive* circuit must operate correctly according to its specification when modeled with *unbounded wire delay*. This definition was used by Seitz ([106]) and Martin *et al.* ([23]). The first definition of delay-insensitivity, due to Molnar *et al.* ([85]), is slightly different and is based on the distinction between a *circuit* (also called "mechanism") and its *environment*. The circuit is embedded in a so-called "**foam rubber wrapper**" that can alter in all possible ways the ordering of signals traveling to and from it on different wires. The protocol of interaction between the circuit and the environment is delay insensitive if it is the same regardless of whether it is specified *inside* or *outside* the wrapper, that is if there is no **computational interference** (no transition is received when the receiver is not ready for it) or **transmission interference** (no two transitions on the same wire can ever "overrun" each other). A more formal definition of the same concept using Trace Theory, due to Udding ([116]), is reported in Section 3.4.3.

The unbounded wire delay model, ideally, would allow the most robust implementation of a given asynchronous circuit specification, since no hypothesis whatsoever is made on the magnitude and distribution of delays. Unfortunately, it can be shown that very few circuits built out of *basic gates* (*and*, *or*, *inverter*, *ex-or*, ...) operate in a delay-insensitive way. For example, Unger ([117]) showed that there is no delay-insensitive implementation of an FT with an *essential hazard*. Then Patil *et al.* ([94]) showed that in any delay-insensitive circuit built out of basic gates every gate must be in every cycle of the circuit. Finally, Martin ([77]) showed that every delay-insensitive circuit that performs non-trivial computations can be composed only of *C* and *inverter* gates.

This apparent dead end for delay-insensitive circuit design can be partially overcome by relaxing some of the above assumptions. For example:

1. Martin *et al.* (Section 2.6.1) chose to relax the hypothesis of having an unbounded delay on *every wire* of the circuit. They made the assumption that some *forks* (a **fork** is a wire that is connected to more than one gate input) in the circuit are **isochronic**, i.e., the difference between the delays of the branches is less than the minimum gate delay. Such circuits are termed **quasi-delay-insensitive**.

   Their synthesis method, described in more detail in the next section, provides as part of its output also a list of forks that must be implemented isochronically. This list of "critical" wires can be used, for example, as a guide during the transistor sizing and layout phases, forcing those wires to be as short as possible. Another phase where isochronic forks must be taken into account is during transistor sizing, because as pointed out by [118] different logic thresholds may invalidate the isochronous hypothesis even without considering wire delays.

2. Ebergen ([40]), on the other hand, lifted the restriction to basic gates, and showed that by interconnecting a small set of primitive components, each designed using a bounded delay model, delay-insensitive operation for a large class of behaviors can be achieved. Both the specification formalism and the syntax-directed translation methodology are somewhat similar to Martin's approach. The isochronic fork was still required, even though Ebergen gave a more complex synthesis procedure that he conjectured could be done without isochronic forks.

3. A similar approach was advocated by Seitz ([106]), in which all forks within a geographically limited region of the chip were supposed to be isochronic, and by Rosenberger *et al.* ([101]), where the circuit was decomposed into modules, each designed assuming bounds on delays (see Section 2.3.6) but whose external wire delays could be unbounded.

4. Brunvand and Sproull ([17]) also proposed to directly translate a concurrent specification (similar to Martin's and Ebergen's) into a delay-insensitive interconnection of complex modules. They used local transformations to optimize the resulting circuit.

The next section describes in more detail the first of the methods listed above, in order to show its commonalities with STG-based methods.

2.6.1 Synthesis of Quasi-delay-insensitive Circuits

Martin and his group developed the method described in ([23, 78, 76, 79] and [24]) for synthesizing quasi-delay-insensitive circuits from a formal specification (adapted from Hoare's **Communicating Sequential Processes**, CSP, [51]). The methodology was used to design various real circuits, including an asynchronous microprocessor ([80]). The data computation part is handled with *self-synchronizing* codes (namely *dual-rail*). Here we briefly outline the specification formalism and the control part synthesis steps.

A CSP process operates on **variables** with basic type *Boolean* and composite types array and record (integer and floating point numbers are pre-defined records, with built-in manipulation functions and operators). The main statement types are:

1. **assignment**, that computes an expression and assigns its value to a variable, e.g., $m[a] := b+c$ assigns the sum of b and c to the a-th element of array m.

2. **composition** operators, which compose two statements into one:

   (a) **sequential composition**, which is denoted by ";" and forces a statement execution to follow another one. $a := b; b := a$ results in a and b having the initial value of b.

   (b) **concurrent composition**, which is denoted by "||" and does not impose any ordering on the components.

   (c) **coincident composition**, which is denoted by "•" and synchronizes communication commands (see below).

3. **control** statements:

   (a) **selection**, which executes one among a set of statements depending on which Boolean **guard** is true. It is denoted by "$[G_{11} \rightarrow S_{11} | \ldots | G_{1n} \rightarrow S_{1n}]$" (if more than one guard is true, one is non-deterministically selected, while if no guard is true, then the execution is suspended until one becomes true). For example, $[x \rightarrow a := b | x' \rightarrow a := c]$ models a multiplexer with selector x, inputs b and c and output a.

   (b) **repetition**, which repeatedly selects one statement, and exits when no guard is true. It is denoted by "$*[G_1 \rightarrow S_1 | \ldots | G_n \rightarrow S_n]$". For example, $*[a < n \rightarrow a := a + 1]$ increments a while it is less than n.

   By convention "$*[S]$" repeats statement S forever.

CSP processes run concurrently and communicate through channels. A **channel** is a one-place unidirectional buffer which can be read, written and checked for pending operations. The **read** operation is denoted by $C?x$, where C is a channel and x is a variable. The **write** operation is denoted by $C!y$, where C is a channel and y is a variable. A pair of **communication** actions $C?x$ and $C!y$ executed on a channel by two processes results in the *synchronization* of the two processes (i.e., each *waits* for the other to be ready) and the *assignment* of the value of y to x.

In order to avoid blocking, a process may use a **probe** on a channel C, denoted by $\overline{C}$. The probe evaluates to **true** if communication on C is possible, i.e., the process at the other end is ready for it. Hence the guarded command $\overline{C} \rightarrow C?x$ is guaranteed not to block (except, of course, for the unbounded but finite delay intrinsic in every CSP action). In order to be consistent with Martin's notation, "—" in this section denotes a probe, while "$\vee$", "$\wedge$", and "$'$" denote the logic operators.

The CSP program is then transformed into a set of production rules that can be, under conditions detailed below, directly interpreted as *logic gates*. A **production rule** is a *guarded assignment* of a logic value to a variable, i.e., a pair $G \rightarrow z^+$, or $G \rightarrow z^-$. We adopt the usual convention that z^+ means $z := 1$, z^- means $z := 0$ and z^* means either of the above. A rule is said to **fire** when the corresponding variable assignment occurs. A pair of rules
$G_1 \rightarrow z^+$,
$G_2 \rightarrow z^+$,
is equivalent to
$G_1 \vee G_2 \rightarrow z^+$,
hence we will always assume that there is exactly one setting and one resetting rule for each variable. Two rules $G_1 \rightarrow z^+$ and $G_2 \rightarrow z^-$, affecting a variable in opposite ways, are **complementary**, and their guards must be disjoint.

Obviously each rule can be interpreted as describing an *excitation function* for the assigned variable, hence from a set of rules:

$$\begin{array}{ccc} S_1 & \rightarrow & z_1^+ \\ R_1 & \rightarrow & z_1^- \\ \dots & \dots & \dots \\ S_n & \rightarrow & z_n^+ \\ R_n & \rightarrow & z_n^- \end{array}$$

we can derive a **circuit implementation**, as $z_i := (z_i \wedge R_i') \vee S_i$, and an associated State Transition Diagram (as in Section 2.5). For example, the pair of rules: $ab \rightarrow c^+$, $a'b' \rightarrow c^-$ describes the behavior of a 2-input C gate.

A rule is **stable** if at any point in the computation either the guard is false, or it remains true until the rule fires. Rule stability is equivalent to STD semi-modularity, because in a set of stable rules no enabled variable is ever disabled without changing value.

The process of CSP translation into production rules is divided into various steps (illustrated by an example below):

1. **process decomposition**: complex statements are decomposed using rewriting rules, which transform them into *calls* to simpler processes. At the end, each process either performs a (possibly infinite) sequence of *communication* actions, or it is an infinite repetition of a *selection*.

2. **handshaking expansion**: each channel is implemented with a *four-phase handshake*. This amounts to rewriting read and write actions with assignments and evaluations of the handshaking variables, and probe values with evaluations of the appropriate handshaking variable.

   The sequence of assignments and evaluations can then be *re-shuffled* to maximize parallelism without altering the flow of the original program.

3. **production rule expansion**: each sequence of evaluations and assignments is translated into a set of production rules such that the original behavior is maintained and each rule is *stable* (i.e., semi-modularity is guaranteed). The execution order for the rules implied by the CSP program is stored with each rule when it is generated from a program statement.

   Stability is ensured using the concept of **acknowledgment**. Given two production rules $r_1 : G_1 \rightarrow z_1^*$ and $r_2 : G_2 \rightarrow z_2^*$ such that r_1 immediately precedes r_2 in the execution order, r_2 is an *acknowledgment* of r_1 if:

   - G_2 implies that z_1^* has occurred (so r_2 cannot fire unless r_1 has fired) and
   - G_1 cannot become false unless z_2^* fires.

   Acknowledgment can be implemented by conditioning G_2 to z_1 (complemented, if necessary), and Martin *et al.* showed that it is *necessary and sufficient* to implement stability.

   Maintaining the original CSP behavior for a set of rules sometimes requires the introduction of *state variables* so that the *execution order* of rules is preserved. For each pair of rules which could fire out of order, a variable set by the first rule is tested by the second rule (and then reset).

4. **operator reduction** and **symmetrization** of production rules are finally applied to map the production rules onto a set of commonly used complex logic gates. This step includes a set of optimization operations which closely resemble those of classical logic synthesis.

Let us examine an example (from [79]) to clarify the methodology. The specification is a one-place buffer, described by the CSP process:

$$*[L?x;\, R!x]$$

with two channels, L and R, and one variable, x.

One possible handshaking expansion (neglecting the data path, which in this case is just a wire implementing variable x) uses a request input li and an acknowledge output variable lo for input channel L, and a request output ro and an acknowledge input variable ri for the output channel R, resulting in the following pair of four-phase handshakes:

$$*[[li'];\ lo^+;\ [li];\ lo^-;\ [ri];\ ro^+;\ [ri'];\ ro^-]$$

where elementary action $[li']$ terminates when li has value 0. This is syntactically equivalent to the set of rules:

$$\begin{aligned} li' &\rightarrow lo^+ \\ li &\rightarrow lo^- \\ ri &\rightarrow ro^+ \\ ri' &\rightarrow ro^- \end{aligned}$$

with initial values $li = 1$, $lo = 0$, $ri = 0$, $ro = 0$.

Now we notice that variables li and lo change value twice without any other variable changing (observe the close similarity with the concept of *normal* STG). This means that rules
$li \rightarrow lo^-$
and
$ri' \rightarrow ro^-$
can immediately fire, without the second rule waiting for a complete cycle on li and lo.

So a state *variable* x is introduced, yielding:

$$*[[li'];\ lo^+;\ x^+;\ [x];\ [li];\ lo^-;\ [ri];\ ro^+;\ x^-;\ [x'];\ [ri'];\ ro^-]$$

that is syntactically equivalent to the set of rules:

$$\begin{aligned} li'\,ro'\,x' &\rightarrow lo^+ \\ lo &\rightarrow x^+ \end{aligned}$$

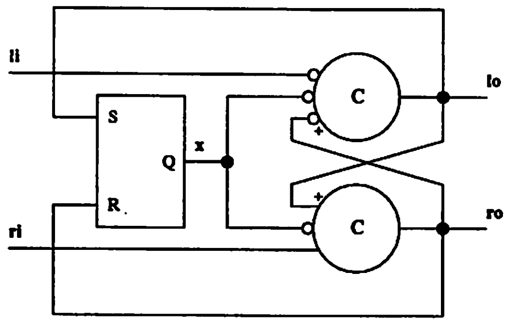


Figure 2.21: The implementation of the one-place buffer

$$\begin{aligned} x\,li &\rightarrow lo^- \\ x\,lo'\,ri &\rightarrow ro^+ \\ ro &\rightarrow x^- \\ x'\,ri' &\rightarrow ro^- \end{aligned}$$

This set of rules now can be implemented by the circuit shown in Figure 2.21 (where inputs marked by "+" influence only the rising transition of the generalized *C* gate).

In some sense the methodology described in this section can be considered as an alternative to that of Varshavsky *et al.* (Section 2.5.2), as the definition of stable rule closely matches the definition of semi-modular circuits. The two are also similar in that both assume some level of custom design for the library, one using generalized *C* gates, and one using arbitrary *and-or-not* gates. Martin's method, though, uses a higher-level specification formalism than Varshavsky's STD (or even than the more recent, signal transition-based, Change Diagram [125]) Martin's approach, obviously, must solve a more difficult synthesis problem that is harder to automate efficiently.

2.7 Hazard Analysis in Asynchronous Circuits

In this section, we will examine a completely different problem: the *analysis* that verifies whether the designer (or the automated synthesis algorithm) has performed his task in a satisfactory way, i.e., without introducing errors or *hazards*. The problem is very complex, and hence a hierarchy of methods exist which provide more and more accuracy at a greater and greater cost.

A logic circuit can be verified with respect to its specification in order to check the correctness of the design procedure both in the case of manual design (where mistakes are obviously common) and in the case of automated design (where bugs in the algorithm implementation can produce incorrect circuits).

Various techniques are available for this purpose, and they differ both in the accuracy of the underlying delay model and in the possibility to reject circuits that would work correctly using a more precise criterion. Here we will only briefly list the methods that have most direct relevance for our work (see [19, 20] for a more complete description):

1. Three-valued simulation ([42], [105]) provides the fastest known algorithm for asynchronous circuit verification, but its usefulness is limited to static and steady-state hazard checking in Fundamental Mode using the pure unbounded wire (or bounded wire) delay model (Section 2.7.1).

2. Muller's analysis ([88], [62]) and Trace Theory ([39]) can be used for hazard checking in Input-Output Mode using the inertial unbounded gate delay model (Section 2.7.2).

3. Chaos-mode discrete-time verification ([22, 21]) and timed automata ([72], [38]) perform circuit verification in Input-Output Mode using the more realistic inertial bounded wire delay model. Unfortunately they can be pessimistic in some cases (Section 2.7.3) or computationally too expensive (Section 2.7.4).

If the circuit is operated in Fundamental Mode or Huffman Mode, rather than Input-Output Mode, then it is possible to use simplified approaches to exact bounded wire delay verification. For example [55] can be used for Fundamental Mode circuits or [37] can be used for Huffman Mode circuits.

2.7.1 Unbounded Delay in Fundamental Mode

The objective of **three-valued simulation** ([42], [105]) is to provide a *fast, pessimistic* (i.e., no incorrect circuit can be accepted, but some correct ones may be rejected) method to check:

- for potential *static hazards* in a *combinational* logic circuit,
- for potential *steady-state hazards* in a *sequential* logic circuit.

Both checks are performed with pure *unbounded wire delay* and *Multiple Input Change* in *Fundamental Mode* (the same considerations about unbounded delays and Fundamental Mode as in Section 2.3.1 apply also here).

The idea of three-valued simulation is to proceed in two steps:

1. Classify all input signals as constant or changing under a specific set of input transitions and propagate the transitions as far as possible in the circuit.

2. Let all input signals stabilize (recall that we are operating in Fundamental Mode), and propagate the constant values as far as possible in the circuit.

Static hazard analysis in combinational circuits requires only step 1, because if a change reaches an output that is supposed to remain constant, then a static hazard condition exists under some wire delay assignment for that circuit. Step 2 verifies if a sequential circuit can have steady-state hazards, because if some state (feedback) signal does not settle to a stable state, then either an oscillation or a critical race can occur for the corresponding transition under some wire delay assignment for that circuit.

In order to describe more precisely the simulation algorithm we need, as usual, some definitions (taken from [19]). Let "$-$" denote a "changing" or "undefined" value. A partial order on the set of ternary values $\{0, 1, -\}$ is given by $0 \sqsubset -$ and $1 \sqsubset -$ (all other pairs are incomparable). This induces a partial order on the set of vectors of m elements of $\{0, 1, -\}$ (let s_i denote the i-th element of vector $s \in \{0, 1, -\}^m$):

$$s \sqsubseteq t \text{ if } s_i \sqsubseteq t_i \text{ for all } i$$

An element u of a partial order is a **least upper bound** (l.u.b.) of two elements s, t, denoted $u = s \sqcup t$, if:

- $u \sqsupseteq s$ and $u \sqsupseteq t$ and

- if there exists u' such that $u' \sqsupseteq s$ and $u' \sqsupseteq t$, then $u' \sqsupseteq u$.

So, for example, $0 \sqcup 1 = -$, $0 \sqcup - = -$, $1 \sqcup 1 = 1$ and so on.

Similarly we can define the **greatest lower bound** (g.l.b.) l of a pair of elements s, t of a partial order, denoted $l = s \sqcap t$, by replacing $\sqsupseteq$ with $\sqsubseteq$ in the previous definition.

Ternary simulation is done using the ternary extension of logic functions. Given a (completely specified) logic function $f : \{0, 1\}^m \rightarrow \{0, 1\}$, its **ternary extension** $\mathbf{f} : \{0, 1, -\}^m \rightarrow \{0, 1, -\}$ is (let $\mathbf{t}$ denote a ternary vector):

$$\mathbf{f}(\mathbf{t}) = \bigsqcup_{t \in \{0,1\}^m \text{ s.t. } t \sqsubseteq \mathbf{t}} f(t)$$

For example, let f be a two-input *or* function. Then

- $f(00) = 0$,
- $f(01) = f(10) = f(11) = f(1-) = f(-1) = 1$, and
- $f(0-) = f(-0) = f(--) = -$

Note that $\{0, 1, -\}$ with the ternary extensions of *and*, *or* and *not* is *not a Boolean algebra*, because $s + \overline{s} \neq 1$ when $s = -$.

The ternary simulation of a logic circuit is composed, as outlined above, of two steps, described in Procedure 2.7.1 and 2.7.2, taking as input:

- a Huffman circuit, with primary input signals x, present state signals y, next state signals Y and output signals z.
- an initial *stable total state* $x'y'$ (stability implies that $Y' = y'$ if we compute Y' from y' and x' by ternary simulation of each gate in the combinational logic part of the circuit)
- a final *input state* x''.

Procedure 2.7.1

```
1. let $y = y'$
2. let $x = x' \sqcup x''$ (i.e., all changing inputs become undefined)
3. repeat
    (a) compute $Y$ from $y$ and $x$ by ternary simulation of each gate in the combinational logic
        part of the circuit
    (b) let $y = Y$
   until convergence (i.e., $y = Y$ immediately after step 3a)
```

Procedure 2.7.2

```
1. let $y$ be computed as in Procedure 2.7.1
2. let $x = x''$ (i.e., all changing inputs stabilize to their final value)
3. repeat
```

```
    (a) compute $Y$ from $y$ and $x$ by ternary simulation of each gate in the combinational logic part of the circuit
    (b) let $y = Y$
until convergence (i.e., $y = Y$ immediately after step 3a)
```

As shown in [19], both Procedure 2.7.1 and Procedure 2.7.2 converge in at most m iterations, where m is the number of state signals, because at least one signal must change at each iteration. Moreover:

- Y (and hence z) is completely defined at the end of Procedure 2.7.2 if and only if the circuit has no critical race and no unbounded oscillation, i.e., it stabilizes to a unique state.
- an output whose initial and final values are the same has no static hazard if and only if its value at the end of Procedure 2.7.1 is defined (i.e., no component has value $-$).

Ternary simulation is *extremely efficient*, because simulating the combinational part of the circuit has a worst-case running time that is also polynomial in the size of the circuit (it can be done by traversing it in topological order from the inputs). Unfortunately, it can give a very *pessimistic* view of the circuit behavior, because it does not use bounds on the delays. We will see in Chapter 5 that ternary simulation can be used to detect *potential hazards* in a circuit, while timing analysis can use delay bounds to verify which hazards can actually show up within the context of Signal Transition Graph-based synthesis.

On a different line of research, Seger ([105, 20]) proposed using ternary simulation also in the bounded delay case, with a worst case time bound that is polynomial in the size of the circuit. The algorithm is based on the idea that in the worst case each changing signal assumes the undefined value at the earliest possible time and a defined value (0 or 1) at the latest possible time. This algorithm still gives a pessimistic result, but it is more accurate than Procedures 2.7.1 and 2.7.2.

This limitation of ternary simulation to *static* hazard analysis was recently lifted by [68], which extended three-valued logic to a nine-valued logic:

- constant 1 and 0,
- rising and falling transition,
- static and dynamic hazards (four in total),

- undefined value.

The partial order is defined as follows: the undefined value is greater than all others, static hazards are greater than the corresponding constant values, and dynamic hazards are greater than the corresponding transitions. The logic functions can be extended to this multi-valued logic and nine-valued simulation can be used to check for *dynamic* and *static hazards* in Fundamental Mode.

2.7.2 Unbounded Delay in Input-Output Mode

The Muller model was originally devised as an asynchronous circuit *hazard analysis* method, where any deviation from *semi-modular* behavior is considered as a *hazard*. The reasons for this interpretation will be explained more precisely in Section 3.2.5. Here we will just note that, intuitively, if an enabled signal is disabled without changing value, then a hazard can occur in the *real* circuit for that signal. This is true even though the Muller model *per se* would not predict any "spurious pulse" on it, and is due to the fact that the *inertial delay* model used by Muller may be overly *optimistic*, say in the presence of transmission-line delays.

Given these premises, verification with the Muller model amounts to building the STD of the given circuit, and then checking for *semi-modularity*. Unfortunately, the STD has, in the worst case, an exponential size in the number of gates, hence the general technique is very expensive to apply in practice.

In order to describe a semi-modular circuit with a polynomial size model, Kishinevsky *et al.* introduced the *Change Diagram* (CD, see also Section 3.5 and [125, 60, 62]). They showed that the Change Diagram has the same expressive power as semi-modular STDs, i.e., an STD is semi-modular if and only if it has a corresponding Change Diagram.

The Change Diagram, discussed in more detail in Section 3.5, is similar to the STG, but it also allows a transition to be enabled when *at least one* of its predecessors is marked, rather than only when *all of them* are marked, as for standard PNs.

A particular sub-class of semi-modular circuits, the *distributive* circuits, can be analyzed with an algorithm that is *polynomial* in the number of gates ([61]). Informally, a semi-modular circuit is distributive if each excitation region of each gate has a *single* entry point that corresponds to the logical *and* of a set of pre-conditions for the corresponding transitions.

The circuit in Figure 2.18 is distributive, while the circuit in Figure 2.17 is not distributive, for two reasons. First, the circuit of Figure 2.17 it is not semi-modular, and second, the excitation region S_z (with states 10*0*, 0*10* and 110*) is entered through two *concurrent* transitions, x^+

and y^+. So z^+ can be enabled by *either* x^+ *or* y^+, whichever fires first, thus describing an *OR-type* causality.

Using the CD-based analysis method, a CD equivalent to a distributive circuit can be derived in polynomial time from the circuit. The construction fails if the circuit contains a hazard or if it contains semi-modular but not distributive behavior. As such it can be considered a partial, efficient, solution to the hazard analysis problem in speed-independent circuits[14].

An interesting alternative solution to the hazard analysis problem with the unbounded gate delay problem was introduced by Dill ([39]). In this approach, which is based on Trace Theory ([100, 119, 120], see also Section 3.1.4), each gate in the circuit is modeled as a *Büchi Finite Automaton*. The FA accepts the set of hazard-free input-output sequences (or *successful traces*) of the gate.

For example, a *nor* gate with two inputs x_1 and x_2 and output z is represented by the FA shown in Figure 2.22 (from [39]). Each state is labeled with a vector of signal values, and each edge is labeled with a transition of an input or output signal. State H has an implicit self-loop under any input event.

The accepting set F in this case contains all the states except H, so any run that does not enter H after a finite number of events is accepting. Otherwise only a finite number of accepting states would appear in it, and the Büchi acceptance condition would not be satisfied. Hence accepting runs correspond to *hazard-free uses* of the gate, where no enabled output transition is ever disabled. For example, output transition z^+ is enabled in state 000. If x_1 or x_2 rises, the transition on z is disabled, and the run is not accepting because it gets trapped in the H state.

The interconnection of a set of gates into a circuit corresponds to the construction of the *product* of their FAs. Whenever a transition belongs to the input states of more than one FA, one of them is assumed to *cause* the transition on its output, and the others are assumed to *observe* it on their inputs. The notion of zero wire delays matches exactly the definition of product given in Section 2.2.3, where an event that is common to the input states of a set of FAs causes *all of them* to have a simultaneous transition. So the verification of semi-modularity amounts to checking for reachability of a state having H as a component in the product FA. This reachability implies that an enabled transition is disabled for the corresponding gate.

Note that all the analysis methods described in this section can deal also with the *unbounded wire delay* model, by inserting a non-inverting buffer gate on each wire (or on selected

[14]This verification method is most likely connected with recent results on polynomial time verification methods for particular subclasses of Petri nets reported in [11].

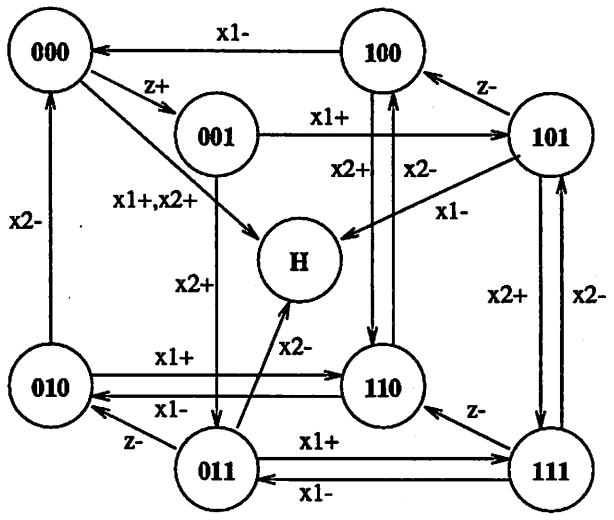


Figure 2.22: The Finite Automaton of a 2-input *nor* gate

wires whose delay is known to be long compared with gate delays, due to, e.g. layout considerations). These methods have higher complexity than *three-valued simulation*, but they can handle verification in the more general Input-Output Mode, rather than being limited to Fundamental Mode.

The *unbounded gate delay* model, though, is rather pessimistic. Circuits that are rejected as hazardous by a verification procedure based on it may actually be correct, due to the bounds on *actual circuit delays*. So other verification mechanisms are required in order to precisely assess whether an asynchronous circuit, designed with delay bounds in mind, actually operates according to the specification, as shown in the next section.

2.7.3 Bounded Delay in Input-Output Mode with Binary Chaos Model

Burch ([21]) recently proposed extending the approach of [39] outlined in the previous section to verify asynchronous circuits modeled with *bounded delay*. The introduction of a periodic signal, or **clock**, ϕ, to describe the *interval of occurrence* of events and the use of a sound underlying gate model allows the derivation of a provably *pessimistic* approximation of the real behavior of the circuit. So the model uses *discrete delay*, but is a pessimistic approximation of the (exact) *continuous delay* model.

Also in this case, every gate is described by an FA that has:

1. a pair of *"stable" states*, where the output of the gate is the same as its logic function (as computed from the output values of the other gates in the circuit and the primary inputs).

2. a set of *"excited" states*, where the output is different from its logic function. These states are used to "count" the time from the last stable condition.

3. a set of *"chaos" states*, reached when some *input* changes while in an excited state, resulting in a potential hazard.

The normal behavior of the gate, under some input change that causes it to go into an excited state, is to move along a sequence of excited states for a number of clock ticks equal to the *lower bound* on its delay. Then it non-deterministically can either make a transition to the stable state with the new output, or keep counting until the *upper bound* on its delay has elapsed, when it is forced to stabilize to the new output value.

The "hazardous" behavior occurs when some input changes while the gate is excited, forcing it to change back to the original value (input changes that do not affect the expected output value, on the other hand, are just ignored by the verification procedure). In this case the gate, rather than going to a "failure" state, as in the unbounded delay case, moves to a chaos state. From this state, the output non-deterministically changes at every clock tick (to allow the corresponding hazard to affect other gates, if this is the case) until the input has remained stable for a time equal to the *upper bound* on the gate delay.

The actual verification procedure is not carried out like a "simulation", as informally described above, but rather by computing the product FA of the various FAs representing each gate and the environment behavior. Verifying the hazard-freeness of the implementation amounts to ensuring that a transition of a *primary output* signal that is not present in the specification (another FA) cannot occur in the implementation. That is, internal gates are allowed to go into chaos states as long as this does not propagate to any output signal. The specification does not prescribe the behavior of *internal* signals, so their abnormal transitions do not constitute hazards *per se*.

Let us examine the circuit described in Figure 2.24 (taken from [22]), where the numbers above each gate represent an upper and a lower bound on its delay (using the gate delay model). Its Signal Transition Graph specification (taken from [27]) appears in Figure 2.23, where L_1^+ and L_2^+ represent two *distinct* rising transitions of the same signal L. The environment delay is assumed to be finite and positive, but no other assumption on its behavior is made except that it follows the STG specification.

The circuit was synthesized using the algorithms described in Chapter 5, and it was verified *before* and *after* hazard elimination. It turns out that the verification before hazard elimination actually found one of the hazards used by the hazard elimination procedure to pad delays. Moreover,

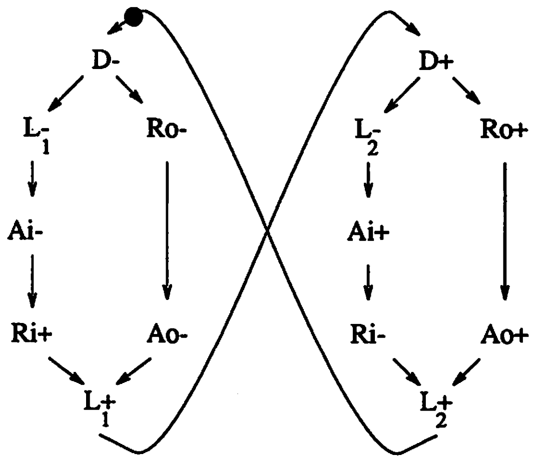


Figure 2.23: A Signal Transition Graph specification (from [27])

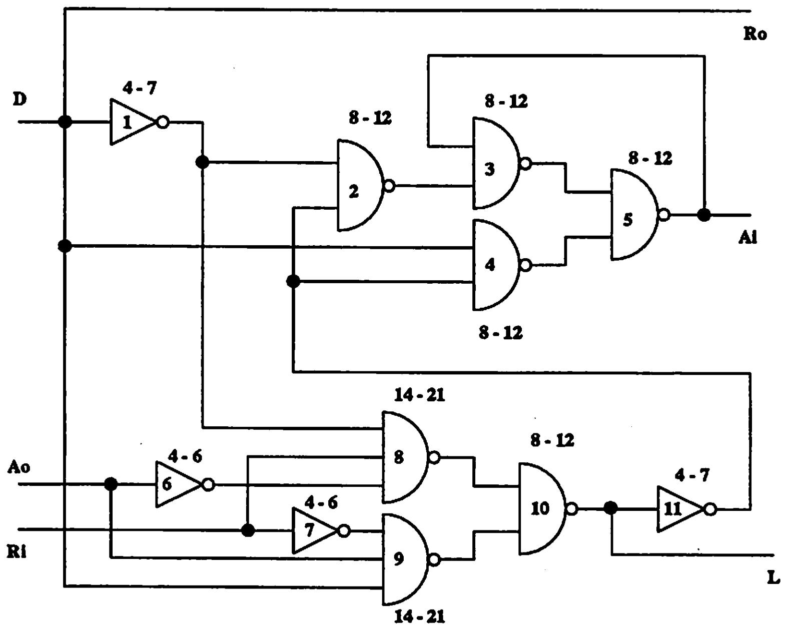


Figure 2.24: A circuit implementation of the STG of Figure 2.23

time	0	4	14	22	26	34	42	50	54	64	72	76	77	78	81	86
events	D^- Ro^- Ao^-	6^+ 1^+	9^+	L^-	11^+	2^-	3^+	Ai^- Ri^+	7^-	8^-	L^+ D^+	11^-	1^-	4^-	4^+	Ai^+

Figure 2.25: A hazardous sequence of events in the circuit of Figure 2.24

the circuit after hazard elimination was verified to be hazard-free, as expected.

The initial state, corresponding to the initial marking shown on the STG, is $Ri = 0$, $Ao = 1$, $D = 1$, $Ro = 1$, $Ai = 1$, $L = 1$. The complete sequence of events (*trace*) leading to the hazard is described in Figure 2.25, and can be summarized as follows.

0 D^- (the first event enabled according to the specification) fires, immediately followed by Ro^- (since Ro is just a wire) and Ao^- (since the environment can instantaneously react to Ro^-). This causes gates 1, 6 and 9 to move to the first unstable state. The first four state transitions (occurring at each clock tick) of 1 and 6 are deterministic to the next unstable state.

4 Both 1 and 6 can have a non-deterministic transition to the stable state and change their output value. In this particular sequence of events which leads to the hazard, they both change *immediately*. However, the verification process uses non-determinism and actually checks for *all possible transition times* between the upper and lower bound. No gate goes to an unstable state due to the change at the output of 1 and 6. Gate 2, for example, has the other input at 0 (due to gate 11). Gate 9, with a minimum delay of 14 time units, keeps counting.

14 Gate 9 now *can* have an output transition, going to the stable state with output value 0, and in this particular sequence it does so. This causes gate 10 to "start counting" (i.e., go to the first unstable state), because now gate 8 has output value 1 but gate 9 has output value 0.

22 Gate 10 changes value.

...

72 L rises and, as specified by the STG, also D rises. This causes 1, 4 and 11 to start counting. In particular, gate 4 has both inputs at 1, so its new output should be 0.

76 Gate 11 goes to the stable state, with output 0 (while in this particular trace gate 1 keeps counting until it reaches the maximum delay).

So gate 4 goes into *chaos* because it has not had time to change its output and one of its inputs goes back to 0. In a real circuit, this may or may not cause a hazard, hence it is a behavior that is considered dangerous and must be avoided. In order to test for hazard propagation to a circuit output (in this case Ai) the chaos model will force the output of gate 4 to have a non-deterministic transition at *every time step*, until its maximum delay has expired and we can be sure that it does not produce hazardous transitions on its output any more. In this case, the other input to gate 5 (gate 3) has value 1 for 3 more time steps. The hazard *can* propagate to the output, as shown by the remaining events.

77 Gate 1 goes to the stable state.

78 Gate 4 changes to 0 (first transition of the hazard) due to D^+ at time 72. Gate 5 starts counting, because gate 3 has value 1.

81 Gate 4 changes back to 1 (second transition of the hazard), due to 11^- at time 76.

86 Gate 5 goes to the stable state with output 1, and the hazard has propagated to the output Ai, where the STG would not allow any transition for Ai in this condition, i.e., after L^+ and D^+ but before L^-.

The chaos mode verifier has the advantage of being able to handle circuits of realistic size in a reasonable amount of CPU time while using a delay model that is not overly pessimistic. It has only one disadvantage: being pessimistic, every detected hazard must (in principle) be re-checked by simulation. On the other hand, a positive result ensures that the circuit will operate according to the specification as long as the delay bounds are met. The only known methodology for exact (i.e., neither pessimistic nor optimistic) hazard analysis with the bounded delay model is based on *timed automata*, and is the topic of the next section.

2.7.4 Bounded Delay in Input-Output Mode with Timed Automata

The most precise verification methodology for asynchronous circuits, called the *timed* FA method, is based on the idea of augmenting Büchi FAs with *timers*, much in the same spirit as in the approach described in the previous section. The methodology was proposed at the same time by [38] (whose notation and terminology we adopt here) and [72]. Time is treated as a *continuous* variable without pessimistic approximations. The result is an *exact* verification method, within the limits of the *bounded wire delay* model (the method can be tailored both for *inertial* and *pure* delays).

A **timer** is an object that has an **integer upper bound** and an **integer lower bound**. It can be either **active** or **inactive**. A timer becomes active as the result of a **set** event, and its **value** is set to a *real* number between its bounds. It becomes inactive as the result of an **expire** event, when its value reaches zero or when it is forced to expire by another event (similar to a *transition disabling* event). Hence the time that elapses between any *set* event and the corresponding *expire* event is greater than the lower bound and smaller than the upper bound associated with the timer.

In this model, one timer is associated with every *signal*. Every *non-forced* occurrence of an *expire* event causes a transition of the corresponding signal. Consequently every gate with that signal as input instantaneously *sets* or *forces the expiration* of its own timer, according to whether an output transition is *enabled* or *disabled* respectively. This means that logic evaluation is considered *instantaneous*, while circuit delays are modeled by the time between set and expire events.

Note that in this case an enabled transition that is disabled without firing does not constitute a hazard, because here *exact* bounds are assumed to be known on the inertial delay magnitude. Of course, using the pure delay model no timer is ever forced to expire (disabled), and the model must be modified to take into account a *sequence* of events that are enabled for each gate.

A **timed trace, consistent** with a set of timers, is a sequence of pairs of *events* and *times* where:

- an event can be:
  - a timer event (i.e., a *set* or *expire* event of some specific timer), or
  - the change of value of some signal in the associated circuit (i.e., a state transition in the corresponding Büchi FA, obtained from the circuit as in Section 2.7.2).
- the sequence of times (real numbers) is monotonically non-decreasing from 0.
- each pair of set/expire events associated with a timer satisfies the corresponding bounds.
- the time of each expire event and of the transition, set and expire events that are caused by it coincide.

A **timer region** associated with a set of timers is a *set of vectors of timer values*, where each vector must be compatible with the timer bounds.

A **timer region** FA associated with a set of timers T is a Büchi FA where:

- the set of states is the set of timer regions associated with all subsets of T.
- the set of inputs is the set of all subsets of the set of events for the timers in T.

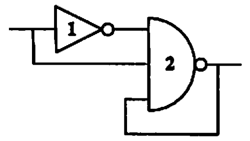
Note that the set of states is apparently *infinite*. [38] and [72] proved that it can be reduced to an *equivalent finite* form, using a finite set of equivalence classes. Unfortunately, the number of such classes is so large that the approach seems practically infeasible for all but the smallest circuits. More precisely, they showed that:

1. Every timer region can be represented as a square matrix d in which each timer is associated with a row and a column, and each entry d_{ij} contains the *upper bound on the difference* between the corresponding timer values v_i and v_j.
2. Every equivalence class has a *canonical representative* that can be found by computing the shortest path between all pairs of nodes in the directed graph represented by the matrix.
3. Given a finite set of timers, there is a *finite set* of canonical timer regions (using the fact that timer bounds are *integers*).

Row 0 and column 0 of matrix d represent a fictitious timer that always has value 0, and are used to store the upper and lower bound on the *value* of each timer. So, for example, d_{20} is the upper bound on timer 2 (upper bound on $v_2 - 0$), d_{02} is the negative of the lower bound on timer 2 (upper bound on $0 - v_2$), d_{22} is the upper bound on $v_2 - v_2$ (obviously always zero), d_{12} is the upper bound on $v_2 - v_1$ and so on.

The next state function of this FA is defined using the timer bound matrix. At each step, for each subset E of active timers (set but not expired):

1. The row corresponding to each expiring timer is set to the minimum between its old value and zero (except for entry d_{j0} for each $j \in E$). If the directed graph described by the matrix has a negative cycle (as detected by the shortest path canonization algorithm), then this set of timers cannot expire together when the FA is in this state due to the bounds on their values, and the new matrix is discarded (this corresponds to an invalid set of events).
2. The non-expiring timers are decremented by the value of the expiring timers, i.e., for each timer i, for each expiring timer $j \in E$:
   (a) d_{ij} is copied to d_{i0} (i.e., the upper bound on the difference between i and j becomes the new upper bound on i).



nand	Min	Max
Rise	2	3
Fall	1	2

inverter	Min	Max
Rise	4	5
Fall	4	5

Figure 2.26: A circuit modeled with bounded delays (from [72])

(b) d_{ji} is copied to d_{0i} (i.e., the upper bound on the difference between j and i becomes the negative of the new lower bound on i).

3. The rows corresponding to expired timers are deleted.

4. A set of new timers is non-deterministically chosen to be set. The relevant rows and columns are added to the matrix, and the matrix is canonized by replacing each d_{ij} with the shortest path between d_i and d_j.

The **timed** FA then is simply the product FA of:

- the Büchi FA associated with the circuit, with the appropriate set/expire events occurring every time a signal changes value, and
- the timer region FA describing the delay bounds of the circuit, as outlined above.

Note that computing the product FA corresponds to choosing the newly set timers at each step 4 above according to the circuit function, thus removing the "excessive non-determinism" of the timer region FA.

The authors then prove that the timed FA accepts *exactly* the set of timed event traces compatible with the circuit function and the given delay bounds. This FA can now be checked with a reachability analysis as in Section 2.7.3 for the presence of hazards. Note that also in this case bounded wire delays can be modeled using non-inverting buffers with non-zero delay.

An example will clarify the methodology. The example is taken from [72], but we will describe it using the notation of [38] for consistency. The primary input of Figure 2.26 rises from 0 to 1 at time 0, and the *nand* and the *inverter* are both initially at 1. The circuit is modeled using inertial bounded gate delays.

The FA corresponding to the circuit of Figure 2.26 modeled with unbounded gate delays appears in Figure 2.27. The label of each state describes the values for the primary input, the *inverter* and the *nand* gates respectively. Edge labels are omitted for the sake of simplicity, as they can be inferred from the state labels. For example, the edge from 011 to 111 is labeled with the

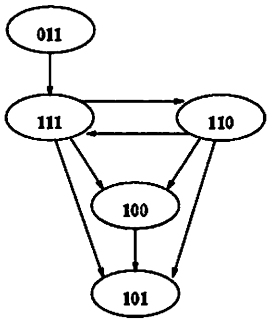


Figure 2.27: The Finite Automaton corresponding to Figure 2.26

rising transition for the primary input. Note that some edges, such as 111 → 100, correspond to the simultaneous transition of *more than one signal.*

The unbounded gate delay model, as shown in Figure 2.27, predicts an *unbounded oscillation*, corresponding to the case when the *inverter* has a very long delay, holding two inputs of the *nand* gate at 1 for that duration. This is obviously a rather pessimistic view, and the objective of the timed FA is to take into account that the *inverter* delay is longer than the *nand* delay, but that *bounds* on both are known.

The timed FA uses two timers (timer 1 associated with the *inverter* and timer 2 associated with the *nand*) and is shown in Figure 2.28. Note the difference between *timer bounds* (fixed, given a circuit) and *timer value bounds*, which are represented in the matrix d, coincide with the timer bounds when the timer is *set*, and are decremented as time passes (as witnessed by some expiring timer).

s_0 : the initial rising transition on the primary input, $s_0 \rightarrow s_1$, sets the timers associated with both gates. So timer 1 is set to a real value $4 \leq v_1 \leq 5$, while timer 2 is set to a real value $1 \leq v_2 \leq 2$ (the bounds for the *falling* transition of the *nand*). So we obtain the (canonical) matrix in state s_1 of Figure 2.28. For example, d_{12} is an upper bound on $v_1 - v_2$, so we can compute it by taking the maximum value for v_1 (i.e., d_{10}) and the minimum value for v_2 (i.e., $-d_{02}$), that is $d_{12} = d_{10} - (-d_{02}) = 4$.

s_1 : timer 2 expires (i.e., the *nand* output goes to 0), leading to state s_2. The new matrix is obtained by:

1. setting each element in row 2 (except d_{20}) to the minimum between its old value and 0, and checking for negative cycles.

2. copying row 2 to row 0 and column 2 to column 0:
   $$\begin{bmatrix} 0 & -2 & 0 \\ 4 & 0 & 4 \\ 0 & -2 & 0 \end{bmatrix}$$
   to yield the new upper and lower bound on timer 1.
3. deleting row 2 and column 2.
4. setting timer 2 (the *nand* is excited, because its output is 0, and one of its inputs is 0) to its initial bounds for the *rising* transition of the *nand*. This yields the new state matrix, which is canonical already.

s_2 : the following events can occur:

1. timer 2 *can* expire again (i.e., the *nand* output goes to 1), leading to state s_3. This state has the same binary label as s_1, but a *different timer region matrix* (because now we *know* that at least 3 time units have passed since timer 1 was set), obtained by:
   (a) setting each element in row 2 to the minimum between its old value and 0, and checking for negative cycles.
   (b) copying row 2 to row 0 and column 2 to column 0:
   $$\begin{bmatrix} 0 & 0 & 0 \\ 2 & 0 & 2 \\ 0 & 0 & 0 \end{bmatrix}$$
   to yield the new upper and lower bound on timer 1.
   (c) deleting row 2 and column 2.
   (d) setting timer 2 (the *nand* is excited, because its output is 1, but all its inputs are 1) to its initial values for the *falling* transition of the *nand*. This yields the new state matrix, which is canonical already.
2. timer 1 *can* expire (i.e., the *inverter* output goes to 0), leading to state s_4. The new matrix is obtained by:
   (a) setting each element in row 1 to the minimum between its old value and 0, and checking for negative cycles.
   (b) copying row 1 to row 0 and column 1 to column 0:
   $$\begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ 1 & 1 & 0 \end{bmatrix}$$

to yield the new upper and lower bound on timer 2.

(c) deleting row 1 and column 1.

(d) timer 1 is not set again, because the input of the *inverter* is now stable, so the matrix is now 2 by 2.

3. timers 1 and 2 *can* expire together, if both were set to the same value. This leads to state s_9, which is stable (no timers are set).

s_3 : the following events can occur:

1. timer 2 *can* expire (i.e., the *nand* output goes to 0), leading to state s_5 . The new matrix is obtained by:

   (a) setting each element in row 2 to the minimum between its old value and 0, and checking for negative cycles.

   (b) copying row 2 to row 0 and column 2 to column 0:
   $$\begin{bmatrix} 0 & 0 & 0 \\ 1 & 0 & 1 \\ 0 & 0 & 0 \end{bmatrix}$$
   to yield the new upper and lower bound on timer 1.

   (c) deleting row 2 and column 2.

   (d) setting timer 2 (the *nand* is excited, because its output is 0, but one of its inputs is 0) to its initial values for the *rising* transition of the *nand*. This yields the new state matrix, which is canonical already.

2. timer 1 *can* expire (i.e., the *inverter* output goes to 0), leading to state s_8. Note that we are modeling this example with inertial delay, so timer 2 is *forced to expire* as well by this transition, without causing a transition on the output of the *nand*.

3. timers 1 and 2 *can* expire together, leading to state s_6. The new matrix is obtained by:

   (a) setting each element in rows 1 and 2 to the minimum between its old value and 0, and checking for negative cycles.

   (b) the intermediate matrix with timer bounds is not needed, as the next step deletes all rows and columns.

   (c) setting timer 2 (the *nand* is excited, because its output is 0, but one of its inputs is 0) to its initial values for the *rising* transition of the *nand*. This yields the new state matrix, which is canonical already.

s_4 : timer 2 expires, leading to s_8.

s_5 : the following events can occur:

1. timer 1 *can* expire, leading to state s_7. The new matrix is obtained by:
   (a) setting each element in row 1 to the minimum between its old value and 0, and checking for negative cycles.
   (b) copying row 1 to row 0 and column 1 to column 0:
   $$\begin{bmatrix} 0 & 0 & -1 \\ 0 & 0 & -1 \\ 3 & 3 & 0 \end{bmatrix}$$
   to yield the new upper and lower bound on timer 2.
   (c) deleting row 1 and column 1.
   (d) timer 1 is not set again, because the input of the *inverter* is now stable, so the matrix is now 2 by 2.
2. note that if we had chosen timer 2 to expire (an impossible event, given the upper and lower bounds on the timer values), the matrix after setting row 2 to 0 would have been:
   $$\begin{bmatrix} 0 & 0 & -2 \\ 1 & 0 & -1 \\ 3 & 0 & 0 \end{bmatrix}$$
   which does have a negative cycle (e.g. between vertices 1 and 2, due to $d_{12} = -1$ and $d_{21} = 0$).

s_6 : timer 2 expires, leading to s_8.

s_7 : timer 2 expires, leading to s_8.

We can see from the timed FA that the *nand* gate can actually oscillate only *twice* before settling to the final value.

In conclusion, this section described various techniques for the *analysis* of asynchronous circuits. The choice of the technique to apply to a particular problem depends mainly on the *circuit size* (the methods were listed in ascending order of complexity) and on the *degree of accuracy* with respect to the timing model and the number of incorrectly rejected circuits. Observe that only the last method can exactly identify hazards in the *bounded wire delay* model, but that other methods are *pessimistic* in the sense that no circuit declared hazard-free can actually have a hazard, so their results can be considered a safe approximation of the real circuit behavior.

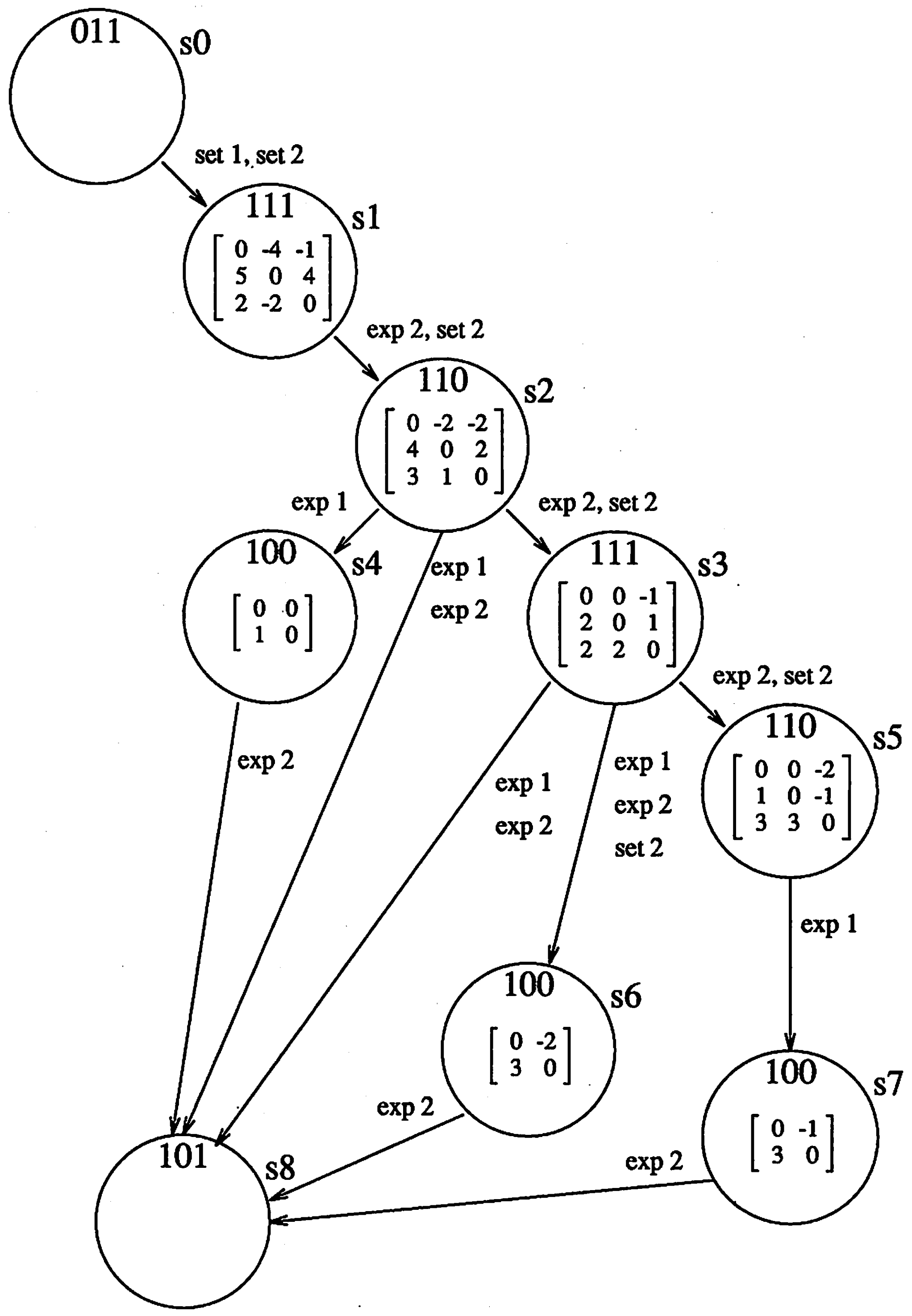


Figure 2.28: The complete timed Finite Automaton corresponding to Figure 2.26

2.8 Conclusion

The design methodology that we are going to describe in the next chapters has many similarities with the various design approaches outlined above. Namely, it relies on a *decomposition* of the circuit behavior into a "*local view*", in which delays must be taken into account during synthesis in order to produce a correct and optimized implementation, and a "*global view*", in which the specification itself ensures that no dependency on delay assumptions *needs to be* introduced (however such assumptions, if given, can be used to optimize the circuit implementation). This decomposition is common, for example, to [106], [85, 101], [53] (within limits) and [113].

The Signal Transition Graph specification describes the *complete system*, circuit and environment, using *events* rather than *logic levels* or *states* (even though, under implementability conditions described in Chapter 3, a state-based model can be derived from the event-based one). The STG definition used in this work is a generalization of the original definition from [27], that permits analysis of the "global view" and helps the designer decide which properties of the specification will ensure a behavior that does not depend on the "global interconnection" delays (as described in Chapter 3).

Our technique to translate an STG into an implementation uses classical state minimization ([46], [65]) and critical race-free state encoding ([115], [117]) in order to ensure implementability (as described in Chapter 4). It also uses analysis techniques closely related to classical *essential hazard* analysis in order to guarantee a hazard-free circuit implementation (as described in Chapter 5).

Finally, a scan-based delay testing technique, similar to that of [101], is used to ensure that the manufactured circuit delays are within the bounds assumed during synthesis, and that hazard-free operation is guaranteed (as described in Chapter 6).

The combination of ideas from all of the above research efforts allows us to derive a complete design methodology that frees the designer from the burden of tedious correctness verification. By automating the lower level synthesis process, the main task of asynchronous design becomes the solution of higher-level problems, such as the trade-off between throughput, latency and cost.

Moreover, our design procedure is aimed at the efficient solution of *low-level*, logic design problems, so it has a potentially wider range of applications than as a stand-alone tool. It can also be used, for example, as a component in higher level synthesis methodologies for asynchronous circuits. Such methodologies, as for example [113], [40] or [3], assume that a library of relatively complex components has been implemented in the available technology. A flexible, automated solution would thus describe the components using an STG, and use the methodology described in Chapters 4 and 5 to implement them correctly and automatically.

Chapter 3

The Signal Transition Graph Model

The classical specification and implementation models for asynchronous circuits described in Chapter 2 suffer from a number of problems, that do not allow to use them directly in our integrated design methodology.

For example, *circuit* models such as the Huffman model (Section 2.3) or the Muller model (Section 2.5), are defined only for circuits built out of components whose output signal behavior can be characterized as a *logic function* of a set of input signals. Such class of circuits excludes some very useful components, for example a large class of arbitration devices, that cannot be described by such interconnections of logic functions ([126]).

Neither model, moreover, allows for the kind of *behavioral abstraction* obtained by modeling some component using *non-determinism* rather than explicitly describing its operation in detail. For example, it is much easier to describe a CPU interacting with a bus interface as a device that can non-deterministically read or write, rather than deterministically describe its instruction memory, program counter, etc. Modeling the CPU as alternating between read and write cycles may not be acceptable either, since the interaction between successive, pipelined cycles can be non-trivial.

At the *specification* level, similarly, there is no known general methodology to decide whether a given STG specification *admits* an implementation that is, for example, hazard-free, or speed-independent, or delay-insensitive. There is also no satisfactory characterization of the above properties if the delays are *pure* (i.e. a translation in time of the input waveform) rather than *inertial* (i.e. short "pulses" are not transmitted). The only effort in this direction, to the best of our knowledge, is the so-called Change Diagram representation, that was shown in [125] to be formally equivalent to semi-modular circuits under the unbounded inertial gate delay model. Change

Diagrams, however, are not general enough, in that they can represent concurrency and causality, but not choice. I.e. they can model only *deterministic* behavior, and as such the description, for example, of a bus protocol with different read and write phases is awkward and imprecise, as we informally argued above.

Furthermore the classical definition of a "valid" Signal Transition Graph specification is unnecessarily restrictive, as [62] and [130] showed by presenting some useful, correctly implementable behaviors that cannot be described with the constrained STGs used by Chu in [26]. For example Chu required the Petri net underlying the STG to be *safe, live and free-choice*, in order to ease the STG analysis/synthesis task. This requirement is not part of the STG definition *per se*, and has little to do with a deeper characterization of the STG behavior as, say, speed-independent or delay-insensitive.

In this chapter we approach the problems mentioned above in the most general way, in the following steps.

- Give a general, low-level model of the *structure* and *behavior* of an asynchronous structure (where with the term "asynchronous structure", or sometimes "asynchronous system", we mean an interconnection of basic components that may be more complex than standard logic gates). This model, called Asynchronous Control Structure (ACS), allows multi-output components, non-determinism, etc. The *structure* of the ACS is a labeled, directed graph, while its *behavior* is described by a state-transition-like representation, that describes the events that can occur in every state, and the corresponding next state of the system (Arc-labeled Transition System, ALTS). We need a structural model because fundamental aspects of asynchronous design, such as delays, are associated with the structural components of the system.

- Describe how a *special* case of ACS, where each component has one output and is described by a logic function, corresponds to the classical model of an asynchronous circuit. The corresponding ALTS behavior specification now is determined by those logic functions changing the values of the circuit outputs in response to input and output signal transitions.

- Relate the local and global properties of the ALTS of a circuit with known low-level properties of the circuit, such as hazards, speed-independent operation, etc., both under inertial delays and pure delays. In order to establish formally this correspondence, we will have to introduce some auxiliary formalisms that capture the "history" of the circuit, beside its "current state", and show how this "history" relates to significant properties of the state-based ALTS description

- Give a general *high-level* model of the *behavior* of an asynchronous system (the associated structure will be described using the same graph-like representation as in the low-level model). This model, the Signal Transition Graph, will not have unnecessary restrictions superimposed, to allow us to prove the correspondence between low-level ALTS properties (and hence circuit properties) and high-level STG properties[1].

At this point the designer can use the framework to verify if a specification meets some circuit-level requirements, or, conversely, given a set of circuit-level properties, what class of specifications needs to be used.

Note that in this chapter we are not concerned with the details of how each component will be implemented in a specific technology. The main concern is to analyze properties that are common to every implementation of the specified behavior, using a model that is general enough to abstract various different implementation techniques, but detailed enough to have practical relevance. Such component implementation issues are dealt with in Chapter 5 (see also Chapter 2 for references to other approaches).

The chapter is organized as follows. Section 3.1 defines the low-level structural and behavioral model of asynchronous systems, called Asynchronous Control Structure and Arc-labeled Transition System, together with the related trace and partial order models. Section 3.2 describes asynchronous logic circuits, a special cases of Asynchronous Control Structures, and relates properties of the two, underlining the effects of the inertial/pure delay model dichotomy. Section 3.3 defines Signal Transition Graphs as interpreted Petri nets and describes the problem of their implementation in asynchronous circuits. Section 3.4 presents a classification of Signal Transition Graphs according to the corresponding circuit properties. Section 3.5 compares the Change Diagram model proposed in [125] with the STG model. Section 3.6 summarizes the use of the proposed model within our integrated design methodology.

3.1 A Low-level Model for Asynchronous Systems

This section introduces a low-level, *state-transition-based*, model of asynchronous systems. It has two *components*: a *structural* component called Asynchronous Control Structure

[1]Such restrictions will be imposed only when useful for the STG implementation algorithms, as in Chapters 4 and 5.

(ACS) and an associated *behavioral* component to describe its evolution in time, the Arc-labeled Transition System (ALTS).

The combination of the two (ACS and ALTS) is somewhat similar[2] to a network of interacting asynchronous Finite State Machines (the structure, describing who communicates with whom) together with the product FSM describing the behavior of the entire network (Section 2.2.2).

The properties of the model are characterized using the concept of Cumulative Diagram, that records the history of changes of each signal in the Asynchronous Control Structure. We then give an example of the power of our model using an asynchronous fair arbiter, that would be impossible to describe using "standard", logic-function based, models of asynchronous circuits.

3.1.1 Asynchronous Control Structures

The notion of Asynchronous Control Structure (ACS) is a generalization of the "interconnection structure" of an asynchronous control circuit. It removes the usual structural limitation (used, e.g. by [83] or [117]) that each component has exactly one output signal. An ACS structure can represent an arbitrary interconnection of modules, with the only restriction that no two modules can drive a single signal. Thus no wired-or or wired-and constructs are allowed, but note that at this level of abstraction they can still be modeled using discrete modules. The *behavior* of this interconnection of modules will be described using an Arc-labeled Transition System, as shown in Section 3.1.2.

An **Asynchronous Control Structure** (ACS) is a directed multi-graph[3] $\langle V, H, Y, \rho \rangle$:

- V is a finite set of *nodes*, each associated with an abstract discrete **component** of the ACS.
- $H \subseteq V \times V \times N$ (where N is the set of non-negative integers), is a finite set of *arcs*, representing the **interconnections** between the components. Note that there can be more than one arc joining a pair of components in each direction, hence the need to distinguish the edges of this multi-graph with an integer index.
- $Y = \{y_1, ..., y_n\}$ is a finite set of finite-state **variables** or **signals** (the names "signal" and "variable" will often be used interchangeably).

[2]This analogy should not be taken literally, and is only given to help the reader understand the general idea of the approach.

[3]A (directed) **multi-graph** is a (directed) graph where more than one edge is allowed between each (ordered) pair of nodes.

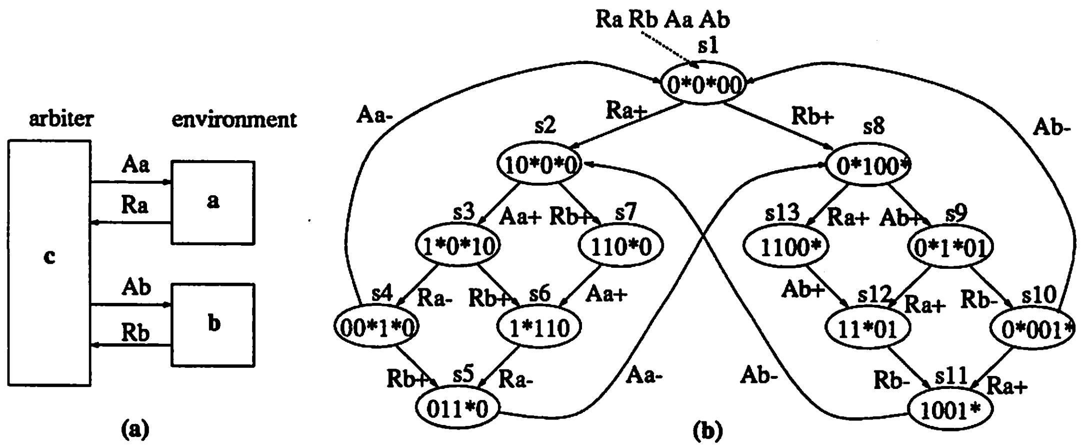


Figure 3.1: A Binary Asynchronous Control Structure and its State Transition Diagram

- $\rho : H \rightarrow Y$ is a *labeling* total function, associating every arc with a signal. Any two arcs labeled with the same signal must have the same source node (i.e. they represent a branching interconnection), so $\forall (v, v', i), (v'', v''', j) \in H$ we must have $\rho(v, v', i) = \rho(v'', v''', j) \Rightarrow v = v''$.

We also denote the sets of *input and output interconnections* for a component v as: $I^H(v) = \{(v', v, i) \in H\}$, and $O^H(v) = \{(v, v', i) \in H\}$, respectively. The sets of input and output *signals*, or simply inputs and outputs, for a component v are denoted: $I^Y(v) = \{y : \rho^{-1}(y) \in I^H(v)\}$, and $O^Y(v) = \{y : \rho^{-1}(y) \in O^H(v)\}$, respectively.

A simple example of an ACS is described in Figure 3.1.(a). Here $V = \{a, b, c\}$, $H = \{(c, a), (a, c), (c, b), (b, c)\}$, $Y = \{R_a, R_b, A_a, A_b\}$, and $\rho(c, a) = A_a$, $\rho(a, c) = R_a$, $\rho(c, b) = A_b$, $\rho(b, c) = R_b$ (we dropped the index 0 from the elements of H for simplicity). Furthermore $I^H(c) = \{(a, c), (b, c)\}$, $I^Y(c) = \{R_a, R_b\}$, and so on.

For every signal $y \in Y$, $S(y) = \{y^0, y^1, ..., y^k\}$ is called the set of *signal values*. We also assume that for each signal a specific set of *allowed changes* of values is defined, $D(y) \subset S(y) \times S(y)$, i.e. $D(y) = \{(y^i \rightarrow y^j) | i, j \in 0, 1, ..., k \wedge i \neq j\}$.

An ACS $\langle V, H, Y, \rho \rangle$ is called a **Binary Asynchronous Control Structure** (BACS) if $\forall y : S(y) = \{0, 1\}$. Hence, for a BACS, the set of allowed changes can be denoted as $Y \times \{+, -\}$, where "+" stands for a signal change from 0 to 1, and "-" for a signal change from 1 to 0. The *behavior* of a BACS is defined by a *binary transition system*, called state transition diagram, which is introduced in the following section.

3.1.2 Transition Systems and State Transition Diagrams

This section describes how the interconnected components of an ACS *behave* in time, that is how the signals associated with them change, using some key concepts from [59].

A **Transition System** (TS) is a pair $\langle S, E\rangle$:

- S is a set of **states**, and
- $E \subseteq S \times S$, is a set of **transitions**.

Note that we do not restrict S and E to be finite. The directed graph representation of a TS is as usual: states are vertices and transitions are arcs. For example, in Figure 3.1.(b) $S = \{s1, \ldots, s13\}$ and $E = \{(s1, s2), (s1, s8) \ldots\}$. We denote $(s, s') \in E$ by sEs'.

An **Arc-labeled Transition System** (ALTS) is a quadruple $\langle S, E, A, \delta\rangle$:

- $\langle S, E\rangle$ is a TS,
- A is a finite **alphabet of actions** and
- $\delta : E \rightarrow A$ is a (total) labeling function, which assigns each transition a single **action name** in A.

Each action name represents a *change of value* of a signal in the associated ACS, and each (possibly infinite) path along the graph represents a valid sequence of such changes in time. Thus the ALTS describes the complete allowed behavior of the associated ACS. For example, in Figure 3.1 we have $A = \{R_a^+, R_a^-, A_a^+, A_a^-, R_b^+, R_b^-, A_b^+, A_b^-\}$, where we use A_a^+ to denote the change of signal A_a from 0 to 1, and A_a^- to denote the change from 1 to 0. Furthermore $\delta(s1, s2) = R_a^+$, $\delta(s2, s7) = R_b^+$, and so on.

For a BACS with a set of n signals Y we define a **Binary Transition System** or **State Transition Diagram** (STD), $\langle S, E, \lambda\rangle$:

- $\langle S, E\rangle$ is a TS and
- $\lambda : S \rightarrow \{0, 1\}^n$ is a (total) labeling function such that each state is encoded with a *binary vector* consisting of the values of logic variables. The i-th component of the vector associated with each state s is denoted as $\lambda(s)_i$.

An STD is called **non-contradictory** if λ is injective. Hence for a non-contradictory STD we can identify the state with its binary label.

For every STD arc, connecting a pair of states s and s', we allow $\lambda(s)$ and $\lambda(s')$ to differ in one and only one component, say the i-th. This component signal, y_i, is called **excited** in state s and its value $\lambda(s)_i$ is marked with a "*" in $\lambda(s)$. Since there can be several outgoing arcs from each state, a number of signals can be excited in it. The signals that are not excited in a state are called **stable** in it. We assume that transitions between the states can have *non-zero arbitrary but finite delays*, and that these delays are associated with the delays of the components in the modeled BACS (similar to the *unbounded gate delay model* in asynchronous circuits, see Sections 2.1 and 2.5). We call an STD **initialized** if it has an explicit initial state. For example, in Figure 3.1.(b) $Y = \{R_a, A_a, R_b, A_b\}$, and $\lambda(s1) = 0000$, $\lambda(s5) = 1000$ and so on. Furthermore, R_a and R_b are excited and A_a and A_b are stable in $s1$.

Note that every STD can be also interpreted as an Arc-labeled Transition System, with the following labeling (consistent, since exactly one signal changes in every arc of an STD):

$$\forall e = (s, s') \in E : \delta(e) = \begin{cases} y_i^+ & \text{if } \lambda(s)_i = 0 \text{ and } \lambda(s')_i = 1 \\ y_i^- & \text{if } \lambda(s)_i = 1 \text{ and } \lambda(s')_i = 0 \end{cases}$$

The following important property of any STD comes directly from its definition:

Property 3.1.1 *No state in an* STD *can have two outgoing transitions labeled with the same signal but with different signs.*

We can now examine in more detail the meaning of Figure 3.1. It represents the interconnection of an *arbiter* and two other components (the arbiter's *environment*), that independently of each other may request access to a single resource, with signals, R_a and R_b. The arbiter grants access with A_a and A_b (which are mutually exclusive).

Note that this BACS/STD pair specifies a **fair arbiter** behavior, because if the arbiter receives a request at one input, say R_a, while it is processing a previous request from R_b, then it must, after finishing the transaction for R_b, respond to R_a before it can react to a new request from R_b again. Our abstract arbiter is capable of distinguishing the order in which the two, possibly concurrent, requests arrive at its inputs, by going to two different states ($s7$ and $s13$), labeled with the same vector 1100 (hence the STD is contradictory).

Reachability and Unique Action Relations

Intuitively, a state s' of a TS $\langle S, E \rangle$ is reachable from a state s if there exists a directed path from s to s'. More formally, the **direct reachability** relation is simply given by the set E. For any pair of states $s, s' \in S$, the state s' is called reachable from s if there is a finite length (including zero length) sequence of transitions leading from s to s'. Therefore **reachability** is given by the reflexive and transitive closure of E, i.e. E^*. In the example of Figure 3.1.(b) all states are mutually reachable.

Similarly for any ALTS $\langle S, E, A, \delta\rangle$ we can define the *reachability through a sequence of actions*. Specifically, for direct reachability through action $a \in A$, we denote $sE(a)s'$ if sEs' and $\delta(s, s') = a$. For example in Figure 3.1.(b) we have $s1E(R_a^+)s2$. Then general reachability through a sequence of actions, denoted $sE(\alpha)s'$ means that there exists a finite sequence of action names $\alpha = a_1, a_2, ..., a_m$ labeling a directed path between s and s'. We can sometimes use the notion of an allowed sequence from a state, i.e. α is *allowed from* s if $\exists s'$ such that $sE(\alpha)s'$. So R_a^+, A_a^+, R_b^+ is allowed in $s1$, but it is not allowed in $s2$ in Figure 3.1.(b)

In the following, whenever we consider an *initialized* ALTS, we will implicitly assume that every state in it is *reachable* from the initial state. Unreachable states, obviously, can be deleted from it without changing the specified behavior.

Note that among the various arcs labeled with the same action in Figure 3.1.(b), some of them actually represent exactly the same "event". For example, arcs $(s2, s3)$ and $(s7, s6)$ both represent the same event, the arbiter acknowledging request R_a^+ from the environment. Now we make this intuitive idea more formal, because it will become important when we relate *state-based* models, such as the State Transition Diagram, with *event-based* models, such as the Signal Transition Graph, where the notion of *unique occurrence of an event* is explicit.

For an ALTS $\langle S, E, A, \delta\rangle$, we define a pairwise relation U on the set E of arcs as $(s, s')U(s'', s''')$ if:

- $\delta(s, s') = \delta(s'', s''')$ and
- sEs'' and $\delta(s, s') \neq \delta(s, s'')$
  (i.e. the "action" $\delta(s, s')$ has not occurred in s'' yet, so in some sense it is "the same action" that is enabled in both s and s'').

Let U^* be the equivalence relation formed by the reflexive, symmetric and transitive closure of U. We call U^* the **Unique-action Relation.** We can easily see that Unique-action Relation partitions the set E into a set of Unique-action Relation-*classes*, $[E]^A$. Each such class, $[e]^a$, is an **action.** The set of actions with the same name, a, is the **action set** of the name a and denoted as $[E]^a$. This notion will be useful later, when we shall associate the transitions of an STG with the transitions of the corresponding STD.

For example, in Figure 3.1.(b) we have $(s2, s7)U(s3, s6)$, and $(s3, s6)U(s4, s5)$, hence, by transitivity, $(s2, s7)U^*(s4, s5)$. Also $(s1, s8)U(s2, s7)$.

Then $[e]^{R_b^+} = \{(s1, s8), (s2, s7), (s3, s6), (s4, s5)\}$. In this very simple case, each action set has a single element, $[E]^{R_b^+} = \{[e]^{R_b^+}\}$ and so on.

For an action $[e]^a$, the set of states, always forming a connected subgraph, which are the sources for the transition arcs in $[e]^a$ is the **excitation region** for action $[e]^a$. So in Figure 3.1.(b) the excitation region of $[e]^{R_b^+}$ is $\{s1, s2, s3, s4\}$, corresponding to the states where the label bit for R_b has value 0 and is tagged with "*".

Interleaving Semantics of Concurrent Actions

Throughout this work we assume that the actions associated with a set of arcs outgoing from the same state can be performed in the modeled system *concurrently*, i.e. independently of each other. See for example R_a^+ and R_b^+ in $s1$ in Figure 3.1.(b), which are "produced" by different and independent components. Since our model is *entirely asynchronous*, we must assume that the changes of corresponding signals can occur *in time in any order*.

Our low-level behavioral model, on the other hand, requires that *a single signal changes for every transition*. We then choose to model such **concurrency** by **interleaving**, i.e. considering all possible alternative chain orderings compliant with the partial order between possibly concurrent actions (in Figure 3.1.(b) this corresponds to paths $s1, s2, s7$ and $s1, s8, s13$). Such a modeling is convenient yet sometimes problematic, because it hides the semantic distinction between *true concurrency* and "shuffled" *alternative selections*. This distinction can be made explicit only in models with explicit causality notions, and we postpone it until Section 3.3, where we will consider Signal Transition Graphs.

3.1.3 Properties of Transition Systems and State Transition Diagrams

In this section we analyze a set of behavioral properties of Transition Systems and State Transition Diagrams that we will show later to be connected with corresponding, important properties of asynchronous systems. For example, the property of *confluence* below is closely connected to the requirement that the "long term behavior" of the system must not be influenced by the relative magnitude of the delay of two components. No matter who "wins the race", we must still be able to reach the same state in the future. Similarly, *local confluence* will be shown to be related to the classical concept of *static hazards* in a circuit.

Following [59], we call an ALTS $\langle S, E, A, \delta \rangle$:

- **confluent,** if $\forall s, s', s'' \in S$, if sE^*s' (i.e. s' is reachable from s) and sE^*s'', then $\exists s''' \in S$ such that $s'E^*s'''$ and $s''E^*s'''$.
- **locally confluent,** if $\forall s, s', s'' \in S$, if sEs' (i.e. there is an arc $(s, s') \in E$) and sEs'', where $s' \neq s''$, then $\exists s''' \in S$ such that $s'Es'''$ and $s''Es'''$. If such s''' is unique, then the ALTS is called **uniquely locally confluent.**

So, Figure 3.1.(b) is confluent (all pairs of states can reach any state), but *not locally confluent*, due to $s1, s2$ and $s8$ ($s1Es2$ and $s1Es8$, but there is no common immediate successor of $s2$ and $s8$).

Keller, in [59], gave three *sufficient* conditions for local confluence (and hence confluence) of an ALTS. An ALTS is:

- **deterministic,** if $\forall s, s', s'' \in S$ and $\forall a \in A$, if $sE(a)s'$ and $sE(a)s''$, then $s' = s''$ (i.e. for each action name there can be only one outgoing transition from a state that is labeled with it).
- **commutative,** if $\forall s \in S$ and $\forall a, b \in A$, if ab and ba are allowed in s, then $\exists s'$ such that $sE(ab)s'$ and $sE(ba)s'$ (i.e. the effect of interleaving two transitions both allowed in a state and not mutually exclusive is the same).
- **persistent,** if $\forall s \in S$ and $\forall a, b \in A, a \neq b$, if a and b are allowed in s, then ab is allowed in s (i.e. no transition can **disable** another one).

The following theorem was proved in [59]:

Theorem 3.1.1 *An* ALTS *is both locally confluent and confluent if it is deterministic, persistent and commutative.*

The definition of STD implies that if an STD satisfies these conditions, then it is uniquely locally confluent.

The ALTS in Figure 3.1.(b) is deterministic and persistent, but not commutative (due to $s1, R_a^+$ and R_b^+ again). So, being confluent, it shows that Keller's conditions are only *sufficient*.

Now, even though our state-based model has no "direct" idea of *causality* between actions, we can still locally verify if some action has "a unique set of predecessors", that can somehow be identified with its causes. Hence we define the property of *strict causality* of an ALTS, which, as the Unique Action Relation, will become more clear when we introduce our event-based model, where such causality is explicit.

Let $\mathcal{S} = \langle S, E, A, \delta \rangle$ be an ALTS and let $[E]^A$ be its set of actions. Let $\mathcal{S}([e]^a)$ denote the excitation region for action $[e]^a \in [E]^A$.

An ALTS is called *strictly causal for action* $[e]^a$ *and state* $s \in S$ if:

- $\forall s', s'' \in \mathcal{S}([e]^a), s' \neq s''$, such that there exists a directed path π_1 between s and s' and a directed path π_2 between s and s'', with $\pi_1 \cap \mathcal{S}([e]^a) = \emptyset$ and $\pi_2 \cap \mathcal{S}([e]^a) = \emptyset$ (i.e. s' is the first state in π_1 where $[e]^a$ is excited, and similarly for s''),
  - $\exists s''' \in \mathcal{S}([e]^a)$ (possibly coincident with s' or s'') such that:
    * for every directed path π_3 between s and s''' we have $\pi_3 \cap \mathcal{S}([e]^a) = \emptyset$ and
    * there exists a directed path π_4 between s''' and s' such that $\pi_4 \subseteq \mathcal{S}([e]^a)$ and
    * there exists a directed path π_5 between s''' and s'' such that $\pi_5 \subseteq \mathcal{S}([e]^a)$.

  I.e. s' and s'' have a common "ancestor" s''', through states where $[e]^a$ is also excited, which is a successor of s and where $[e]^a$ is also excited for the first time.

An ALTS is called **strictly causal** if it is strictly causal for all actions in $[E]^A$ and all states in S.

This definition means, informally, that each excitation region of each action has a single "top" state (or a "cycle" of such states, as in the example below), where it becomes excited for the first time, and all other states in the region (which is connected by definition) are successors of it through paths *within the region.* So actions leading into this "top" state (or cycle) can be informally identified with its causes.

On the other hand, if the ALTS is not strictly causal, it means that some action has "many alternative ways" of becoming allowed.

The ALTS in Figure 3.1.(b) is strictly causal, because, for example, for action $[e]^{R_a^+}$ the states in its excitation region $\{s1, s8, s9, s10\}$ form a cycle. So, for example, from state $s4$ we can reach both $s1$ and $s8$ (through $s5$), and in this case the third state in the definition coincides with $s1$, whence $s8$ is reachable without leaving the excitation region. Similarly for all other triples of states and actions.

Finally, when analyzing the behavior of an ALTS we are interested in checking if we have a point where future behaviors diverge completely. Such behavior is, in general, not desirable, and hence the *liveness* of an ALTS is important to check. We define it only for *finite* ALTS. For a *finite* ALTS, with the reachability relation E^* between states, we define the **mutual reachability** relation between any two states, $s, s' \in S$ if both sE^*s' and $s'E^*s$ hold for them. This is an equivalence relation, so it gives rise to a set of equivalence classes. Built for a given initial state, these classes form a partial order induced by the reachability relation, i.e. $c_1 \sqsubseteq c_2$ if the states in c_2 can be reached from the states in c_1 (but not vice-versa, otherwise the two classes would not be distinct). Each *maximal* class in this partial order[4] is called a **final class** (i.e. once we enter one of these classes, we can never leave it).

A *finite* ALTS is **live** if for every state $s \in S$ and every action name $a \in A$, there exists a state $s' \in S$, reachable from s, in which a is allowed. A live ALTS is **strongly live** if it forms a *single equivalence class* for any initial state. Such a TS is represented by *a strongly connected graph*[5]. The ALTS in Figure 3.1.(b) is strongly live.

3.1.4 Trace Models

For an ACS defined by an ALTS we can define another representation, called Trace Structure ([100, 119, 120], [13], [39]), or Trace Model[6], of its behavior. This representation will be needed in Section 3.4.3, because delay-insensitive circuits were defined in the literature using Trace Models, so in order to define delay-insensitivity within our framework we must relate Trace Models with Arc-labeled Transition Systems.

A **Trace Model** representation of the behavior (described by an Arc-labeled Transition System) of a structure (described by an Asynchronous Control Structure) is a pair $\langle A, \Sigma \rangle$:

- A is a finite **alphabet of actions** and
- $\Sigma \subseteq A^\infty$ is a *prefix-closed* set of **traces** or sequences of actions.

Here A^∞ denotes the set of all possible finite strings and infinite sequences of actions in A (i.e. according to a more standard notation, $A^\infty = A^* \cup A^\omega$). A set Σ of traces is **prefix-closed** if whenever a trace belongs to Σ, then all its prefixes belong to Σ. A *finite* trace α is a **prefix** of a trace β if there exists a (finite or infinite) sequence of actions γ such that $\beta = \alpha\gamma$.

This model is defined with respect to a given initial state, and represents the execution *history* of the ALTS. Each trace then stands for a sequence of actions that can be performed on the signals of the ACS. The set of traces in Σ contains those traces that are allowed by the behavioral

[4]An element of a partial order is a **maximal element** if it is larger than any element comparable with it.

[5]A directed graph is **strongly connected** if there exists a directed path between every pair of nodes.

[6]We are forced to introduce this term here, as a synonym of the more common "Trace Structure", only to avoid confusion with the abbreviation of the term "Transition System" (TS).

specification. Such traces are the **successful** traces, as opposed to the sequences of actions that are not in Σ, the **failure** traces.

For a BACS with an associated initialized STD and a set of signals Y, we can also think about a **Binary Trace Model**, which is a pair $\langle Y, \Sigma \rangle$, where $\Sigma \subseteq (Y \times \{+,-\})^\infty$ is a prefix-closed set of traces, or sequences of signal changes as allowed by the STD.

In the example in Figure 3.1, initialized in state $s1$, we have (let ϵ denote the empty trace):

$$\begin{aligned} A &= \{R_a^+, R_a^-, A_a^+, A_a^-, R_b^+, R_b^-, A_b^+, A_b^-\} \\ \Sigma &= \{\epsilon, R_a^+, R_b^+, R_a^+R_b^+, R_b^+R_a^+, R_a^+A_a^+, R_b^+A_b^+, R_a^+A_a^+R_b^+, \ldots\} \end{aligned}$$

The following property of the Binary Trace Model generated from an STD is the result of Property 3.1.1 and of the definition of STD.

Property 3.1.2 *In a Binary Trace Model $\langle Y, \Sigma \rangle$, for every trace in Σ and any signal $y \in Y$, all the occurrences of y have alternating signs, i.e. between any two consecutive changes of the same sign there is at least one opposite change.*

3.1.5 Cumulative Diagrams

In order to characterize classes of behaviors of asynchronous systems, we need the concept of *history of the execution* of a state-based specification. The complete history of the system is represented by a set of traces, where each trace records *exactly* the order of occurrences of actions. The state of the Arc-labeled Transition System, on the other hand, describes only the *final result* of such execution. In this section, following [83], we introduce a model to describe this history, called a Cumulative Diagram (CD), where only the *number of occurrences* of each action is recorded. Hence this representation will be of intermediate "precision" between a Trace Model and an ALTS.

The Trace Model description of the operation of an initialized ALTS contains all the traces of the ALTS starting from its initial state. The mechanism of trace generation induces a natural mapping between traces and sets of states, where each trace maps to the set of states where the ALTS may be at the end of its generation. Note that this mapping is *functional* if the ALTS is *deterministic*. In this case, the state where the ALTS arrives for a given trace with respect to an initial state is uniquely determined through the reachability relation.

On the other hand, for *any* ALTS (not just deterministic ones) we can think about another mapping from the set of traces. This mapping defines, for every trace $\alpha \in \Sigma$, a *multiset* of action names μ, with the multiplicity of each name $a \in A$, denoted by μ_a, equal to the number of occurrences of a in α. A multiset obtained in this way is called a **cumulative state**. It is convenient to represent a cumulative state by a *vector* of non-negative integers with dimension $|A|$.

We can easily see from the above definition that a cumulative state defines an equivalence class between traces of the ALTS which are simple permutations of each other (note that not every permutation may be a valid trace). Let $[\alpha]$ be such equivalence class for trace α. Every trace $\beta \in [\alpha]$ brings the ALTS to the same cumulative state μ. Therefore we can identify this μ with $[\alpha]$.

The set of cumulative states generated by an ALTS through its Trace Model model is a partial order. This order is a subset of the natural integer vector ordering[7] and is built upon the *prefix order* between traces up to permutations. Formally, $[\alpha] \sqsubseteq [\beta]$ if there exist $\alpha \in [\alpha]$ and $\beta \in [\beta]$ such that α is a prefix of β.

The partial order of cumulative states, built as above, is called the **Cumulative Diagram** (CD). It will be represented graphically using its Hasse diagram, where the reflexive and transitive edges are removed.

The mapping between cumulative states and ALTS states is a *function* for every initialized, deterministic, persistent and commutative ALTS, where all the traces in $[\alpha]$ bring the ALTS in the same state.

The above definitions are easily adapted to the case of a BACS and its STD. By Property 3.1.1, we can change the notion of cumulative state and build the CD for a set of signal names Y rather than their changes $Y \times \{+, -\}$. This modified version of CD is isomorphic to the original version because *all the changes of the same signal are linearly ordered* (see also Property 3.1.2).

For example, Figure 3.2 contains an initial fragment of the CD for the STD described in Figure 3.1, with signal ordering R_a, R_b, A_a, A_b. We have $0000 \sqsubseteq 1000$ because trace R_a^+ is a valid trace of the STD, and $0000 \sqsubseteq 1100$ because trace $R_a^+ R_b^+$ (as well as $R_b^+ R_a^+$) is a valid trace of the STD. Note that the CD model *cannot describe the local divergence* after traces $R_a^+ R_b^+$ and $R_b^+ R_a^+$. In fact both traces lead to the same cumulative state 1100, that corresponds to states $s7$ and $s13$ in Figure 3.1.(b). It is possible to develop, in order to take care of this mismatch, a modified definition of CD, where a cumulative state is a *pair* composed of a multiset of action names and the *state* reached after the corresponding trace ([47]). Here we only need the weaker definition of CD. The example of Figure 3.2 also illustrates that the mapping between CD states and ALTS states in general is a relation, not a function.

Now we have all the necessary information to define the notion of speed-independent

[7]If v and v' are two integer vectors, $v \sqsubseteq v'$ if for all i $v_i \leq v'_i$, otherwise v and v' are incomparable.

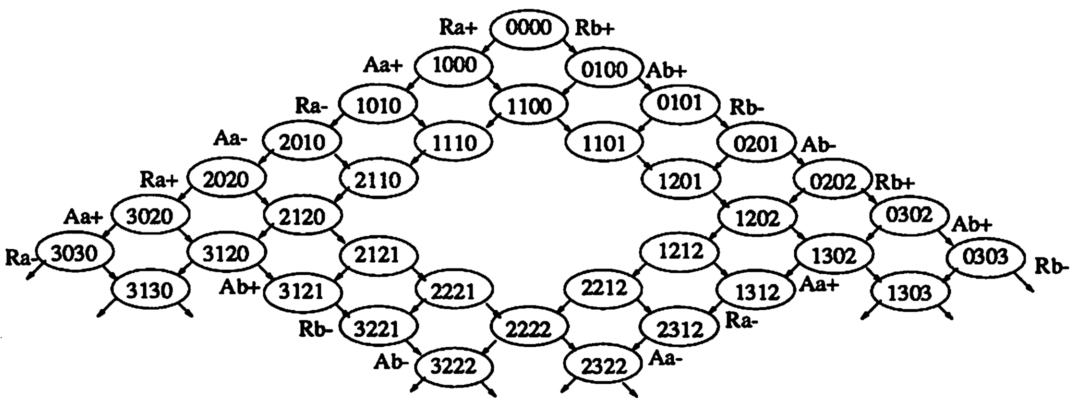


Figure 3.2: The Cumulative Diagram of Figure 3.1

and semi-modular behavior, which are crucial (as we will see in Section 3.2.5) for more practical purposes, such as the analysis and synthesis of asynchronous control circuits.

3.1.6 Speed-independence and Semi-modularity

The intuitive notion of speed-independent behavior can be more formally described using *confluence*, that ensures that the "long term" behavior of the modeled asynchronous system does not depend on the winner of a race between concurrent transitions. In this section we give a set of *alternative* definitions of some ALTS properties. These properties will be shown to correspond to:

- interesting circuit properties, such as the absence of hazards in Section 3.2, and
- high-level specification properties in Section 3.4.

So this section provides the desired *bridge* between the two domains.

Let us first recall some definitions from lattice theory. Let C be a partial order. An element $z \in C$ is a **zero element** if for all $c \in C$ we have $z \sqsubseteq c$. An element c_1 of a partial order C **covers** another element c_2 of C if $c_2 \sqsubseteq c_1$ and there is no c_3 such that $c_2 \sqsubseteq c_3 \sqsubseteq c_1$.

A **lattice** is a partial order where every pair of elements has a *greatest lower bound* ($\sqcap$) and a *least upper bound* ($\sqcup$). A lattice is **semi-modular** if for every pair of elements c_1 and c_2 that cover a third element c_3 (then obviously $c_3 = c_1 \sqcap c_2$), they are both covered by $c_1 \sqcup c_2$. A semi-modular lattice is **distributive** if the g.l.b. and l.u.b. operations are mutually distributive.

We can now formally define:

1. An ALTS (STD) is called **speed-independent-1** if it is *confluent*.

2. A *finite* ALTS (STD) is called *speed-independent-2 with respect to a state* if the CD generated for this state is a *lattice with a zero element*, according to the partial order defined in Section 3.1.5.

   The ALTS (STD) is called **speed-independent-2** if it is speed-independent-2 with respect to every state.

3. A *finite* ALTS (STD) is called *speed-independent-3 with respect to a state s* if it has a single *final class* when initialized in s

   The ALTS (STD) is called **speed-independent-3** if it is speed-independent-3 with respect to every state.

For example, the ALTS in Figure 3.1.(b) satisfies speed-independent-3, because it is finite and it has a single final equivalence class for every initial state. We can easily see that the TS is confluent, and hence speed-independent-1. Furthermore its CD, represented in Figure 3.2, is a lattice (zero is cumulative state 0000), so the STD is speed-independent-2.

It can be shown that, despite our intuition, the definitions above are not strictly equivalent for a given finite ALTS (STD). Speed-independent-1 is equivalent to speed-independent-3 in the finite case, while speed-independent-1 and speed-independent-2 are equivalent if (but not only if) the given finite ALTS (STD) is *deterministic, persistent* and *commutative*:

Theorem 3.1.2

1. *A finite initialized* ALTS *(*STD*) is speed-independent-1 if and only if it is speed-independent-3.*

2. *A finite initialized* ALTS *(*STD*) is speed-independent-2 if it is deterministic, persistent and commutative.*

3. *An* ALTS *(*STD*) is speed-independent-1 if it is deterministic, persistent and commutative.*

In the following $\alpha, \beta, \ldots$ denote finite traces, $a, b, \ldots$ denote actions, $\mu, \nu, \ldots$ denote CD states and $s, t, \ldots$ denote ALTS states. We first need the following lemma:

Lemma 3.1.3 *Let S be an initialized, deterministic, persistent and commutative* ALTS.

Then for each valid trace α of S, all the valid permutations $\beta \in [\alpha]$ lead the ALTS *exactly to the same state.*

Proof We will prove the lemma by induction on the length of α.

- base case: the only permutation of a trace of length 1 leads the ALTS to a single state, because it is deterministic.
- induction step: every permutation of a trace $\alpha = a\alpha'$ of length $n + 1$ (with $n > 0$) that yields another valid trace $\beta = b\alpha''$ can be composed as sequence of three permutations:

  1. a permutation of α' into $b\alpha'''$,
  2. an exchange of a and b and
  3. a permutation of $a\alpha'''$ into α''.

  Let s denote the initial state of $\mathcal{S}$, let s' be the (unique, by determinism) state reached from s through action a, and let s'' be the state reached from s through action b.

  All the above permutations are valid partial traces of $\mathcal{S}$ (starting from the appropriate state s' or s''), because both a and b are enabled in s, hence by persistency b must still be enabled in s'. This means that b can be the first action of a valid permutation of α' (and vice-versa).

  By determinism and commutativity, the state reached from s after a and b (in any order) is the same, call it s'''. The first and the last permutation involve traces of length n, so by the induction hypothesis both α' and $b\alpha'''$ lead into the same state, as well as α'' and $a\alpha'''$. But this state must be the same, as it is reached from s''' after partial trace α'''.

■

Now it should be clear that for every initialized, deterministic, persistent and commutative ALTS, where all the traces in $[\alpha]$ bring the ALTS in the same state, there exists a *functional mapping* between cumulative states and ALTS states.

We are now ready to prove Theorem 3.1.2

Proof

1. $\Rightarrow$ If the ALTS has a single final equivalence class, then all the states in that class are mutually reachable and reachable from any other state by definition. So for each pair of states s, s' there exists at least one state, any member of this final class, that is reachable from both, and hence the ALTS is confluent.

   $\Leftarrow$ Suppose, for the sake of contradiction, that the ALTS had more than one final class. Then there would exist two states (belonging to different final classes) both reachable from the initial state, without a state reachable from both. This would contradict the assumption of confluence.

2. We will prove a stronger claim, that if an ALTS is deterministic, persistent and commutative, then its CD is a *semi-modular* lattice with a zero element. We prove the existence of the zero element, then the existence of the g.l.b. and of the l.u.b., and finally semi-modularity:

   - The zero element is clearly the all-zero vector, corresponding to the empty trace, that is less than any other element of the CD.
   - L.u.b.: let μ, μ' be a pair of incomparable cumulative states, reached after traces α and α' respectively. Lemma 3.1.3 implies that every cumulative state has a unique corresponding ALTS state. Let s and s' denote the states corresponding to μ and μ' respectively.

     Then we claim that the cumulative state μ'', where $\mu''_a = \max(\mu_a, \mu'_a)$ for all action names a, is reachable from μ, by following edges labeled with actions a such that $\mu'_a > \mu_a$ an appropriate number of times.

     Suppose the claim is not true. Then there exists an action that appears in α' but is not enabled in any ALTS state reachable from s through actions only in α' but not in α. Let a denote the first action in α' with this property, and let β' be the prefix of α' up to this a (i.e. $\alpha' = \beta' a \gamma'$).

     By the choice of a, there exists a trace β such that, extension of a prefix of α, that is a permutation of β', and leads (by Lemma 3.1.3) the ALTS to the same state as β'.

     But then, by persistency, we can extend α with the actions in β and not in α. This leads to a state that is a successor of s and where, again by persistency, a is enabled.

     This shows that μ'', as defined above, exists. It is clearly an upper bound of μ and μ'. Moreover it is clearly the unique least upper bound, because no smaller cumulative state can be larger than both μ and μ'.
   - G.l.b.: the lattice zero is clearly a lower bound. Suppose that a pair of cumulative states μ and μ' had two incomparable lower bounds ν and ν' such that no element greater than ν is a lower bound and no element greater than ν' is a lower bound. We claim that the l.u.b. ν'' of ν and ν', that exists by the above argument, is also a lower bound on μ and μ'.

     By the definition of lower bound, for all action names a we have $\mu_a \geq \nu_a$ and $\mu_a \geq \nu'_a$ (and similarly for μ'). For the l.u.b. constructed above, also $\mu_a \geq \nu''_a$ for all a (and similarly for μ').

Moreover, since ν and ν' are incomparable, ν'' must be greater than both, so we have reached a contradiction, and the g.l.b. must be unique. This completes the proof that the CD is a *lattice* with a zero element.

- The proof of semi-modularity is immediate. Let μ' and μ'' be two CD states covering μ. Let a and b the actions leading from μ to μ' and μ'' respectively. By persistency a must be enabled in the (unique, by Lemma 3.1.3) ALTS state corresponding to μ'' and b must be enabled in the ALTS state corresponding to μ'. The state reached from μ' through action b and from μ'' through action a clearly covers both.

3. The third result is immediate by Theorem 3.1.1.

■

It is very easy to construct examples showing that removing any of persistency, commutativity or determinism can falsify Theorem 3.1.2. On the other hand, the example in Figures 3.1 and 3.2 shows that our conditions for the equivalence between speed-independent-1 and speed-independent-2 are only *sufficient* and *not necessary*, because this ALTS is both speed-independent-1 and speed-independent-2 but *not commutative*.

Semi-modularity, that we will relate to hazard-freeness, is a stronger property than speed-independence. Again, two alternative definitions can be formulated.

1. An ALTS (STD) is called **semi-modular-1** if it is *locally confluent*.

2. A *finite* ALTS (STD) is called *semi-modular-2 with respect to a state* if the CD generated for this state is a *semi-modular lattice with a zero element*, according to the partial order defined in Section 3.1.5.

   The ALTS (STD) is **semi-modular-2** if it is semi-modular-2 for every state.

Note that now CD semi-modularity, determinism and commutativity are enough to ensure *persistency*:

Lemma 3.1.4 *If an* ALTS *is semi-modular-2, deterministic and commutative, then it is persistent.*

Proof Let s be any ALTS state that is not persistent. I.e. there exist states s' and s'' and actions a and b such that $sE(a)s'$ and $sE(b)s''$, but a is not enabled in s'' or b is not enabled in s'.

Let μ be the zero element of the CD built from s, μ' the cumulative state reached from μ through a and μ'' be the state reached from μ through b. They both cover μ, so by semi-modularity they are both covered by $\mu''' = \mu' \sqcup \mu''$.

By determinism, μ' can correspond only to s' and μ'' only to s''. Moreover, μ''' covers μ', so b must be enabled in s', and similarly for μ'', s'' and a. We have reached a contradiction, so the ALTS must be persistent. ■

A Theorem analogous to Theorem 3.1.2 can also be shown to hold about semi-modular-1 and semi-modular-2:

Theorem 3.1.5

1. *A finite initialized* ALTS *(*STD*) is semi-modular-2 if it is deterministic, persistent and commutative.*

2. *An* ALTS *(*STD*) is semi-modular-1 if it is semi-modular-2, deterministic and commutative.*

Note that the *sufficient* conditions that ensure equivalence between semi-modular-1 and semi-modular-2 are almost the same as for speed-independent-1 and speed-independent-2, and so are the proofs of the two theorems. The only difference is that, as shown in Lemma 3.1.4, semi-modularity, determinism and commutativity imply persistency. The situation is very similar to the distinction between *local* and *global confluence*, where sufficient conditions are known only for the *stronger* definition (local confluence and semi-modularity respectively).

The ALTS in Figure 3.1.(b) is not locally confluent, hence not semi-modular-1. It is not semi-modular-2 either, because the corresponding CD in Figure 3.2 is not semi-modular, due for example to cumulative states 1110 and 1101, that both cover 1100, but are not covered by their least upper bound 2222 (recall that covering means being immediately above in the partial order).

Another class of ALTSs, significant because of some interesting analysis and synthesis results (see Section 2.7.2 and [126]) is connected with the definition of *strict causality* (or, informally, of a "unique set of actions causing an action") described in Section 3.1.3.

1. An ALTS (STD) is called **distributive-1** if it is *strictly causal* and *locally confluent.*

2. A finite ALTS (STD) is called *distributive-2 with respect to a state* if the CD generated for this state is a *distributive lattice with a zero element*, according to the partial order defined in Section 3.1.5.

   The ALTS (STD) is **distributive-2** if it is distributive-2 for every state.

A Theorem analogous to Theorem 3.1.2 can also be shown to hold about distributive-1 and distributive-2:

Theorem 3.1.6

1. *A finite initialized* ALTS *(*STD*) is distributive-2 if it is distributive-1, deterministic, persistent and commutative.*
2. *An* ALTS *(*STD*) is distributive-1 if it is distributive-2, deterministic and commutative.*

Proof

1. We will show that if an ALTS is strictly causal, deterministic, persistent and commutative, then for each pair of cumulative states their greatest lower bound is the component-wise minimum. We have already shown, in the proof of Theorem 3.1.2, that the least upper bound of each pair is the component-wise maximum, so the distributivity of min and max will be enough to show that the CD is distributive-2.

   Let μ and μ' be a pair of cumulative states and let s and s' be their (unique, by Lemma 3.1.3) corresponding ALTS states. Let $\mu'' = \mu \sqcap \mu'$ be their g.l.b., corresponding to state s''.

   Let us assume, for the sake of contradiction, that μ'' is not the component-wise minimum (it must obviously be less, according to the integer vector order). Let a be an action such that $\mu''_a < \min(\mu_a, \mu'_a)$. There must exist at least one such action, since μ'' is not the component-wise minimum.

   So there must exist a pair of states t and t' such that:

   - a is enabled in t for the first time on a path between s and s'.
   - a is enabled in t' for the first time on a path between s and s''.

   But by strict causality there must exist another state t'' that is a predecessor of both and a successor of s where a is also enabled. This contradicts the assumption that μ'' is an l.u.b., since the cumulative state corresponding to t'' would be also an upper bound, and strictly greater than μ''.

2. Assume that the CD is distributive (hence it is also a semi-modular lattice with the zero element, by definition) and that the ALTS is deterministic and commutative. The ALTS is also persistent, due to Lemma 3.1.4.

   Suppose, for the sake of contradiction, that the ALTS is not strictly causal. This means that there is some action a whose excitation region can be entered through a pair distinct states,

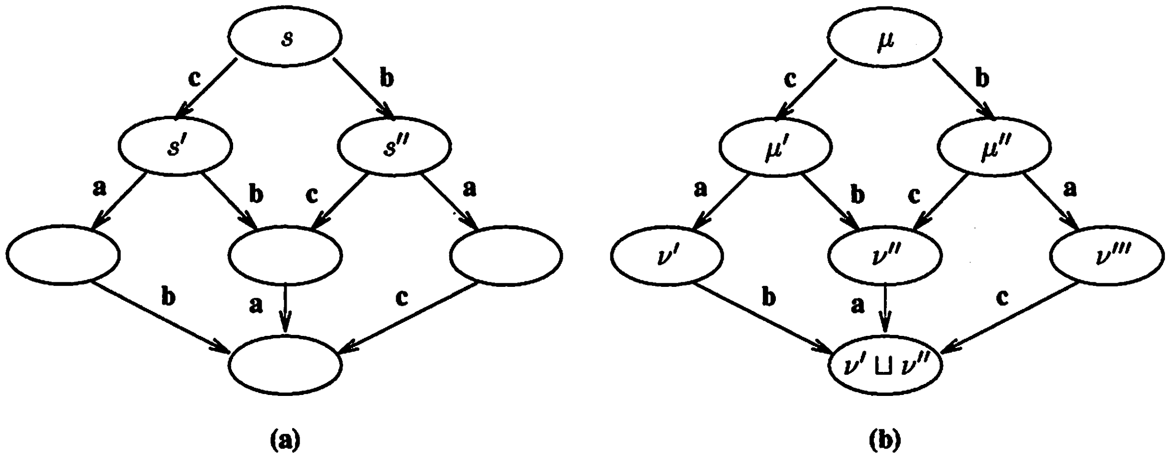


Figure 3.3: Illustration of the proof of Theorem 3.1.6

s' and s'', that have no common predecessor through states in the excitation region itself. Among such pairs of states there must exist one with an immediate predecessor, call it s, that does not belong to the excitation region. Such an ALTS fragment is depicted in Figure 3.3.(a).

Let us consider the CD built from s (a fragment appears in Figure 3.3.(b)). Every cumulative state has a unique corresponding ALTS state by Lemma 3.1.3, so let μ, μ' and μ'', be the smallest cumulative states corresponding to s, s' and s''. But this contradicts the assumption that the CD is distributive, because ν''' is $(\nu' \sqcup \nu'') \sqcap \nu'''$, but $\nu' \sqcap \nu'''$ is μ''' and $\nu'' \sqcap \nu'''$ is μ'', so $\mu''' \sqcup \mu''$ is μ''', which is different from ν'''.

■

In the following we shall use the term speed-independent for an ALTS (STD) if it is *both* speed-independent-1 and speed-independent-2, and similarly for semi-modular and distributive.

It should be obvious from the lattice theory definitions given above that the following inclusion holds for the classes of ALTSs (STDs): distributive $\subset$ semi-modular $\subset$ speed-independent.

3.2 Modeling Asynchronous Logic Circuits

In this section we will show how "real" asynchronous circuits, built out of gates and wires, fit as a special case of our Binary Asynchronous Control Structures and State Transition Diagrams.

We will use two different delay models, pure and inertial, to describe the behavior of the circuit, and characterize circuit properties such as hazards in terms of ALTS properties such as local confluence.

3.2.1 A Low-level Model for Asynchronous Logic Circuits

Here, as in Section 3.1.1, we describe a circuit as the conjunction of a *structure* (a graph) and a *behavior* (a set of logic functions and delays, see [88] and [83]).

An **asynchronous logic circuit** (also called simply a **circuit**) is a triple $\langle X, Z, F\rangle$:

- X is a set of m **input** (or **primary input**) **signals.**
- Z is a set of n **output signals.**
- $F = \{f_1, f_2, ..., f_n\}$ is a set of **circuit element functions,** where $f_i : \{0,1\}^{d_i} \rightarrow \{0,1\}$ computes the value of z_i as a logic function of d_i signals in $Y = X \cup Z$ (i.e. Y denotes the set of signals of the circuit).

In this definition we associate a node with each input of a circuit to model the delay of the input wires of the circuit.

The **fanin** of a circuit node is the set of nodes connected to its inputs. The **fanout** of a circuit node is the set of nodes connected to its outputs.

The set of **primary output signals** of the circuit is a subset of Z that denotes signals that are directly observable from the environment, i.e. the set of signals the behavior of the environment can depend on. All other output signals are also called **internal signals.** This distinction will become important in Chapter 5, where we will be concerned in circuits that are hazard-free with respect only to the externally visible signals (see also Section 2.7.3, where an internal hazard was not considered a problem).

The structure of a circuit can be represented by a directed graph with one node for each signal, and an arc connecting the node corresponding to each input of f_i with the node corresponding to y_i. The circuit is **combinational** if the graph is *acyclic,* **sequential** otherwise.

Structurally, a circuit is a special case of a BACS. The difference is that every structural **circuit component** of the circuit is uniquely associated with a single signal, thus implying that each component, v_i, has *only one output,* $\{y_i\} = O^Y(v_i)$. The value of this output can be characterized either by the value of the corresponding logic function f_i (if y_i is an output signal) or by the value of the signal itself (if y_i is an input signal).

A circuit is **initialized** if its initial state is defined, as a binary vector $s \in \{0,1\}^{n+m}$. A circuit is **autonomous** if $X = \emptyset$ (i.e. it has no input signals).

Figure 3.4.(a) describes a very simple autonomous circuit, where $Z = \{z_1, z_2, z_3\}$ and $f_1 = \overline{z_3}$, $f_2 = \overline{z_3}$, $f_3 = z_1 + z_2$.

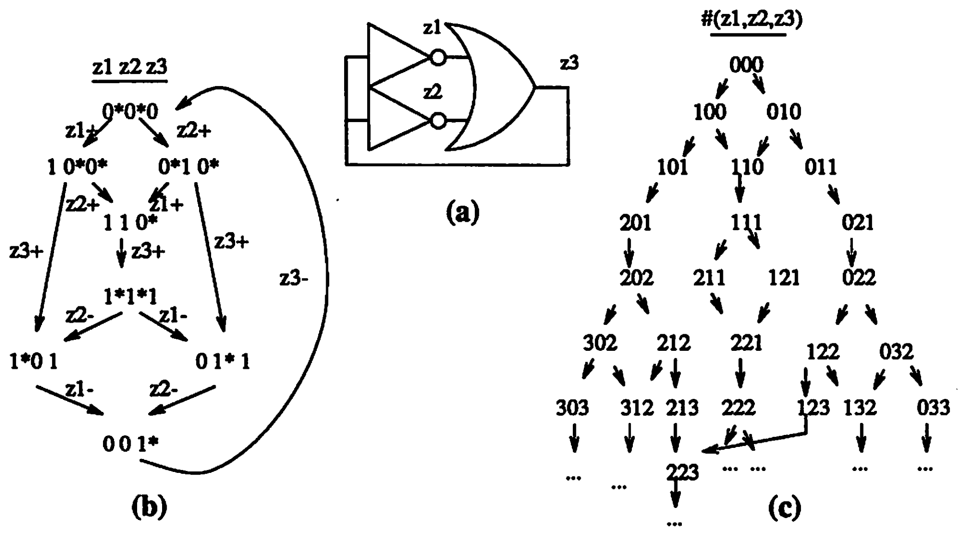


Figure 3.4: State Transition Diagram and Cumulative Diagram with inertial delay

A **gate** is an output node of a circuit. A gate is **combinational** if its function does not essentially depend on its output signal, otherwise it is **sequential**.

3.2.2 Delay Models of an Asynchronous Circuit element

Let a circuit $\langle X, Z, F\rangle$ be initialized in some state s. Every element z_i of the circuit is modeled as a delay-free logic function followed by an unbounded delay element[8]. For every element z_i we call it (and the corresponding output signal) **stable** in s if its current value $\lambda(s)_i$ is equal to the value of its function f_i. Otherwise we call it **excited**. We assume that if a signal is excited, then this signal *may* change its state after some finite time interval, which we call the **delay**. For example, in the circuit in Figure 3.4.(a), initialized in state 000, z_3 is stable while z_1 and z_2 are excited. What happens next, whenever z_1 or z_2 changes value, depends on whether we use the *pure* or *inertial* delay model.

Inertial Delay Model

In the *inertial delay* model (ID), an excited signal *may* change its state after a finite delay. This means that for any excited signal z_i there are two possibilities. One is that its value, 0 or 1, changes to the opposite, i.e. to 1 or 0, after a finite but unbounded amount of time. The other

[8]I.e. we use the unbounded gate delay model, but as usual the results can be extended to the unbounded wire delay model by using gates computing the identity function, or non-inverting buffers.

possibility is that the value of its function f_i is changed before z_i manages to change. In this case the output z_i of the element ceases to be excited and retains its previous value, which becomes stable (hence the term "inertial"). Speaking in more quantitative terms, the ID model means that if an element has a switching delay of d time units, pulses generated by the logic evaluator with duration less than d are filtered out, while pulses longer than d units appear at the output z_i shifted in time by d units (see Figure 2.1.(a)).

Since we are dealing with completely asynchronous circuits, we cannot precisely say whether in the second situation above the element has or has not produced a short pulse at its output. We shall therefore regard the behavior when f_i changes before z_i as anomalous (a *hazard*).

Pure Delay Model

We can alternatively assume that the delay block of each element is not inertial when it becomes excited, i.e. it cannot filter out the pulses whose duration is less than a given value d (this behavior is close to reality for long wire delays). Therefore, even though the function value changes before the output z_i has changed, the element remains excited, and just shifts in time the complete sequence of its "expected" output transitions (see Figure 2.1.(a)). With this *pure delay* model (PD), the value of the element in state s of the circuit, must be modeled by a pair, (r_i^s, τ_i^s), where the first component is the current binary value of z_i, i.e. $r_i^s \in \{0,1\}$, while the second component is the **excitation number** (recording how many excitations have been registered by the functional evaluator since the element was last stable), $\tau_i^s \in \{0,1,2,...\}$. In this model, the state of the circuit is a vector of length $|Y|$, with each component being of the above form.

We can now define an element z_i to be **stable**, according to the PD model, in state s if $\tau_i^s = 0$. Otherwise it is **excited**. The normal operation of the element is described by the following sequence of transitions: $(r_i^s, 0) \rightarrow (r_i^{s'}, 1) \rightarrow (\overline{r_i^{s''}}, 0)$. The anomalous operation (*hazard*), on the other hand, is described by the following sequences of transitions: either $(r_i^s, \tau_i^s) \rightarrow (r_i^{s'}, \tau_i^{s'}+1) \rightarrow (\overline{r_i^{s''}}, \tau_i^{s''})$ (if $\tau_i^s > 0$) or $(r_i^s, \tau_i^s) \rightarrow (r_i^{s'}, \tau_i^{s'} + 1) \rightarrow (r_i^{s''}, \tau_i^{s''} + 2)$.

Now, with this definition of the behavior of each gate, we can describe the operation of the entire circuit for both the ID and the PD models.

3.2.3 Circuit Behavior Description with Inertial Delays

Let us consider, as an example, the circuit shown in Figure 3.4.(a). If we assign the all zero vector as the initial state of this circuit, signals z_1 and z_2 are excited in this state. As usual, we designate this fact by labeling the value of an excited signal with an asterisk (*). Therefore the initial state is labeled as 0*0*0. Using the ID model of an element we can think about two possible states directly reachable from this state through the element normal switching behavior, 10*0* and 0*10*. Although signals z_1 and z_2 are excited concurrently and can switch independently, our interleaving semantics of concurrent actions requires that the first of the above two states is reached if signal z_1 changes before z_2 (and vice-versa). In both cases signal z_3 now becomes excited.

We can thus use a depth-first search procedure to generate the set of states reachable from the initial state. This set, together with the relation of direct reachability between states, can be represented by a graph, which satisfies our definition of an STD. Note that the transitions in this graph are labeled by the changes of signal values. Since the number of signals in the circuit is fixed ($|Y| = n + m$), it is obvious that the size of the STD, in terms of the number of its state labels, is bounded by 2^{n+m}.

Theorem 3.2.1 *The* STD *for any circuit, under the* ID *element model, is* deterministic, commutative *and* non-contradictory.

Proof The result follows directly from the ID model of an element and the uniqueness of the result of logic function evaluation for any given binary encoded state. ■

Therefore determinacy and commutativity are the intrinsic properties of the STD description for any circuit obtained using the ID model. This implies that *confluence* and *local confluence* are determined (up to sufficiency) by whether the circuit is *persistent.*

We can now define speed-independence, semi-modularity, and so on for a circuit modeled with inertial delays. Let $C = \langle X, Z, F \rangle$ be a circuit modeled with *inertial* delay and let $\mathcal{S} = \langle S, E, \lambda \rangle$ be its associated STD. According to the classification of Section 3.1.6 and Theorem 3.2.1:

1. C is **speed-independent** if the STD $\mathcal{S}$ is *confluent.*

2. C is **output-persistent** if it is speed-independent and for each pair of edges $sE(y_1^*)s'$ and $sE(y_2^*)s''$, if $y_1 \in Z$, then y_1^* is enabled in s'' (i.e. no output signal can ever be disabled).

3. C is **semi-modular** if the STD $\mathcal{S}$ is *persistent.*

4. C is **distributive** if the STD $\mathcal{S}$ is *strictly causal* and *persistent.*

Analogous definitions exist for a BACS (except for output-persistent).

Output-persistency guarantees that no transition of an output signal will ever be disabled, thus guaranteeing a correct behavior (recall that disabling a transition means possibly causing a spurious pulse on the signal).

Obviously
ID-distributive$\subset$ID-semi-modular$\subset$ID-output-persistent$\subset$ID-speed-independent.

Note that semi-modular as defined above is equivalent to the "operational" definition due to Muller ([88]). A circuit is called semi-modular with respect to a given state if the STD built from this state has no transition from a state where some z_i is excited to another state where z_i is stable but has the same value.

Also, note that Theorem 3.1.2 and 3.2.1 immediately imply that for a circuit with the ID model speed-independent-1 (confluent) is implied by speed-independent-2 (lattice), and similarly for semi-modular-1 versus semi-modular-2 and distributive-1 versus distributive-2.

The STD and an initial fragment of the CD for the circuit example in Figure 3.4.(a) are shown in Figure 3.4.(b) and (c). The STD is live, speed-independent but not semi-modular (persistency is violated in states 1*01 and 01*1, where z_2 and z_1 are disabled after the transition z_3^+ from states 10*0* and 0*10*, respectively). It is not strictly causal because states 10*0* and 0*10*, in which action $e[z_3^+]$ is enabled, can be reached from state 0*0*0 by two different paths. The STD is confluent but not locally confluent. Furthermore it is strongly connected, thus having a single final equivalence class, and it contains only non-transient cycles of states. The CD is a lattice (one can easily prove that every pair of cumulative states has its least upper bound in the CD), but not semi-modular. It has a zero element, the empty multiset (or all zero vector).

3.2.4 Circuit Behavior Description with Pure Delays

Throughout this section we will refer to properties of the circuit under consideration *when analyzed with the inertial delay model* by prefixing them with ID. Properties without the prefix, on the other hand, refer to the circuit analyzed with the pure delay model.

An asynchronous circuit can be analyzed with the pure delay model in a similar way as in the inertial delay case, by building its CD with a depth-first search procedure, from the initial state. According to the notation introduced earlier, each state, s, is labeled by a vector of $|Y| = n + m$ pairs (r_i^s, τ_i^s), where the first component is the binary value on the output of the delay block, z_i, and the second component is the *number of potential changes* that the element will generate on its own before it will become stable. Let us call such a vector the **excitation vector**. This graph does not satisfy the definition of an STD, given in Section 3.1.2, because the label is not binary. Nevertheless, it satisfies the definition of an ALTS, and again, due to the unique evaluation of the logic functions describing the circuit, we can make use of the fact that every state in set S is uniquely labeled by the excitation vector, and show that the following Theorem holds.

Theorem 3.2.2 *The* ALTS *for any circuit, under the* PD *element model, is* deterministic, persistent *and* non-contradictory.

Proof The result again follows directly from the PD model of an element and the uniqueness of the result of logic function evaluation for any given binary encoded state. ■

Equality between excitation vectors requires also equality of the second component, so *commutativity does not hold* in this case. However, due to the non-inertiality of elements behavior, we can claim *persistency*, because every element records its excitation, and cannot be disabled by changing the value at the output.

The latter detail drastically changes the role of the CD that can be built from the ALTS associated with the circuit. Such a CD is no longer a description that can be meaningfully used for characterizing the confluence properties of the circuit behavior.

The definition of a *bounded circuit* becomes crucial in such characterization. The PD model of a circuit is called k-**bounded** (or simply "bounded") if for every reachable state s in the associated ALTS, $\forall z_i : \tau_i^s \leq k$. An immediate consequence of boundedness is the *finiteness* of the ALTS of a PD-circuit.

The following Theorem is the implication of the fact that a PD-modeled circuit can accumulate unbounded "switching events" in its elements if its operation is cyclic. Recall that "ID-strongly live" means that the STD of the *same circuit*, modeled with *inertial delay*, is strongly live, and similarly for "ID-persistent" (Section 3.1.3).

Theorem 3.2.3 *The* PD *model of a circuit, which is* ID-strongly live *and* non-ID-persistent, *is* unbounded.

Proof In a strongly live, non-persistent ID model of a circuit there is no bound to the number of times the circuit can reach a state s where a signal y_i becomes disabled before it has a chance to fire. So every new arrival in this state will increment the corresponding τ_i^s. ■

We can now define speed-independence, semi-modularity, and so on for a circuit modeled with pure delays. Let $C = \langle X, Z, F \rangle$ be a circuit modeled with *pure* delay and let $\mathcal{S} = \langle S, E, \lambda \rangle$ be its associated STD. According to the classification of Section 3.1.6 and Theorem 3.2.1:

- C is **speed-independent** if the ALTS S is *finite* and *confluent.*
- C is **semi-modular** if the ALTS S is *finite* and *commutative.*
- C is **distributive** if the ALTS S is *finite, strictly causal* and *commutative.*

Analogous definitions exist for a BACS.

This definition and Theorem 3.2.3 imply the following important result.

Theorem 3.2.4 *The* PD *model of a circuit, which is* ID-strongly live, *is* PD-speed-independent *if and only if it is* ID-semi-modular[9].

This theorem in practice claims that for the PD model of an asynchronous circuit, speed-independence amounts to semi-modularity, if one considers a cyclically operating circuit, which is important for the synthesis of circuits operating in input-output mode. This result provides a crucial justification for the restriction to semi-modularity, when we look for necessary and sufficient conditions for the hazard-free, speed-independent implementation of a circuit.

Note also that this equivalence result strongly favors the use of semi-modularity rather than speed-independence as the characterization of a "correct" circuit in the ID case. The ID model can be too optimistic in many practical cases, so a design made for semi-modularity (or output-persistency) will be more "robust" with respect to technology changes, different implementations of other components of the system, and so on.

Obviously PD-distributive$\subset$PD-semi-modular.

Our circuit example in Figure 3.4.(a) generates the ALTS whose initial fragment is shown in Figure 3.5. This ALTS is persistent but non-commutative. It is infinite and not locally confluent. The PD-circuit is unbounded, therefore it is not speed-independent.

Our analysis of ID and PD models of circuits has an interesting by-product. It gives a formal, concise and general characterization of hazards in the circuit behavior.

3.2.5 Static and Dynamic Hazards

The traditional definition of hazards (Section 2.3.4) assumes two types of hazards, *static* and *dynamic*. Now we can examine how this behavior can be analyzed within the *pure* and *inertial* delay models.

[9]I.e. the same circuit modeled with inertial delays is semi-modular.

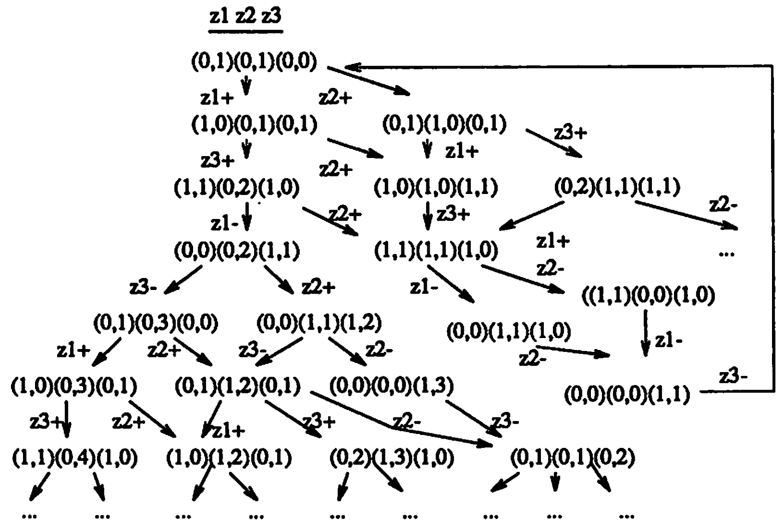


Figure 3.5: Arc-labeled Transition System with pure delay

For a given circuit, the ID model represented by its STD can only depict *static hazards*. They are present if the ID-circuit is *not persistent*. An element whose excitation is disabled, without switching is defined to have a static hazard. Strictly speaking, this is not a hazard in the "ideal" model, because the output of the delay block does not change. Due to the physical considerations above, though, this kind of situation can actually generate a spurious pulse on the output. Conversely, if the circuit is ID-semi-modular (or ID-output-persistent), then it is free from hazards.

For a given circuit, the PD model represented by its ALTS can describe all hazards. An element whose excitation number τ_i^s is greater than 1 has a hazard: the "standard" static hazard corresponds to $\tau_i^s = 2$, while the "standard" dynamic hazard to $\tau_i^s > 2$.

3.3 A High-level Behavioral Model for Asynchronous Systems

The previous sections discussed various inter-related models of Asynchronous Control Structures and logic circuits, and the relationship between model properties, such as confluence, and circuit properties, such as hazards.

In this section we develop a very general, *event-based*, model of BACS and circuit that, unlike STDs and Trace Models, has an explicit notion of *causality* and *concurrency*. So for example we will be able to distinguish the cases where events a and b are truly concurrent, independent of each other, and the case where either a can happen, and then cause b, or b can happen, and then cause a (an example of this distinction will be given in Figure 3.6).

The model, called Signal Transition Graph (STG), is based on interpreted Petri nets, and is a development of similar, but less general, models presented by [102] and [26].

In this section we establish relationships between the STG model and the lower level models described in the previous sections.

3.3.1 Arc-labeled Transition System and Cumulative Diagram of a Petri net

A marked Petri net $\mathcal{P} = \langle T, P, F, m_0 \rangle$ such that *no two transitions have exactly the same set of predecessor and successor places*[10] generates an Arc-labeled Transition System $\langle [m_0 >, E, T, \delta \rangle$ as follows. For each edge $(m_1, m_2) \in E$, where $m_1[t > m_2$, we have $\delta(m_1, m_2) = t$. Under this mapping, each Unique-action Relation class $[e]^t$, with $e = sE(t)s'$ corresponds to *a particular firing* of a transition t.

The following Theorem is an obvious consequence of the PN firing rule and of Theorem 3.1.1:

Theorem 3.3.1

1. *The* ALTS *corresponding to a marked* PN *is* finite *if and only if the* PN *is* bounded.

2. *The* ALTS *corresponding to a marked* bounded live PN *is* live.

3. *The* ALTS *corresponding to a marked* PN *is* deterministic *and* commutative.

4. *The* ALTS *corresponding to a marked* persistent PN *is* persistent. *Hence it is also* locally confluent *and* confluent.

As in Section 3.1.5, we can define the Cumulative Diagram (CD) of a Petri net, and analyze its properties as a lattice. This will be useful in order to establish the desired correspondence between PN properties and circuit properties.

According to [132], we define the **Cumulative Diagram** of a marked PN as follows. Given a firing sequence $m_0[t_1 > m_1[t_2 > \ldots m_n$ of a marked PN $\mathcal{P} = \langle T, P, F, m_0 \rangle$, the corresponding **firing vector** is a mapping $V : T \rightarrow \{0, 1, 2, \ldots\}$ such that for each transition t, $V(t)$ is the number of occurrences of t in the sequence.

Let $\mathcal{V}$ be the set of all firing vectors of $\mathcal{P}$. We define a mapping $\mu : \mathcal{V} \rightarrow [m_0 >$ that associates each firing vector with the final marking of the corresponding firing sequence. Note

[10]Such pairs of transitions are forbidden because they would create problems when defining the STD associated with an *interpreted* PN. For example they would make the enforcement of Property 3.1.1 rather difficult.

that the mapping is well defined, since in any marked PN the marking reached after a sequence of transition firings from m_0 depends only on the number of occurrences of each transition in the sequence, not on the *order* of occurrence.

The set $\mathcal{V}$, called the *Cumulative Diagram* of $\mathcal{P}$, was shown in [132] to be a partial order when we define $V_1 \sqsubseteq V_2$ if:

- $V_1(t) \sqsubseteq V_2(t)$ for all t and
- marking $\mu(V_2)$ is reachable from marking $\mu(V_1)$.

The following Theorem was proved in [12]:

Theorem 3.3.2 *The* ALTS *of a* PN *is* confluent *if the net is* free-choice, bounded *and* live.

The following Theorems were proved in [132]:

Theorem 3.3.3 *The* CD *of a marked* PN *is a* semi-modular *lattice with a zero element if the net is* persistent.

Theorem 3.3.4

1. *The* CD *of a marked* PN *is a* distributive *lattice with a zero element if the net is* safe *and* persistent.
2. *The* CD *of an* MG *is a* distributive *lattice with a zero element.*
3. *Let $\mathcal{P}$ be a* bounded PN *whose* CD $\mathcal{V}$ *is* distributive. *Then there exists a* safe *and* persistent PN $\mathcal{P}'$ *whose transitions are labeled with the transitions of $\mathcal{P}$ and whose* CD *is isomorphic to $\mathcal{V}$.*
4. *Let $\mathcal{P}'$ be a* safe persistent PN, *let $\mathcal{V}$ be its* CD. *Then there exists a safe* MG $\mathcal{P}''$ *whose transitions are labeled with the transitions of $\mathcal{P}'$ and whose* CD *is isomorphic to $\mathcal{V}$.*
5. *Let $\mathcal{S}$ be a* finite distributive ALTS *with alphabet of actions T. Then there exist*
   - *a* safe persistent PN $\mathcal{P}$ *and*
   - *a* safe MG $\mathcal{P}''$

   whose transitions are labeled with those in T and whose CDs *are isomorphic to the* CD *generated by $\mathcal{S}$.*

3.3.2 Signal Transition Graphs

An interpreted Petri net, where transitions represent changes of values of circuit signals, was proposed independently as a specification model for asynchronous circuits by [102] (where it was called Signal Graph) and [26] (where it was called Signal Transition Graph, STG). Both papers proposed to **interpret** a PN as the specification of a circuit $C = \langle X, Z, F\rangle$ (where Y denotes, as usual, $X \cup Z$), by labeling each transition with an element of $Y \times \{+, -\}$. A label y_i^+ means that signal $y_i \in Y$ changes from 0 to 1, and y_i^- means that y_i changes from 1 to 0, while y_i^* denotes either y_i^+ or y_i^-.

A **Signal Transition Graph** is a quadruple $\mathcal{G} = \langle \mathcal{P}, X, Z, \Delta\rangle$ where $\mathcal{P}$ is a *marked* PN, X and Z are (disjoint) sets of *input* and *output* signals respectively and $\Delta : T \rightarrow (X \cup Z) \times \{+, -\}$ labels each transition of $\mathcal{P}$ with a signal transition. An STG is **autonomous** if it has no input signals (i.e. $X = \emptyset$).

In order to simplify some of the practical design issues (see, for example, Sections 1.2.4 and 2.5.3), *unlabeled* or **empty transitions** can also be allowed in the STG. Their introduction in this chapter would unnecessarily clutter the notation, so we will not consider them here. The restriction does not imply a loss of generality, because it can be easily shown that a *free-choice* STG with empty transitions can always be translated into an *extended free-choice* STG without empty transitions, and vice-versa. Moreover, as shown in [10], all results proved for free-choice Petri nets are true also for extended free-choice ones, and vice-versa.

3.3.3 Signal Transition Graphs and State Transition Diagrams

Both [102] and [26] gave also synthesis methods to translate the PN into an STD (called Transition Diagram in [102] and State Graph in [26]) and hence into a circuit implementation of the specified behavior.

Given an STG $\mathcal{G} = \langle \mathcal{P}, X, Z, \Delta\rangle$ and the ALTS $\langle [m_0 >, E, T, \delta\rangle$ corresponding to its PN $\mathcal{P}$, we define the associated STD $\mathcal{S} = \langle [m_0 >, E, \lambda\rangle$ as follows. For each $m \in [m_0 >$, we have $\lambda(m) = s^m$, where s^m is a vector of signal values. Let s_i^m denote the value of signal y_i in marking m.

Obviously the STD labeling must be **consistent** with the interpretation of the PN transitions, so we must have for all arcs $e = (m, m')$ in the STD:

- if $\Delta(\delta(e)) = y_i^+$, then $s_i^m = 0$ and $s_i^{m'} = 1$.

- if $\Delta(\delta(e)) = y_i^-$, then $s_i^m = 1$ and $s_i^{m'} = 0$.
- otherwise $s_i^m = s_i^{m'}$.

Figure 1.3.(c) shows an example of the STD corresponding to the STG of Figure 1.3.(b).

An STG is defined as **valid** ([102]) if its STD is *finite* (i.e. the PN is *bounded*) and has a *consistent labeling*. In this work we will only consider valid STGs (otherwise their interpretation as control circuit specifications would lose its meaning).

One consequence of this requirement is similar to Property 3.1.1:

Property 3.3.1 *In a valid* STG, *for all firing sequences of its* PN, *the signs of the transitions of each signal alternate.*

Two STG markings m_1 and m_1' are **equivalent** if for each finite firing sequence $m_1[t_1 > m_2[t_2 > \ldots$ there exists a firing sequence $m_1'[t_1' > m_2'[t_2' > \ldots$ such that $\Delta(t_i) = \Delta(t_i')$ for all i. This relation partitions the set of reachable markings into equivalence classes. The equality between STG markings (and hence STD states) in the sequel will always be modulo this equivalence, which amounts to observable behavior equivalence.

An STG is **input free-choice** if its underlying PN is free-choice with respect to the set of transitions labeled with input signals.

An STG is **persistent** if for each reachable marking m_1, if t_1 is enabled in m_1 and $m_1[t_2 > m_2$, with $\Delta(t_1) \neq \Delta(t_2)$, then there exists a transition t_3 enabled in m_2 such that $\Delta(t_1) = \Delta(t_3)$. An STG is **output-persistent** if the above definition holds for all t_1 such that $\Delta(t_1) \in Z \times \{+, -\}$. Note that this definition is consistent with the interpreted PN definition of persistent transition. It is not related with the definition given by Chu ([27]), that will be denoted *Chu-persistent* and is discussed in more detail in Section 5.3.3.

This definition of STG persistence allows a case like that in Figure 3.6 to be treated as *persistent*, even though the underlying PN is *not persistent*. So PN persistency is a *stronger* condition than STG persistency. Only the transition labeling Δ that maps two different PN transitions into y^+ (and similarly for z^+) "erases" the distinction that was present between the PNs underlying Figures 3.6 and 1.3.(b), so that the two STGs generate isomorphic STDs. This Figure illustrates clearly the "semantic gap" arising from using a purely *interleaving* semantics (as in the STD) versus using a **true** *concurrent* semantics (as in the STG).

The following Theorem is a direct consequence of Theorems 3.3.2 and 3.3.3:

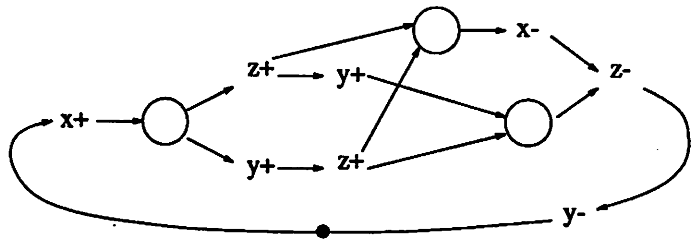


Figure 3.6: A Persistent Signal Transition Graph with non-persistent underlying Petri net

Theorem 3.3.5

1. *For every* finite deterministic, commutative *and* persistent STD S *there exists a valid* STG *whose* PN *is* persistent *and generates* S.
2. *For every* finite distributive STD S *there exists a valid* STG *whose* PN *is a* safe MG *and generates* S.
3. *For every* finite distributive STD S *there exists a valid* STG *whose* PN *is* safe, persistent *and generates* S.

3.3.4 Signal Transition Graphs and Asynchronous Circuits

Signal Transition Graphs were introduced to specify asynchronous circuits, a special case of Binary Asynchronous Control Structures. We are now ready to establish a correspondence between the STD associated with a valid STG and the ALTS associated with a BACS or a circuit.

Intuitively, our target is to implement the STG as a circuit with one signal for each STG output signal, where the logic function computed by each gate maps each STD binary label into the corresponding *implied value* for that signal. The **implied value** for signal y_i in state s^m is defined as the complement of s_i^m if y_i is excited in s^m, s_i^m otherwise. So for example if $s^m = 00^*1$ for signal ordering $y_0y_1y_2$, then the implied value of y_0 is 0, the implied value of y_1 is 1 and the implied value of y_2 is 1.

Both [102] and [26] recognized that an STG has an STD-isomorphic circuit implementation if (but not only if) the STD is *non-contradictory*. As a matter of terminology, [121] introduced the term Unique State Coding to denote an STG with a non-contradictory STD.

Chu also formulated a *necessary and sufficient* condition for the existence of a circuit implementation of a valid STG, calling it "a problem with state-assignment" ([27]). We will use the more explanatory term Complete State Coding (CSC), due to [87].

An STG has the **Complete State Coding property** if *all markings with the same binary label have the same set of enabled output signal transitions.* So we can state the following Theorem (a straightforward extension of the results of [27]).

Theorem 3.3.6 *Let S be the* STD *of a valid* STG. *Let $Y = X \cup Z$ be its set of signals. Let $\mathcal{C} = \langle V, H, Y, \rho \rangle$ be a* BACS *whose* STD *is isomorphic to S.*

The output signals Z of S can be characterized as logic functions of signals in Y if and only if the STG *has the* CSC *property.*

Proof If the STG has the CSC property, then the implied value rule defines a unique logic function between the set of STD labels and the value of each signal.

On the other hand, suppose that the STG does not have the CSC property. Then two states with the same label have a different implied value for some output signal z_i, and there is no logic function of the set of STG signals that can characterize z_i. ■

Theorem 3.3.6 means that each output signal of an STG can be implemented as a logic function if and only if its value and excitation in each STD state are uniquely determined by the STD state binary label itself.

We describe and solve in Chapter 4 the problem, given a valid free-choice STG $\mathcal{G}$ without the CSC property, to determine an externally compatible STG $\mathcal{G}'$ with the CSC property. External compatibility means that the set of traces of $\mathcal{G}'$, restricted to the signals of $\mathcal{G}$, is a subset of the traces of $\mathcal{G}$.

Note that non-contradictory STD (Unique State Coding) is only a *sufficient* condition for the implementability of a generic STG, but it becomes *necessary and sufficient* for the implementability of an *autonomous* STG, due to the following:

Corollary 3.3.7 *Let S be the* STD *of a valid* autonomous STG *(i.e. $X = \emptyset$).*

There exists an autonomous *circuit such that its* STD *is isomorphic to S if and only if all the states of S have distinct labels.*

3.4 Classification of Models of Asynchronous Circuits

We are now ready to proceed with the main topic of this chapter, a classification of the Signal Transition Graph models according to the type of Asynchronous Control Structure or asynchronous circuit they give origin to.

Note that the major focus of this section is the analysis of the "protocol of interaction" between the nodes in a BACS or a circuit, rather than a synthesis methodology for the internal structure of each node (described in Chapters 4 and 5). We are interested in checking when a given STG describes a speed-independent or semi-modular interaction between nodes, and we assume that each node can be realistically modeled using an instantaneous logic function and a pure delay or an inertial delay (see Sections 3.2.3 and 3.2.4). This point of view is interesting, for example, if the interconnections between nodes occur outside a single integrated circuit, so that the delays between them are large compared with the delays inside the circuit, and are outside of the direct designer's control.

The practical relevance of this section is that an STG that specifies a non-speed-independent behavior will almost certainly cause a malfunctioning implementation, since the behavior of the implementation will depend on the delays of the components. Similarly, an STG that specifies a non-output-persistent behavior will most likely be implemented with a circuit that suffers from hazard problems (due to the analysis in Section 3.2.4) unless a special care is devoted to its low-level design. Analogous considerations apply to the usefulness of having the STG describe a delay-insensitive behavior when the interconnection delays between the circuit components are widely variable and unpredictable.

3.4.1 Speed-Independence with Inertial Delay

Using the definitions from Section 3.2.3, we can now check, given an STG, if the associated BACS or circuit belongs to any of the speed-independence classes. We will give sufficient conditions for each class (necessary and sufficient conditions can be derived using the definitions directly on the STD, but they can be in general more expensive to evaluate).

The following Theorem is a result of Theorems 3.3.2, 3.3.3 and 3.3.4.

Theorem 3.4.1 *Let $\mathcal{G} = \langle \mathcal{P}, X, Z, \Delta \rangle$ be a valid* STG, *and let $\mathcal{S}$ be its associated* STD. *Let $\mathcal{A} = \langle V, H, Y, \rho \rangle$ be a* BACS, *modeled with* inertial *delay, such that its* STD *is isomorphic to $\mathcal{S}$ and node v_0 has $I^Y(v_0) = Z$ and $O^Y(v_0) = X$ (i.e. inputs Z and outputs X: this special node represents the* environment *where the circuit described by the* STG *will operate).*

- *$\mathcal{A}$ is* ID-speed-independent *if $\mathcal{P}$ is* free-choice *and* live.
- *$\mathcal{A}$ is* ID-semi-modular *if $\mathcal{P}$ is* persistent.
- *$\mathcal{A}$ is* ID-distributive *if $\mathcal{P}$* safe *and* persistent.

- *$\mathcal{A}$ is* ID-distributive *if $\mathcal{P}$ is a* Marked Graph.

Obviously if the STG has the CSC property and it has a circuit implementation due to Theorem 3.3.6, then the above results hold for the circuit as well, because it is just a special case of BACS.

The following Theorem is a result of Theorem 3.3.2.

Theorem 3.4.2 *Let $\mathcal{G} = \langle \mathcal{P}, X, Z, \Delta \rangle$ be a valid* STG *with the* CSC *property, and let $\mathcal{S}$ be its associated* STD. *Let $\mathcal{C} = \langle X, Z, F \rangle$ be a circuit, modeled with* inertial *delay, such that its* STD *is isomorphic to $\mathcal{S}$.*

- *$\mathcal{C}$ is* ID-output-persistent *if $\mathcal{P}$ is* live *and $\mathcal{G}$ is* input free-choice *and* output-persistent.

The following Theorem is a result of Theorems 3.3.4 and 3.3.5.

Theorem 3.4.3 *Let $\mathcal{A} = \langle V, H, Y, \rho \rangle$ be a* BACS, *modeled with* inertial *delay, and let $\mathcal{S}$ be its associated* STD.

- *if $\mathcal{A}$ is* ID-semi-modular, *then there exists an* STG *whose* PN *is* bounded *and* persistent, *and whose* STD *is isomorphic to $\mathcal{S}$.*
- *if $\mathcal{A}$ is* ID-distributive, *then there exists an* STG *whose* PN *is* safe *and* persistent, *and whose* STD *is isomorphic to $\mathcal{S}$.*
- *if $\mathcal{A}$ is* ID-distributive, *then there exists an* STG *whose* PN *is a* Marked Graph *and whose* STD *is isomorphic to $\mathcal{S}$.*

3.4.2 Speed-Independence with Pure Delay

We will now give conditions, similar to the previous section, in order to characterize a circuit implementation of an STG using the *pure* delay model (Section 3.2.4). We will only consider circuits that exhibit a *cyclic* behavior, that is whose operation does not "stop" after a finite number of transitions, because they have the greatest practical interest.

The following Theorem is a result of Theorems 3.3.2, 3.3.3 and 3.3.4.

Theorem 3.4.4 *Let $\mathcal{G} = \langle \mathcal{P}, X, Z, \Delta \rangle$ be a valid* STG *with a live* PN, *and let $\mathcal{S}$ be its associated* STD. *Let $\mathcal{A} = \langle V, H, Y, \rho \rangle$ be a* BACS, *modeled with* pure *delay, such that its* ALTS *is finite and isomorphic to $\mathcal{S}$, and node v_0 has $I^Y(v_0) = Z$ and $O^Y(v_0) = X$ (i.e. inputs Z and outputs X: this special node represents the* environment *where the circuit described by the* STG *will operate).*

- *$\mathcal{A}$ is* PD-speed-independent *and* PD-semi-modular *if $\mathcal{P}$ is* persistent.
- *$\mathcal{A}$ is* PD-distributive *if $\mathcal{P}$* safe *and* persistent.
- *$\mathcal{A}$ is* PD-distributive *if $\mathcal{P}$ is a* Marked Graph.

Obviously if the STG has the CSC property and it has a circuit implementation due to Theorem 3.3.6, then the above results hold for the circuit as well, because it is just a special case of BACS. Theorem 3.4.3 holds also in the pure delay case.

3.4.3 Delay Insensitivity

Informally, a *delay-insensitive* circuit operates correctly under the inertial gate and wire delay model in Input-Output Mode (see also Section 2.6). A more formal definition was given by Udding in [116], using Trace theory.

Let $\mathcal{A} = \langle V, H, Y, \rho \rangle$ be a BACS, let $\mathcal{S} = \langle S, E, \lambda \rangle$ be its associated STD, and let $\mathcal{T} = \langle A, \Sigma \rangle$ be its associated Trace Model. Let $\nu(y)$ for signal $y \in Y$, denote the node $v \in V$ such that $y \in O^Y(v)$ (i.e. y is an output of v). Let v_0 be a special node qualified as "the *environment*" (this distinction is necessary to speak about "output persistency"). The delay insensitivity of $\mathcal{A}$ is captured by the following four rules (given in [116] and adapted to our notation):

R_0 For all $s \in A^*$ and $y_1^* \in A$, we must have $sy_1^*y_1^* \notin \Sigma$, i.e. Property 3.1.2 must hold for $\mathcal{T}$.

R_1 For all $s, t \in A^*$ and $y_1^*, y_2^* \in A$, such that $\nu(y_1) = \nu(y_2)$ (i.e. both are outputs of the same node), we must have $(sy_1^*y_2^*t \in \Sigma) \Leftrightarrow (sy_2^*y_1^*t \in \Sigma)$. So signals that are output of one node cannot be ordered (i.e. they commute).

R_2 This rule, dealing with commutativity, takes two forms, in decreasing order of strictness:

R_2' For all $s, t \in A^*$ and $y_1^*, y_2^* \in A$, such that $\nu(y_1) \neq \nu(y_2)$ (i.e. outputs of different nodes), if $sy_1^*y_2^* \in \Sigma \wedge sy_2^*y_1^* \in \Sigma$ then $(sy_1^*y_2^*t \in \Sigma) \Leftrightarrow (sy_2^*y_1^*t \in \Sigma)$. So no change in the ordering between two transitions produced by two different nodes can change the subsequent behavior of *any* node. This rule is satisfied if $\mathcal{S}$ is *commutative*.

R_2'' For all $s, t \in A^*$ and $y_1^*, y_2^*, y_3^* \in A$ such that $\nu(y_1) \neq \nu(y_2)$ and $\nu(y_3) \neq \nu(y_2)$ (i.e. y_2 is the output of a node different from y_1 and y_3) if $sy_1^*y_2^*ty_3^* \in \Sigma \wedge sy_2^*y_1^*t \in \Sigma$ then $sy_2^*y_1^*ty_3^* \in \Sigma$. So no change in the ordering between two transitions produced by two different nodes can change the subsequent behavior of *another* node (i.e. y_1 and y_2 can "have memory" of this change in the ordering, but no-one else can).

R_3 This rule, dealing with persistence, takes three forms, in decreasing order of strictness:

- R_3' For all $s \in A^*$ and $y_1^*, y_2^* \in A$, $y_1^* \neq y_2^*$, if $sy_1^* \in \Sigma \wedge sy_2^* \in \Sigma$ then $sy_1^*y_2^* \in \Sigma$. This rule is equivalent to S being *persistent*.
- R_3'' For all $s \in A^*$ and $y_1^*, y_2^* \in A$, $y_1^* \neq y_2^*$, such that either $\nu(y_1) \neq v_0$ or $\nu(y_2) \neq v_0$, if $sy_1^* \in \Sigma \wedge sy_2^* \in \Sigma$ then $sy_1^*y_2^* \in \Sigma$, i.e. only transitions produced by the environment can disable each other. This rule is similar to the idea of output persistency.
- R_3''' For all $s \in A^*$ and $y_1^*, y_2^* \in A$, $y_1^* \neq y_2^*$, such that $\nu(y_1) \neq \nu(y_2)$, if $sy_1^* \in \Sigma \wedge sy_2^* \in \Sigma$ then $sy_1^*y_2^* \in \Sigma$, i.e. no transitions produced by different nodes can disable each other.

In cases R_3'' and R_3''' it is assumed that the implementation of nodes with mutually disabling transitions can do so without hazards, using appropriates circuit design techniques.

All the circuits that Udding considers must satisfy R_0 and R_1. In addition:

1. the circuits that satisfy R_2' and R_3' are called the **synchronization** class, C_1.
2. the circuits that satisfy R_2' and R_3'' are called the **data communication** class, C_2.
3. the circuits that satisfy R_2' and R_3''' are called the **arbitration** class, C_3.
4. the circuits that satisfy R_2'' and R_3''' are called the **delay-insensitive** class, C_4.

Obviously $C_1 \subset C_2 \subset C_3 \subset C_4$.

Now we can verify, given an STG, whether the BACS (or circuit, if it exists) implementing its output signals belongs to any of the delay insensitivity classes. An STG generates a Trace Model as the set of its firing sequences. So in principle a check whether an STG specification describes a system in C_1, C_2, C_3 or C_4 would require a check whether a potentially infinite set of traces satisfies the above rules.

Fortunately we can do better than that, and examine the STD generated by the STG, which is finite for valid STGs[10].

The following Theorem is a straightforward consequence of the PN firing rule:

Theorem 3.4.5 *Let* $\mathcal{G} = \langle \mathcal{P}, X, Z, \Delta \rangle$ *be a valid* STG, *and let* $\mathcal{S}$ *be its associated* STD. *Let* $\mathcal{A} = \langle V, H, Y, \rho \rangle$ *be a* BACS *such that its* STD *is isomorphic to* $\mathcal{S}$ *and node* v_0 *has* $I^Y(v_0) = Z$ *and* $O^Y(v_0) = X$ *(i.e. inputs* Z *and outputs* X*).*

[10]This can still be very costly, in practice, since the size of the STD can be exponential in the size of the STG.

R_0 *is automatically satisfied, since $\mathcal{G}$ satisfies Property 3.3.1.*

R_1 *is satisfied if and only if for all states m_1, m_2 and m_3 of S such that $m_1E(y_1^*)m_2E(y_2^*)m_3$ and $\nu(y_1) = \nu(y_2)$ there exist m_2' and m_3' such that $m_1E(y_2^*)m_2'E(y_1^*)m_3'$, and $m_3 = m_3'$ (modulo the equivalence described in Section 3.3.2).*

R_2 :

R_2' *is satisfied if and only if for all states m_1, m_2, m_3, m_2' and m_3' of S such that $m_1E(y_1^*)m_2E(y_2^*)m_3$ and $m_1E(y_2^*)m_2'E(y_1^*)m_3'$, we have $m_3 = m_3'$ (modulo the equivalence described in Section 3.3.2).*

R_2'' *is satisfied if and only if for all pairs of simple paths $m_1E(y_1^*)\ m_2E(y_2^*)\ m_3E(y_3^*) \ldots m_{i-1}E(y_{i-1}^*)\ m_iE(y_i^*)\ m_{i+1}$ and $m_1E(y_2^*)\ m_2'E(y_1^*)\ m_3'E(y_3^*) \ldots m_{i-1}'E(y_{i-1}^*)\ m_i'$ we have $E(y_i^*)$ enabled in m_i'.*

R_3 :

R_3' *is satisfied if and only if $\mathcal{G}$ is* persistent.

R_3'' *is satisfied if and only if $\mathcal{G}$ is* output-persistent.

R_3''' *is satisfied if and only if for all states m_1, m_2, m_2' of S such that $m_1E(y_1^*)m_2$ and $m_1E(y_2^*)m_2'$ and $\nu(y_1) \neq \nu(y_2)$ there exist m_3 and m_3' such that $m_2E(y_2^*)m_3$ and $m_2'E(y_1^*)m_3'$.*

Note that our example of a fair arbiter, described in Figure 3.1, does not even belong to the delay insensitive class, C_4. Its STD fails on rule R_2'', if we identify y_1^* with R_a, y_2^* with R_b, y_3^* with A_a, and t with the empty trace.

3.5 Signal Transition Graphs and Change Diagrams

Change Diagrams, described in more detail in [125, 60, 62], are an event-based model for asynchronous circuits that bears some resemblance to Signal Transition Graphs, but has some interesting properties of its own. In this section we compare the two models, and show how, when we limit ourselves to semi-modular circuits, they have similar modeling power. So the choice between the two is just dependent on the need for choice modeling (as in output-persistent circuits) and on the availability of good analysis and synthesis algorithms.

The definition of Change Diagrams is based on two types of precedence relations between transitions in asynchronous circuits.

1. the *strong precedence relation* between transitions a^* and b^*, usually depicted by a solid arc in the graphical representation of Change Diagrams, means that that b^* *cannot occur without the occurrence of* a^*.

2. the *weak precedence relation* between transitions a^* and b^*, usually depicted by a dashed arc in the graphical representation, means that b^* *may occur after an occurrence of* a^*. But b^* may also occur *after some other transition* c^*, which is also weakly preceding b^*, without the need for a^* to occur.

A **Change Diagram** is therefore formally defined as a quintuple $(A, \rightarrow, \vdash, M, O)$:

- A is a set of **transitions or events**, generally labeled with transitions of a set of signals Y (as in Section 3.3.2).

- $\rightarrow \subseteq (A \times A)$ is the **strong precedence relation** between transitions.

- $\vdash \subseteq (A \times A)$ is the **weak precedence relation** between transitions.

- $M \subseteq (\rightarrow \cup \vdash)$ is a set of **initially active arcs.**

- $O \subseteq \rightarrow$ is a set of so-called **disengageable arcs.**

The relations $\rightarrow$ and $\vdash$ are mutually exclusive (i.e. $(a^*, b^*) \in \rightarrow$ implies that $(a^*, b^*) \not\subseteq \vdash$ and vice-versa), and all the predecessor arcs of a transition a^* must be either of the *strong* type or of the *weak* type. Hence the set of transitions A is partitioned into **AND-type** transitions (with strong predecessors) and **OR-type** transitions (with weak predecessors).

The firing rule of Change Diagrams is similar to that of PNs, with arcs playing the role of places and flow relation elements at the same time. Each arc is assigned an integer **activity** which, unlike PN marking, can be *negative.* Initially each arc in M has activity 1, and each arc not in M has activity 0.

- An *AND-type* transition is **enabled** if *all* its predecessor arcs have activity greater than 0.

- An *OR-type* transition is **enabled** if *at least one* predecessor arc has activity greater than 0.

When an enabled transition fires, the activity of each predecessor arc is decremented, and the activity of each successor arc is incremented. The activity of a predecessor arc $a^* \vdash b^*$ of an *OR-type* transition b^* can become negative as a consequence of a firing of b^* due to positive activity on some other arc $c^* \vdash b^*$. It can then become positive again when a^* fires in turn.

A Change Diagram is **bounded** if the activity on each arc is bounded (both above and below) in all possible firing sequences.

Each *disengageable arc* is "removed" from the Change Diagram after the *first firing* of its successor transition. They are used to represent the *initialization sequence* of a circuit, and we will not enter into details concerning their usage.

Following [125], a *State Transition Diagram* $\mathcal{S} = \langle S, E, \lambda \rangle$ can be associated with a Change Diagram, as we did in Section 3.3.2 for STGs. Let S be the set of **reachable activity vectors** (similar to PN markings). An arc $(s, s') \in E$ joins two activity vectors $s, s' \in S$ if there exists a transition $y_i^* \in A$ that is enabled in s and whose firing produces s'. As usual, we need a *consistent labeling*, so for each arc $(s, s') \in E$ corresponding to transition y_i^* we must have:

- $\lambda(s)_i = 0$ and $\lambda(s')_i = 1$ for an arc associated with y_i^+.
- $\lambda(s)_i = 1$ and $\lambda(s')_i = 0$ for an arc associated with y_i^-.
- otherwise $\lambda(s)_i = \lambda(s')_i$.

A Change Diagram is **correct** if it satisfies the following conditions, ensuring that the above labeling is consistent:

- for all firing sequences, the signs of the transitions of each signal alternate (similar to Property 3.3.1).
- no two transitions of the same signal can be concurrently enabled in any reachable activity vector.
- the Change Diagram is connected and *bounded* (i.e. the set S is *finite*).

The main theoretical result concerning Change Diagrams is stated in [125] (the proof will appear in [63]). A **transient cycle** in an STD is defined as a cycle where at least one signal is continuously excited with the same value (see, for example, the cycle $s1, s2, s3, s4$ for signal R_b in Figure 3.1).

Theorem 3.5.1 *Each* semi-modular STD *without transient cycles has a corresponding* correct *Change Diagram.*

Each correct *Change Diagram has a corresponding* semi-modular STD *without transient cycles.*

Thus Change Diagrams are *equivalent in modeling power* to semi-modular STDs without transient cycles, and hence to most practical semi-modular circuits.

Change Diagrams are useful in practice because of the availability of low-complexity polynomial time *analysis* algorithms to decide, e.g.:

- whether a given Change Diagram is correct, and hence it can be used as a valid specification of a semi-modular circuit.
- whether a given circuit has a distributive Change Diagram, and hence a distributive STD. Note that this analysis, as outlined in Section 2.7.2, can be performed by direct construction of the Change Diagram, without going through the exponential size STD.

Furthermore synthesis algorithms from Change Diagrams to circuits in various technologies were outlined in [60].

The main limitation of Change Diagrams is their inability to describe *choice* among alternative behaviors, as modeled by places with more than one successor in PNs. So a designer faced with the description, for example, of a self-timed memory element, must describe the various possible read/write cycles of each data value as an *alternation* rather than a *choice* between them.

Given Theorems 3.4.3 and 3.5.1, we can see that there is a strong modeling similarity between Change Diagrams and STGs. Both can model all *semi-modular* circuits, that is a broad class of interest for asynchronous design. At the same time this similarity is only limited because STGs can describe choice, and there is a difference in the modeling power when *unbounded* PNs and Change Diagrams are considered.

Figure 3.7.(a) shows a cyclic finite Change Diagram with *unbounded* arc activity[11]. Such unbounded behavior, in which the i-th ($i = 1, 2, ...$) occurrence of event c is caused either by the i-th occurrence of a or by the i-th occurrence of b, is represented in Figure 3.7.(b) as a Change Diagram unfolding ([60]). In the unfolding each event a^i represents a *unique occurrence* of the corresponding event a in a firing sequence of the Change Diagram (similarly for b^i with respect to b and c^i with respect to c).

[11]Here events are not labeled with signal transitions for the sake of simplicity. It is possible to construct a correct interpreted Change Diagram showing the same type of behavior.

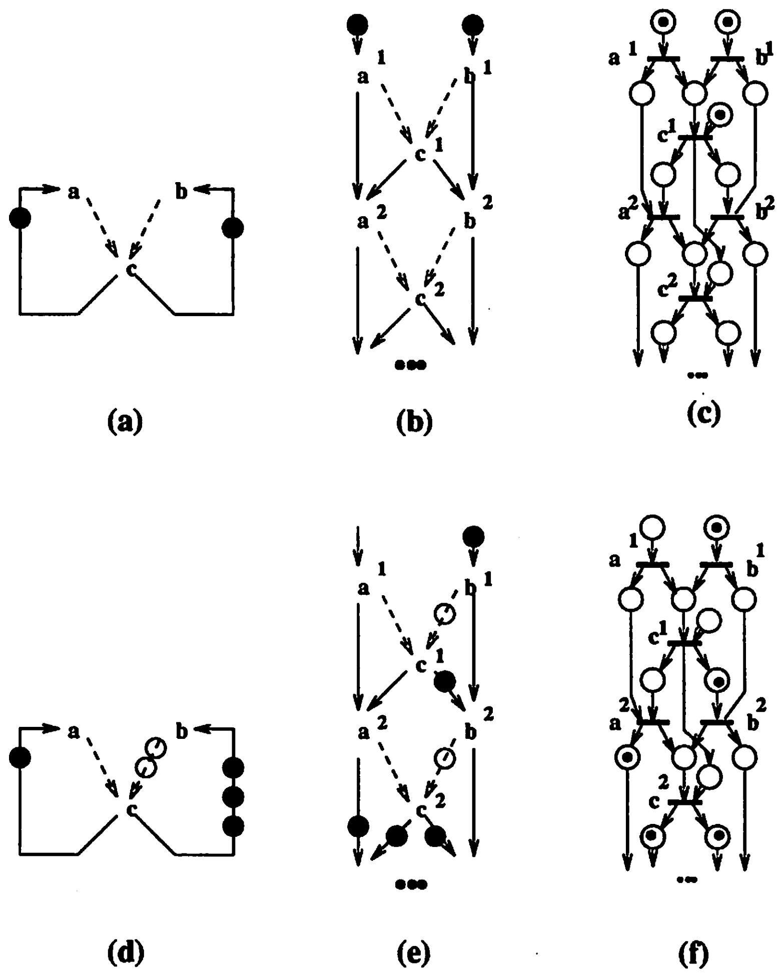
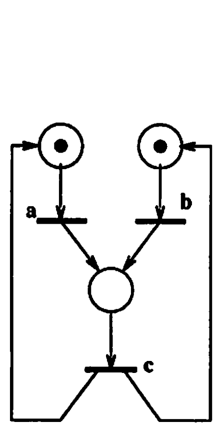
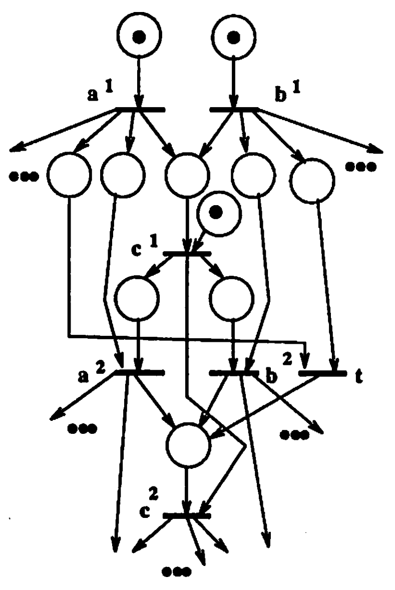


Figure 3.7: A Change Diagram without an equivalent finite Petri net



(a)



(b)

Figure 3.8: A Petri net only seemingly equivalent to Figure 3.7

The same behavior can be represented by the *infinite* PN in Figure 3.7.(c). We conjecture that there exists no finite PN representing it. The seemingly equivalent PN shown in Figure 3.8.(a) describes in effect a different behavior. Its "unfolding" into an execution net ([132]) in Figure 3.8.(b) shows that here the i-th occurrence of event c can be caused by any combination of pairs of the form a^k and b^{i-k} where k can be of any value between 0 and i. The difference between these behaviors is obvious.

The Change Diagram is able to remember the number of occurrences of events a and b, using the ***negative activity*** mechanism. So if a fires twice, as represented in Figures 3.7.(d), 3.7.(e) and 3.7.(f) (empty circles represent negative activity on the arc between b and c), and then it stops firing, c can fire again only after b has fired *three* times, in order to "re-absorb" the negative activity. On the other hand in Figure 3.8.(a), if a fires twice and then stops, c can begin firing again as soon as b fires, because there is no way to remember an *unbounded* "debt" of tokens.

As a final example, consider Figure 3.9.(a). It represents an initially one-place buffer that becomes two-place when event b occurs. The behavior is *semi-modular*, because no event is disabled. Yet there is no bounded equivalent Change Diagram, because bounded Change Diagrams can represent only semi-modular behaviors without *transient cycles*. In this example event b is continuously enabled during the cyclic firing of a and c. The Change Diagram shown in Figure 3.9.(b), whose behavior is only a subset (in terms of the Trace Model) of that of the PN

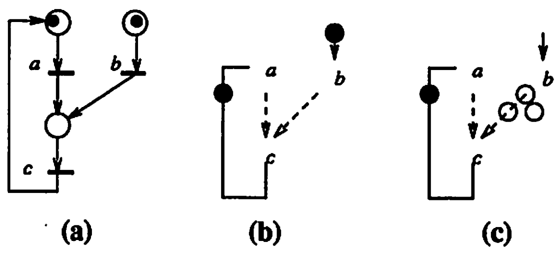


Figure 3.9: An unbounded Change Diagram with an almost equivalent bounded Petri net

in Figure 3.9.(a), has an *unbounded* negative activity on the arc between *b* and *c*. Figure 3.9.(c) shows such negative activity after the occurrence of *b*, followed by four occurrences of *a* and *c*. The difference in their behaviors begins after the occurrence of event *b*.

In summary, both Signal Transition Graphs and Change Diagrams can represent semi-modular circuits, and their modeling power differs only when it comes to describe unbounded behaviors (not interesting for speed-independent circuit design), transient cycles and *choice*. The designer can choose between them depending on the need, respectively, for an explicit representation of choice in Signal Transition Graphs and polynomial time analysis and synthesis algorithms in Change Diagrams.

3.6 Conclusion

Now we can examine how the designer can use the results of this chapter in the initial phase of the specification of an asynchronous circuit behavior using a *Signal Transition Graph*. First of all, we distinguish a *local view* and a *global view* of the system (Figure 3.10):

1. The **global view** is a *Binary Asynchronous Control Structure*, where all the STG signals are modeled. Such signals can belong to one of two classes, according to their behavior in the *State Transition Diagram* derived from the STG:

   - *output-persistent* signals, whose implementation can be automatically derived using the methods described in the following chapters,
   - *non-output-persistent* signals (e.g. some arbitration grant signals), whose implementation is produced using some other technique or by hand.

   The delay model used in the global view can be either *gate* or *wire delay*, depending on the technology (e.g. Figure 3.10 uses a gate delay model).

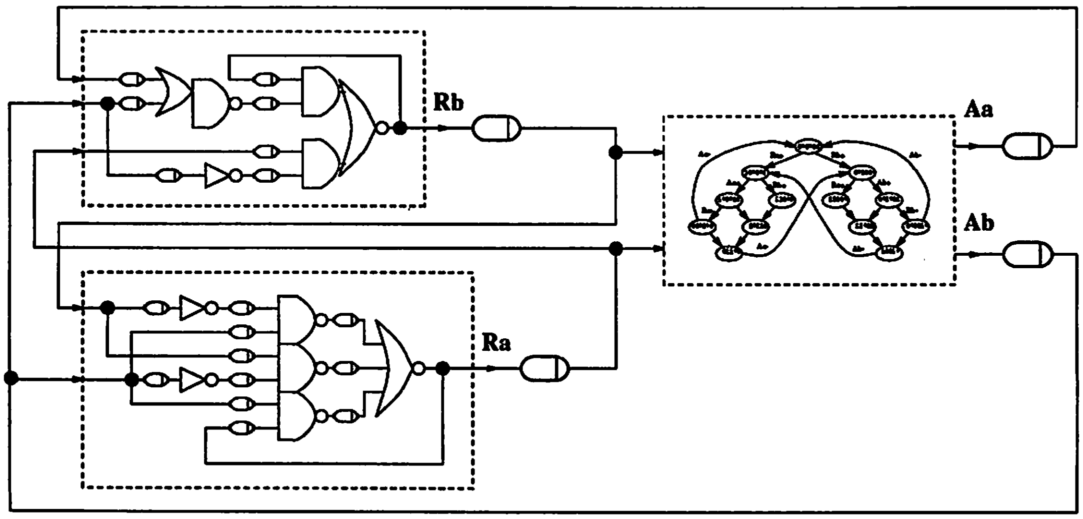


Figure 3.10: The global and local views of an asynchronous circuit

2. The **local view** is:

   - an *Input-Output Mode bounded wire delay* circuit, for output-persistent components (e.g. the components with output signals R_a and R_b in Figure 3.10[12]),
   - an *Arc-labeled Transition System* for non-output-persistent components (e.g. the component with output signals A_a and A_b in Figure 3.10).

   The local view uses a consistent, unified, verifiable model, because the STD can be interpreted as a *Finite Automaton*. Hence there are efficient techniques (see, e.g., Sections 2.7.3 and 2.7.4) to:

   - derive an STD description from a bounded wire delay circuit, and
   - merge STD descriptions of interconnected components into a single STD, and verify the latter against the original specification.

This decomposed model preserves modularity, thanks to the speed-independence or delay-insensitivity properties of the global view and allows the integration of:

- automated hazard-free design, optimized using delay bounds *both* on the circuit being designed (where they are required to produce a correct implementation, see Section 5.4) and, if available, on the environment, and

[12]The logic circuits appearing within these components serve illustrative purposes only, and do not implement any meaningful behavior.

- manual design of modules, such as arbiters, that cannot be designed using automated techniques, or whose efficient implementation is already known, e.g. from a cell library.

So the synthesis can proceed along the following steps:

1. The design process begins when the designer selects:
   - The circuit and delay models that must be used (Section 2.1), for example depending on:
     - the chip implementation technology, where *gate* delays or *wire* delays may dominate,
     - the partitioning of the system into chips, boards and cabinets, that may dictate to consider some wire or component delay to be *unbounded*,
     - the trade-off between modularity (when little is assumed on the environment where the circuit will operate) and performance (when some knowledge of the *delay bounds* can greatly help to improve the performance).
   - The class of behavior that best fits the synthesis algorithms and the system-level requirements. For example:
     - good synthesis algorithms with unbounded gate delays exist for *distributive* specifications ([126]),
     - using a purely *speed-independent* specification, rather than a *semi-modular* one, increases the risk of malfunctioning due to hazards, as we argued in Section 3.2.4,
     - on the other hand, there are inherently non-semi-modular behaviors, such as *arbitration*, which may be required by the type of application. In this case, one would like to use the *output-persistency* criterion to at least make sure that the parts of the system that need to be designed with standard logic gates will not be *inherently* (i.e. independent of the implementation style) prone to hazards.
   - The Petri net class (underlying the Signal Transition Graph specification) that offers the best trade-off between ease of analysis and descriptive power:
     - *Structural properties* (i.e. properties that depend on the graph underlying the PN, and not on its marking), such as being a *Marked Graph* or being *free-choice*, are very easy to analyze. Yet they can be used, for example, with Theorem 3.4.1 to design distributive circuits.

- *Behavioral properties* (i.e. properties that depend both on the graph and on the marking of the PN), such as being *live*, *safe*, *bounded* or *persistent*, are in general harder to compute, but the PN literature offers a wealth of efficient algorithms. They allow to use again Theorem 3.4.1 to design *semi-modular* circuits.
- Conversely, results like Theorem 3.4.3 show that if the designer chooses to describe, say, a *distributive* behavior, then it can be specified using an STG whose underlying PN is a *Marked Graph*.

2. Once the above choices have been made, the STG describing the desired behavior can be verified against the chosen properties, using the results from the literature summarized in this chapter.

3. An implementation is produced using the synthesis algorithms described in Chapter 5 (or, for example, in [126] and [8]).

4. The resulting circuit can be verified (using, for example, [39] or [61]) *against the very same properties selected in the first step*, since our framework provides a *uniform representation* for such properties both at the specification and at the implementation level.

Chapter 4

The State Encoding Methodology

As described in Section 3.3.4, a valid Signal Transition Graph specification can be implemented with a logic circuit if and only if the STG has the Complete State Coding property ([27], [87]). This, informally, amounts to say that the signals specified by the STG completely define the circuit state. Until now the burden of satisfying the CSC property has been placed mostly on the designer. Kondratyev *et al.* ([66, 133]) and Vanbekbergen *et al.* ([122]) addressed this problem, but only in the limited case of STGs whose underlying PN is a Marked Graph.

In this chapter we propose a new framework to satisfy the CSC property for a live, safe, free-choice, valid STG by solving a constrained state minimization problem and a critical race-free state encoding problem. We first give a procedure to derive from an STG an equivalent FSM representation that completely captures the state information implicit in the STG. Minimization of this FSM allows us to prove *necessary conditions* on the number of state signals required to implement it. This proposed framework is general enough to embed also previous methods to solve the CSC problem for some specific sub-classes of STGs, such as [122].

We also address how to extract from the minimized FSM *sufficient conditions* for the STG to have the CSC property. We propose to apply a critical race-free state encoding algorithm (such as the one proposed by Tracey [115]). Using the new states codes, we insert appropriate signal transitions in the STG. At this point, the STG can be implemented using the technique described in Chapter 5.

The initial research in the area of Unique State Coding enforcement (for example [133] and [121]) concentrated on the introduction of constraints within an STG whose underlying Petri net is a Marked Graph, using a sufficient condition as a guidance. Namely both [133] and [121] recognized that if all pairs of signals in the STG are *locked* using a chain of handshaking pairs, then the MG satisfies the Unique State Coding condition. We will briefly show in Section 4.3 how to interpret this sufficient condition within our framework, and how to use it to *reduce* the number of state signals.

Also the synthesis methodology developed by Martin *et al.* (see Section 2.6.1), even though it starts from a different specification formalism, basically guarantees CSC (called "program execution order") by handshaking, if possible, otherwise by heuristic state variable insertion ([76]).

Vanbekbergen *et al.* ([122]) proposed a technique to transform an STG whose underlying Petri net is a Marked Graph, so that it satisfies CSC. The technique is based on the identification of pairs of STD states that cause a CSC violation. Each pair of such states is connected by an edge in an undirected graph called the *constraint graph*[1]. Then coloring this graph provides, according to [122], a lower bound on the number of state signals required for CSC. This approach, even though it was originally proposed only for MGs, bears some resemblance with the framework that we propose for general STGs. In fact graph coloring has been used since long ago as a heuristic for FSM minimization ([127]). Vanbekbergen apparently did not recognize the need to use critical race-free encoding. Critical races would show up as further violations of CSC in the encoded STG. Thus the main advantage of our framework over [122] is the recognition that CSC falls within a much more general problem, previously known as *state minimization/critical race-free encoding* (Sections 2.3.2 and 2.3.3), and that constraint graph coloring can be considered only as a *heuristic technique* to solve the general problem.

Kishinevsky *et al.* ([60]), in the context of their Change Diagram synthesis methodology, also used graph coloring to identify regions of the State Transition Diagram that must be distinguished using state signals. They recognized the problem of critical races, but solved it through *iteration* of the encoding procedure.

A very general framework to solve the CSC problem for a general STD (i.e. not limited to those generated by MGs or FCPNs) was recently proposed by Vanbekbergen *et al.* ([124]). The authors recast the CSC problem as a **Boolean satisfiability problem**[2], by giving the conditions under which state signals can be added to the STD to ensure that it has CSC while keeping the "expanded" STD semi-modular.

The formulation is very interesting and general, but it seems premature to judge its practical utility. It is well-known ([44]) that many combinatorial optimization problems can be

[1]The authors also add constraints to allow optimization of the combinational logic implementing some particular signal, but this has no direct relevance to the CSC problem per se.

[2]I.e. the problem, given a product-of-sums Boolean expression, to find an assignment of values to the variables such that the expression evaluates to true.

transformed into instances of Boolean satisfiability, but the instances of the satisfiability problem derived from "real life" STGs may be too large to be efficiently solved. This is especially true since the known heuristic techniques to speed up the solution of satisfiability problems are often very dependent on the nature of the problem that is being reduced to satisfiability (e.g. stuck-at fault testing, in [70]) and may not be suited for this particular case.

Our approach, on the other hand, capitalizes on well-studied problems for which very efficient exact and heuristic methods have been found. Also note that the main focus of our approach is to produce as output a *Signal Transition Graph*, rather than a *State Transition Diagram* as [124]. The encoded STG can be used to *document* to the designer the changes made to the specification to make it implementable. Of course the same information could be reconstructed, in principle, from the STD, but it would be much harder to interpret, due to the much larger size (in general) of the STD.

The chapter is organized as follows. Section 4.1 gives an overview of the proposed approach. Section 4.2 describes how STGs are translated into FSMs. Section 4.3 describes the state minimization procedure. Section 4.4 describes how to insert state signals in the STG so that it can be implemented. Section 4.5 gives some experimental results.

4.1 Overview of the State Encoding Methodology

Our approach consists of several steps. First the STG is checked to see if it satisfies the CSC property. If it does, then we can directly synthesize logic from it, as shown in Chapter 5. If the STG does not satisfy the CSC property, then new *state signals* must be added to distinguish the STD states that have the same binary label but different output transitions. Our approach tries to *minimize the number of such signals*, and to *change the* STG as specified by the designer *as little as possible*.

We now need a few definitions to formalize the notions of "valid STG modifications" and "changing the STG as little as possible".

Throughout this chapter we restrict the class of Signal Transition Graphs to *"correct"* STGs, in order to prove the *correctness of the encoding procedure*. Chu ([27]) used the term *live* to denote a similar notion. We prefer the term *"correct"*, borrowed from the literature on the Change Diagram model (see e.g. [60]), to avoid confusion between the very different concepts of *live Petri net* and *live Signal Transition Graph*.

A Signal Transition Graph is defined to be **correct** if:

1. the underlying Petri net is *live*, *safe* and *free-choice*, and
2. its State Transition Diagram is *strongly connected*, and
3. for each signal z_i there exists *at least one* SM component, initially marked with exactly one token, such that:
   (a) it contains all transitions z_i^* of z_i,
   (b) each path from a transition z_i^* to another transition z_i^* (i.e. both rising or falling) contains also the complementary transition $\overline{z_i^*}$ (i.e. $z_i^+ \rightarrow z_i^- \rightarrow z_i^+$ or $z_i^- \rightarrow z_i^+ \rightarrow z_i^-$).

STG correctness is a *sufficient condition* to ensure that the STD derived from it has a *consistent labeling*:

Theorem 4.1.1 *A* correct STG *is* valid.

Proof For each signal z_i, there exists one SM component, initially marked with one token, to which all the transitions of z_i belong. By the results of Hack ([48]), this component remains marked with one token after any firing sequence of the STG. This ensures that the firing sequences are consistent (i.e. rising and falling transitions alternate), and hence that the value of z_i in each marking is unique.
■

Strong connectedness of the STD avoids pathological cases such as, for example, the initial marking p_1, p_4 of the FCPN in Figure 1.2. This marking is live and safe, but cannot be reached again through any firing sequence[3].

A transformation M from an STG $\mathcal{G}$ to an STG $\mathcal{G}'$ is **environment-preserving** if

- M preserves correctness (i.e. if $\mathcal{G}$ is correct, then $\mathcal{G}' = M(\mathcal{G})$ is correct), and
- M does not remove any constraint (i.e., any element of the flow relation) from $\mathcal{G}$, and
- M does not change the predecessors of any *input signal transition* in $\mathcal{G}$.

The insertion of internal signals and the modification of the predecessors of non-input signal transitions are allowed, because they do not modify the *environment behavior*. Furthermore, the restriction that no constraint can be *removed* from the specification ensures that the implemented behavior (in terms of *successful traces*) is a *subset* of the original specification. For example, forcing

[3]This marking, even though it marks every SM component of the PN exactly once, is *not* a live safe marking of any MG component, thus contradicting Theorem 3.15 of [27].

a particular ordering of two originally concurrent transitions may not be considered *desirable* (since it can reduce the total system throughput), but is in general considered *acceptable* in an implementation.

We will give a procedure that transforms any correct STG $\mathcal{G}$ given by the designer into a correct STG $\mathcal{G}'$ with CSC, by applying to it a transformation M such that:

1. if there exists a correct $\mathcal{G}''$ with CSC that can be derived from $\mathcal{G}$ with an environment-preserving transformation, then M is also environment-preserving.
2. otherwise, the amount of environment modification required by M is minimized, both exactly and heuristically.

We transform a given STG into an STG with CSC as follows. First we interpret the STD derived from the STG as an FSM, and apply to it classical *state minimization* and *state encoding* techniques. From this minimized encoded FSM we extract information sufficient to define how many state signals are needed and where their transitions must be inserted.

More in detail, the model behind the STG synthesis algorithms appears in Figure 4.1.(a): a combinational circuit with inputs *all the* STG *signals* and computing the implied value of all *the* STG *output signals*. If the STG does not have the CSC property, then the model fails, because the logic function that must be implemented by the combinational circuit is not well defined.

The desired relationship between the input and the output signals of the STG in this case can be described as a *sequential circuit*, rather than a *combinational* one, because now the sequential circuit has some additional "internal" state information (Figure 4.1.(b)). The behavior of this circuit is specified by an FSM representation, derived from the STG, that will be implemented in turn with a combinational circuit and some *state signals* (Figure 4.1.(c)). If we now consider also the state signals as output signals, we obtain an STG with a well-defined logic function for each non-input signal, that can be implemented using the techniques described in Chapter 5.

The key point in the approach is the correspondence between classical *state compatibility* and *Complete State Coding*. Distinguishing between incompatible markings amounts to forcing a state signal change between states in the minimized FSM.

There is an important difference between the approach presented here and classical FSM minimization techniques. We consider also the *output* signals as feedback signals, rather than only state signals. This allows to drastically reduce the number of states in the minimized FSM.

The procedure then becomes quite natural:

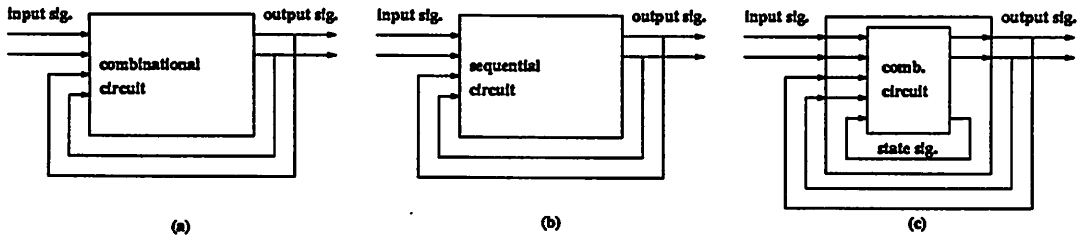


Figure 4.1: Circuit structures for state encoding

1. Derive from the STG an FSM with the input state defined as a binary vector of *both input and output signals*, and output state a binary vector of *only output signals*. The internal states correspond to the STD states.

2. Minimize the FSM so that we obtain a lower bound on the number of state signals required to implement the given STG specification (a *necessary condition*, independent of the synthesis methodology).

3. Encode the states of the minimized FSM so that there are no critical races.

4. Insert state signal transitions in the STG to distinguish between every pair of incompatible FSM states, and also every pair of STD states that violate the CSC property. We currently can prove only *sufficient* conditions to guarantee CSC using this procedure. These conditions then must guide the minimization process, to do the least amount of "damage" to the STG specification, especially in terms of the loss of concurrency.

4.2 From Signal Transition Graph to Finite State Machine

Once a State Transition Diagram is derived from the STG, as described in Section 3.3.2, we obtain an FSM representation of the STD as follows:

Procedure 4.2.1

1. *The set of internal states has one element s for each* STD *state (in the rest of the chapter we will liberally identify* STD *states and* FSM *states based on this one-to-one correspondence).*

2. *The set of input states has one element i for each binary label of an* STD *state (using both input and output signals).*

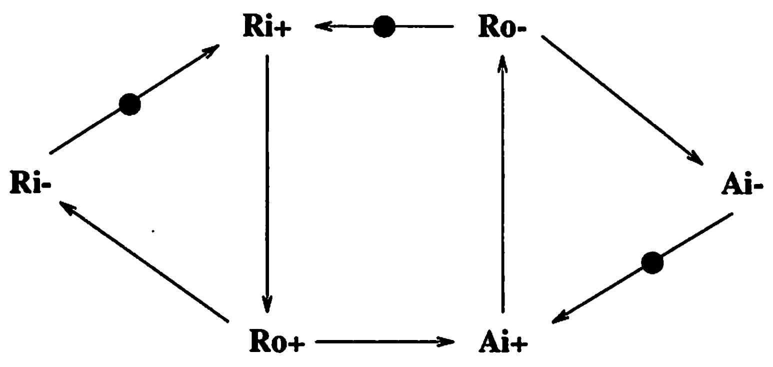


Figure 4.2: A Signal Transition Graph without Complete State Coding

3. *The set of output states has one element o for each implied value label of an* STD *state.*

4. *For each internal state s:*

   (a) *Let i be the binary label of the corresponding* STD *state.*

   (b) *Let $N(i, s) = \{s\}$, i.e. every state has itself as next state as long as no* STG *signal changes its value.*

   (c) *Let o be the implied value label of the corresponding* STD *state, i.e. the "next value" of the output signals.*

   (d) *Let $O(i, s) = \{o\}$.*

5. *For each edge among two* STD *states s' and s'':*

   (a) *Let i'' be the binary label of state s''.*

   (b) *Let $N(i'', s') = \{s''\}$.*

   (c) *Let o'' be the implied value label of s''.*

   (d) *Let $O(i'', s') = \{o''\}$.*

Figure 4.2 describes an STG, whose STD appears in Figure 4.3, that does not have the CSC property initially. States s_1 and s_5 are assigned the same binary label 110 but s_5 has an enabled output transition Ro^+ which s_1 does not have. State pairs causing CSC violations ((s_1, s_5), (s_2, s_6), (s_7, s_9) and (s_{10}, s_{12})) are identified by similar dotted and dashed lines.

An FSM for the STD of Figure 4.3 is given in Figure 4.4. For each state s_i in Figure 4.3 there exists a corresponding state i in Figure 4.4 with appropriate input and output states. For example, state s_1 of Figure 4.3, has binary label 110 (Ri = 1, Ai = 1, Ro = 0) and enables transitions

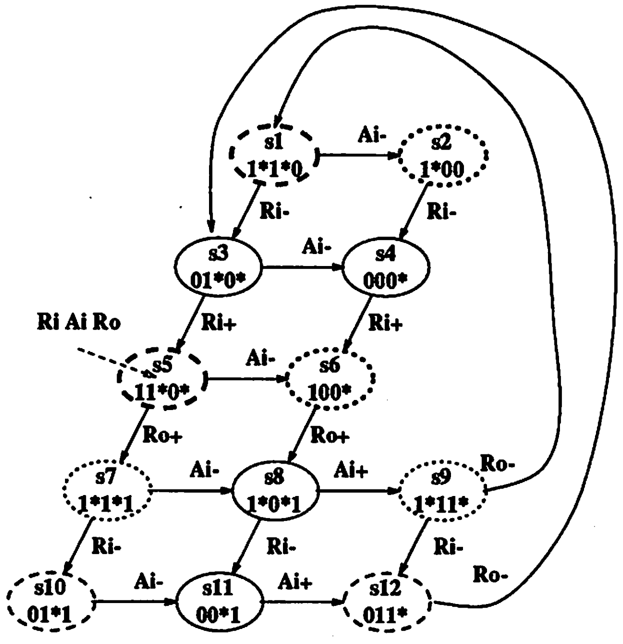


Figure 4.3: State Transition Diagram obtained from the Signal Transition Graph of Figure 4.2

Ai^- and Ri^-. It becomes state 1 in Figure 4.4, with a self-loop edge labeled 110/0, an edge to state 2 labeled 100/0 (corresponding to the firing of Ai^-), and an edge labeled 010/0 (corresponding to the firing of Ri^-). State 1 must produce output 0 on inputs 110, 100 and 010 because the implied value for signal Ro is 0 in states s_1, s_2 and s_3 (no transition of Ro^+ is enabled in any of them). Similarly state 5 must produce output 1 on input 110 because the implied value for signal Ro is 1 in state s_5 (Ro^+ is enabled in it).

The FSM is *normal* since each unstable state leads directly to a stable state. This is obviously true also of any FSM derived from an STD by Procedure 4.2.1.

Note that CSC violations translate into output-incompatibilities among states, namely (1,5), (2,6), (7,9), (10,12). For example, state 1 must produce output 0 on input 110, while state 5 must produce output 1 on the same input.

The relationship between this FSM and the STG can be seen in the following theorem:

Theorem 4.2.1 *An* STG *satisfies the* CSC *property if and only if the* FSM *derived from the* STG *according to Procedure 4.2.1 can be minimized to a single state.*

Proof

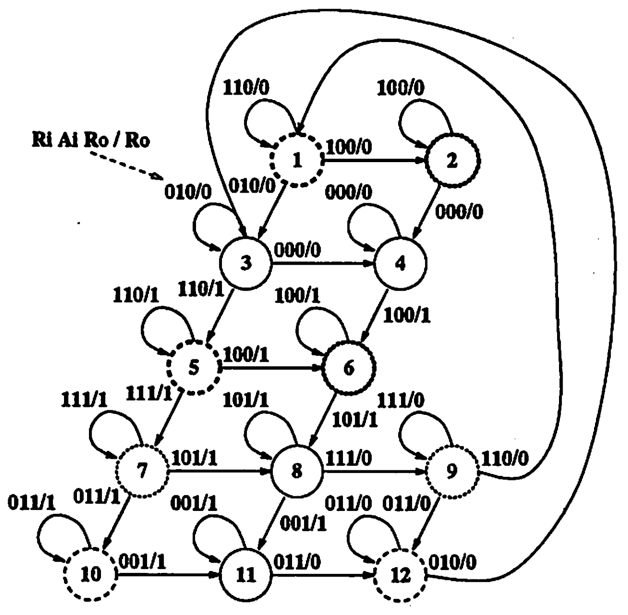


Figure 4.4: Finite State Machine corresponding to the State Transition Diagram of Figure 4.3

⇒ If an STG satisfies the CSC property, STD states with the same input label have the same implied output label, and pairs of FSM edges with the same input state have the same output state. Then all state pairs are output compatible, and there is a single compatible, which is of course closed.

⇐ If there is a single compatible, then pairs of FSM edges with the same input state must have the same output state. But then STD states with the same input label have the same implied output label, and the STG has the CSC property.

■

4.3 Constrained Finite State Machine Minimization

We now describe how the Finite State Machine obtained by direct translation of the State Transition Diagram, according to Procedure 4.2.1, can be minimized taking into account the need to derive a Signal Transition Graph with Complete State Coding.

This can be accomplished (as outlined in Section 4.1) by assigning each STG marking to a state in the minimized FSM. Then for each pair of adjacent markings belonging to two different states, we modify the STG to constrain some of the transitions, enabled in the second marking but not in the first one, to be enabled only after the transition of the relevant state signals. This corresponds to adding new internal signals and transitions to the initial STG.

Then the objective function that the minimization procedure must optimize is not only the number of state signals (as it was in the classical FSM minimization theory), but also the amount of "damage" done to the STG by the insertion of state signals:

- Increase in combinational logic size, due to the need to synthesize logic also for the state signals and (possibly) to the added constraints between signal transitions.
- Reduction in throughput, due to
  - the increase in propagation delay through a larger combinational logic, and
  - the added constraints between signal transitions.
- Constraints on input signal transitions. This, strictly speaking, is not a legal operation, because the behavior of the *environment* in general cannot be modified. In some cases, though, it may be acceptable, either if also the environment is being synthesized separately, or if it is known a priori that those constraints are already satisfied, for example because it is known that the environment is "slow" to react (an example of the latter case was discussed in Section 1.2.5).

  In any case, the procedures that we will give below are guaranteed to find a solution that *does not constrain the environment, if such a solution exists* within the search space.

Hence the objective function that the proposed minimization procedure optimizes is, in decreasing order of priority,

1. the number of state signals, then
2. the number of *input* signals in the STG whose transitions may be constrained to follow some other signal transition, then
3. the number of *output* signals in the STG whose transitions may be constrained to follow some other signal transition.

Given the generality of the proposed framework, though, it is possible to use different cost functions, e.g. to optimize the estimated area of the circuit.

The following result of [117] shows that using this framework we obtain *a lower bound on the number of state signals*:

Theorem 4.3.1 *Given any minimized* FSM *with s_i stable states under input label i, any sequential circuit realizing this* FSM, *in which stable* FSM *states are represented by stable circuit states, must have* at least $\lceil \log_2(\max_i(s_i)) \rceil$ *feedback signals.*

In our case every FSM state is stable, because it has a self-loop if no STG signals change. Furthermore every FSM stable state is represented by a stable circuit state (using the methodology described in Chapter 5), because if the synthesized circuit implements the STG specification, then after all the enabled transitions fire, they cannot be enabled again until some STG signal changes again.

Note that this lower bound may not be attained by the algorithms proposed in the following sections, because for example we may have to use more state signals in order to avoid critical races among them.

We can also reduce the number of state signals below the given bound if we are allowed to *remove some state* from the STD, because then we are modifying the FSM before realization. For example the method described in [121] adds constraints to the STG to remove from the FSM states that can cause incompatibilities, so that the resulting FSM has only one compatible and, by Theorem 4.2.1, the STG has CSC. However, this is not general and cannot always be done without adding state signals.

We need a few definitions to formally state the *constrained* state minimization problem.

A closed cover π of an FSM (see Section 2.3.2 for the terminology) is a **closed partition** of the states of the FSM if every state belongs to one and only one compatible (called **block**). A closed partition π is **derived** from a closed cover C if every block of π is a subset of a compatible of C.

Let $\mathcal{G}$ be a correct STG and let $\mathcal{F}$ be the corresponding FSM, derived by Procedure 4.2.1. Let D be a subset of the signals appearing in $\mathcal{G}$. Let π be a closed partition of the states of $\mathcal{F}$. Let π_i denote the block of π to which state s_i belongs.

A pair of *adjacent* states $s_1, s_2 \in \mathcal{F}$ belonging to *distinct blocks* $\pi_1 \neq \pi_2$ is **locally distinguishable using** D if for every transition t_i^*, such that

- t_i^* is enabled in s_2, and
- t_i^* is not enabled in any predecessor of s_2,

then the corresponding signal t_i is in D.

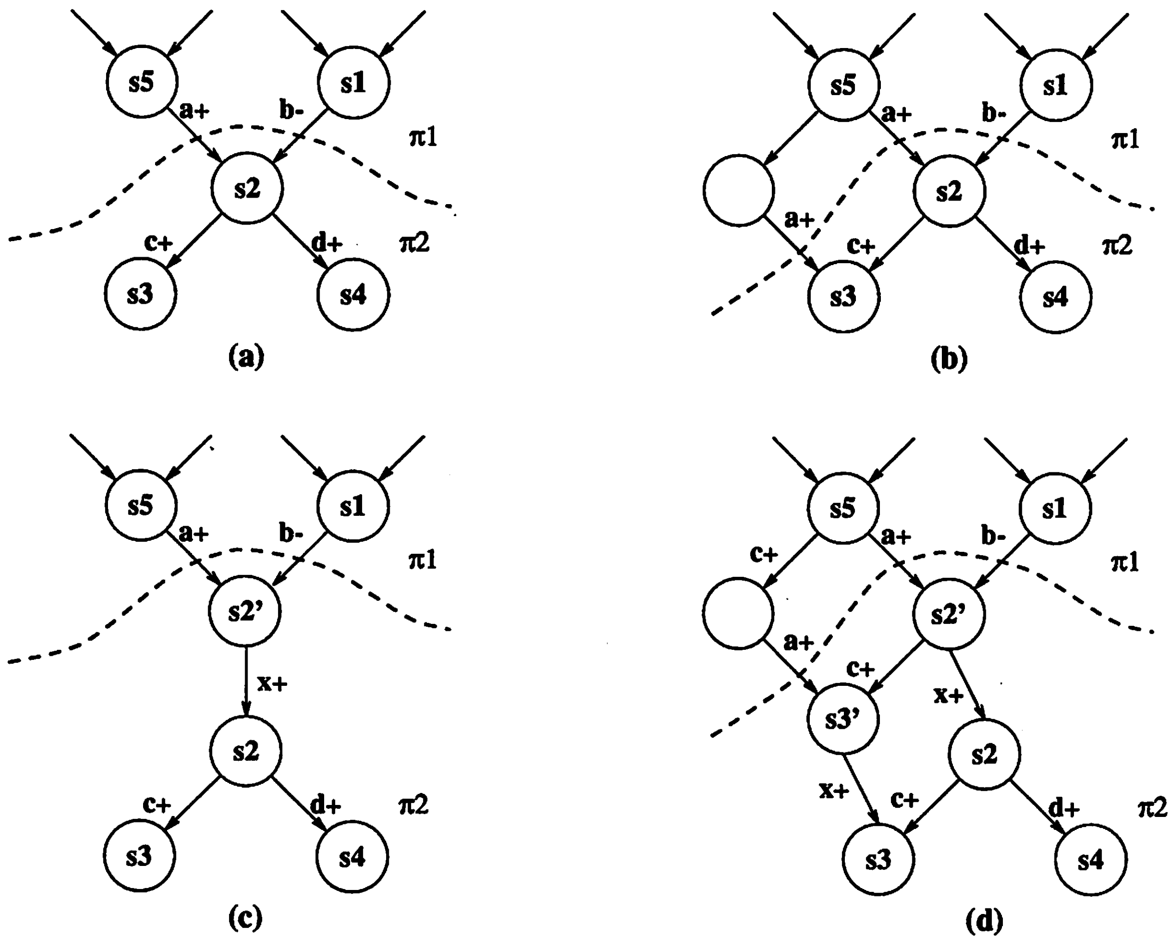


Figure 4.5: Locally distinguishable states

A pair of states s_1, $s_2 \in \mathcal{F}$ belonging to *distinct blocks* $\pi_1 \neq \pi_2$ is **distinguishable using** D if for every path from s_1 to s_2 and for every path from s_2 to s_1 there exists a pair of states that are locally distinguishable using D.

In Figure 4.5.(a), both signals c and d must be in D, for s_1 and s_2 to be locally distinguishable, because neither c^+ nor d^+ were enabled in any predecessor of s_2 (namely s_1 and s_5). So in order for the binary labels of s_1 and s_2 to have a different value of the state signals, c^+ and d^+ must be enabled only after some state signal transition has fired (x^+ in Figure 4.5.(c)). On the other hand in Figure 4.5.(b) only signal d must be in D, for s_5 and s_2 to be locally distinguishable, because c^+ was enabled in a predecessor of s_2, namely s_5.

A set D of STG signals is a **partitioning** set of signals with respect to π if every pair of states s_1, s_2 of $\mathcal{F}$ belonging to *distinct blocks* in π is *distinguishable* using D.

A partitioning set D is **minimal** if no signal can be removed from it, while remaining a partitioning set.

A partitioning set D is **optimal** with respect to a closed cover C if it is minimal and it has:

1. the minimum number of input signals n_i among all partitioning sets with respect to some closed partition π derived from C, and

2. the minimum number of output signals n_o among all such partitioning sets with n_i input signals.

A partitioning set D is **optimum** if it has the least cost (as in the definition of optimal) among all partitioning sets with respect to any closed cover C of $\mathcal{F}$.

This definition of optimality obviously takes into account the need not to constrain input signals and the need to constrain the least number of output signals, as was intuitively justified above.

In order to guarantee that the partitioning set that we obtain is *optimum* we need to find all the *prime compatibles* of the FSM, and then find a subset C of these such that:

- C is closed, and

- there exists an optimum partitioning set D whose associated closed partition π is derived from C.

Given the above partitioning set cost function, if the STG can be implemented with an *environment-preserving* transformation, then D will not include any input signal. Otherwise D constrains the minimum number of such signals.

Unfortunately the enumeration of all maximal compatibles is infeasible in most cases of practical interest. To obtain the initial closed cover we are currently using a heuristic classical state minimization procedure ([127]). We are interested in minimizing the cardinality of the cover, because the minimized FSM will have one state for each block in π. So minimizing the cardinality of C minimizes the cardinality of π, and hence the number of state signals.

When using a classical FSM minimization procedure we must also make sure that *pairs of adjacent states s_1, s_2, where every transition enabled in s_2 is also enabled in s_1*, belong to the *same compatibles*. Otherwise the partitioning set algorithm will not be able to find a solution, because s_2 has no transitions that are not enabled in s_1 and thus we cannot insert a state signal transition

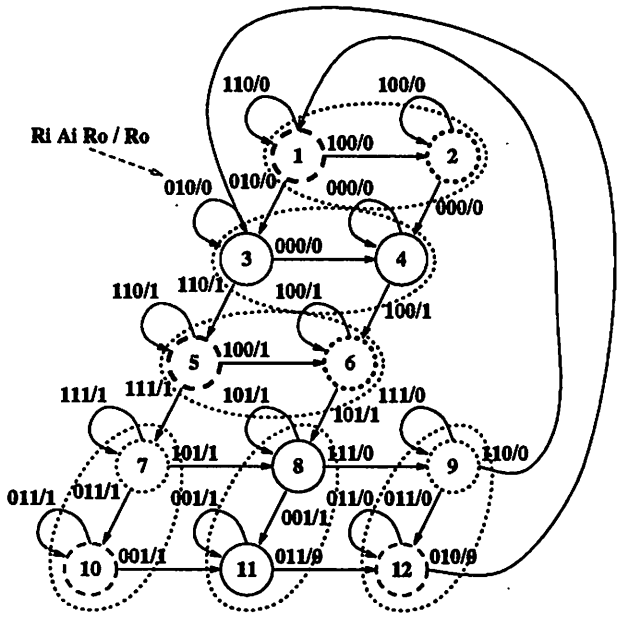


Figure 4.6: Pre-minimized Finite State Machine of Figure 4.4

to enforce a state change from s_1 to s_2. This can be accomplished by a *pre-minimization* step that iteratively merges all such state pairs. The pre-minimization merges only compatible state pairs, because in an FSM derived from an STG s_1 and s_2 are obviously output compatible, and the set of possible implications from s_2 is a subset of those from s_1.

The result of this pre-minimization step for the FSM of Figure 4.4 is shown in Figure 4.6. For example, in state 1 both Ri and Ai can have a falling transition. In its successor state 2 only Ri can have a falling transition, so the two can never be distinguished and they *must* belong to the same π_i in the final partition. Then we merge them before performing the actual FSM minimization. On the other hand, in state 3 not only Ai can have a falling transition, but also Ri can have a *rising* transition, different from the falling transition of Ri enabled in state 1. So states 1 and 3 cannot be merged.

We will now give an exact algorithm to find an *optimal* set of partitioning signals D and an associated closed partition π starting from an initial closed cover C (Section 4.3.1). We will also give a heuristic algorithm to find a *minimal* such set, in case the exact solution is too expensive to compute (Section 4.3.2).

4.3.1 Optimal Partitioning Set Derivation

The exact algorithm is divided into three steps:

1. Formulation, as a conjunction of logic expressions over a set of multi-valued variables, of the conditions for a set D of STG signals to be a partitioning set with respect to any closed partition π derived from a closed cover C.
2. Partial solution of the clauses, to find a partitioning set D of minimum cost.
3. Derivation of π from C and D.

Let C be the initial closed cover of an FSM $\mathcal{F}$ derived from a correct STG $\mathcal{G}$. As described above, a partitioning set D and the related closed partition π must satisfy the following conditions, expressed as a *conjunction* of clauses. The clauses depend on the following set of variables:

- A multi-valued variable S_s for each state s, with allowed values in $p_s = \{c \in C \mid s \in c\}$. This variable also expresses the fact that s must belong to one and only one block.
- A binary-valued variable y_i for each signal z_i defined by the STG, that has value 1 if z_i is in D, and 0 otherwise.

The following procedure builds the appropriate set of clauses:

Procedure 4.3.1

- *for each state* s_1
  - *for each successor* s_2 *of* s_1 *such that* s_1 *and* s_2 may *belong to different blocks, create a clause stating that:*
    * *either* s_1 *and* s_2 *are assigned to the same block,*
    * *or all the signals required to distinguish between them are in* D.

    I.e., let $D(s_1, s_2)$ *denote the set of signals required to distinguish between* s_1 *and* s_2. *Then the clause is of the form:*

$$(S_1 = S_2) \vee \left(\bigwedge_{z_i \in D(s_1,s_2)} y_i \right)$$

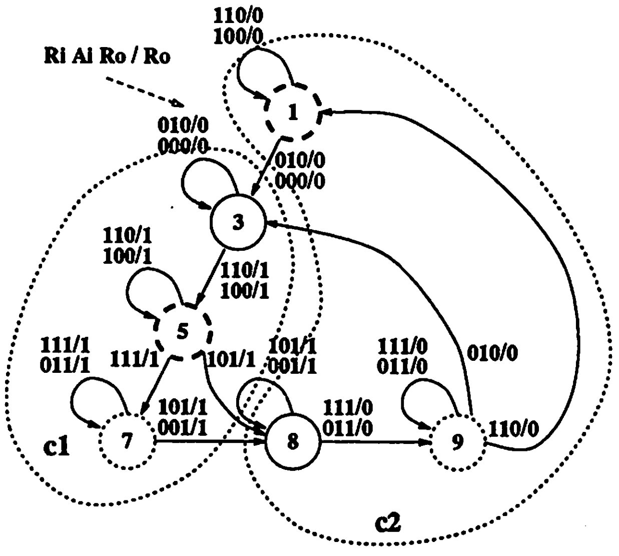


Figure 4.7: Closed cover of the Finite State Machine of Figure 4.6

The worst-case running time of this algorithm is $O(c^2n^2m)$ (m is the number of signals, n is the number of states, c is the number of compatibles). This is also a bound on the size of the conjunction of clauses.

The FSM in Figure 4.7 requires the introduction of the following clauses:

1. $(S_1 = c_2) \wedge (S_8 = c_2) \wedge (S_9 = c_2) \wedge (S_3 = c_1) \wedge (S_5 = c_1) \wedge (S_7 = c_1)$
   to express the admissible compatibles for each state (in this example there is no choice).

2. $(S_1 = S_3) \vee y_{Ri}$ to guarantee that s_3 is distinguishable from s_1, because the only signal enabled in s_3 and not in s_1 is Ri.

3. $(S_9 = S_3) \vee y_{Ri}$ to guarantee that s_3 is distinguishable from s_9.

4. $(S_5 = S_8) \vee y_{Ai}$ to guarantee that s_8 is distinguishable from s_5.

5. $(S_7 = S_8) \vee y_{Ai}$ to guarantee that s_8 is distinguishable from s_7.

One way to find a minimum cost partitioning set given the clauses defining it is to combine the approach described in [73] to solve the *binate covering problem* using binary decision diagrams with the multi-valued extension of binary decision diagrams.

A Multi-valued Decision Diagram (MDD, see [18] and [58]) is a rooted directed acyclic graph where:

- every leaf node is labeled with either the value 1 or 0.
- every non-leaf node is labeled with a multi-valued variable.
- every edge is labeled with one value of the variable corresponding to its source node.

An MDD represents a multi-valued logic function over the variables associated with the MDD labels in the following way. Every element of the domain of the function defines a unique path from the root to a leaf, by following edges labeled with the variable values associated with the domain element. The function has value 1 on all the points that correspond to paths to the leaf labeled with 1, and 0 on all the other points (they obviously correspond to paths to the leaf labeled with 0).

Conversely every path from the root to the leaf labeled with 1 defines a (possibly partial) assignment of values that makes the associated function evaluate to 1 (and similarly for 0).

So, given an MDD representing a conjunction of the clauses, any path from the root to the leaf labeled with 1 corresponds to a partial assignment of values to the variables that satisfies the clauses. Hence this partial assignment represents a family of partitioning sets and associated closed partitions.

We then assign a weight to each edge in the MDD, according to the cost function defined above, as follows:

1. zero for all edges whose source is an S_s variable (i.e. associated with a state).
2. zero for all edges corresponding to the 0 value of y_i variables (i.e. associated with signals).
3. one for all edges corresponding to the 1 value of y_i variables associated with *output* signals.
4. the number of output signals plus one for all edges corresponding to the 1 value of y_i variables associated with *input* signals.

Then, as shown in [73], a shortest path from the root to the leaf labeled with 1 corresponds to a minimum cost assignment that satisfies all the constraints. The proof was given for Binary Decision Diagrams, but since every MDD can be translated into a Binary Decision Diagram with an appropriate encoding and the weights assigned to the multi-valued variables are all zero, the result applies directly to this case as well. The (non-trivial) problem of optimal ordering of the MDD variables belongs to future research.

Now D consists of all the signals whose variables are assigned a value of 1.

In the example above, all the clauses could be satisfied by the assignment of each state to the only compatible it originally belonged to, and by an assignment of 1 to y_{Ai} and y_{Ri}. Note that the state signal insertion Procedure 4.4.1 will actually condition both those signals to follow state signal transitions, as shown in Figure 4.10.

The worst case running time of this step can be exponential in the number of variables, and hence in the number of states. So it can be doubly exponential in the number of STG signals, motivating the introduction of a heuristic algorithm (Section 4.3.2).

The assignment corresponding to a shortest path gives also a compatible for each state, that unfortunately cannot be used as its partition block, because the resulting partition *may not be closed*. In principle, we could add further clauses expressing the closure conditions, and use the shortest path formulation as shown above. Alternatively we can use an iterative algorithm, described below, which has a better worst-case bound.

Let $p_s = \{c \in C \mid s \in c\}$ be the set of initially allowed compatibles for state s according to the closed cover C. We can derive π with the following procedure:

Procedure 4.3.2

```
• Repeat as long as some compatible is removed by some $p_s$:
    1. for each state $s$
        – for each compatible $c \in p_s$
            (a) if $s$ cannot be distinguished with $D$ from all states $s'$ such that $c \notin p_{s'}$
                * remove $c$ from $p_s$
    2. for each input state $i$
        – for each compatible $c \in C$
            (a) let $p' = C$
            (b) for each state $s$ such that $c \in p_s$ and such that $N(i, s) = \{s'\}$
                * let $p' = p' \cap p_{s'}$
                (this step computes a closure condition under input $i$ for all successors of states
                that might be in $c$)
            (c) for each state $s$ such that $c \in p_s$ and such that $N(i, s) = \{s'\}$
                * let $p_{s'} = p'$
                (this step updates the allowed compatibles according to the above computation)
```

The output of this procedure is a closed cover, where every pair of states that *cannot be distinguished* with D are forced to belong to the same subset of compatibles. This is because step 1 guarantees that every state can be distinguished from any other state that does not belong to the same set of compatibles. Furthermore step 2 ensures that the compatibles satisfy the closure condition. Choosing any element of each p_s as the block of s gives the desired closed partition π.

The result of the above algorithm on the FSM of Figure 4.4 appears in Figure 4.7. In this case each state belongs only to one compatible, so there was no choice. The closed partition has two blocks, π_1 and π_2, corresponding to compatibles c_1 and c_2 respectively.

Note also that this algorithm can be used to *check whether a given set of signals is a partitioning set*, because of the following theorem.

Theorem 4.3.2 *Let $\mathcal{F}$ be an* FSM *derived from a correct* STG $\mathcal{G}$ *according to Procedure 4.2, let C be a closed cover of $\mathcal{F}$, let D be a set of signals of $\mathcal{G}$.*

Procedure 4.3.2 terminates with a non-empty p_s for all s if and only if D is a partitioning set *for some π that can be derived from C.*

Proof

$\Leftarrow$ Suppose that D is a partitioning set, with associated closed partitions π. Let π_s be the the block to which s belongs.

Then step 1 can never remove π_s from p_s, because s can be distinguished from all states that are not in π_s (including those s' for which $\pi_s \notin p'_s$), due to the definition of partitioning set.

Furthermore step 2 will not remove π_s from p_s, since π is closed.

$\Rightarrow$ Suppose that the algorithm terminates with a non-empty closed cover for all states. Then choosing the first (with any total ordering) compatible in each set gives a closed partition π. But all state pairs s and s' that had $p_s \neq p_{s'}$ are distinguishable (otherwise step 1 would have made them equal), so all state pairs that belong to different blocks are distinguishable, and D is a partitioning set.

■

A pessimistic analysis of the running time of this algorithm leads to a worst case bound of $O(c^2n^4)$, because in step 1 each state can be checked with a depth-first search on the FSM, so it is $O(cn^3)$, in step 2 each edge in the FSM needs to be checked only once, so it is $O(cn^2)$ (n is the number of states, c is the number of compatibles). Furthermore each iteration removes at least one allowed block from one state, so we can iterate at most $O(cn)$ times. In practice the algorithm is very fast even for large FSMs.

4.3.2 Minimal Partitioning Set Derivation

In case the exact algorithm given in Section 4.3.1 cannot find the optimal solution in a reasonable amount of space or time (as shown above, the critical step is finding a partitioning set of signals) we can:

- either try to reach an optimal solution with exhaustive search over all possible subsets of signals, at most 2^m times, using Procedure 4.3.2 to check if a given set of signals is a partitioning set. This has a bound of $O(2^m c^2 n^4)$, and it is not exponential in n (where m is the number of signals, n is the number of states, c is the number of compatibles).
- or use greedy search to find a *minimal* partitioning set, trying to remove each signal in fixed order.

In the latter case we can apply the following algorithm:

Procedure 4.3.3

```
1. let $D$ be the set of all signals in $\mathcal{G}$
2. for each signal $z_i$, beginning from input signals
    (a) if $D - \{z_i\}$ is a partitioning set (checked using Procedure 4.3.2)
        • then let $D = D - \{z_i\}$
```

In this case we can take advantage of the monotonicity of the problem, to state the following Theorem:

Theorem 4.3.3 *Let $\mathcal{F}$ be an* FSM *derived from a correct* STG *according to Procedure 4.2, let C be a closed cover of $\mathcal{F}$.*

If there exists an optimal *partitioning set D' of $\mathcal{F}$ (with respect to any closed partition π derived from C) such that D' does not contain any* input *signal, then Procedure 4.3.3 finds a partitioning set D that does not contain any input signal.*

If it is possible at all given the initial closed cover, then the heuristic algorithm does not change the *environment* specification.

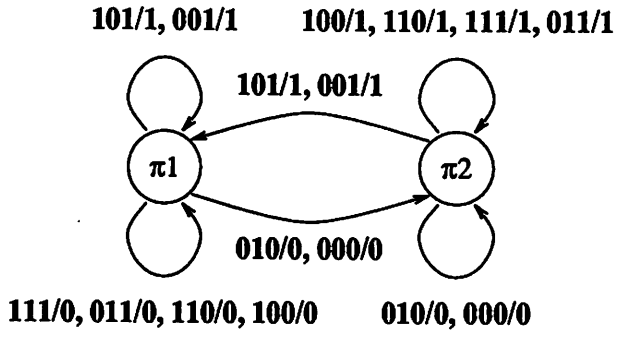


Figure 4.8: Minimized Finite State Machine of Figure 4.6

Proof If D'' is a partitioning set, then for any signal z_i, also $D'' \cup \{z_i\}$ is a partitioning set. So let D' be an optimal partitioning set without any input signal. Then the set obtained by adding all the output signals to D' is also a partitioning set, and so is every set obtained adding to it any number of input signals. But Procedure 4.3.3 tries to remove all the input signals first (if feasible), and it will certainly find a partitioning set without input signals, if one exists. ■

A worst case running time bound for Procedure 4.3.3 is $O(mc^2n^4)$ (m is the number of signals, n is the number of states, c is the number of compatibles).

4.3.3 Minimized Finite State Machine Derivation

The algorithms described above return an assignment of each state in the pre-minimized FSM $\mathcal{F}$ to one of the blocks in the closed partition π.

We can now create a minimized FSM $\mathcal{F}'$ from $\mathcal{F}$ and π as follows. Let π_j denote the block to which state s_j belongs.

Procedure 4.3.4

```
1. for each element $\pi_j$ of $\pi$
    (a) create one state $s'_j$ of $\mathcal{F}'$
2. for each input $i$, for each pair of states $s_1$ and $s_2$ of $\mathcal{F}$ such that $N(i, s_1) = \{s_2\}$:
    (a) let $N'(i, s'_1) = \{s'_2\}$.
    (b) let $O'(i, s'_1) = O(i, s_1)$
```

The result of the above algorithm on the FSM of Figure 4.4 appears in Figure 4.8.

It is easy to show that, since π satisfies the closure condition, then $\mathcal{F}'$ is a deterministic FSM. We can now proceed to encode the states of $\mathcal{F}'$, using any *critical race-free* encoding algorithm. We use Tracey's algorithm, because it is Single Transition Time, hence it allows state signals to change as soon as possible. Moreover, as we outlined in Section 2.3.3, the basic algorithm can be augmented to take into account also the optimization of the *state signal implementation.*

The complexity of satisfying the dichotomy constraints is $0(2^n)$, where n is the number of states. Because the number of the states in the minimized FSM is usually quite small, we were able to find an exact solution to the encoding problem for all our examples. For large FSMs we can use heuristic techniques which do not generate all the prime dichotomies (e.g. [134] or [136]).

4.4 State Signal Insertion

We have partitioned, as shown in Section 4.3, the states in the original FSM $\mathcal{F}$ into a closed partition π, whose blocks are distinguishable using a selected subset of signals D. Then we have used the partition to obtain a minimized FSM $\mathcal{F}'$ and we have assigned, as shown in Section 2.3.3, binary codes to the states of $\mathcal{F}'$ without critical races.

In this section we will prove that an appropriate insertion of state signal transitions in the original STG can guarantee that it still satisfies the original specification, it has the CSC property, and it can be implemented as a logic circuit.

In brief, for each pair of adjacent un-minimized FSM states s_1, s_2 that were assigned to different states in the minimized FSM (say s_1' and s_2' respectively), we condition the original STG signal transitions which are enabled in s_2 but not in any of its predecessors, to be enabled only after the state signals that differ between the codes of s_1' and s_2' have changed.

Here we will use an algorithm that gives only *sufficient* conditions for the STG to have the CSC property. The algorithm takes as input the original STG $\mathcal{G}$, with initial marking m_0, the pre-minimized FSM $\mathcal{F}$ derived from it, the partitioning set D and the associated closed partition of the states of $\mathcal{F}$, and the encoding of each block (i.e. state of the minimized FSM).

Procedure 4.4.1

- *for each pair of maximal sets S_1 and S_2 of states of $\mathcal{F}$ such that*
  - *S_1 is connected and all the states in S_1 belong to the same block π_1 and*
  - *S_2 is connected and all the states in S_2 belong to the same block π_2 and*

```
    – each state in $S_1$ is a predecessor of at least one state in $S_2$ and each state in $S_2$ is a
      successor of at least one state in $S_1$
do:
    1. let $P'$ be the set of places that:
        – are marked in all states in $S_2$ and
        – are not marked in some state in $S_1$ and
        – are predecessors of a transition enabled in some state in $S_2$
       (i.e. the places whose marking denotes the change of state from $\pi_1$ to $\pi_2$).
    2. let $T'$ be the set of non-state signal transitions that:
        – are enabled in some state in $S_1$ and
        – are predecessors of some place in $P'$
       (i.e. the transitions whose firing denotes the change of state from $\pi_1$ to $\pi_2$).
    3. create two new empty transition $\epsilon'$ and $\epsilon''$
    4. for each state signal $x$ that changes value between $\pi_1$ and $\pi_2$
        – create a new transition $x^*$ and constrain $\epsilon' \rightarrow x^* \rightarrow \epsilon''$
    5. for each place $p \in P'$:
        (a) delete all the edges between transitions in $T'$ and $p$ (deleting $p$ from the STG if it
            is left without predecessors)
        (b) let $p'$ be a new place with predecessors the predecessors of $p$ that are in $T'$
        (c) let $\epsilon'$ be the successor of $p'$
        (d) let $p'$ be marked if $p$ was marked in $m_0$
        (e) let $p''$ be a new place with successors the successors of $p$
        (f) let $\epsilon''$ be the predecessor of $p''$
       (i.e. split place $p$ into two, separated by the empty transitions and the state transitions)
```

Figure 4.9 illustrates steps 3, 4 and 5, assuming that $P' = \{p_1, p_2\}$, $T' = \{b^+, c^-, d^-\}$ and that π_1 had code 010 while π_2 had code 100. Figure 4.9.(a) is a fragment of the original STG, Figure 4.9.(b) is the corresponding STD fragment and Figure 4.9.(c) is the encoded STG fragment.

Figure 4.10 describes the result of the application of Procedure 4.4.1 to the STG of Figure 4.2. In this case, the main loop is repeated twice:

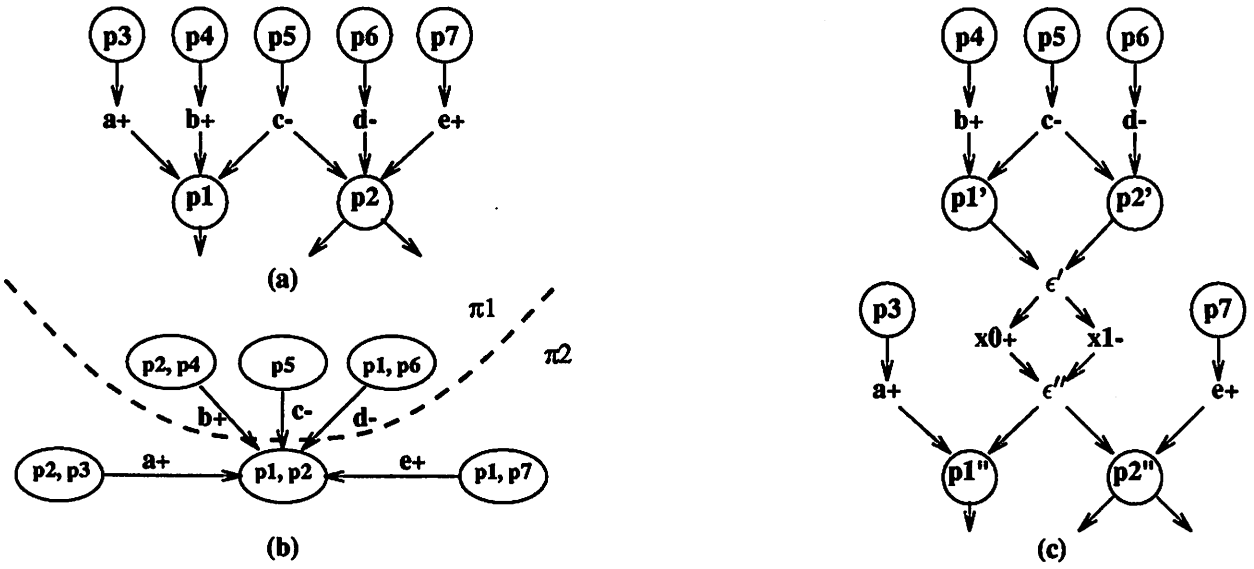


Figure 4.9: Illustration of Procedure 4.4.1

1. $S_1 = \{1, 9\}$ and $S_2 = \{3\}$ (in Figure 4.7).

   1 let $P' = \{(Ri^-, Ri^+), (Ro^-, Ri^+)\}$ (where an implicit place is denoted by its predecessor and successor transitions).

   2 let $T' = \{Ri^-, Ro^-\}$.

   3, 4, 5 create $\epsilon'_1, \epsilon''_1$ and x^-.

2. $S_1 = \{5, 7\}$ and $S_2 = \{8\}$.

   1 let $P' = \{(Ro^+, Ai^+), (Ai^-, Ai^+)\}$.

   2 let $T' = \{Ro^+, Ai^-\}$.

   3, 4, 5 create $\epsilon'_2, \epsilon''_2$ and x^+.

The next sections are devoted to the proof of correctness of Procedure 4.4.1. First of all, we will justify the assumption that step 1 can always find a non-empty set of places that define the new state. Then we will derive the correctness result.

4.4.1 State Splitting

Procedure 4.4.1 requires that for each change of state in the minimized FSM we can find a set P' of places whose marking defines the change itself. Note that this restriction is due to

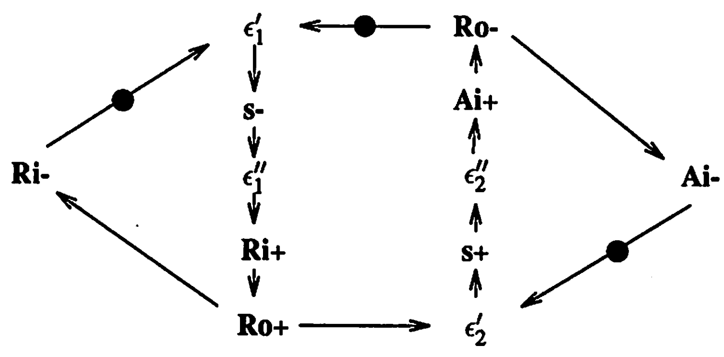


Figure 4.10: Encoded Signal Transition Graph of Figure 4.2

the *algorithm* used to enforce CSC, but it is not inherent in the proposed framework. A *sufficient* condition to ensure that Procedure 4.4.1 always terminates successfully can be obtained by splitting the states in the minimized FSM, as we show below.

See for example Figure 4.11.(a) and 4.11.(b), where Procedure 4.4.1 would not be able to assign s_2 and s_3 to set S_2, because they do not form a connected set of states.

Operating twice on the given STD fragment would be erroneous, because it would introduce two concurrent sets of transitions of the *same* state signals in the same direction, as shown in Figure 4.11.(c).

If, on the other hand, we modified the procedure to allow for S_2 to be not connected, then we would introduce a synchronization between previously concurrent sections of the STG, as shown in Figure 4.11.(d). This can be a problem, as shown in Figure 4.12, if there exists some MG reduction of the STG to which only one of the synchronized transitions belongs. In this example, the synchronization between $c^+ \rightarrow g^-$ and $d^+ \rightarrow f^-$ does not preserve liveness because x^- can never be enabled again if c^+ and e^+ fire when p_1 and p_2 are marked.

So the problem outlined above can be solved:

1. *manually* in *some* cases, because the designer may decide that such added edges in the STG are acceptable (i.e. they preserve liveness, safeness, the desired behavior and throughput, etc.). Such a choice was made, for example, in the example described in Section 1.2.5, in order to deal with a similar problem.

2. *automatically* in *some* cases by dealing with it during the *minimization* procedure. For example in Figure 4.11.(a) there is no problem if a valid partition exists where s_1 is assigned to the same block as s_2 and s_3 (as in the case illustrated in Figure 4.9.)

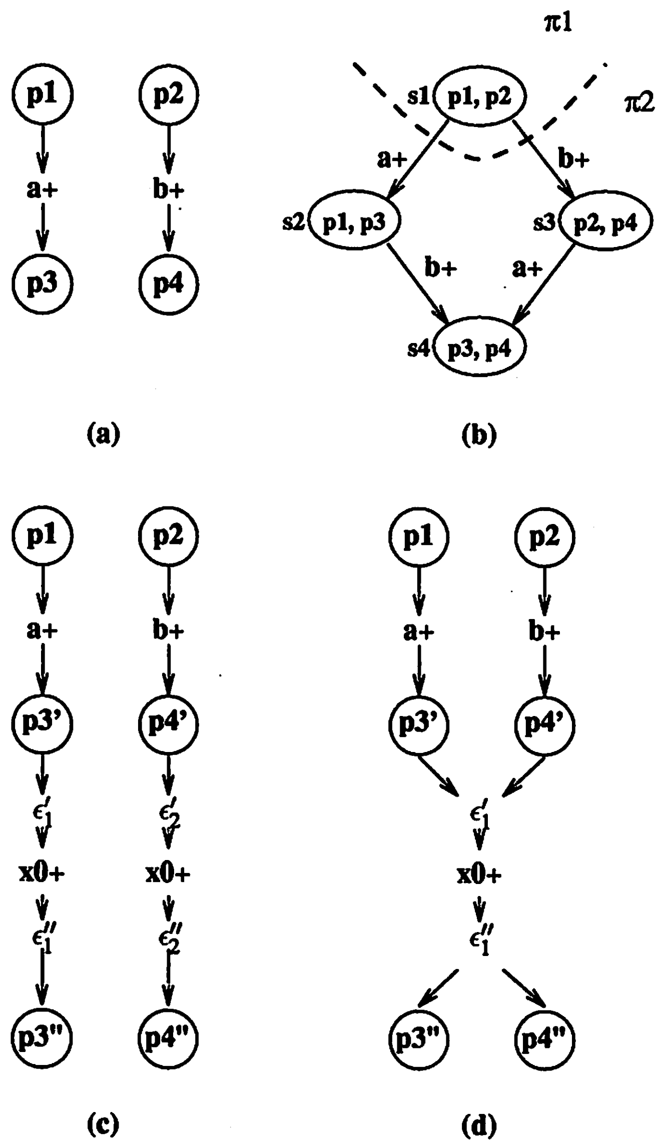


Figure 4.11: A case where Procedure 4.4.1 fails

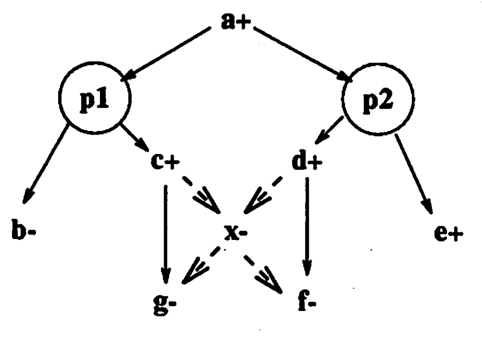


Figure 4.12: Illegal synchronization in free-choice Signal Transition Graph

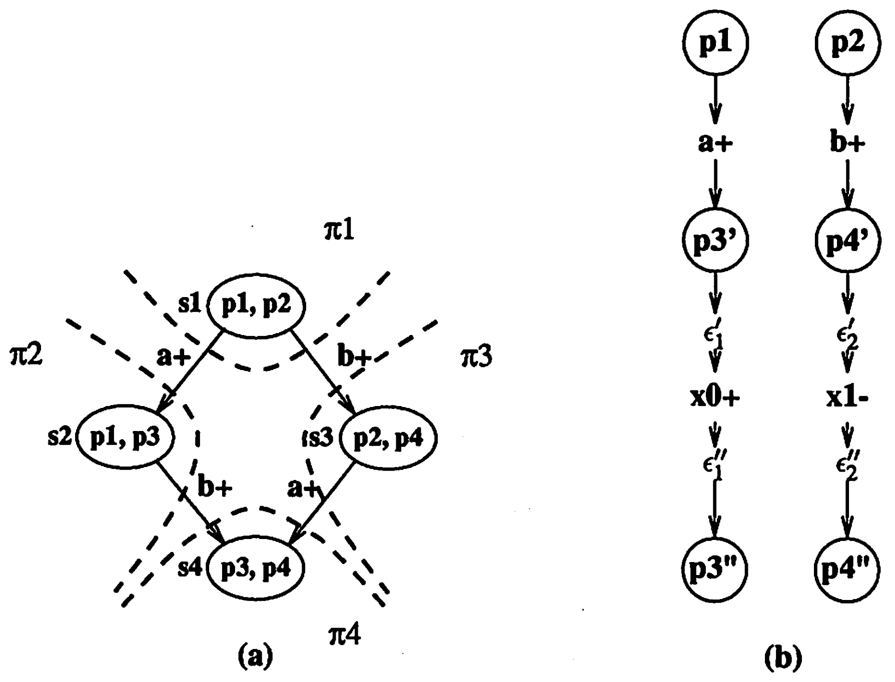


Figure 4.13: State splitting to solve the problem of Figure 4.11

3. *automatically* in *all* cases, possibly requiring more state signals, by *splitting* the state transition. See, for example, Figure 4.13.(a) and 4.13.(b), where we split the transition between π_1 and π_2 of Figure 4.11 into four states, corresponding to blocks π_1, π_2, π_3 and π_4. Now we force different state signals to change going from π_1 to π_2 and from π_1 to π_3, encoding, for example π_1 as 01, π_2 as 11, π_3 as 00 and π_4 as 10.

The algorithm to perform this state splitting is as follows. Let s_0 be a state of the original (pre-merged) FSM. Let $S = \{s_1, \dots s_n\}$ be a *maximal* set of successors of s_0 such that

- all $s_i \in S$ belong to the same minimized FSM state s_1', different from the minimized FSM state of s_0 (call it s_0'), and
- the transitions labeling each edge $s_0 \rightarrow s_i$ are all concurrent.

Let $T_2 = \cup_{s_i \in S} T_{2,i}$, where $T_{2,i}$ is the set of transitions enabled in s_i and not enabled in any predecessor of s_i. Let T_1 be a set, with *minimum* cardinality, of transitions such that each transition in T_2 has at least one predecessor in T_1. Let n be the cardinality of T_1. If $n > 1$, then we split the transition from s_0' to s_1' into 2^n states. Furthermore we assign state codes so that there are n disjoint groups of state signals that change value from one state to the other. Each state signal will have a transition that is a successor of one of the n concurrently enabled signals, so Procedure 4.4.1 can terminate successfully. State splitting terminates, in the worst case, when all the states of the un-minimized FSM have been reinstated, because then each T_1 has cardinality 1.

The constraint on the disjointness of the state codes can be easily embedded in the dichotomy-based state encoding procedure. We can just forbid to generate any prime dichotomy that contains s_1 on one side and s_2, s_3 on the other side, because this avoids to produce a solution where the same state signal changes value going *both* from s_1 to s_2 *and* from s_1 to s_3.

4.4.2 Proof of Correctness of the Encoding Algorithms

We will now show that the output of Procedure 4.4.1, called the **encoded** STG, is a correct STG with CSC, and that it specifies a behavior (i.e. a set of successful traces) that is a *subset* of the original STG, and as such it is compatible with the original specification.

The first Lemma shows that the procedure preserves every *Marked Graph reduction* and every *State Machine covering* of the initial STG.

Lemma 4.4.1 *Let $\mathcal{G}$ be a correct* STG, *let $\mathcal{G}'$ be an encoded* STG *derived from $\mathcal{G}$ as shown in Sections 4.2, 4.3 and 4.4.*

Then:

1. *every* Marked Graph *reduction $\mathcal{G}$ has* exactly *one corresponding strongly connected, non-empty* MG *reduction of $\mathcal{G}'$.*

2. *every* State Machine *covering of $\mathcal{G}$ corresponds to an* SM *covering of $\mathcal{G}'$.*

Proof

1. Steps 3, 4 and 5 of Procedure 4.4.1 replace a set of places P' with an MG fragment, as shown in Figure 4.9.

   We have the following cases:

   - P' appeared together in all MG reductions of $\mathcal{G}$. Then the procedure replaces an MG fragment with another MG fragment, hence no reduction can become empty or not strongly connected. So no MG reduction of $\mathcal{G}$ is changed (except for the fragment replacement).

- there exist $p_1, p_2 \in P'$ and MG reductions $\mathcal{M}_1, \mathcal{M}_2$ of $\mathcal{G}$ such that $p_1 \in \mathcal{M}_1$, $p_2 \notin \mathcal{M}_1$ and $p_2 \in \mathcal{M}_2$, $p_1 \notin \mathcal{M}_2$. Then we can partition P' so that the previous case applies, and use the state splitting technique outlined in Section 4.4.1 to make sure that each subset of places belonging to the same set of MG reductions is assigned a set of state variable changes.

2. For each predecessor choice (partial *State Machine allocation*) in the MG fragment introduced by steps 3, 4 and 5 of Procedure 4.4.1 we can find a corresponding SM fragment in $\mathcal{G}$.

   So each SM component in the cover of $\mathcal{G}$ corresponds to a set of SM components of $\mathcal{G}'$ (one for each combination of choices in the MG fragments). Obviously these components *cover* $\mathcal{G}'$.

■

We can now state the correctness results, in the following three Theorems.

Theorem 4.4.2 *Let $\mathcal{G}$ be a correct* STG, *with initial marking m_0, let $\mathcal{G}'$ be an encoded* STG *derived from $\mathcal{G}$ as shown in Sections 4.2, 4.3 and 4.4, with initial marking m'_0.*

Then $\mathcal{G}'$ is correct.

Proof Theorem 2.2.1 and Lemma 4.4.1 can be used to show that $\mathcal{G}'$ is *live*, because every MG reduction of $\mathcal{G}'$ is strongly connected and not empty.

So the new initial marking m'_0 is also *safe*, because:

- $\mathcal{G}$ was live and safe, so there exists an SM cover where m_0 marks each component with exactly one token.

- each component of that cover corresponds to a set of components that cover $\mathcal{G}'$ and are marked with exactly one token by m'_0.

We must still show that the reachability graph of $\mathcal{G}'$ can be labeled consistently with signal values.

- This is guaranteed for non-state signals by Lemma 4.4.1, because the SM component that ensured it in $\mathcal{G}$ has at least one corresponding SM component in $\mathcal{G}'$ that is still marked with one token.

- For each state signal x:

  - there can be no two concurrently enabled transitions of x because Procedure 4.4.1 (with the state splitting outlined in Section 4.4.1, if necessary) creates only one such transition between connected sets of boundary states.
  - moreover transitions of x alternate because Procedure 4.4.1 inserts a state signal transition only if the two blocks were assigned different codes in that signal.

■

Theorem 4.4.3 *Let $\mathcal{G}$ be a correct* STG, *with initial marking m_0, let $\mathcal{G}'$ be an encoded* STG *derived from $\mathcal{G}$ as shown in Sections 4.2, 4.3 and 4.4, with initial marking m_0'.*

Then $\mathcal{G}'$ has Complete State Coding.

Proof Let $\mathcal{F}$ be the un-minimized FSM derived from $\mathcal{G}$ as shown in Section 4.3. We can consider two types of markings of $\mathcal{G}'$:

- Markings where no state signal transition is enabled. The minimization Procedure 4.3.4 assigned states of $\mathcal{F}$ with the same binary label but with different implied values for output signals to different blocks. These blocks are assigned different values of the state signals as shown in Section 4.3.
- Markings where some state signal transition is enabled. Let m_1' be any one of these markings.
  - any marking m_2' reachable from m_1' without firing any state signal was assigned to the same block as m_1', so there can be no CSC problem between m_1' and m_2'.
  - if a state signal fires, we can use the fact that the state encoding given in Section 2.3.3 is critical race-free. The implied value of both state signals and output signals does not depend on the outcome on the race, and there is no CSC problem either.

■

Theorem 4.4.4 *Let $\mathcal{G}$ be a correct* STG, *with initial marking m_0, let $\mathcal{G}'$ be an encoded* STG *derived from $\mathcal{G}$ as shown in Sections 4.2, 4.3 and 4.4, with initial marking m_0'.*

Then $\mathcal{G}'$ specifies a subset of the possible execution traces specified by $\mathcal{G}$ with respect to the non-state signals.

Proof Procedure 4.4.1 does not remove any constraint between signals from $\mathcal{G}$. ■

name	initial			final				CPU
	sig.	trans.	states	sig.	trans.	states	lit.	sec
alloc-outbound	7	18	17	9	22	3	19	6
nak-pa	9	18	18	10	22	2	30	9.9
pe-rcv-ifc	8	38	27	9	45	2	50	12.1
pe-send-ifc	8	41	54	11	54	4	33	82.2
ram-read-sbuf	10	20	16	11	22	2	20	8
sbuf-ram-write	10	20	24	12	24	3	30	11.5
sbuf-read-ctl	6	12	10	7	14	2	13	5.2
sbuf-send-ctl	6	18	14	8	24	3	34	8.4
sbuf-send-pkt2	6	20	15	7	24	2	11	5.7
sendr-done	3	6	5	4	8	2	5	3.5
atod	6	12	11	7	14	2	14	5.2
nousc	3	6	6	4	8	2	9	4.1
nousc.ser	3	6	4	4	8	2	8	3.8
master-read	14	28	132	17	36	5	77	1635.1
vbe4a	6	12	34	8	16	4	22	10.1
vbe6a	8	16	16	10	20	4	30	18.4

Table 4.1: Results of the state encoding procedure

3. building the MDD for the optimum procedure was not always possible, so the table uniformly shows the result of the heuristic procedure,

We are now ready to tackle the second half of the Signal Transition Graph synthesis problem. Every STG signal can be implemented with a logic function, so we need to find a circuit that realizes this function correctly, i.e. without *hazards*, as described in the next chapter.

4.5 Experimental Results

The algorithms described above have been implemented within the SIS sequential synthesis system developed at U.C. Berkeley. The FSM minimizer must be able to handle very large FSMs, with thousands of states. So we resorted to using a heuristic minimizer, that could solve all the cases presented so far in a matter of minutes ([127]).

Table 4.1 contains the result of our procedure on a set of industrial and literature examples, some of which are free-choice STGs. The columns labeled "initial signals" and "initial transitions" contain the initial size of the STG. The column labeled "initial states" contains the number of states of the FSM before minimization, and the column labeled "final states" contains the number of states after minimization. The columns labeled "final signals" and "final transitions" contain the size of the encoded STG. The difference between initial signals and final signals is the number of state signals. The column labeled "lit." contains the number of factored form literals in a hazard-free implementation of the encoded STG. The column labeled "CPU time" contains the CPU time (in seconds, on a DEC5000/125) for the whole minimization, encoding and synthesis procedure. Example "master-read" took a relatively long time to complete because most states were assigned to all the compatibles by the initial FSM minimization procedure, and then Procedure 4.3.2 was slow to converge.

In comparison, the algorithm described in [122] found a solution[4] for STGs *vbe4a* and *vbe6a* with 5 and 34 literals respectively. Our procedure obtains a significantly larger circuit for *vbe4a* because we use state signals to eliminate CSC violation but preserve the concurrency of the specification as much as possible, while [122] does not add state variables but reduces the concurrency. Presently we do not attempt to remove any state (by adding causality constraints), but we always add state signals, which accounts for the large area penalty in this case. Applying to *vbe4a* the *state removal technique* of [122] *within our framework* led to a result that *did not require any state signal and had 13 literals* but more concurrency than the result shown in [122].

Summarizing the experimental results of the proposed methodology:

1. all the STG examples known to us that required encoding were solved *automatically* by the procedure, leading to results similar to hand-encoding (when such an encoding was known).

2. the FSM that must be minimized may have a *very large number of states*, so that heuristic minimization techniques must be used.

[4]It required no state signals, of course, since it just adds constraints to the STG in order to remove the STD states that cause CSC problems.

Chapter 5

The Synthesis Methodology

This chapter describes a synthesis procedure that transforms a correct Signal Transition Graph specification with *Complete State Coding* into a logic circuit implementing it. The implementation can be shown to be hazard-free, using the *bounded wire* delay model if:

- the circuit operates in an environment that obeys the STG specification, and
- the bounds on the delays are met by the circuit after manufacture.

Within this chapter we will always consider *pure* (rather than *inertial*) and *wire* (rather than *gate*) delays. Both the inertial and gate delay models can be too optimistic as the scale of integration decreases and transmission line delays are no longer negligible in comparison with gate switching delays. The gate delay model may also fail, as shown by [118], if two gates have a different *logic threshold* voltage.

The approach avoids two major drawbacks that were present in the original STG synthesis methodology, as defined by Chu (see [27]). Namely, Chu's methodology:

- required the STG to be *persistent* in order to be able to produce a hazard-free circuit, while we show that this requirement is *not necessary*, and
- could guarantee to automatically produce a hazard-free circuit only using an unrealistic delay model where no restrictions are imposed on the logic function implemented by each gate.

Our methodology can be divided in the following steps:

1. Derive from an STG a circuit implementation C with two-level combinational functions and flip-flops.

2. Characterize all potential hazards in C, because we have delays *inside* the logic block implementing each STG signal.

3. Show that constrained multi-level logic synthesis can be used without altering hazard properties of C.

4. Remove all hazards from C by delay padding, using bounds on gate, wire and environmental delays.

The methodology is based on the following key results:

- hazards can occur in a two-level implementation of an STG if and only if, due to delays inside the circuit, the STG-specified ordering of two signal transitions is reversed,

- the above property is preserved by constrained logic synthesis, and

- all hazards can be removed by increasing the delay of some STG signal.

The chapter is organized as follows. Section 5.1 performs a *static* hazard analysis using *unbounded wire* delays on a two-level circuit whose inputs are constrained to follow an STG specification, and extends the results to multi-level circuits. Section 5.2 uses that analysis to synthesize an *initial* implementation that is hazard-free with respect to *concurrent* STG transitions. Section 5.3 shows that hazards can still occur because the order of *non-concurrent* transitions can be reversed by the wire delays, and discusses the implications of Chu's definition of STG persistency. Section 5.4 gives greedy and optimum algorithms to produce a hazard-free circuit implementation of the STG using delay padding based on delay bounds. Section 5.5 shows how the results of the previous sections imply that the circuit is free of *dynamic* hazards as well. Section 5.6 gives experimental results of the implementation of the algorithms described in this chapter.

5.1 Hazard Analysis and Signal Transition Graphs

As outlined in Section 3.3.4, the synthesis procedure for a Signal Transition Graph assumes the circuit structure described in Figure 4.1 for each *output* signal of the STG. The **next state function** f computed by each combinational logic block maps each STD binary label into the corresponding *implied value* for that signal (i.e. the complement of the value of the signal in the label if a transition for that signal is enabled in the corresponding marking, the same value otherwise). Note that every vertex in the domain of f that does not correspond to any STD label belongs to the *dc-set* of f. Such don't cares will be exploited by our synthesis algorithm to optimize the combinational logic implementation.

This section establishes some basic results concerning the *possible* hazards in the combinational logic blocks implementing the STG signals. We first analyze the behavior of a *two-level* implementation under input changes constrained by the STG specification, then extend the results to *multi-level* circuits.

5.1.1 Static Hazard Analysis of a Two-level Logic Circuit

This section assumes that a two-level logic implementation $\mathcal{C}$ has been derived for the next state function f of each output signal z_i of an STG $\mathcal{G}$, and establishes the conditions under which *static* hazards can occur in it. The next sections will then proceed to show that:

- such results hold also for multi-level implementations obtained with constrained logic synthesis.
- $\mathcal{C}$ can be synthesized ensuring that there are no hazards due to *concurrent* transitions.
- hazards due to transitions that are not concurrent can be eliminated by inserting delays into appropriate points of the circuit.

Section 5.5 shows how to extend these hazard analysis results to handle *dynamic* hazards as well.

The circuit *environment* is assumed to operate according to the STG specification. So $\mathcal{G}$ limits the number of possible changes that can be applied to the inputs of $\mathcal{C}$ in two ways:

- if two sets of transitions are separated by a transition of signal z_i, then the second set cannot fire unless z_i has had time to fire.
- if some signal z_j is constant between two transitions of z_i, then no matter what the circuit delays are, z_j will not change between them. This means that the *environment* is supposed to be hazard-free (see [131] for a discussion of techniques that increase circuit robustness with respect to environmental hazards).

These assertions should be taken with care, as we are dealing with *wire* delays and we cannot assume, in general, that the STG describes a *delay-insensitive* behavior. We can assume them to be true for the time being, and give a more detailed justification for this in Section 5.5.

Remembering Eichelberger's result on static hazard analysis with *three-valued simulation* (Section 2.7.1), we can describe the set of signals that can change between two consecutive transitions of signal z_i using the concepts of *valid state pair* and *transition cube.*

Let (s', s'') be a pair of states of the STD $\mathcal{S}$ associated with an STG $\mathcal{G}$. Recall that $\lambda(s')$ is the binary vector of signal values labeling s' in $\mathcal{S}$. Let C be a circuit implementation of the next state function f of an output signal z_i of $\mathcal{G}$.

We define (s', s'') to be a **valid state pair** if:

- $f(\lambda(s')) = f(\lambda(s''))$ (since we are analyzing *static* hazards), and
- there exists at least one directed STD path between s' and s'' traversing at most one transition of z_i.

We exclude paths with more than one transition of z_i from the analysis because we are interested in *static* hazard analysis only, while these paths would actually involve *dynamic* hazards. On the other hand, we cannot just exclude z_i^* from being the first or the last transition of the path, because we want to make sure that there cannot be hazards *while* signal z_i is changing value.

A **transition cube** for a circuit implementation C of the next state function f of output signal z_i of an STG $\mathcal{G}$ is a three-valued vector of values for the *input* signals of C. Signals with a values of "0" or "1" are assumed to be constant, while signals with value of "$-$" are assumed to be changing. The transition cube associated with a valid state pair can be derived by assigning:

- a value of "$-$" to all signals that change on some STD path between s' and s'' including at most one transition of z_i, and
- the value in the label of s' to all the other signals.

Let F denote the cover corresponding to C (one cube per *and* gate). It is easy to show using three-valued simulation that the only case when a two-level combinational circuit can have a static hazard under transition cube c is when c intersects some cube of F without being completely covered by any single one of them.

For example, a next state circuit synthesized from the STG in Figure 1.3.(b) (whose STD appears in Figure 1.3.(c)) to implement output signal z_i must consider the pair of states labeled (10*0*,00*1) as valid, with corresponding transition cube $c = -0-$, while the state pair (1*0*1,011*) is not valid, because the value of the next state function of z_i is 1 and 0 respectively.

5.1.2 Logic Synthesis and Hazards

We shall now prove that for every multi-level combinational circuit C_M derived in a constrained way from a two-level circuit C_T, for every assignment of delays to the wires in C_M, then there exists an assignment of wire delays in C_T such that the behavior of C_M and C_T is the same. Namely we will show that applying the associative and the distributive law, or adding an *inverter* to any output of the circuit, does not add any new hazard to the circuit. So if we analyze, as in Section 5.1.1, what classes of input changes may cause a hazard in C_T under some particular combination of wire delays, then we can check only those input changes and the corresponding wire delays in order to examine all possible hazard causes in C_M. Furthermore if we derive, as we will show in Section 5.2.1, a two-level implementation C_T of a logic function f that does not exhibit hazards for some class of input changes, then we can use constrained multi-level synthesis techniques to obtain a multi-level implementation of f that has the same hazard properties.

A similar result was proved in [117], but only under the *Fundamental Mode* assumption, while here we lift such restriction. Recently a larger class of *hazard-non-increasing transformations* has been reported in [68], using an extension of three-valued simulation to handle dynamic hazards as well, but the applicability of such extension to our non-Fundamental Mode context is not clear yet.

A **circuit family** is a set of circuits with the same *structure* but different *delays*. Each **instance** of a circuit corresponds to a specific assignment of values to the pure delay elements.

The **topological distance** of a node (i.e. a gate or a primary input) v in a combinational logic circuit from the primary inputs, $d(v)$, is recursively defined as follows:

- if v is a primary input, then $d(v)$ is 0.
- otherwise, $d(v)$ is 1 more than the maximum $d(v')$ of any *fanin* v' of v.

Lemma 5.1.1 *Let C_M be an instance of a combinational circuit family. Let C'_M be a combinational circuit derived from C_M by duplicating every gate that has more than one fanout (copying the delay assignment of the fanin and fanout wires of every duplicated gate), until every gate in the resulting circuit has exactly one fanout.*

Then the primary outputs of C_M and C'_M have exactly the same behavior in time.

Note that in C'_M primary input nodes still have more than one fanout, and that we do not make any hypothesis on how and when the primary inputs can change (i.e. no *Fundamental Mode* hypothesis).

Proof We will proceed by induction on the distance of each node w in C_M from the primary inputs.

- Base case, $d(w) = 1$: all the fanin nodes of w are primary inputs. Then all the corresponding duplicated gates w_i in C'_M have exactly the same fanin set and exactly the same input wire delay assignment, so their temporal behavior reproduces the behavior of w exactly.
- Induction step. Let us assume that each gate u in C_M such that $d(u) < d(w)$ has exactly the same temporal behavior as its duplicates u_j in C'_M. Then all the duplicates w_i of w have all their inputs with exactly the same behavior as the inputs of w, and the same wire delay assignment. So their temporal behavior is the same as that of w.

■

Lemma 5.1.2 *Let C_M be an instance of a combinational circuit family, such that no gate has more than one fanout. Let C'_M be a combinational circuit derived from C_M by recursively moving the delays back, up to wires fanout of* primary inputs. *That is for each node v, in order of increasing distance from primary outputs, let $e(v)$ be the delay associated with its single fanout wire, assign zero to the fanout delay of v, and add $e(v)$ to each fanin delay of v.*

Then the primary outputs of C_M and C'_M have exactly the same behavior in time.

Proof Pure delay, being a translation in time, can be moved from the fanout wire of a single-fanout node of a combinational circuit to each of its fanin wires without changing its behavior. Then each step of the procedure transforming C_M into C'_M preserves the temporal behavior of each node. ■

Theorem 5.1.3 *Let $\mathcal{T}$ be a two-level combinational circuit family, such that only primary inputs can have multiple fanout and all wires except the fanouts of primary inputs have zero delay. Let $\mathcal{M}$ be a multi-level combinational circuit family derived from $\mathcal{T}$ by applying the distributive and associative laws, and/or by adding an* inverter *to some primary output, with unbounded delays on every wire.*

Then for each instance C_M of $\mathcal{M}$ there exists an instance C_T of $\mathcal{T}$ whose primary outputs have exactly the same behavior in time (modulo complementation of value if an inverter *is added).*

Proof We will give a constructive method to derive C_T given $\mathcal{T}$, $\mathcal{M}$ and C_M, and prove that each step preserves temporal behavior.

Given C_M, transform it into C'_M where each gate has exactly one fanout, using Lemma 5.1.1. Then transform C'_M into C''_M where only wires fanout of primary inputs have a non-zero delay, using Lemma 5.1.2.

Now apply the *inverse transformations* of those used to obtain $\mathcal{M}$ from $\mathcal{T}$, in order to derive a two-level circuit $C_{\mathcal{T}}$. No transformation changes the primary output temporal behavior, because:

- Only wires fanout of primary inputs have non-zero delay.
- Removing an *inverter* driving a primary output changes only the polarity of the primary output itself, not its temporal behavior.
- Inverse application of the associative law:

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

  merges two gates into one. Direct application of the associative law to derive $\mathcal{M}$ from $\mathcal{T}$ can never duplicate fanins, so these gates did not have any common fanin node. Then the wires and their delays are simply copied to the merged node, and the temporal behavior is obviously not changed.

- Inverse application of the distributive law:

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

  transforms two nodes into three nodes. Now we must duplicate the delay on the wire between node a with the *and* node, and again the temporal behavior does not change.

■

Note that, since $\mathcal{M}$ was obtained from $\mathcal{T}$ in a very restricted way, its gates can only be *inverters*, *ands* and *ors*. But we can use the following procedure to replace some sub-circuit of $\mathcal{M}$ with a single gate that computes the same function (**technology mapping**, [32]) and still be able to apply Theorem 5.1.3. The procedure takes as input a multi-level circuit $\mathcal{M}$ composed only of *and*, *or* and *inverter* gates and produces a multi-level circuit $\mathcal{M}'$ composed of arbitrary gates.

Procedure 5.1.1

1. *Transform $\mathcal{M}$ into $\mathcal{M}'$ by replacing each node v of $\mathcal{M}$ and its set of fanout wires $< v, u >$ with:*

   - *either*
     - *a node computing the same function as v,*
     - *an inverter i_v with fanin v,*
     - *an inverter $i_{<v,u>}$ with fanin i_v for each wire $< v, u >$.*
   - *or*
     - *a node computing the complement of the function of v*
     - *an inverter $i_{<v,u>}$ with fanin v for each wire $< v, u >$.*

2. *Repeatedly replace some two-level sub-circuit in $\mathcal{M}'$ with a new node w implementing the same function as the sub-circuit.*

Theorem 5.1.4 *Let $\mathcal{M}$ be a combinational circuit family. Let $\mathcal{M}'$ be a combinational circuit family derived from $\mathcal{M}$ by Procedure 5.1.1.*

Then for each instance C'_M of $\mathcal{M}'$ there exists an instance C_M of $\mathcal{M}$ whose primary outputs have exactly the same behavior in time.

Proof As before, we apply the reverse transformations of Procedure 5.1.1 and show that they preserve the temporal behavior.

Each two-level sub-circuit inherits the input wire delays from the replaced node w. It acts as an instantaneous logic evaluator, so it replaces each node w without affecting the behavior of the primary outputs.

Similarly the insertion of inverters transforms a single node in a delay-free evaluator for the same function, and a set of wires with exactly the same delay as before (the delays can be copied to the input wire of each $i_{<v,u>}$). ∎

5.2 Circuit Implementation of the Next State Function

This section uses the results developed so far on hazard analysis to derive a circuit implementation of each output signal of an STG. The main result is that the implementation can be shown not to have static hazards due to the firing order of *concurrent* transitions. Section 5.2.1 describes the initial two-level synthesis algorithms and proves hazard-freeness. Section 5.2.2 uses the results of Section 5.1.2 to show how an *efficient* multi-level implementation can be derived while keeping this essential property.

5.2.1 Initial Two-level Circuit Implementation

The purpose of this section is to make sure that the synthesized circuit behavior is correct for each possible ordering of *concurrent* transitions. In the example of Figure 1.3.(b), nothing is said about the ordering of z^+ and y^+, then no output signal can have static hazards regardless of their firing order. So we must make sure, remembering the analysis in Section 5.1.1, that the on-set and off-set covers F and R that we synthesize for the next state function of each output signal z_i have the following property:

Property 5.2.1 *Let $\mathcal{G}$ be a correct* STG *with Complete State Coding. Let F be an on-set cover and R be an off-set cover of the next state function of signal z_i, synthesized from $\mathcal{G}$ with the property that the intersection of F and R is empty*[1].

Then for each reachable marking m of $\mathcal{G}$ and for each set T of transitions concurrently enabled in m such that the next state value for z_i does not change during any sequence of firings in T:

1. *if z_i must be 1, then there exists at least one cube $c' \in F$ such that:*
   - *c' covers the vertex corresponding to marking m and*
   - *no signal whose transition is in T appears in c'.*
2. *otherwise, if z_i must be 0, then there exists at least one cube $c'' \in R$ such that:*
   - *c'' covers the vertex corresponding to m and*
   - *no signal whose transition is in T appears in c''.*

Case 1 guarantees that the output of F, if so required, remains at 1 independent of the firing order in T. Case 2 guarantees that the output of F, if so required, remains at 0 independent of the firing order in T, even though it is stated in terms of the off-set cover R. This is because the intersection of F and R is empty, so the intersection of $c' \in R$ with any cube $c'' \in F$ is not defined, hence every $c'' \in F$ must have at least one literal that is the complement of a literal in c'. Then

[1]In general an on-set and an off-set cover of a given logic function can have a non-empty intersection, consisting only of *dc-set* vertices.

there exists at least one stable literal (i.e. no transition for its signal is in T) that keeps each $c'' \in F$ at 0.

For example, the set of concurrently enabled transitions in the marking corresponding to vertex 10*0* in Figure 1.3.(c), is $T = \{y^+, z^+\}$. Assume that one of the cubes in the on-set cover of the next state function for z, which must be a constant 1 independent of the firing order in T, is exactly x. Then the value of the next state function for signal z will remain consistently at 1 regardless of the firing order of y^+ and z^+.

In this way we are able to guarantee that the circuit implementation does not have hazards under *any transition cube corresponding to the firing of concurrent transitions* in the STG specification. This is particularly important, because the firing order of *concurrent* transitions cannot be affected by delay padding. Padding adds a *fixed* delay, while concurrent transitions can fire at an arbitrary distance. So we would not be able to use methods based on padding, such as those described in Section 5.4.3, to eliminate such hazards.

The following procedure derives an on-set cover F and an off-set cover R for the next state function f of signal z_i, receiving as input a *correct* STG $\mathcal{G}$, having the *Complete State Coding* property, with initial marking m. Let v denote a vector of values for the n signals that appear in $\mathcal{G}$, $v \in \{0, 1\}^n$, and let v_j denote the value of signal z_j in v.

Procedure 5.2.1

1. *Initialization:*
   (a) *for each signal z_j in the* STG *determine its initial value, as follows:*
      i. *let M_j be an* SM *component of $\mathcal{G}$, initially marked with one token, that contains all the transitions for z_j, and let m_j be its initial marking (a subset of m).*
      ii. *find on M_j the first transition z_j^* that can be reached from m_j.*
      iii. *if z_j^* is z_j^+, then let $v_j = 0$, otherwise let $v_j = 1$.*
   (b) *let $F' = \phi$, $R' = \phi$.*
2. *Recursive step:*
   (a) *if z_i^+ is enabled in m then let $v_i = 1$.*
   (b) *else if z_i^- is enabled in m then let $v_i = 0$.*

(c) *for each maximal subset T (possibly containing a transition z_i) of transitions concurrently enabled in marking m such that z_i^* is* not enabled *in the marking m' obtained from m firing all transitions in T do:*

   i. *let $c = \{v_j : z_j^* \notin T\}$.*

   ii. *if $v_i = 1$ then let $F' = F' \cup \{c\}$, otherwise let $R' = R' \cup \{c\}$.*

(d) *for each transition z_j^* enabled in m such that marking m', obtained from m firing z_j^*, has not been reached yet, do:*

   i. *let $v' = v$.*

   ii. *if z_j^* is z_j^+, then let $v_j' = 1$, otherwise let $v_j' = 0$.*

   iii. *recursively call step 2 with v' and m'.*

3. *Minimization:*

   (a) *let F'' be the set of all prime implicants of the incompletely specified logic function with on-set vertices covered by F' and off-set vertices covered by R'.*

   (b) *Let F be a set of cubes of F'' such that:*

      i. *every cube in F' is covered by at least one cube in F and*

      ii. *F has minimum cardinality.*

The initial values determined by step 1a are well defined if the STG is *correct*, because:

- at least one M_j containing all transitions of signal z_j must exist.
- the direction of the first transition of z_j that can fire on the SM component M_j starting from a marking m must be independent of the path on M_j, or else there would be a firing sequence where two transitions z_j^* (both rising or falling) are not separated by a $\overline{z_j^*}$.
- if there are two (or more) SM components containing all transitions of z_j, say M_j' and M_j'', then the first transition of z_j reachable from a marking m restricted to M_j' or M_j'' must be the same, since the intersection of the two machines obviously contains all such z_j^*'s, and the SMs are synchronized in their mutual intersections.

Note that in practice exhaustive exploration of the STD can be a faster way of determining the initial values, rather than SM decomposition. Both have exponential worst case complexity, but the constant factor for STD analysis is in practice much better.

F and R are constructed by the above procedure exactly to have Property 5.2.1. This guarantees that the next state function remains *constant* whenever the STG specifies so, independent of the firing order of a set of concurrently enabled transitions.

Moreover the vector of values v generated at each step is consistent with the firing, so it coincides with the label of the STD state s corresponding to marking m.

Note that the two-level synthesis methodology described by Moon *et al.* ([87]) is potentially faster at the expense of optimality, as it performs only a single expansion of the cubes in F, removing only non-prime cubes. Here, on the other hand, we guarantee that the two-level implementation has the *minimum* number of cubes among those that ensure no hazards due to the firing order of concurrent transitions. The initial two-level implementations produced by the two methods, however, are closely related, and can be used more or less interchangeably. The hazard elimination procedures, on the other hand, are very different between our approach and [87]. We perform delay padding on an optimized implementation that uses gates from an arbitrary library (standard cell or hand designed) with bounded wire delays. Moon *et al.* use a two-level implementation with unbounded gate delays, and add literals to cubes to keep them stable and prevent them from causing hazards.

Now we can prove the correctness of Procedure 5.2.1, showing that F and R are on-set and off-set covers of the next state function f, computing the *implied value* of z_i.

Theorem 5.2.1 *Let f be the incompletely specified next state function of signal z_i, obtained from a correct* STG $\mathcal{G}$. *Let F and R be the covers obtained from $\mathcal{G}$ using Procedure 5.2.1.*

Then

- *F and R are valid on-set and off-set covers of f, and*
- *the intersection of F and R is empty.*

Proof We have the following cases:

1. Some transitions enabled in the current marking m leads directly to a marking m' where no transition of z_i is enabled.

   Then we generate a set of cubes such that the vertex v corresponding to m is covered. Moreover all the cubes belong to either the on-set or to the off-set according to whether v_i is 1 or 0, and the decision about v_i is made exactly as in the definition of *implied value*.

All covered vertices correspond to markings where the next state value of z_i is the same, so *if the* STG *has the* CSC *property* then no vertex where f must have a different value can be covered. Furthermore the intersection of F and R is empty, because all covered vertices can be reached by some firing sequence of the concurrent transitions, so no generated cube covers any dc-set vertex.

2. Every transition firing from m directly reaches a marking m' where a transition of z_i is enabled.

   We will show that in this case no transition of signal z_i can be enabled in m.

   Suppose that z_i^* is enabled in m and let m' be the marking reached from m firing z_i^*. Then

   (a) z_i^* could not be enabled again in m', since the STG is correct (otherwise two rising or falling transitions of z_i could fire in sequence).

   (b) we can also show that $\overline{z_i^*}$ cannot be enabled in m'. We have the following cases:

      i. if z_i^* is not enabled in marking m''', reached from m' firing $\overline{z_i^*}$, then we have a contradiction, because m and m''' have the same label but different enabled output signals, so the STG would not have the CSC property. Similarly if z_i^* is enabled in m''' and there exists some marking, reachable by m''' by firing only transitions of z_i, where no transition of z_i is enabled.

      ii. otherwise, signal z_i can have an unbounded number number of transitions without any other signal changing:

         A. either the STG is simply $z_i^+ \rightarrow z_i^- \rightarrow z_i^+$, that can be shown by inspection to satisfy the Theorem,

         B. or z_i^* is enabled by the transition of some other signal. Then if the conditions that first enabled it occur again, the corresponding place can be marked twice (actually an unbounded number of times), thus contradicting the safeness hypothesis.

   We also know that the STG is correct, so there exists some marking m'' from which m can be obtained by firing some transition z_j^*. Whenever the procedure reaches m'', it generates a cube covering also the vertex corresponding to m, because m is obtained from m'' by firing a transition that does not enable z_i^*.

■

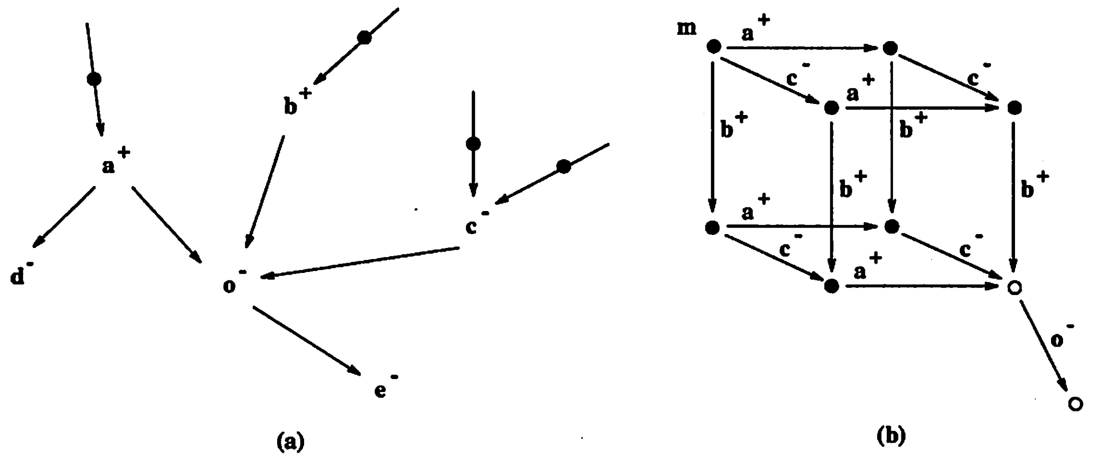


Figure 5.1: Illustration of case 1 of Theorem 5.2.1

Figure 5.1 contains an STG fragment and the corresponding STD fragment to illustrate case 1. Let o be the signal for which we are generating cover cubes in marking m. Black dots in the STD represent on-set vertices of f, white dots represent off-set vertices.

1. The sets of transitions that can fire without enabling o^- are: $T' = \{a^+, b^+\}$, $T'' = \{a^+, c^-\}$ and $T''' = \{b^+, c^-\}$.

2. The vector corresponding to marking m is: $a = 0, b = 0, c = 1, d = 1, e = 1, o = 1$.

3. The generated cubes are: $c' = cdeo$, $c'' = \bar{b}deo$ and $c''' = \bar{a}deo$.

4. Each cube covers vertex $\bar{a}\bar{b}cdeo$, corresponding to m, and belongs to the on-set cover. So on-set vertex $\bar{a}\bar{b}cdeo$ of f is covered.

5. Every cube covers only vertices where o^- is not enabled, so no off-set vertex (such as $ab\bar{c}deo$) is improperly covered.

Figure 5.2 contains an STG fragment and the corresponding STD fragment to illustrate case 2. Let o be the signal for which we are generating cover cubes in marking m.

1. The only two transitions that can fire in m are either a^+ or c^- (not both, since this is a multi-successor place, so it enables only one successor transition). Both enable o^-.

2. One example of a marking m'' predecessor of m is represented as white dots on the STG (replacing the token on $b^+ \rightarrow o^-$).

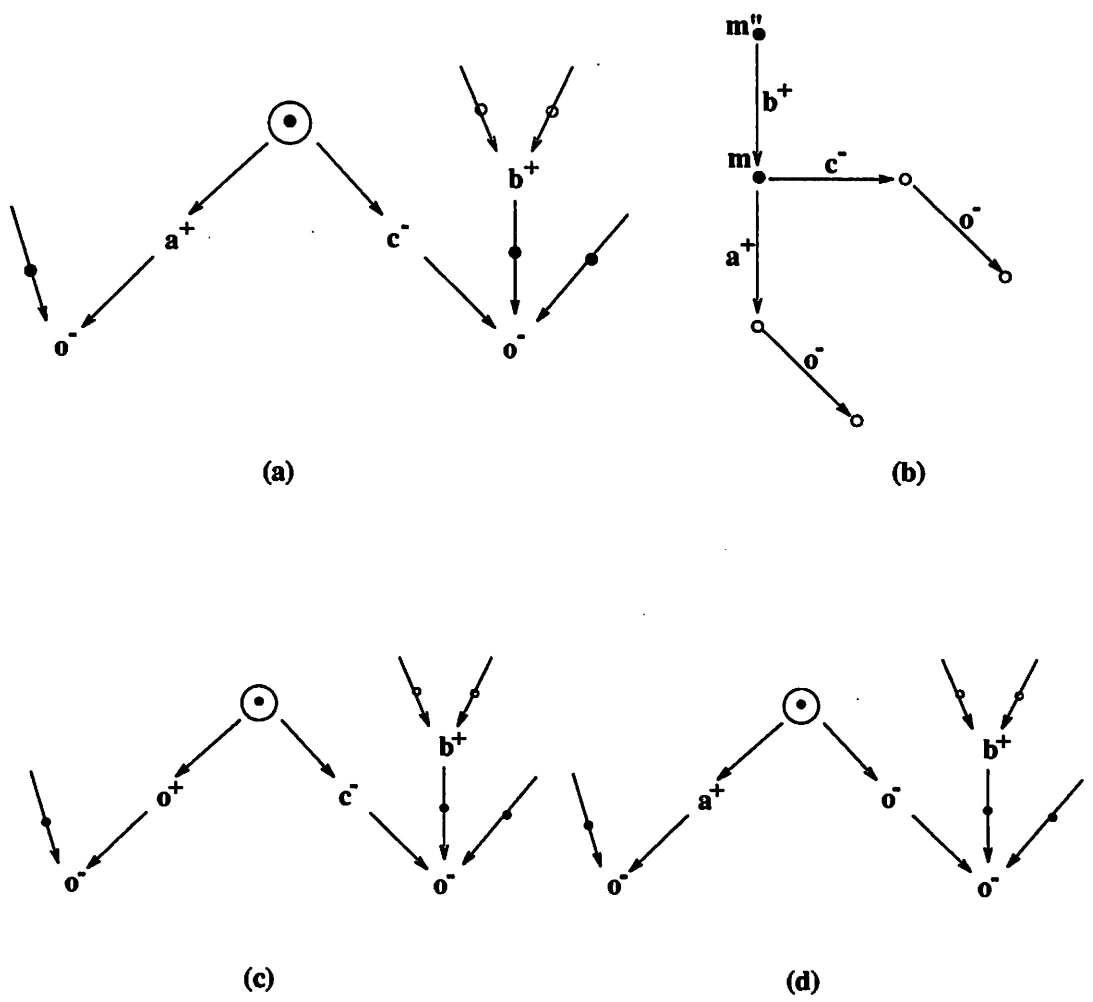


Figure 5.2: Illustration of case 2 of Theorem 5.2.1

3. One of the cubes generated in m'' is $c' = \overline{a}co$, so on-set vertex $\overline{a}\overline{b}co$ corresponding to m is covered.

4. If one of the enabled transitions in m had been either o^+ or o^-, instead of a^+ or c^-, then it is clear that

   - either the STG would not have had the CSC property (o^+ followed by o^-, Figure 5.2.(c)),
   - or it would not have been correct (o^- followed by o^- Figure 5.2.(d)).

Various authors (see [62], [130] and Chapter 3), pointed out that the assumption to start from a *correct* STG can be unnecessarily *restrictive*, as there are some useful asynchronous circuit behaviors that cannot be described with live safe free-choice Petri nets. The reader can check that Procedure 5.2.1 and the proof of Theorem 5.2.1 rely only on a more relaxed set of assumptions. Namely we only need the STG to be:

1. *valid*, i.e. the associated STD is finite and has a consistent labeling with vectors of signal values. Obviously step 1a of Procedure 5.2.1 in this case must be replaced with an exhaustive reachability analysis.
2. with *Complete State Coding*.
3. *output-persistent*, because we can synthesize only deterministic circuits.

5.2.2 Optimized Multi-level Circuit Implementation

Once we have the on-set and off-set covers of the next state function f for each output signal, we can choose how to implement the feedback loop (sequential part), and apply known logic synthesis techniques in order to obtain a minimal implementation of the combinational part.

The feedback loop can always be implemented using a simple flip-flop, due to the following Theorem, first proved in [86] in the restricted case when the STG is *persistent*. Note the close analogy with the result on the *perfect implementation* of semi-modular STDs ([126], see also Section 2.5.2).

Theorem 5.2.2 *Let $\mathcal{G}$ be a correct* STG *with* Complete State Coding. *Let F be an on-set cover of the next state function f of signal z_i derived from $\mathcal{G}$ according to Procedure 5.2.1.*

Then F is monotone increasing *in z_i.*

An intuitive reason for this is that if f is binate or decreasing in z_i, then there is a set of input values for which z_i oscillates. Moreover, if f is monotone increasing, then every prime cover of f must be monotone increasing.

Proof Let us assume, for the sake of contradiction, that F is binate or monotone decreasing in z_i.

Then there exists at least one vertex $v' = \overline{z_i}\beta$, where $\beta \in \{0,1\}^{n-1}$, belonging to the on-set of f, whose corresponding vertex $v'' = z_i\beta$ belongs to the off-set of f (otherwise we could always cover both v' and v'' with a prime implicant not depending on z_i).

The value of f in vertex v' is the complement of the value of z_i in v', so z_i^+ is enabled in the marking m' corresponding to v'. The marking obtained firing z_i^+ corresponds exactly to v'', since v'' differs from v' only in the value of z_i.

Similarly the value of f in vertex v'' is the complement of the value of z_i in v'', so z_i^- is enabled in the marking m'' corresponding to v''.

m'' is obtained by m' through the firing of z_i^+, so a firing of z_i^+ would immediately enable z_i^-. The same argument as in case 2 of Theorem 5.2.1 can be used to show that this case is either trivial or impossible. ■

A function with a monotone increasing cover must be monotone increasing, so:

Corollary 5.2.3 *Let* $\mathcal{G}$ *be a correct* STG *with* Complete State Coding. *Let* f *be the next state function of signal* z_i*, derived from* $\mathcal{G}$ *according to Procedure 5.2.1. Then* f *is* monotone increasing *in* z_i*.*

If f is monotone increasing in z_i, then there exist two logic functions s and m that do not depend on z_i, such that $f = s + z_i m$. So we can partition the circuit into two purely combinational parts, s and m, and a *set-dominant SM* flip-flop, that is a flip-flop with logic equation $Q = S + QM$ ([9]). This decomposition, when applied to the two-level implementation of cover F, uses only the distributive property, so according to Theorem 5.1.3 it does not change the hazard properties of the circuit.

We can also use Theorem 5.1.3, adding an inverter to M, to implement the feedback loop with the more usual *set-dominant SR* flip-flop, with logic equation $Q = S + Q\overline{R}$, without changing the hazard properties of the circuit.

Similarly we can use a C element, with logic equation $Q = S\overline{R} + SQ + \overline{R}Q$, if the cover F can be factored according to the given equation. There is no guarantee, though, that this factorization is always possible, unlike with *SM* or *SR* flip-flops, where Theorem 5.2.2 guarantees the existence of the appropriate factorization.

We assume in the following that the flip-flop is ready to accept a new transition on its inputs (i.e. that the internal feedback loop is in a stable state) whenever the output Q makes a transition. This means that the internal feedback loop delay must be smaller than the delay on any other circuit path from the flip-flop output to one of its inputs.

The combinational logic part of the circuit implementing each STG signal can be derived indifferently either from an *on-set* cover or an *off-set* cover of each signal (inverting the output if necessary). In the following we will discuss only the on-set cover implementation, but the results apply also to the off-set cover implementation.

The objective of logic synthesis is to obtain an implementation that is minimal with respect to some cost function, usually a combination of delay, area and testability. The starting point should ideally be a *prime* and *irredundant* cover of the desired function, because:

1. heuristic two-level logic minimizers can obtain prime and irredundant covers whose implementation has a nearly minimum area among all two-level implementations of the function ([15]).

2. a two-level implementation of a logic function obtained from a prime and irredundant cover is fully testable for single stuck-at faults ([64]).

3. a prime and irredundant cover is a good starting point of multi-level logic synthesis systems ([16]).

On the other hand we want to preserve Property 5.2.1, because it is tightly connected with the hazard properties of the implemented circuit. Unfortunately we cannot remove *redundant* cubes from the output of Procedure 5.2.1, because we must guarantee that ***each cube in the original on-set cover is covered by some prime.***

Figure 5.3.(a) contains an example of a correct STG with CSC whose two-level implementation of the flip-flop excitation function m according to Procedure 5.2.1 is redundant. Figure 5.3.(b) contains the STD (input variables are ordered a, b, c, t), while Figure 5.3.(c) contains the Karnaugh map of the function m.

The initial cover F of f is:

$$a\bar{b}t + a\bar{c}t + \bar{a}b + \bar{a}ct + b\bar{c}t$$

The cover of s (i.e. the set of cubes in F that do not depend on t) is:

$$\bar{a}b$$

while the cover of m (i.e. the set of cubes in F that depend on t, cofactored against t) is:

$$a\bar{b} + a\bar{c} + b\bar{c} + \bar{a}c$$

where the implicant $a\bar{c}$ is redundant (it is shown by the dashed oval on the Karnaugh map, while non-redundant implicants are shown by dotted ovals). If the redundant implicant is removed from the cover, then a hazard can occur when the circled b^- transition fires, because the implicant $b\bar{c}$ could go to 0 before the implicant $a\bar{b}$ goes to 1. This causes a static 1-hazard, and possibly a malfunction in the circuit, since the *SM* flip-flop can be set incorrectly due to this hazard.

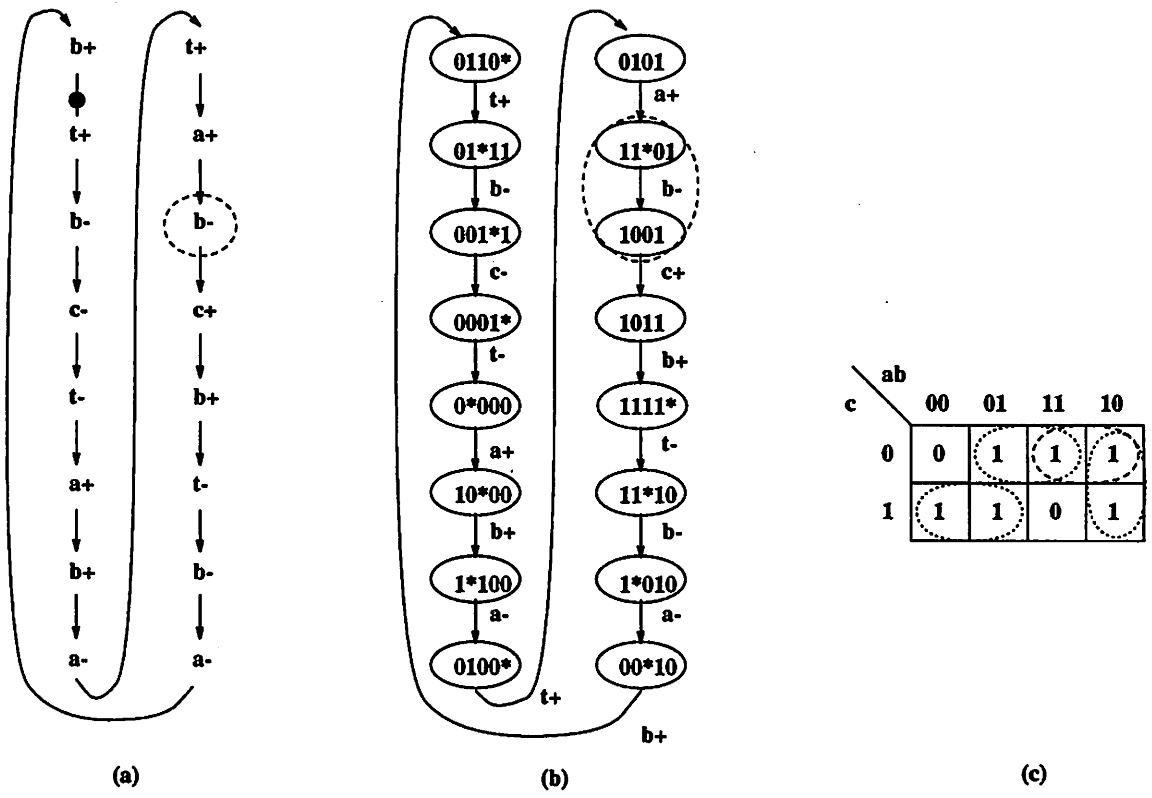


Figure 5.3: A Signal Transition Graph requiring a redundant two-level implementation

A more realistic example of the same kind of problem can be found in Figure 2.23 (taken from [27]). An on-set cover for signal A_i is $D\overline{L} + D\overline{R_i} + L\overline{R_i}$ and an off-set cover for it is $R = \overline{D}\,\overline{L} + \overline{D}R_i + LR_i$. Both covers are redundant (cube $D\overline{R_i}$ can be removed from the on-set cover and cube $\overline{D}R_i$ can be removed from the off-set cover). Unlike Figure 5.3, Ai has also valid covers that are not redundant, but it shows that redundant covers can indeed arise in practice, and not only from "crafted" STG's.

Up to now, we have dealt mainly with two-level implementations of logic functions. A two-level implementation assumes that there is no limit on the fanin and fanout of a gate, and this may not be realistic in the chosen technology. So if we want to implement the covers of s and m in a specific technology and improve the area and/or delay performance of the circuit, then we can use some multi-level logic synthesis techniques (e.g. those described in [16], [32] and [108, 107]). In order to retain the hazard properties of the two-level circuit, though, we must restrict ourselves to the transformations listed in Theorems 5.1.3 and 5.1.4 (and, possibly, those listed in [68]). The algebraic factorization operation, for example, is a direct application of associative and distributive laws, so it is covered by Theorem 5.1.3, while tree-based technology mapping is covered by Theorem 5.1.4.

5.3 Static Hazard Analysis of the Circuit Implementation

The previous sections described the hazard behavior of the circuit implementing each STG output signal under *concurrent* transitions. We still have to analyze what may happen under *non-concurrent* transitions, because, as we shall see, delays in the circuit may *violate* this ordering, and cause hazards.

We will use again the techniques described in Section 5.1.1 on the initial *two-level* implementation. The results automatically hold also for *multi-level* circuits obtained with constrained logic synthesis, thanks to Theorems 5.1.3 and 5.1.4. Section 5.3.1 gives necessary and sufficient conditions on the *circuit delays* for this implementation to be hazard-free. Section 5.3.2 describes a practical procedure to perform hazard detection on a circuit implementation given *delay bounds*. Section 5.3.3 compares our results with the claims of [27].

5.3.1 Necessary and Sufficient Conditions for Hazard-free Implementation

We must analyze the hazard behavior of a two-level implementation of the *next state* function of each STG output signal, as obtained from Procedure 5.2.1, under each *transition cube*

c corresponding to a *valid state pair* (s', s''). Recall that $\lambda(s')$ denotes the STD label of state s'.

A **path** on a transition cube c associated with a state pair (s', s'') is defined as a sequence of vectors of signal values $\lambda(s') = v^0 \rightarrow v^1 \rightarrow v^2 \ldots \rightarrow v^n = \lambda(s'')$ such that the Hamming distance between each pair $d(v^i, v^{i+1})$ is 1 and $v^i \sqsubseteq c$ for all i. Note that we do not require the path to be simple[2], i.e. vectors can repeatedly appear in it.

The following Theorem is the fundamental result of this chapter, and is the basis of the hazard elimination procedure as well:

Theorem 5.3.1 *Let $\mathcal{G}$ be a* correct STG *with* Complete State Coding. *Let C be a two-level circuit implementation of the next state function f for signal z_h, derived according to Procedure 5.2.1 from $\mathcal{G}$. Suppose that the interaction between C and its environment obeys $\mathcal{G}$.*

Then no static hazard *can occur with respect to signal z_h if and only if:*

- *for each valid state pair (s', s''), such that the associated transition cube c covers both on-set and off-set vertices of f,*
  - *for each pair of transitions (z_a^*, z_b^*) such that*
    - *they are not concurrent and*
    - *they belong to a firing sequence of $\mathcal{G}$ from the marking corresponding to s' to the marking corresponding to s'', in the order $z_a^* \rightarrow \ldots \rightarrow z_b^*$,*
    - *there exists an* STD *state s''', labeled v''', between s' and s'' such that:*
      - · *z_a^* is enabled in the corresponding marking and*
      - · *the label obtained from v''' by toggling signal z_b has a next state function value different from v''',*

    (i.e. an inversion in the firing order of z_a^* and z_b^* causes the value of f to change)

  the effect of z_b^* cannot, under the bounded wire *delay hypothesis, propagate to z_h before the effect of z_a^*.*

Proof

$\Leftarrow$ Suppose that there exists a pair of transitions such that the effect of z_b^* can propagate to z_h before the effect of z_a^* and such that an inversion in the firing order of z_a^* and z_b^* can cause the value of f to change. Then obviously we have a hazard at z_h.

[2]A **simple path** of a directed graph is a path such that no node appears twice in it.

$\Rightarrow$ Suppose that a static hazard can occur in the circuit. We will show that the order of a pair of transitions as described in the Theorem statement is reversed.

According to [42] a static hazard occurs only if we have applied to C a transition cube c that was not covered by a single on-set or off-set cube. Moreover, the transition cube must be associated with a *valid* state pair (s', s'') because the hazard is *static*.

We have the following cases:

1. $f(\lambda(s')) = f(\lambda(s'')) = 1$. Let us assume that two distinct cubes of C, say c' and c'', are required to cover $\lambda(s')$ and $\lambda(s'')$ (if more than two cubes are required to cover c, then we can just pick another valid state pair, belonging to some STD path between s' and s'', such that two cubes are sufficient to cover their associated transition cube). Then there exists some path $v' \rightarrow v''$ on c that is not completely contained in the on-set of C (i.e. some vertex v belonging to that path is not covered by any cube of C). Otherwise we could cover both c' and c'' with a single cube, contradicting the assumption that C is prime.

   For each such path that leaves and re-enters the on-set of C at most once[3], we can find two transitions z_a^* and z_b^* such that:

   - z_b^* and z_a^* are not *concurrent* (otherwise they would be covered by a single cube),
   - z_b^* turns c' off. I.e. there exists a pair of STD states s_{b1} and s_{b2} such that $\lambda(s_{b1}) \sqsubseteq c'$, $\lambda(s_{b2}) \not\sqsubseteq c'$ and $s_{b1} E(z_b^*) s_{b2}$.
   - z_a^* turns c'' on. I.e. there exists a pair of STD states s_{a1} and s_{a2} such that $\lambda(s_{a1}) \not\sqsubseteq c''$, $\lambda(s_{a2}) \sqsubseteq c''$ and $s_{a1} E(z_a^*) s_{a2}$.

   Note that the transitions must occur in the order $z_a^* \rightarrow \ldots \rightarrow z_b^*$ in the STG, in order for c' and c'' to change value in the right order and keep z_h at 1.

2. The case when $f(\lambda(s')) = f(\lambda(s'')) = 0$ and c intersects some cube c' of C is completely analogous, as we can find a pair of non-concurrent transitions that turn on and off c' respectively.

■

The pair of states labeled (0*00, 110)* for the STD in Figure 1.3.(c) is an example of case 1 for output x. Here $c' = \overline{y}\,\overline{z}$, $c'' = x\overline{z}$, $z_b^* = y^+$ and $z_a^* = x^+$, ordered $x^+ \rightarrow y^+$ on the STG.

[3]Again, if the path leaves and re-enters more than once, we can find another path that does it only once.

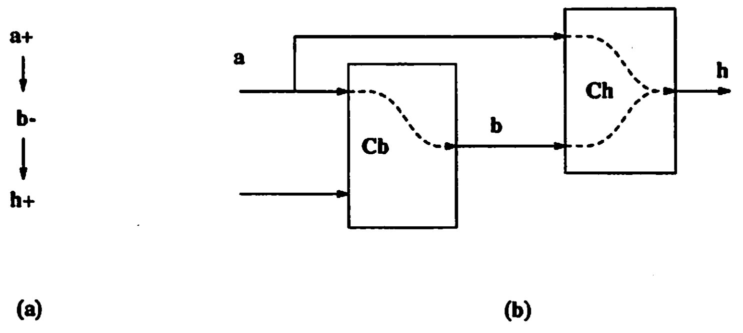


Figure 5.4: A Signal Transition Graph fragment and its implementation.

Note that the implied value for x is 1 in all the states. The pair of states labeled (001, 010) for the same STD is an example of case 2 for output x.

Both hazard cases occur, informally, when the sub-circuit behaves as though *the* STG-*specified ordering of* z_b^* *and* z_a^* *had been reversed.* This means that the physical circuit implementation must preserve the transition ordering:

Property 5.3.1 *Given an* STG $\mathcal{G}$ *and a circuit* $\mathcal{C}$ *implementing it, if a transition* z_a^* *on the input of a sub-circuit* C_b *causes a transition* z_b^* *on its output, then no other sub-circuit* C_h *can produce a sequence of events on its output as a delay-free implementation would have produced if* z_b^* *had preceded* z_a^* *in time.*

Note the close similarity between Property 5.3.1 and the classical definition of *essential* hazards. We will see that the methods for eliminating both are also quite similar.

For example in the circuit fragment schematically shown in Figure 5.4 we must make sure that every circuit path connecting $a \rightarrow b \rightarrow h$ through C_b and C_h has a longer delay than any circuit path $a \rightarrow h$ in C_h only.

A suitable global handshaking protocol can guarantee that Property 5.3.1 is met under the *unbounded wire* delay model.

For example we can encode each external signal z_i with two wires, $z_{(i,0)}$ and $z_{(i,1)}$, carrying complementary values during quiescent conditions (*dual-rail* encoding), and driven with wired-or logic.

Under quiescent conditions all sub-circuits drive one of the wires to 0, say $z_{(i,0)}$, and leave the other wire at 1. When a transition z_i^* must occur on signal z_i, the sub-circuit that generates its transitions drives $z_{(i,1)}$ to 0 and releases $z_{(i,0)}$. All other sub-circuits release $z_{(i,0)}$ as soon as they recognize the transition. Then $z_{(i,0)}$ will go to 1 only when all sub-circuits agree that the transition fired. Now any transition enabled by z_i^* can fire, and so on.

This approach has the obvious disadvantage of requiring additional logic for dual-rail encoding and completion detection, but it can be useful when wire delays are very large or impossible to control.

Another way to obtain the same result, under the *bounded wire* delay model, is to *slow down* the output of C_b so that we are sure that every circuit having a as input is stable when we generate a transition on b. Section 5.4 will give an *optimum* algorithm, based on linear programming, for this purpose. The circuit will still work properly with this increased delay on the output of C_b, because we are changing the delay of an STG signal. So the circuit still follows the specification, since the firing time of an STG transition can be arbitrary.

The next section describes a practical algorithm to analyze, given a set of known bounds on the circuit delays, when some pair of transitions can cause a hazard on another circuit signal.

5.3.2 Hazard Detection with Bounded Delays

In the previous section we described the precise conditions under which a static hazard can occur, namely when *the difference between the delays along two paths in one sub-circuit is greater than the delay between two transitions*. In this section we give an algorithm for detecting such hazards in practice.

The hazard detection procedure below must be called once for each output signal z_h, with next state function f, implemented by cover F according to Procedure 5.2.1. Let R' be a cover of a completely specified logic function that has value 0 on the vertices covered by a cube in F and value 1 otherwise (it can be obtained by complementing cover F, as described, e.g, in [15]). Let $d(c', c'')$ be the Hamming distance between cubes c' and c''.

Procedure 5.3.1

For each STD *state pair* (s', s''), *labeled* $v' = \lambda(s')$ *and* $v'' = \lambda(s'')$ *respectively, such that*

- $f(v') = f(v'') = f(\lambda(s'''))$ *for all* s''' *on an* STD *path between* s' *and* s'', *and*
- s' *and* s'' *have a maximal distance (i.e. number of traversed edges) among such* STD *states do:*

1. *let* c *be the transition cube associated with* (s', s'').
2. *if* $f(v') = f(v'') = 1$ *then:*

```
    (a) for each pair of cubes $c^{Rb}, c^{Ra} \in (R' \cap c)$ such that
        – $d(v', c^{Rb}) \leq d(v', c^{Ra})$,
        – $d(c^{Rb}, v'') \geq d(c^{Ra}, v'')$
          (i.e. they occur in order $c^{Rb} \rightarrow c^{Ra}$ on a path on c between $s'$ and $s''$),
      do:
        i. for each pair of cubes $c^{Fb}, c^{Fa} \in (F \cap c)$ such that
           – $d(v', c^{Fb}) \leq d(v', c^{Fa})$,
           – $d(c^{Fb}, v'') \geq d(c^{Fa}, v'')$,
             (i.e. they occur in order $c^{Fb} \rightarrow c^{Fa}$ on a path on c between $s'$ and $s''$),
           – $d(c^{Fb}, c^{Rb}) = 1$,
           – $d(c^{Fa}, c^{Ra}) = 1$,
           do:
          A. let $z_b^*$ be the transition moving from $c^{Fb}$ to $c^{Rb}$
             (i.e. $z_b^-$ if $z_b \in c^{Fb}$, $\overline{z_b} \in c^{Rb}$, and $z_b^+$ if $\overline{z_b} \in c^{Fb}$, $z_b \in c^{Rb}$; there exists
             only one such signal $z_b$, since $d(c^{Fb}, c^{Rb}) = 1$).
          B. let $z_a^*$ be the transition moving from $c^{Ra}$ to $c^{Fa}$.
          C. if $z_b^*$ and $z_a^*$ both occur on at least one STD path between $s'$ and $s''$ where
             $f(v') = f(v'') = f(\lambda(s'''))$ for all $s'''$ on the path, then:
             – let $d_{bh}$ be a lower bound on the delay for transition $z_b^*$ along the circuit
               path from input $z_b$ to $z_h$ corresponding to cube $c^{Fb}$
               (there exists only one such path if we optimize the initial two-level function
               only with the distributive and associative law).
             – let $d_{ah}$ be an upper bound on the delay for transition $z_a^*$ along the circuit
               path from input $z_a$ to $z_h$ corresponding to cube $c^{Fa}$.
             – let $d_{ab}$ be a lower bound on the delay between transition $z_a^*$ and $z_b^*$.
             – if $d_{ah} > d_{ab} + d_{bh}$ then a hazard condition exists.
3. else ($f(v') = f(v'') = 0$):
    (a) do the same, exchanging $F \leftrightarrow R'$
```

This procedure obtains exactly the same set of triples (z_a^*, z_b^*, z_h) as the case analysis in Section 5.3.1, because:

- z_b^* turns c^{Fb} off and z_a^* turns c^{Fa} on, as required. Moreover there exists at least one path from s' to s'' including z_b^* and z_a^*. Finally, the change in order between them causes to enter an off-set cube intersecting the transition cube. So the algorithm finds only correct transition pairs.

- Any path on c that leaves and re-enters F:

  1. leaves F at the boundary between some cube $c^{Fb} \in F$ and some cube $c^{Rb} \in R'$, with some transition z_b^*,
  2. re-enters F again at the boundary between some cube $c^{Ra} \in R'$ and some cube $c^{Fa} \in F$, with some transition z_a^*.

  Moreover c^{Fb} and c^{Rb} differ in the value of z_b, so their distance is 1, and c^{Ra} and c^{Fa} differ in the value of z_a, so their distance is 1. So the algorithm finds all the correct transition pairs.

Note that similar conditions for hazard manifestation could be derived using the approach of [55, 56] for hazard analysis in combinational circuits with bounded delays, but only under the *Fundamental Mode* hypothesis. We do not require such hypothesis, because our only constraint is that the environment where the circuit operates satisfies the STG specification.

The worst case running time of the algorithm can be pessimistically estimated as follows. Suppose that the STG has n signals. Then there can be at most $O(2^n)$ STD states and $O(2^n)$ cubes in F and R'. The main loop selects each state in turn, and performs a depth-first search of the STG with complexity $O(2^{2n})$ in order to find the second member of each pair, with total complexity $O(2^{3n})$. The loop body is executed $O(2^{2n})$ times, and in the worst case it can involve $O(2^{4n})$ cubes. Each pair can be checked for Hamming distance in $O(n)$ operations. So the worst case running time of the algorithm is $O(n2^{6n})$.

In practice, though, a very simple depth-first strategy allows to select only STD states with a maximal distance, so that much fewer pairs must be examined, and the number of cubes both in F and R' is rather small.

The analysis in Section 5.3.1 was performed on a *two-level implementation* of the next state function, using the *unbounded wire* delay model. But, as proved in Theorems 5.1.3 and 5.1.4, we can use constrained multi-level logic synthesis to optimize and implement the function, and obtain a circuit without any new *potential* hazard that uses logic gates from any available library (*nand*, *nor*, *inverter*, *ex-or*, *SR* flip-flops, *C*-elements, ...).

Now we can use the delay information from the implementation in order to measure bounds on d_{ah}, d_{bh} and d_{ab} and hence verify which hazards, predicted using the pessimistic *unbounded* wire delay model, will actually be present if we use the more realistic *bounded* wire delay model. Moreover we can use the same information to *remove* those hazards, and produce a *hazard-free* implementation of the STG specification, as shown in Section 5.4.

Basically, we will perform a *timing analysis* step to check which inequalities, of the form $d_{ah} > d_{ab} + d_{bh}$, are satisfied, thus producing a hazard, and then *pad delays* in order to make those inequalities false.

5.3.3 Signal Transition Graph Persistency and Hazards

Given the results on hazards in a circuit implementation of an STG specification obtained in the previous section, we now turn our attention towards a property of the STG specification that was previously considered *necessary and sufficient* to obtain a hazard-free implementation.

An STG transition z_i^* is defined to be **Chu-persistent** ([27]) if for each immediate predecessor z_j^* of z_i^*, z_i^* and $\overline{z_j^*}$ are not concurrent. Note that this definition of persistency is *different* from the classical definition of PN persistency that we used in Chapter 3. An STG is defined to be **Chu-persistent** if all its transitions are Chu-persistent. For example in Figure 1.3.(b) transition y^+ is not Chu-persistent, because it has x^+ as a predecessor, but x^- and y^+ are concurrent. Note that the same transition is persistent according both to the classical PN definition and our definition given in Section 3.3.2.

In a Chu-persistent STG, whenever a transition z_i^* becomes enabled, none of its enabling signals z_j can change level before z_i^* has fired. Chu-persistency was considered to be a necessary and sufficient condition for the existence of a hazard-free circuit implementation, due to the following Theorem, taken from [27] (only the notation is changed here, for consistency with the rest of the chapter):

Theorem 5.3.2 *Let $\mathcal{G}$ be a live* STG[4].

For each output signal z_i, there exists a signal z_j and a marking m in $\mathcal{G}$ such that:

- *z_j^* and z_i^* are enabled in m and*
- *z_j^** disables *z_i^* and*

[4]Recall that Chu's definition of *live* STG is similar to, but stronger than, our definition of *correct* STG.

- z_i^* does not disable z_j^* *(this means that z_j^* and z_i^* are not both successors of a multi-successor place, otherwise z_i^* would disable z_j^* as well)*

if and only if $\overline{z_j^*}$ is a predecessor of z_i^*, and z_i^* and z_j^* are concurrent (that is z_i^* is non-Chu-persistent).

The case in which z_i^* is disabled by z_j^* but not vice-versa intuitively seems to be dangerous, because if the circuit implementing signal z_i is not "fast enough" to fire after $\overline{z_j^*}$ fires, then a firing of z_j^* may prevent z_i^* from firing. So we could have a potential hazard, depending on the delay of the circuit implementing z_i and the time from $\overline{z_j^*}$ to z_j^*. On the other hand a simple check of the STG Chu-persistency would be enough to guarantee that no such hazard occurs.

The proof of this Theorem was based not only on STG properties, since strictly speaking no output signal transition can be disabled an a *well-formed* STG. It was in fact based on specific assumptions on the *circuit implementation* derived from the STG, namely that if z_i^* is enabled by $\overline{z_j^*}$, then an occurrence of z_j^* must disable z_i^* in the circuit implementation of signal z_i. This is not true in general, but only if the output of the sub-circuit implementing signal z_i is sensitive to the value of signal z_j in marking m.

See for example the STG in Figure 1.3.(b), where y^+ is not Chu-persistent. The logic equation for y is $y = x + z$, and using the *unbounded gate* delay model, as assumed by [27], we know that x^- is caused by z^+, so when x^- fires z is already at 1 (which determines the output of the *or* gate independent of the value of x), and x^- cannot disable y^+. So *Chu-persistency is not necessary for a hazard-free implementation*, because the implementation described in Figure 1.5 of this non-Chu-persistent STG is hazard-free using the unbounded gate delay model that [27] used.

On the other hand Theorem 5.3.2 guaranteed hazard-free implementation only if the whole sub-circuit implementing each signal z_i could be satisfactorily modeled as a *single gate* with the *unbounded gate* delay model. But the gate delay model is a reasonable approximation of reality only for *simple gates*, such as a *nand* or a *nor*, or, in the most optimistic case, for *and-or-not* gates as in the approach described in Section 2.5.2.

We can examine now the example in Figure 5.5 (from [27]), where a circuit implementation was derived from a Chu-persistent STG (only a fragment is shown here). The value of signals in the given marking is $La = 0$, $Lr = 1$, $Sa = 0$, $Sr = 1$, $Ca = 0$ and $Cr = 0$. When La^+ fires, then the output of a_3 has a rising transition. Suppose that the gate delay of a_3 is greater than the gate delay of a_2 plus the delay between Sa^+ and Sr^-. Then a hazard occurs on Lr. The only way to avoid the hazard in the unbounded gate delay model is to assume that the whole group of gates i_1, a_3 and o_1 can be modeled with a *single delay* on the output of o_1. So *Chu-persistency is not*

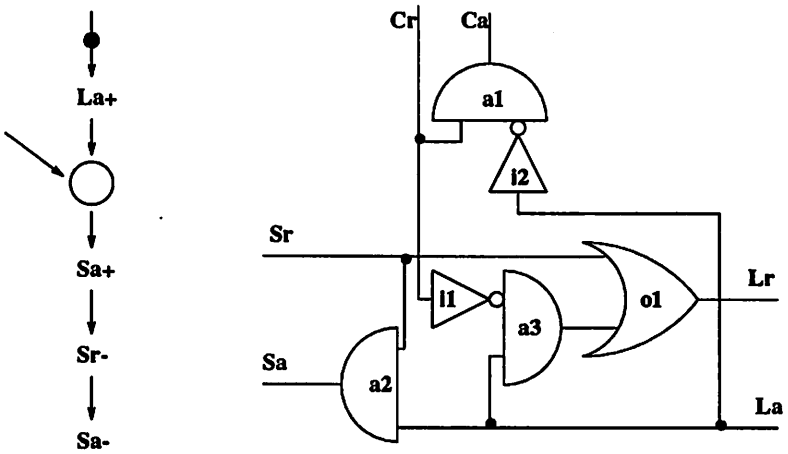


Figure 5.5: A Signal Transition Graph fragment and its implementation

$$\begin{tikzcd}
 & z3- \arrow[r] & z2+ \arrow[drr] & & & & & \\
z2- \arrow[ur] \arrow[dr] & & & & z3+ \arrow[r] & z1- \arrow[r] & z5- \arrow[r] & z4+ \arrow[lllllll, bend right] \\
 & z5+ \arrow[r] \arrow[uur] & z4- \arrow[r] & z1+ \arrow[ur] & & & &
\end{tikzcd}$$

Figure 5.6: A Chu-persistent Signal Transition Graph with a term takeover

sufficient for a hazard-free implementation, unless under the complex gate delay model (that may be considered too unrealistic for most practical purposes).

We may now think that Chu-persistency ensures at least that each output signal has an implementation with an *SR* flip-flop and a pair of *and* gates (or, strictly speaking, an *and* gate for each excitation region of that signal in the STD). In other words, Chu-persistency would be sufficient to guarantee an implementation without any *term takeover*. Unfortunately even this turns out to be false, because [126] reports an example of a Chu-persistent STG whose STD has a term takeover. The STG appears in Figure 5.6, and it has a takeover for the excitation region S_1 of signal $z_1 = (\overline{z_2} + \overline{z_3})\overline{z_4}$ due to the STD sequence 0*01*01 $\rightarrow$ 0*0*001 $\rightarrow$ 0*1001 (for signal ordering $z_1 z_2 z_3 z_4 z_5$).

In conclusion we can state that:

Theorem 5.3.3 *Let $\mathcal{G}$ be a correct* STG *with* Complete State Coding. *Let C be a circuit implementation of the signals in $\mathcal{G}$ according to Procedure 5.2.1 and the decomposition described in Section 5.2.2. If the implementation of each combinational circuit exciting each flip-flop input, as* *well as each flip-flop, can be modeled as a single gate* with non-zero delay, *then C is hazard-free with the* unbounded gate *delay model.*

Proof Each signal z_i in $\mathcal{G}$ is implemented by a circuit with non-zero delay, so every $d_{ab} > 0$. Moreover both d_{bh} and d_{ah} are delays within the same gate, implementing either s or m, so $d_{bh} = d_{ah}$, and $d_{ah} - d_{bh} = 0 < d_{ab}$. ■

Note that the assumption that each excitation function may be modeled as a single gate was used, as shown above, by [27] and [81].

This completes the hazard analysis task. Now we can use the results derived so far to obtain a circuit implementation that is free from hazards, as described in the next section.

5.4 Hazard Elimination by Linear Programming

This section concludes our hazard-free synthesis methodology. We first show that the problem of minimum delay padding to ensure hazard elimination is NP-hard. Then we give a greedy procedure that shows the existence of a solution to this problem, a branch-and-bound procedure that finds an optimum solution by repeated application of a linear program, and a heuristic procedure that improves the worst case running time at the (possible) expense of optimality. Finally we show that there are cases in which the *environment* must be slowed down in order to achieve a hazard-free implementation, when pairs of events occur too close together for the underlying technology to be able to discriminate between them

The procedures described in this section can take into account delay constraints originated both from the specification and from the need to produce a hazard-free circuit. Note that *minimum* delay constraints can always be satisfied by delay padding, while *maximum* delay constraints may not. This can be due either to the delays inherent in the chosen logic implementation, and in this case logic synthesis ([110]) can be used to solve the problem, or to inconsistencies between minimum and maximum constraints.

5.4.1 Complexity of the Optimum Delay Padding Problem

This section proves that the optimum delay padding is at least as hard as the class of NP-complete problems ([44]). So it is highly unlikely that a polynomial time algorithm to find an exact solution to it will ever be found, justifying the introduction of a potentially exponential branch-and-bound procedure (Section 5.4.2).

The **delay padding** problem is as follows:

Problem 5.4.1 Given*:*

- *a Signal Transition Graph specification $\mathcal{G}$,*
- *lower bounds on delays between each input transition of $\mathcal{G}$ and its predecessors,*
- *an asynchronous circuit implementation $\mathcal{C}$, modeled with* pure bounded wire delays, *of the output signals of $\mathcal{G}$,*
- *a maximum* delay padding *value D, and*
- *a weight associated with padding each signal of $\mathcal{G}$.*

Question*: there exists a set of delays padded after signals specified by $\mathcal{G}$ such that:*

- *the weighted sum of the padded delays is at most D, and*
- *the new circuit $\mathcal{C}'$, with the modified lower and upper bounds on the padded delays, is* hazard-free.

A necessary and sufficient condition for the *existence*, given an STG/circuit pair as above, of some value of D such that Problem 5.4.1 has an affirmative solution, is that *every* signal in the STG can be padded, as shown in Theorem 5.4.4 below.

Problem 5.4.1 *most likely is not* NP-complete because, as shown in [20], *checking* that a circuit is hazard-free is NP-hard. So a guessed answer most likely cannot be verified in polynomial time (i.e. unless the hazard checking problem can be solved in polynomial time).

On the other hand, the problem is *at least as hard* as NP-complete problems because the known *hitting set* NP-complete problem can be solved using a polynomial time reduction to Problem 5.4.1.

The **hitting set** problem ([44]) is as follows:

Problem 5.4.2 Given*:*

- *a finite set $U = \{e_1, e_2, \ldots, e_n\}$,*
- *a finite collection of subsets of U, say S_i, $i = 1, \ldots, m$, and*
- *an integer K.*

Question: *there exists a subset H of U such that:*

- *$|H| \leq K$, and*
- *for each S_i in the collection, $\exists e_j \in H$ such that $e_j \in S_i$.*

We will show that for each instance I of Problem 5.4.2 there exists an instance I' of Problem 5.4.1 such that:

- I' can be obtained from I in polynomial time, and
- the answer (yes/no) to I' is the same as the answer to I.

Given I create an STG $\mathcal{G}$ with:

1. one input signal x_j, $j = 1, \ldots, n$ for each element e_j of U,
2. one output signal y_i for each set S_i,
3. one input signal z_i for each set S_i and
4. one additional input signal s.

The structure of the STG $\mathcal{G}$ is shown in Figure 5.7. Let X_i denote $\{x_j | e_j \in S_i\}$. Assume that all environment transitions are sufficiently separated (say by 5 delay units) except for the $s^- \rightarrow X_i^-$ and $X_i^- \rightarrow z_i^-$ transitions, which for each i have a delay of 0 (i.e. X_i^- and z_i^- occur at the same time as s^-).

Note that the STG $\mathcal{G}$ has a size that is polynomial in the size of the hitting set instance. We need to give an asynchronous implementation consistent with the signal transitions defined by $\mathcal{G}$. The function of each y_i for each subset S_i is chosen to be $\overline{s} + z_i$, i.e. an *or* gate with the inverted value of s and the signal z_i as its inputs (Figure 5.8). The circuit has a size that is polynomial in the size of the hitting set instance. There is a delay of 1 through each *or* and of ϵ (where $0 < \epsilon < 1$) through each inverter.

Let us examine the following hazard at y_i(see Figure 5.8). Assume that $s = 1$, $x_j = 1$, $\forall x_j \in S_i$ and $y_i = 1$. At $t = 0$, s falls, causing every x_j in S_i to fall which in turn cause z_i to fall. These transitions are instantaneous. During $0 \leq t < \epsilon$, the output of the delay element driven by the inverter is still 0 and there will be a glitch ($1 \rightarrow 0 \rightarrow 1$) at y_i in the time interval between 1 and 1+ϵ. This hazard can be eliminated by delaying the fall of some of the X_i or delaying the

$$\begin{tikzcd}
s^+ \arrow[r] & z_1^- \arrow[r] & y_1^- \arrow[r] & z_1^+ \arrow[r] & y_1^+ \arrow[r] & \cdots \arrow[r] & z_n^- \arrow[r] & y_n^- \arrow[r] & z_n^+ \arrow[r] & y_n^+ \arrow[drr, bend left] & & \\
z_n^+ \arrow[u, bend left] & X_n^+ \arrow[l] & z_n^- \arrow[l] & X_n^- \arrow[l] & s^- \arrow[l] & \cdots \arrow[l] & s^+ \arrow[l] & z_1^+ \arrow[l] & X_1^+ \arrow[l] & z_1^- \arrow[l] & X_1^- \arrow[l] & s^- \arrow[l]
\end{tikzcd}$$

Where: $\leftarrow z_i^- \leftarrow X_i^- \leftarrow s^- \leftarrow$ **represents:**

$$\begin{tikzcd}
& & x_{i1}^- \arrow[dll] & & \\
& & x_{i2}^- \arrow[dll] & & \\
z_i^- & & \cdots \arrow[ll] & & s^- \arrow[ll] \arrow[ull] \arrow[uull] \arrow[dll] & \arrow[l] \\
& & x_{il}^- \arrow[ull] & &
\end{tikzcd}$$

Figure 5.7: Signal Transition Graph structure for the NP-completeness proof

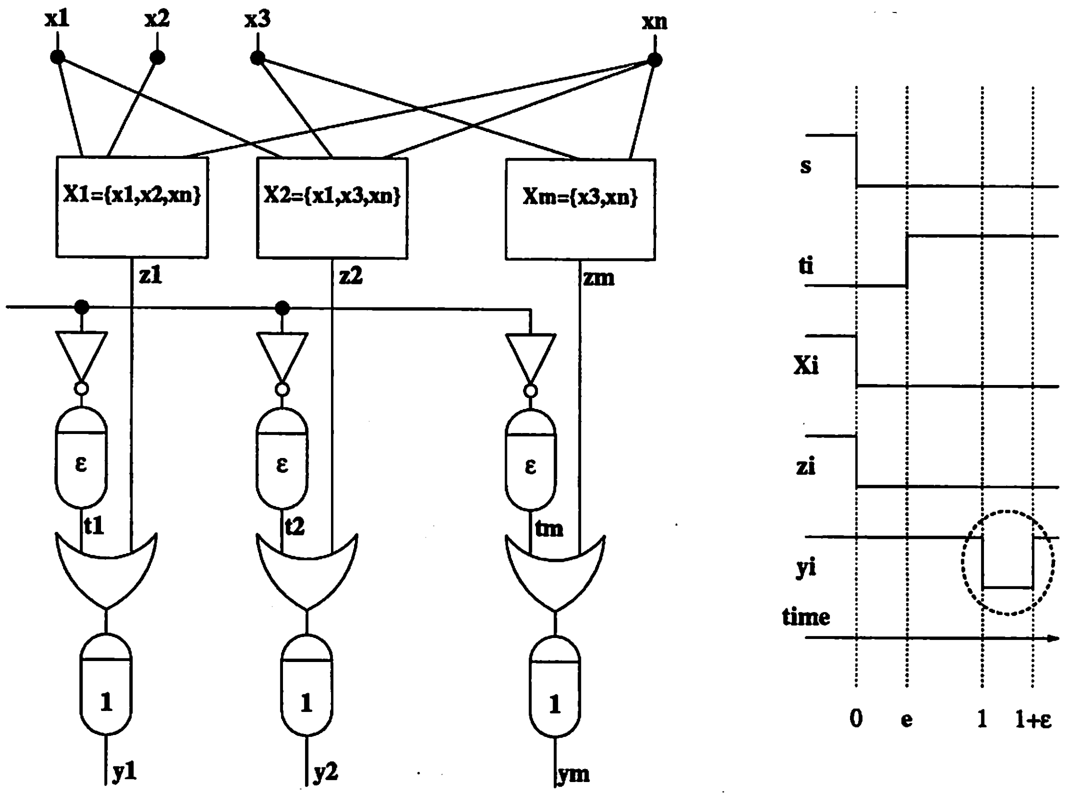


Figure 5.8: An asynchronous implementation of the Signal Transition Graph of Figure 5.7

signal z_i. If the circuit signals are weighted so that the weight of z_i for all i is larger than n, and the weight of x_i is 1, then a minimum weighted padding will always select a subset of the x's.

The reader can check that this is the only possible hazard for each output signal, assuming that the circuit is operated according to the STG with the given lower bound on the separation between input transitions.

We now claim that the circuit can be made hazard-free by *slowing down* a subset of the x_j signals by exactly ϵ units of delay each. Moreover, due to the STG construction, this can be done with a cost of $\leq K\epsilon$ if and only if the original hitting set problem had a solution with $\leq K$ elements.

Theorem 5.4.1 *The hitting set problem has a solution of cardinality less than or equal to K if and only if C can be made hazard free with a weighted sum less than or equal to $K\epsilon$*

Proof Since all environment transitions except the $s^- \rightarrow X_i^- \rightarrow z_i^-$ transitions (for each i) are slow, the only hazards in the circuit are those caused at each y_i due to these instantaneous transitions.

$\Leftarrow$ If the hitting set problem has a solution H with $|H| \leq K$, pad each x_i associated with $e_i \in H$ with a delay of ϵ. Since z_i^- can fire only after every x_j^- ($j \in S_i$) has fired, each of the z's is delayed by ϵ and the hazards are eliminated at a cost $\leq K\epsilon$.

$\Rightarrow$ If the circuit is hazard free, a subset of the x's and/or the z's must have been padded. If the weighted sum of the paddings $D(= K\epsilon)$ is greater than or equal to $n\epsilon$, we can choose the hitting set set to be U itself. So assume without loss of generality that $K < n$. The cost of padding a z_i is greater than n, so in effect if any z_i was padded with a delay θ we can delete this padding and pad every x_i by θ to obtain a hazard-free circuit at a lesser cost. Since all the hazards were eliminated, every z_i must have been delayed by at least ϵ, implying that for each z_i there must be at least one $x_j \in X_i$, which has been delayed (by at least ϵ). If any x_j was delayed more than ϵ we can decrease the weighted sum by reducing this amount to ϵ without introducing hazards. So we have a new hazard-free circuit, obtained as a result of the above transformations on the hazard-free circuit produced by solving Problem 3.1. The above procedure only decreases the amount of padding. So we have a circuit with padding only a subset of the x's by ϵ and the sum (weighted) of the padding is $\leq K\epsilon$. Hence the subset has size $\leq K$. This gives us a hitting set.

■

Most authors believe that NP-complete problems cannot be solved in polynomial time, so there is little hope that a polynomial time algorithm to optimally eliminate all hazards can ever be found. The next section is devoted to a global hazard elimination algorithm, using branch-and-bound and linear programming, that is shown in Section 5.6 to be able to solve asynchronous circuits of practical size in a reasonable running time.

5.4.2 A Branch-and-bound Global Hazard Elimination Procedure

In order to produce a hazard-free circuit, we can now estimate an **upper bound** on d_{ah} and a **lower bound** on d_{bh} and d_{ab} for each potential hazard detected by Procedure 5.3.1. If some inequality of the form $d_{ah} > d_{ab} + d_{bh}$ (or, equivalently, $(d_{ah} - d_{bh}) > d_{ab}$) is satisfied, then we have the following options to eliminate the corresponding hazard:

1. reduce the difference between each pair of failing d_{ah} and d_{bh}, by logic synthesis or by transistor sizing.

2. pad delays to slow down some STG signal, to increase the failing d_{ab} enough to satisfy all inequalities.

3. pad delays on the wires inside the sub-circuit implementing each output signal, to reduce the difference between each pair of failing d_{ah} and d_{bh}, in addition to increasing some failing d_{ab}.

Note that in cases 2 and 3 we have to satisfy *one-sided* inequalities on the delays, so there is no need to precisely control delays (even though more control over the delays means better performance for the resulting circuit).

This analysis is still *pessimistic*, because even a hazard whose associated inequality is satisfied may not manifest itself in the actual circuit because of the *inertial* nature of real circuit delays. In a real circuit, hazards shorter than a certain minimum amount are in fact *absorbed* by the gates, rather than just delayed as the *pure* delay model, used in our analysis, would predict. We can partially take this into account by subtracting from each $d_{ah} - d_{bh}$ the maximum width of a hazard that is guaranteed to be absorbed by any gate.

The operations in case 1 are hard to automate, because reducing the delay of some path to eliminate some hazard can actually make some other hazard *worse*. They may be usefully applied, however, to some parts of the circuit that are particularly critical with respect to timing. In the rest of the chapter we will assume that the designer has already used this option to the fullest extent, if desired.

Case 2 is obviously sufficient to eliminate all hazards. This can be shown by giving a a simple algorithm, based on the observation that increasing d_{ab} by padding delay after the corresponding signal z_b does not change the value of the *difference* between d_{ah} and the corresponding d_{bh} in *any hazard*.

Procedure 5.4.1

- *for each* STG *signal b*
    1. *let D_b be the maximum $(d_{ah} - d_{bh}) - d_{ab}$ among all the potential hazards for signal b, detected by Procedure 5.3.1.*
    2. *add a delay buffer with magnitude greater than D_b* after *the output of the circuit implementing signal b*

This greedy algorithm is not optimal, because it exploits the fact that delay padding to eliminate a hazard cannot make any other hazard *worse*, but it does not exploit the fact that the same padding can also *help eliminate another hazard.*

Section 5.4.3 shows how a *global optimum* solution to the delay padding problem of case 2 can be solved as a set of linear programs.

Of course, if we use more freedom during the delay optimization phase, as described in case 3, we can potentially obtain a better result at the expense of an increased complexity in the linear program solution, as outlined in Section 5.4.4.

As hinted in Section 2.8 we can observe some similarity between this approach and the classical definition of *essential hazards*:

1. An essential hazard manifests itself, in a circuit operated in *Fundamental Mode*, as the fact that a state variable change is *fast* propagating. It "overruns" the input change that caused it, and causes a hazard in some gate that was expecting the effect of the input change *before* the effect of the state variable change.

   Such hazards required (see, e.g., [117]) to *increase the delay* of the state signals, so that input changes finish their propagation before the changes due to the state transition are initiated.

2. The hazards detected by our methodology manifest themselves in a circuit operated according to an STG specification (that is we *do not require Fundamental Mode operation*) if two signal transitions that are causally related "overrun" each other.

   Such hazards require to increase the delay of the second signal, so that changes due to the first transition finish their propagation before the changes due to the second transition are initiated.

So, in some sense, we take care of *Fundamental Mode* and of *essential hazards* at the same time, with a formal automated procedure.

We can also find a parallel between our proposed methodology and what is classically done in *synchronous circuit* synthesis:

- in the *synchronous* case we slow down the *clock* signal until *no more events* are propagating along the whole circuit.
- in the *asynchronous* case we slow down *each* signal until *no more events that caused its change* are propagating along the whole circuit.

So this approach, even if it does not generate *locally* speed-independent or delay-insensitive circuits, can still be considered faithfully adherent to the "asynchronous philosophy", in that every element must obey a "locally defined" protocol, and elements that are logically far apart must not be slowed down due to each other. Moreover, the "global view" of the circuit, as described in Chapter 3, is still speed-independent, or delay-insensitive, thanks to the STG specification intrinsic properties.

5.4.3 Delay Padding Outside the Circuit

In this section we show how, given a set of potential hazards output by Procedure 5.3.1, we can optimally pad delays after each STG signal to make sure that the final circuit is hazard-free.

Computation of the Delay Bounds

The first step to analyze when a hazard may occur in the circuit is to obtain an *upper bound* on $d_{ah} - d_{bh}$ for each potential hazard. We can perform a timing analysis of the final implementation of the circuit along the corresponding circuit paths (as reported by Procedure 5.3.1), and measure d_{ah} and d_{bh}. Of course, to obtain an upper bound on $d_{ah} - d_{bh}$, we must use:

- the *upper* bound on the wire delays when traversing wires *only on the path corresponding to* d_{ah}.
- the *lower* bound on the wire delays when traversing wires *only on the path corresponding to* d_{bh}.
- any value (it will cancel out when computing the difference) when traversing wires that are *on both paths*.

See, for example, the circuit shown in Figure 5.9, synthesized from the STG of Figure 2.23 using a library of n-input *nand* gates[5]. The pair of numbers near each gate represent its minimum and maximum delays. The wires are assumed, for the sake of simplicity, to have no delay.

The hazard analysis algorithm reports, for example, the following potential hazard: (L_1^+, D^+, Ai). In this case, d_{ah} is measured along gates 11, 4 and 5, while d_{bh} is measured along 4 and 5. So we must use the *upper bound* on the delay of gate 11, and the delays of gates 4 and 5 (that are on both paths) cancel out. Hence the upper bound on $d_{ah} - d_{bh}$ is 14. Note that the two paths that must be used to measure d_{ah} and d_{bh} are also output by Procedure 5.3.1. The reader can check that, while the STG required Ai to stay at 0 between L_1^+ and D^+, if D^+ propagates to Ai before L_1^+, then Ai has a 0-hazard (see also Section 2.7.3).

The *lower bound* on each delay d_{ab} between a pair of transitions z_a^* and z_b^* can be computed, in the general case, using the STG as follows. We want to identify the earliest time at which z_b^* can fire in any execution of the STG, knowing that:

- z_a^* fires at time 0, and
- a lower bound $\delta(i, j)$ on the difference between the firing time of each transition z_i^* in the STG and each of its successors z_j^* is obtained in the following way:
  - if z_j is an output signal, then it can be computed from the circuit, using timing analysis on the shortest circuit path between signal z_i and output signal z_j. Note that if z_i^* is a predecessor of z_j^*, then z_i is an input of the circuit implementing z_j.
  - otherwise, it can be either given to the synthesis system by the designer (who has some information on the behavior of the circuit environment) or assumed to be zero.

In the example of Figure 2.23, we can use timing analysis to label each edge in the STG with a *minimum* delay between the predecessor and successor transition, as shown by the edge

[5]The delays are *different* from the similar implementation shown in Figure 2.24, that was taken from [22].

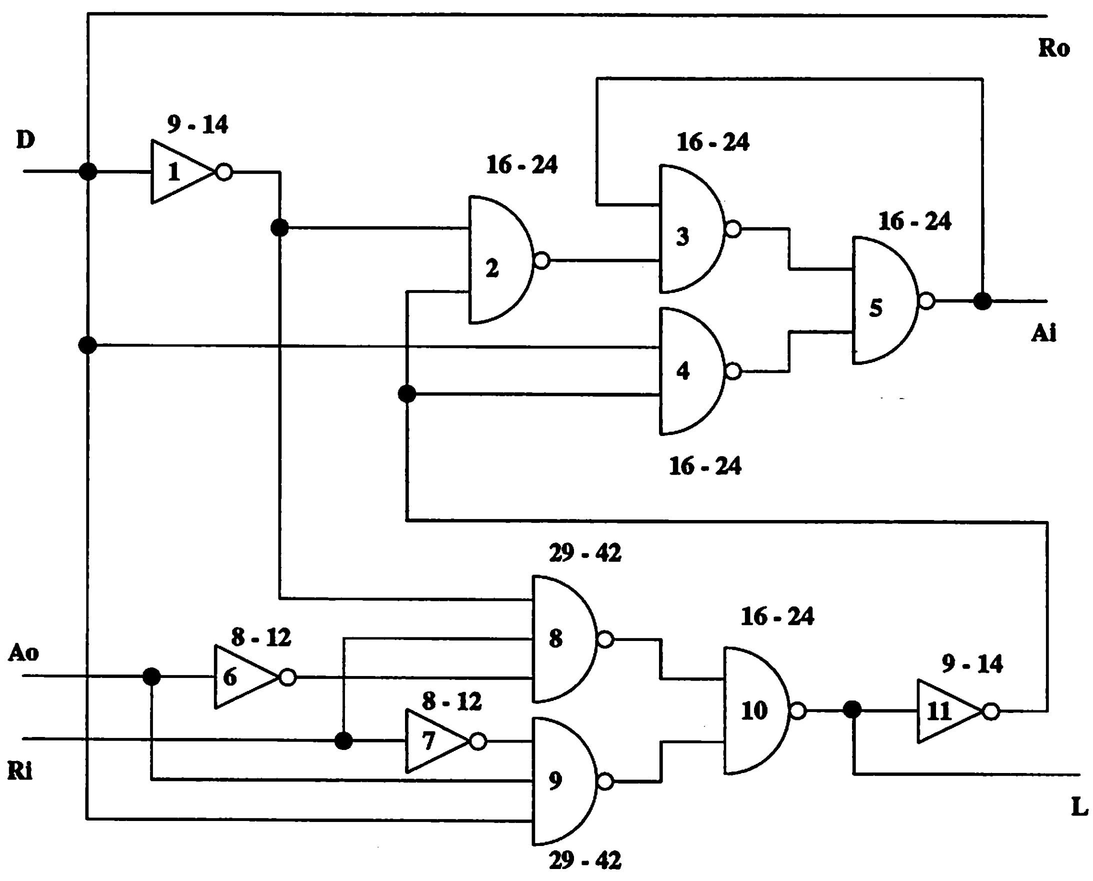


Figure 5.9: Implementation of the Signal Transition Graph of Figure 2.23

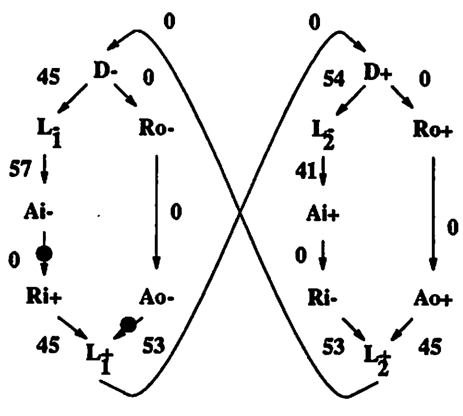


Figure 5.10: The Signal Transition Graph of Figure 2.23 with delay bounds

labels. The result, using the delays of Figure 5.9, is shown in Figure 5.10. In this case, the input signal delays were conservatively assumed to be zero. For example the minimum delay between D^- and L_1^- is 45, because we must traverse gates 9 and 10, with a minimum delay of 29 and 16 respectively, to cause a falling transition on L with a falling transition on D.

If we did not pad delays, then we could compute a lower bound on d_{ab} by examining each MG component to which both z_a^* and z_b^* belong (there exists at least one such component, because z_a^* and z_b^* are *ordered* in the STG). Each edge (z_i^*, z_j^*) in the MG component is given a weight equal to the lower bound between the firing time of z_i^* and z_j^*. So, for example, if we want to estimate the minimum delay between D^- and L_1^+, we can look at the longest path on the STG, among $D^- \rightarrow L_1^- \rightarrow Ai^- \rightarrow Ri^+ \rightarrow L_1^+$ and $D^- \rightarrow Ro^- \rightarrow Ao^- \rightarrow L_1^+$.

This simple solution, though, does not work as expected, because the target of the hazard elimination procedure is exactly to *increase* some of those lower bounds between transition firing times. Fortunately we can formulate the problems of d_{ab} estimation *and* delay padding *simultaneously*, to obtain a delay padding sufficient to eliminate all hazards from the circuit.

For each triple $H = (z_a^*, z_b^*, z_h)$ potentially producing a hazard, we identify the set of MG components to which both z_a^* and z_b^* belong. For each MG component M we identify the set of transitions $T_{(H,M)}$ that *must* fire between z_a^* and z_b^*, i.e. the set of transitions that lie on a *simple path* of M between z_a^* and z_b^*. This set of transitions can be identified by a depth first traversal of M beginning from z_a^*. In our example, there is only one MG component (the STG itself), and the set of transitions between D^- and L_1^+ (included) is $\{D^-, L_1^-, Ai^-, Ri^+, L_1^+, Ro^-, Ao^-\}$.

For each MG component M for each hazard $H = (z_a^*, z_b^*, z_h)$, we then associate a variable $D_{(j,H,M)}$ with each transition z_j^* between z_a^* and z_b^*. This variable represents the *earliest* possible firing time of z_j^*, given that:

- we are checking hazard H and
- the STG is executing the MG component M and
- each predecessor z_i^* of z_j^* has fired:
  - either at time $-\infty$ if z_i^* does not belong to $T_{(H,M)}$ (this provides a worst case, because it allows for the earliest possible firing, but the bound could be improved using techniques such as those described in [123] or [82]),
  - or at time $D_{(i,H,M)}$ if z_i^* belongs to $T_{(H,M)}$.

Let $\Delta_{(i,j)}$ denote the minimum delay between the firing of z_i^* and the firing of z_j^*. Let $P_{(j,H,M)}$ denote the set of transitions $z_i^* \in T_{(H,M)}$ such that z_i^* is a direct predecessor of z_j^*. Then each $D_{(j,H,M)}$ must satisfy the following constraint, because a transition in an MG component can fire only if *all its predecessors have fired*:

$$D_{(j,H,M)} \geq \max_{z_i^* \in P_{(j,H,M)}} D_{(i,H,M)} + \Delta_{(i,j)} \tag{5.1}$$

So d_{ab} can be computed as $D_{(b,H,M)}$, since we defined the firing time of z_a^* (i.e. $D_{(a,H,M)}$), to be zero. Furthermore we need each $D_{(j,H,M)}$ to be equal to its lower bound to obtain the lower bound on d_{ab}, i.e. equation 5.1 becomes:

$$D_{(j,H,M)} = \max_{z_i \in P_{(j,H,M)}} D_{(i,H,M)} + \Delta_{(i,j)} \tag{5.2}$$

Delay Padding Problem Formulation

We can now introduce a variable S_i for each signal z_i in the STG, representing the amount of delay padded after it. Then we can express $\Delta_{(i,j)}$ as:

$$\Delta_{(i,j)} = \delta_{(i,j)} + S_j$$

(recall that $\delta(i,j)$ is a known lower bound on the difference between the firing time of transition z_i^* and of its successor z_j^*). We have completed the formulation of the hazard elimination problem, because for each hazard H we have:

- a set of constraints of the form given by equation 5.2, yielding a lower bound on each d_{ab} in function of the padded delays S_i.

- an inequality that must hold in order for the hazard not to manifest itself in the implemented circuit:

$$D_{(b,H,M)} > (d_{ah} - d_{bh})$$

So we can state the delay padding problem as:

Problem 5.4.3 Given:

- *a Signal Transition Graph specification $\mathcal{G}$,*
- *lower bounds on delays between each input transition of $\mathcal{G}$ and its predecessors,*
- *an asynchronous circuit implementation C, modeled with* pure bounded wire delays, *of the output signals of $\mathcal{G}$,*
- *a weight associated with padding each signal of $\mathcal{G}$,*
- *the list of potential hazards detected by Procedure 5.3.1, yielding the following set of constraints:*
  - *for each hazard $H = (z_a^*, z_b^*, z_h)$:*
    - *for each* MG *component M of the* STG *to which both z_a^* and z_b^* belong:*
      - $D_{(a,H,M)} = 0$
      - $D_{(b,H,M)} > (d_{ah} - d_{bh})$
      - *for each transition z_j in the set $T_{(H,M)}$ of transitions that must fire between z_a^* and z_b^*:*
        $D_{(j,H,M)} = \max_{z_i \in P_{(j,H,M)}} D_{(i,H,M)} + \delta_{(i,j)} + S_j$
        (i.e. with i ranging among all predecessors of z_j that are in $T_{(H,M)}$)

Where $\delta_{(i,j)}$, d_{ah} and d_{bh} are constants, while $D_{(j,H,M)}$ and S_j are variables, constrained to be non-negative.

Find: *the minimum weighted assignment of values to the S_j's that satisfies all the constraints.*

As above, S_j represents the amount by which *every* transition of signal z_j is slowed down, i.e. the amount of delay padded after z_j.

If the circuit specification includes constraints on the *minimum* and *maximum* separation in time between pairs of transitions (z_i^*, z_j^*), they can:

- be *used* here, if z_j is an *input* signal, determining $\delta_{(i,j)}$, or
- be *enforced* here, if z_j is an *output* signal. In this case, they are just added to the list of potential hazards, where now $D_{(b,H,M)} > (d_{ah} - d_{bh})$ is replaced by an appropriate inequality. Note that, as explained above, *minimum* delay constraints can always be satisfied by delay padding, while *maximum* delay constraints may not.

Constraints that are automatically satisfied by assigning a value of zero to all the S_j's can be discarded: the corresponding potential hazards cannot show up in the synthesized circuit, with the given delay bounds.

Note that Problem 5.4.3 always has a feasible (albeit possibly not optimal) solution, that can be obtained using Procedure 5.4.1.

In the example of Figure 5.9 we have the following set of constraints for hazard $H = (Ai^-, D^+, Ai)$, with $T_{(H,0)} = \{Ai^-, Ri^+, L_1^+, D^+\}$ for the only MG component of Figure 5.10 (we omitted the subscripts $H, 0$ for the sake of simplicity):

$$\begin{aligned}
D_{Ai^-} &= 0 \\
D_{Ri^+} &= D_{D^-} + 0 + S_{Ri} \\
D_{L_1^+} &= D_{Ri^+} + 45 + S_L \\
D_{D^+} &= D_{L_1^+} + 0 + S_D \\
D_{D^+} &> 8
\end{aligned}$$

The right hand side in the last equation is obtained by measuring an upper bound on d_{ah} along gates 3 and 5 and a lower bound on d_{bh} along gates 4 and 5 (remembering that the same value must be used for the delay of gate 5 in both cases). We can easily see that this set of constraints is automatically satisfied by setting all the S_j's to zero.

We must now choose the weight of each STG signal, i.e. the objective function to be minimized. These weights can reflect either a user-defined cost of slowing down "critical" signals, or a measure of the area cost of the delay padding.

If we choose to assign the same weight to all variables, then we minimize an estimate of the area impact of delay padding. This is similar to the delay *re-padding* done in [129], where, for example, two equal delays padded before a two-input gate are moved after the gate, halving the area cost of the buffers.

If we want to have a better estimate of the global impact of the delay padding on the *system performance*, then we can analyze the length of the *cycles in the* STG *specification*, since they are an estimate of the total system throughput [81]. In this case we minimize $C_{\max}$, where $C_{\max}$ is the maximum among the lengths C_c of the simple cycles[6] ξ_c of the STG:

$$C_c = \sum_{z_i^*, z_j^* \in \xi_c} \delta_{(i,j)} + S_j$$

In general an STG can have an exponential number of cycles but, as shown in [24], we can limit our analysis to a smaller set of cycles, and still reliably estimate the throughput.

If the target technology is semi-custom, such as standard cells or gate array, then the delay padding must be done in a *discrete* fashion, adding enough buffers to satisfy the constraints. In this case any delay larger than the value obtained solving Problem 5.4.3 does eliminate all the hazards, but a closer approximation of reality is obtained by constraining the S_j variables to be *discrete*. This will entail the repeated solution of an *Integer Linear Programming* problem ([92]), because we can introduce one variable for each available type of delay buffer, for each STG signal, and a linear function of the number of such buffers can then be derived to represent the delay padded after the signal.

The next Section describes how the mixed linear and *max* constraints of Problem 5.4.3 can be optimally solved. Note that we cannot claim an optimum solution to Problem 5.4.1 as well. As shown by the construction for the proof of Theorem 5.4.1, satisfying *at least* one constraint for each potential hazard is sufficient, in some case, to make other constraints *redundant*, and thus automatically satisfied. The problem of detecting when a constraint becomes redundant (that involves also the *circuit functionality*, and as such cannot be handled by analyzing the set of constraints alone), is a subject of further investigation.

A Branch-and-bound Delay Padding Procedure

The optimum solution to Problem 5.4.3 is attained when *at least one* equality holds in each *max* type constraint. So we can find it by repeatedly solving a *Linear Program* where we choose for each max constraint *one* element of the right hand side to be *equal* to the left hand side, and the others to be less than or equal.

That is, for each

$$D_{(j,H,M)} = \max_{z_i \in P_{(j,H,M)}} D_{(i,H,M)} + \delta_{(i,j)} + S_j$$

[6]A simple cycle of a directed graph is a cycle without repeated nodes.

we choose some $z_k \in P_{(j,H,M)}$ and we decompose the constraint into the *conjunction* of:

- $D_{(j,H,M)} = D_{(k,H,M)} + \delta_{(k,j)} + S_j$ and
- $D_{(j,H,M)} \geq D_{(i,H,M)} + \delta_{(i,j)} + S_j$ for each $i \neq k$ such that $z_i \in P_{(j,H,M)}$

We can use a branch-and-bound procedure to prune the search and hopefully avoid to explore all the possible combinations of choices. The procedure is based on the observation that if we "*relax*" some *max* constraint, leaving *all* the right hand side elements *less than or equal* to the left hand side, then the solution with one or more elements *equal* is contained in the solution space of the "relaxed" problem. So the minimum cost of the "relaxed" problem is *less than or equal* to the minimum cost of the original problem. If the "relaxed" problem is infeasible, so is the original problem.

The branch-and-bound algorithm proceeds as follows, given any ordering of the *max* type constraints. Let $\mathcal{M}$ be the set of n *max* constraints, each of the form:

$$D_{(j,H,M)} = \max_{z_i \in P_{(j,H,M)}} D_{(i,H,M)} + \delta_{(i,j)} + S_j$$

Let $\hat{c}$ (initialized to ∞) be the best solution seen so far for which all the *max* constraints are satisfied. Let k (initialized to 1) be the index of the currently considered *max* constraint in $\mathcal{M}$, denoted by $\mathcal{M}_k$.

Procedure 5.4.2

1. *let c be the optimum cost of a Linear Program where*
   - *each* $\mathcal{M}_l$, *for* $l = 1, \ldots, k-1$ *has been "resolved" by previous recursive calls, and*
   - *each* $\mathcal{M}_l$, *for* $l = k, \ldots, n$ *is "relaxed" as:*
     $D_{(j,H,M)} \geq D_{(i,H,M)} + \delta_{(i,j)} + S_j$ *for all* $i \in P(j, H, M)$
2. *if the Linear Program is infeasible or* $c \geq \hat{c}$ *then return*
   (no better solutions in this sub-tree)
3. *if its solution satisfies all the* max *constraints then* $\hat{c} = \min\{\hat{c}, c\}$ *and return*
   (no better solutions in this sub-tree)
4. *for each* $z_l \in P_{(j,H,M)}$*:*

(a) set

- $D_{(j,H,M)} = D_{(l,H,M)} + \delta_{(l,j)} + S_j$ *and*
- $D_{(j,H,M)} \geq D_{(i,H,M)} + \delta_{(i,j)} + S_j$ *for each* $i \neq l$ *such that* $z_i \in P_{(j,H,M)}$

("resolve" $\mathcal{M}_k$ *with respect to* z_l*)*

(b) recur incrementing k by 1

The branch and bound procedure, if each *max* constraint has m elements, can be seen to be an m-ary tree, with a Linear Program (LP) associated with each node. The depth of a node is the number of nodes along any path from the root to it. The root has a depth of zero. At depth d there are m^d nodes. The LP at the root has only inequality constraints. Every edge downwards from depth $k-1$ to depth k, replaces an inequality constraint from the relaxed k^{th} *max* constraint by an equality constraint in the LP associated with the upper node, to give the LP associated with the lower node. The tree has depth n and m^n leaves. At each leaf, every *max* constraint has an equality constraint associated with it. So any solution to the original problem with *max* constraints is a solution to the LP at one of the leaves. Let p be a root-leaf path and let P_d denote the feasible region of the LP at a node with depth d on p and let c_d denote the minimum objective function value of the LP associated with that node. We note the following along the path p:

1. $P_n \subseteq P_{n-1} \subseteq \ldots P_0$
2. consequently, $c_n \geq c_{n-1} \geq \ldots c_0$

This implies that, if at any node P_d is feasible and $c_d \geq \hat{c}$, the sub-tree rooted at the node cannot contain a better solution. In addition if the LP is infeasible at a node, then the LPs at all the nodes in the sub-tree rooted at that node are infeasible, and hence can be pruned. Note that P_0 is feasible because the original problem with *max* constraints has a solution (Procedure 5.4.1).

This procedure evaluates $O(m^{n+1})$ Linear Programs in the worst case, so it can be worse than the straightforward approach evaluating the cost only at the bottom of the recursion ($O(m^n)$ times). But an adequate selection of the order of resolution of the constraints *and* of the order of z_l at each execution of step 4 at each recursion level should provide a better average case performance.

In the example in Figure 5.9, the only potential hazard that produces a set of constraints that are not trivially satisfied is $H = (L_1^+, D^+, Ai)$, with $T_{(H,0)} = \{L_1^+, D^+\}$:

$$\begin{aligned} D_{L_1^+} &= 0 \\ D_{D^+} &= D_{L_1^+} + 0 + S_D \end{aligned}$$

$$D_{D+} > 14$$

Where the right hand side in the last equation is obtained by measuring an upper bound on d_{ah} along gates 11, 4 and 5 and a lower bound on d_{bh} along gates 4 and 5 (remembering that the same value must be used for the delay of gates 4 and 5 in both cases). We can easily see that this set of constraints is satisfied by padding at least 14 delay units after signal D.

The branch-and-bound procedure requires an exponential number of calls to the LP solver in the worst case. We can take advantage of *monotonicity* properties of the problem, similar to those used for the bounding above, to derive a heuristic procedure. This procedure, described in the next section, requires only a polynomial number of LP solutions to find a set of delay paddings that yield a hazard-free circuit.

A Heuristic Delay Padding Procedure

The key observation leading to a heuristic solution to the delay padding problem is that the set of constraints of Problem 5.4.3 defines a *directed acyclic graph*. So a topological argument allows us to prove that there exists a criterion to resolve each *max* constraints into an equality and a set of inequalities such that the resulting Linear Program is feasible, and its optimum solution yields a valid (albeit possibly suboptimal) solution to the hazard elimination problem.

The set of constraints of the form:

$$D_{(j,H,M)} = \max_{z_i \in P_{(j,H,M)}} D_{(i,H,M)} + \delta_{(i,j)} + S_j$$

of Problem 5.4.3 can be associated with a directed acyclic graph, called *constraint graph*, with a node for each variable $D_{(k,H,M)}$, and a directed edge from the elements of each right hand side to its left hand side. Each edge is labeled with the corresponding S_j variable.

For example, given the STG fragment in Figure 5.11.(a), suppose that the hazard detection procedure found two potential hazards, (v^+, x^+, u) and (w^-, y^-, u). We would then produce the following set of constraints:

$$\begin{aligned}
D_{v^+} &= 0 \\
D_{w^-} &= D_{v^+} + \delta_{(v,w)} + S_w \\
D_{z^+} &= D_{v^+} + \delta_{(v,z)} + S_z \\
D_{y^-} &= \max\{(D_{w^-} + \delta_{(w,y)} + S_y), (D_{z^+} + \delta_{(z,y)} + S_y)\}
\end{aligned}$$

$$\begin{tikzcd}
 & w^- \arrow[r] & y^- \arrow[ddr] & \\
v^+ \arrow[ur] \arrow[dr] & & & \\
 & z^+ \arrow[uur] \arrow[rr] & & x^+
\end{tikzcd}$$

(a)

$$\begin{tikzcd}
 & D_{w^-} \arrow[r, "S_y"] & D_{y^-} \arrow[ddr, "S_x"] & \\
D_{v^+} \arrow[ur, "S_w"] \arrow[dr, "S_z"] & & & \\
 & D_{z^+} \arrow[uur, "S_y"] \arrow[rr, "S_x"] & & D_{x^+} \\
 & D'_{w^-} \arrow[r, "S_y"] & D'_{y^-} &
\end{tikzcd}$$

(b)

Figure 5.11: A Signal Transition Graph fragment and its associated constraint graph

$$\begin{aligned}
D_{x^+} &= \max\{(D_{z^+} + \delta_{(z,x)} + S_x), (D_{y^-} + \delta_{(y,x)} + S_x)\} \\
D_{x^+} &> d_{vu} - d_{xu} \\
D'_{w^-} &= 0 \\
D'_{y^-} &= D'_{w^-} + \delta_{(w,y)} + S_y \\
D'_{y^-} &> d_{wu} - d_{yu}
\end{aligned}$$

The corresponding constraint graph is shown in Figure 5.11.(b). Each set of constraints derived from each hazard and each MG component produces a connected component of the directed graph[7]. Each connected component can be viewed as a labeled acyclic subgraph of the MG component.

In Figure 5.11.(b) there are two connected components, one with vertices D_{v^+}, D_{w^-}, D_{z^+}, D_{y^-}, D_{x^+} and one with D'_{w^-}, D'_{y^-}. Note that the same edge label may appear in different connected components, because for example both D_{y^-} and D'_{y^-} are associated with transitions of the *same signal.*

We will now show that for each node we can choose one successor to become an *equality* constraint, and produce a feasible solution to the original problem.

Let $\mathcal{M}$ denote the set of constraints defining the solution of an instance of Problem 5.4.3. Let $\mathcal{I}$ denote a set of *linear constraints* obtained by selecting one element from each *max* constraint in $\mathcal{M}$ to become an equality and relaxing the others. Let $\mathcal{G}$ denote the constraint graph corresponding to $\mathcal{M}$.

[7]A connected component of a directed graph is a set of vertices such that there exists a path in the underlying undirected graph between every pair of vertices in the component.

The set of edges in the constraint graph selected by $\mathcal{I}$ to become equalities, denoted by bold edges in Figure 5.11.(b), is a *spanning tree* of each component[8]. The set of edges in $\mathcal{I}$ clearly defines a tree, because exactly one immediate predecessor is selected for each node, so each node has (by induction) a unique path to the root. Moreover, the tree is rooted in the node whose associated variable is fixed at 0 ($D_{v}+$ and $D'_{w}-$ in the example).

Suppose that we have chosen one element of each *max*, and obtained a set of linear constraints. Given an assignment of values to the S_i variables, the value of each D_j variable is *uniquely* determined by the equalities, because each D_j is now equal to one of its predecessors on the constraint graph plus some constant ($\delta_{jk} + S_j$).

The other inequalities relative to each D_j may or may not be satisfied. But if we *increase* by some ϵ the value of some S_i we can show that all satisfied inequalities remain satisfied.

Lemma 5.4.2 *Let S be an assignment of values to the S_i variables, and let D be the corresponding value of the D_j variables, obtained using the equalities in $\mathcal{I}$. Let S' be an assignment of values derived from S by increasing by ϵ the value of a single S_i, and let D' be the corresponding value of the D_j variables.*

If an inequality in $\mathcal{I}$ is satisfied by D and S, then it is satisfied by D' and S'.

Proof If we increase by ϵ the value of some S_i, then we increase by exactly ϵ the value of its corresponding node D_i and of all the successors to D_i on the spanning tree. For example, increasing S_z would change the values of $D_{z}+$, $D_{y}-$ and $D_{x}+$.

This increase could violate only constraints of the form:

$$D_k \geq D_l + \delta_{lk} + S_k$$

where D_l has been increased by ϵ, while D_k is unchanged. That is D_k is a *predecessor* of D_i, while D_l is a *successor* of D_i (or D_i itself).

The presence of that constraint implies the presence of an edge (D_l, D_k), and hence a cycle in the graph, because D_k is also a *predecessor* of D_l. We have reached a contradiction.

Hence this increase in value can never violate any of the *inequality constraints* that were previously satisfied. ■

On the other hand, we can show that if we always choose as equality at each node D_j a constraint from a node D_k such that the only path from D_k to D_j is though this single edge, then we can always find a feasible solution by assigning large enough values to the S_i's.

[8]A spanning tree of a connected component of a graph is a subset of edges of the graph defining a tree where every node of the component appears.

Theorem 5.4.3 *Let $\mathcal{M}$ be a set of constraints as above. Then:*

1. *There exists a set of linear constraints $\mathcal{I}$ derived from $\mathcal{M}$ such that for all edges (D_k, D_j) corresponding to equality constraints there is no directed path from D_k to D_j besides (D_k, D_j).*
2. *There exists a pair of assignments D, S that satisfies all constraints in $\mathcal{I}$ (i.e. $\mathcal{I}$ is* feasible*).*

Proof

1. Let $\mathcal{G}_j$ be the set of immediate predecessors of node D_j on the constraint graph $\mathcal{G}$ corresponding to $\mathcal{M}$.

   Suppose, for the sake of contradiction, that there exists D_j such that for all $D_k \in \mathcal{G}_j$ there exists a path from D_k to D_j besides (D_k, D_j). Call the last node on this path (before D_j) D_l. Note that $D_l \neq D_k$ by assumption, but $D_l \in \mathcal{G}_j$. Then for each $D_k \in \mathcal{G}_j$ there exists $D_l \in \mathcal{G}_j$ such that there exists a path between D_k and D_l. This is clearly a contradiction, because $\mathcal{G}$ is acyclic.

   So we can always select $\mathcal{I}$ satisfying this requirement.

2. Let S be an arbitrary assignment of values to the S_i variables, and let D be the corresponding assignment to the D_j's (obtained with the equalities in $\mathcal{I}$).

   Let (D_f, D_j) be the edge corresponding to an inequality that is *not* satisfied. Then we claim that there exists some S_k that can be increased enough to satisfy this inequality.

   Let (D_k, D_j) be the edge corresponding to the equality constraint determining the value of D_j.

   We know that, by construction of $\mathcal{I}$, there is no path between D_k and D_j besides (D_k, D_j). So there exists no path between D_k and D_f (otherwise we could augment it with (D_f, D_j) and obtain a path between D_k and D_j). So increasing the value of the S_k variable associated with D_k does not modify the value of D_f, while it obviously increases the value of D_j.

   We can then increase S_k enough to satisfy (D_f, D_j). By Lemma 5.4.2 this does not violate any previously satisfied inequality, so by induction we can satisfy all the inequalities in $\mathcal{I}$.

■

Note that the construction is used only to determine the *feasibility* of the problem, while the actual determination of the *optimum* values of the S_i variables, ***given the choice of equality constraints*** is left to the Linear Program solver.

In Figure 5.11 the bold edges satisfy Theorem 5.4.3, while choosing (D_{z^+}, D_{x^+}) would not have been valid. We can also see how the latter choice would have caused infeasibility, because the set of constraints:

$$\begin{array}{rcl} & \dots & \\ D_{y^-} & = & D_{z^+} + \delta_{(z,y)} + S_y \\ D_{x^+} & = & D_{z^+} + \delta_{(z,x)} + S_x \\ D_{x^+} & \geq & D_{y^-} + \delta_{(y,x)} + S_x \\ & \dots & \end{array}$$

with $\delta_{(z,y)} = \delta_{(z,x)} = 0$ and $\delta_{(y,x)} = 1$ clearly is not feasible, because it implies

$$D_{z^+} + S_x \geq D_{z^+} + S_y + S_x + 1$$

that is a contradiction, since all variables are non-negative.

A set of *candidate edges* satisfying Theorem 5.4.3 can be derived from the constraint graph in polynomial time. The set of candidates for each node D_j is obtained by a reversed depth-first search from D_j, i.e. traversing edges in the opposite direction to the arrow. In this search we remove from the list of candidates every edge (D_k, D_j) such that D_k is reached more than once.

We must now choose one edge, among those candidates, to be made an equality constraint. A reasonable heuristic at this step is to select the candidate with a longest path (determined using the $\delta_{(i,j)}$ weights) from the root. This would be the "critical path" determining the firing time of the transition corresponding to D_j in absence of delay paddings, hence this is the equality that is "most likely" to hold after the optimum padding as well. This heuristic can also be used in the branch-and-bound Procedure 5.4.2 to *order* the exploration of the solution tree in a way that hopefully maximizes the possibility of early bounding of suboptimal solutions.

The heuristic procedure then requires a polynomial pre-processing step to ensure feasibility, and a *single* LP solution, so it can be used where the large number of *max* constraints would make the exact branch-and-bound procedure impractical.

Moreover note that this heuristic procedure can also be used as a subroutine in the branch-and-bound procedure to provide an *upper bound* to the cost of the best solution under a node in the search tree. So at every node in the search tree in Procedure 5.4.2 we solve two LPs:

- one with all the unresolved constraints *relaxed*, that provides a *lower bound* on the cost of feasible solutions below that node, and
- one with all the unresolved constraints *resolved according to the heuristic procedure*, that provides an *upper bound* on the cost of feasible solutions below that node.

The search is pruned if the lower bound is larger than the current best feasible solution or than the current best upper bound (as found by some node above the current one).

The next section describes how we can improve the result of both the branch-and-bound and the heuristic procedures by allowing more flexibility in the delay padding, especially for circuit parts where maximum speed is required.

5.4.4 Delay Padding Inside the Circuit

If we are willing to pad delays *also inside* the combinational logic blocks that are being synthesized, we can in general improve the results of the procedure described in Section 5.4.3, at the price of a more complex problem to be solved.

In this case, in addition to the variables associated with externally padded delays as described in the Section 5.4.3, we can create one variable for each *connection* inside the combinational logic implementing each output signal. Namely (see Figure 5.12):

- each wire w_{ij} from node i to node j inside each combinational sub-circuit is labeled with:
  - an upper and lower bound U_{ij} and L_{ij} on the delay between a transition on the output of i and a transition on the output of j, and
  - a variable S_{ij}, describing how much delay is padded (after the fanout).
- wires that belong to both paths associated with d_{ah} and d_{bh}, π_{ah} and π_{bh} respectively, can be ignored because they cancel out.
- d_{ah} can be statically computed as an upper bound on the delays of wires that belong only to its associated path π_{ah}, plus the variables associated with them:

$$d_{ah} = \sum_{w_{ij} \in \pi_{ah} \wedge w_{ij} \notin \pi_{bh}} U_{ij} + S_{ij}$$

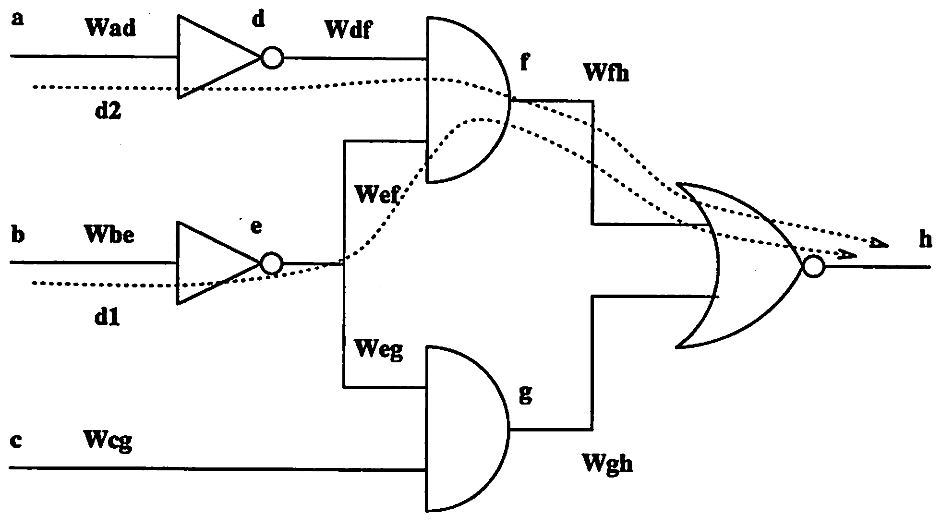


Figure 5.12: Wire labels for delay padding inside the circuit

- d_{bh} can be statically computed as a lower bound on the delays of wires that belong only to its associated path π_{bh}, plus the variables associated with them:

$$d_{bh} = \sum_{w_{ij} \in \pi_{bh} \wedge w_{ij} \notin \pi_{ah}} L_{ij} + S_{ij}$$

The computed expressions for d_{ah} and d_{bh} replace the corresponding constants in the linear program described in Section 5.4.3. In the example in Figure 5.12 we would have:

$$\begin{aligned} d_{ah} &= S_{ad} + U_{ad} + S_{df} + U_{df} \\ d_{bh} &= S_{be} + L_{be} + S_{ef} + L_{ef} \end{aligned}$$

Similarly, each Δ_{ij} is no longer $\delta_{(i,j)} + S_j$, but is the minimum, among the circuit paths between signals z_i and z_j, of the corresponding sums of L_{kl} and S_{kl}'s.

The problem *always has a feasible solution*, because setting the variables associated with internal wires to zero reduces it to the case of Section 5.4.3 (hence the solution can be found using Procedure 5.4.1).

The number of variables may increase considerably with respect to the case when only STG signals are considered, so this approach should ideally be used only for highly critical sections of the circuit, where maximum operating speed must be achieved.

We shall briefly observe here that another refinement of the hazard elimination procedure could take into account the possibility of padding delay *after the fanouts* also with respect to external signals. Again this would increase the number of variables in the linear program, but it could improve the throughput of the final solution.

At this point we may ask ourselves whether we can carry the refinement one step further, and *fix some padded delay to zero*. If we can eliminate all hazards without padding some delay, then we can handle cases where *the environment is not under control*, and we cannot arbitrarily pad delays in it. For example two input signals may be provided by an external module, and they may satisfy a causal relationship, but their transitions may be "too close together". It would be nice if we could eliminate all hazards by only slowing them down *outside* the module that produces them.

Unfortunately the answer to this question is **no**, as we will show in the next section.

5.4.5 Environment Accessibility Conditions

In this section we show that *all the fanouts in the environment* must be available for delay padding, in general, to be able to eliminate all hazards. The proof is simply an example of a valid STG specification where if some wire is not accessible, then some hazard cannot be eliminated by only padding external signal delays.

There is also a more intuitive explanation for this impossibility. Suppose that two input transitions, say x^+ and y^-, occur too close together (i.e. y^- should be slowed down in order to eliminate a potential hazard). Suppose also that, due to some other potential hazard, another pair of transitions *of the same signals in the opposite order*, say y^+ and x^- occur too close together (i.e. x^- should be slowed down in order to eliminate this potential hazard). It should be clear that no amount of delay padding for signal y that does not also influence the *firing time* of the transitions of x can eliminate the first hazard without making the second hazard worse. In other terms, in some cases we must be able to really slow down the *environment behavior*, and not only the local view of the environment signals that reaches the circuit that we are synthesizing.

Note that this result does not contradict the results described in Section 2.5.2, stating that hazard-free circuits can be obtained by *logic transformations* only. The *gate delay* model, used in Section 2.5.2, assumes that two transitions that occur arbitrarily close can always be discriminated without hazards. Our *wire delay* model, on the other hand, is more conservative, and can require a more expensive solution to ensure hazard-freeness. The choice of which delay model is best suited to describe the behavior of the *actual circuit*, depends essentially on system-level technology and reliability considerations, as outlined in Section 3.6.

The STG that formalizes the above intuitive reasoning is a cycle with the following

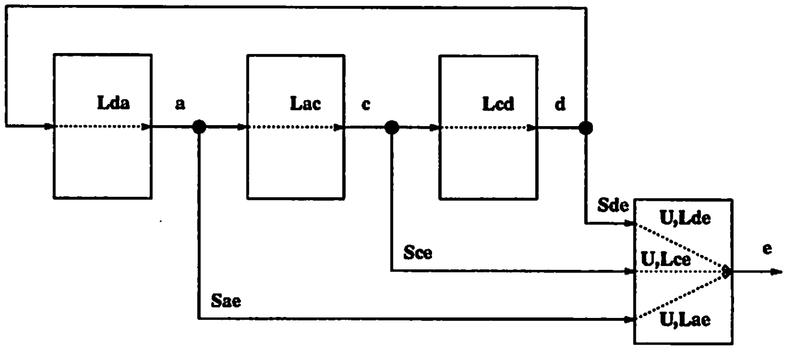


Figure 5.13: An example where delay padding must be performed on the environment

transitions:

$$a_1^+ \rightarrow c^+ \rightarrow d_1^+ \rightarrow f_1^+ \rightarrow e_1^+ \rightarrow d_1^- \rightarrow a_1^- \rightarrow b^+ \rightarrow e_1^- \rightarrow b^- \rightarrow f_1^- \rightarrow e_2^+ \rightarrow d_2^+ \rightarrow f_2^+ \rightarrow e_2^- \rightarrow f_2^- \rightarrow c^- \rightarrow a_2^+ \rightarrow e_3^+ \rightarrow d_2^- \rightarrow a_2^- \rightarrow e_3^- \rightarrow a_1^+$$

The resulting circuit structure is described in Figure 5.13, where we only synthesize the circuit for e. The next-state function for e has the following (unique) two-level prime and irredundant representation:

$$a\bar{c}d + ae + af + \bar{a}c\bar{d}\bar{f} + \bar{b}\bar{d}ef + ce\bar{f}$$

The hazard detection procedure finds the following three potential hazards (plus other hazards that are not relevant here): (a^+, c^+, e), (c^+, d^+, e), and (d^-, a^-, e).

Let variable S_{ij} denote the delay padding between signal z_i and signal z_j, let U_{ij} be a (constant) upper bound on the delay between signal z_i and the output of the circuit for signal z_j, and similarly let L_{ij} be a corresponding (constant) lower bound. Let us also suppose that *the outputs of a, c and d cannot be slowed down before the fanouts*, so $S_{cd} = S_{da} = S_{ac} = 0$. If the only thing that we can do is delay padding on STG signals, then we end up with the following inequalities that must be satisfied to eliminate the above hazards:

$$\begin{aligned} L_{ac} + S_{ce} &> S_{ae} + U_{ae} - L_{ce} \\ L_{cd} + S_{de} &> S_{ce} + U_{ce} - L_{de} \\ L_{da} + S_{ae} &> S_{de} + U_{de} - L_{ae} \end{aligned}$$

Let us suppose that we have, for example:

$$U_{ae} - L_{ce} = U_{ce} - L_{de} = U_{de} - L_{ae} = 1, \quad L_{ac} = L_{cd} = L_{da} = 0$$

Then the resulting set of constraints

$$\begin{aligned} S_{ce} &> S_{ae} + 1 \\ S_{ae} &> S_{de} + 1 \\ S_{de} &> S_{ce} + 1 \end{aligned}$$

is cyclic, so it does not have a finite solution.

Theorem 5.4.4 *Let $\{\mathcal{G}\}$ be the set of live* STG*s with* CSC*. Let $\{\mathcal{C}\}$ be the set of logic circuits synthesized from elements of $\{\mathcal{G}\}$ using Procedure 5.2.1 and optimized using only commutativity, associativity and inverter insertion.*

Let $\{\mathcal{C}'\}$ be the set of hazard-free circuits, modeled with bounded wire delays, obtained from $\{\mathcal{C}\}$ by padding delays at STG *signals.*

Then every STG *in $\{\mathcal{G}\}$ has at least one circuit implementation in $\{\mathcal{C}'\}$ if and only if* every fanout of STG signals can be padded by an arbitrary amount.

Proof

- the *if* part is implied by Theorem 5.3.1, the proof of correctness of Procedure 5.3.1 and Procedure 5.4.1.
- the *only if* part is implied by the counter-example above.

■

Note that the theorem does not imply that *every* STG requires complete environment accessibility in order to be synthesized without hazards. For example, it can be shown that this is not the case when the STG describes a *delay-insensitive* behavior.

This section completes the hazard-free circuit implementation procedure, giving a hazard elimination algorithm that can be applied to the circuit synthesized from an STG with Procedure 5.2.1, and then optimized with constrained logic synthesis. Note that all the results given so far concern *static* hazards only. The next section shows that, with some caution, we can extend the validity of those results to *dynamic* hazards as well.

5.5 Dynamic Hazard Analysis

Dynamic hazard analysis is much more difficult than static hazard analysis, and we do not have yet powerful formal techniques to reason about this class of hazards. Even the nine-valued simulation procedure described in [68] (see also Section 2.7.1) could not be applied here. Static hazards have a very precise "boundary" condition: the next state function value must remain constant. This allowed us to define the concept of valid state pair and prove Theorem 5.3.1. A dynamic hazard on a signal z_h, on the other hand, can occur an arbitrary amount of time after a transition of z_h, so we cannot easily restrict the set of STG transitions that we must analyze.

So this section will be less precise and more based on intuition than its counterparts on static hazards. On the other hand, dynamic hazards can be considered less "critical" than static hazards, as the *inertial* nature of circuit delays (as opposed to the *pure* model used in this chapter) tends to eliminate them, rather than propagating them to cause deviations from the specified behavior.

In a two-level implementation of the on-set of a logic function F, *assuming that we have performed the static hazard elimination procedure* described in Section 5.4, we can analyze the circuit operation as follows.

The STG specification describes a *constrained way* to walk along the cubes implementing the function, and the hazard elimination procedure guarantees that:

- If we are walking along on-set vertices and we are about to "leave" a cube and the function value must not change in the next vertex, then Procedure 5.2.1 guarantees that there is *another on-set cube* covering both the current vertex and the next one. The hazard elimination procedure delays, if necessary, the transition among those two vertices to make sure that the logic implementing this second cube keeps the output high before the first cube can cease to keep it high.

  So we "orderly walk" on the cubes making sure that *whenever we "enter" a cube* it has time to "turn on".

- If we are walking along off-set vertices and we might "enter" a cube, then the hazard elimination procedure delays (if necessary) the transition to ensure that we remain "outside" the on-set.

So our static hazard elimination algorithm actually guarantees that *we follow only paths that are legal according to the specification*, even though we are using the bounded wire delay model, and so the delays in the implemented circuits might have led us away from legal STD paths.

But in this case *we cannot have dynamic hazards* because:

- While we are walking on the on-set (and then we could have $1 \rightarrow 0 \rightarrow 1 \rightarrow 0$ hazards when entering the off-set), we are sure that every on-set cube that we can enter has time to be turned on before we proceed. Then we will not output a spurious "1-pulse" due to some slow cube whose effect is felt on the output only after we have left the on-set.

- While we are walking on the off-set (and then we could have $0 \rightarrow 1 \rightarrow 0 \rightarrow 1$ hazards when entering the on-set), we are sure that no on-set cube can ever be "turned on". Then we will not output a "1-pulse".

The only problem can be caused by cubes that happen to cover a vertex due to *some other*, independent firing sequence of the STG. This is due to the fact that the Complete State Coding property ensures only that different markings with the same binary label (i.e. corresponding to the same vertex) have the same set of enabled output transitions. So we can reach the same vertex more than once in different markings, and have a different set of *input* transitions enabled, thus producing different cubes in Step 2(c)i of Procedure 5.2.1. The hazard elimination procedure would then ignore this second cube, and may leave some dynamic hazard in the final implementation.

This problem can be solved by enforcing the more restrictive *Unique State Coding* property on the STG and avoiding to expand the initial cover of cubes if they can intersect each other, as in the approach described by [91]. Note that Unique State Coding can be enforced using a straightforward extension of the algorithms given in Chapter 4, where the state minimization step is modified to take into account that now all pairs of states entered under the same input are incompatible (not only those that have incompatible outputs).

Constrained logic synthesis, based only on distributivity and commutativity, does not change substantially the "cube-based" picture, so the intuitive view can be used also for a multi-level implementation of the same function.

5.6 Experimental Results

The algorithms described in Section 5.4 have been implemented within the sequential synthesis system SIS ([108, 107]). We applied them to a set of STGs taken both from the literature and from a real industrial application, a multi-processor interconnection system ([111]).

We will show the influence of the following factors on the final result:

1. the use of some information on the delay between output transitions and input transitions due to the non-zero response time of the environment (while all the other results in this section are obtained by conservatively estimating the environment delays to be *zero*).
2. the type of gate library available for the implementation.
3. the difference between the results of the greedy Procedure 5.4.1 and of the optimum Procedure 5.4.2.

We will also give a comparison with a straightforward *synchronous* implementation of (roughly) the same functionality. This implementation was obtained interpreting the STD as a Finite State Machine (as outlined in Chapter 4), minimizing it and then performing standard state assignment and synchronous logic synthesis. So the it would implement exactly the same behavior if the flip-flops had zero setup and hold time and infinite clock frequency.

In all the tables, the columns labeled "Area" give the total area (excluding routing) of each circuit, using a "generic" standard cell library. The columns labeled "Delay" give the maximum combinational logic delay between any STG signal and any STG output signal. The columns labeled "Pad" give the *total* amount of delay padding required by Procedure 5.4.1 and by Procedure 5.4.2 respectively. As a reference point, in this library the *inverter* gate area is 16 units and its delay is 1 unit plus 0.2 units for each driven gate. Note that the delay column is not meant to give an absolute measure of operating speed, but only an idea of the trade-offs involved. Moreover, the padded delay is in general distributed among various output signals, so it does not precisely reflect the decrease in throughput due to delay padding.

Table 5.1 shows how some knowledge about the delay of the environment can greatly improve the synthesis result. The column labeled "Zero-delay" shows the result of Procedure 5.4.2 if we assume that the environment instantaneously responds to an output transition with a new set of input transitions. The column labeled "2 *inverter* delays" shows the result if we suppose to know that the delay in the environment is at least equal to the delay of *two inverter* gates in our standard cell library.

Table 5.2 shows the influence of the available library on the synthesis results. We used two different libraries, one (Library 1) with a large set of combinational functions and one (Library 2) with only a few gates with delays widely unbalanced both among gates and among different inputs to the same gate, to simulate somehow the influence of very long routing lines. Example *pe-send-ifc* had too many linear constraints (about 3000) to be solved with the implementation of the simplex algorithm available in SIS in a reasonable amount of time.

example			Zero-delay		2 *inverter* delays	
	With hazards		Without hazards		Without hazards	
	Area	Delay	Area	Pad	Area	Pad
chu133	224	4.4	224	0.0	224	0.0
chu150	200	4.6	200	0.0	200	0.0
chu172	104	1.6	104	0.0	104	0.0
converta	368	5.0	368	0.0	368	0.0
ebergen	216	3.2	216	0.0	216	0.0
full	192	4.6	208	1.2	192	0.0
hazard	200	4.2	200	0.0	200	0.0
hybridf	242	5.0	258	1.2	242	0.0
nowick	232	4.6	232	0.0	232	0.0
alloc-outbound	272	4.0	272	0.0	272	0.0
mp-forward-pkt	232	3.4	232	0.0	232	0.0
nak-pa	256	3.4	256	0.0	256	0.0
pe-rcv-ifc	752	6.6	784	2.4	752	0.0
pe-send-ifc	912	7.4	1056	10.8	976	4.8
ram-read-sbuf	384	4.2	416	2.4	384	0.0
rcv-setup	128	2.8	128	0.0	128	0.0
sbuf-ram-write	296	4.0	296	0.0	296	0.0
sbuf-read-ctl	272	4.2	272	0.0	272	0.0
sbuf-send-ctl	280	3.2	280	0.0	280	0.0
sbuf-send-pkt2	320	4.4	416	7.2	352	2.4
sendr-done	96	3.0	112	1.2	96	0.0
qr42	216	3.2	216	0.0	216	0.0
rpdft	168	5.2	232	4.8	168	0.0
trimos-send	576	4.6	576	0.0	576	0.0
vbe10b	688	7.0	784	7.2	752	4.8
vbe5b	208	4.0	208	0.0	208	0.0
vbe5c	160	3.0	176	1.2	160	0.0
wrdatab	664	4.8	680	1.2	664	0.0
total	8858	119.6	9402	40.8	9018	12.0

Table 5.1: Effect of knowledge of environment delay on hazard elimination

example	Library 1				Library 2			
	With hazards		Without hazards		With hazards		Without hazards	
	Area	Delay	Area	Pad	Area	Delay	Area	Pad
chu133	224	4.4	224	0.0	296	7.8	296	0.0
chu150	200	4.6	200	0.0	280	9.0	296	1.2
chu172	104	1.6	104	0.0	128	6.6	144	1.2
converta	368	5.0	368	0.0	504	10.4	520	1.2
ebergen	216	3.2	216	0.0	320	14.4	320	0.0
full	192	4.6	208	1.2	224	9.0	240	1.2
hazard	200	4.2	200	0.0	216	8.2	248	2.4
hybridf	242	5.0	258	1.2	328	7.8	328	0.0
nowick	232	4.6	232	0.0	320	19.4	400	6.0
alloc-outbound	272	4.0	272	0.0	296	9.2	296	0.0
mp-forward-pkt	232	3.4	232	0.0	304	13.2	304	0.0
nak-pa	256	3.4	256	0.0	328	9.0	328	0.0
pe-rcv-ifc	752	6.6	784	2.4	1008	15.8	1136	9.6
ram-read-sbuf	384	4.2	416	2.4	472	14.2	648	13.2
rcv-setup	128	2.8	128	0.0	176	7.8	272	7.2
sbuf-ram-write	296	4.0	296	0.0	400	15.2	400	0.0
sbuf-read-ctl	272	4.2	272	0.0	312	8.0	328	1.2
sbuf-send-ctl	280	3.2	280	0.0	368	19.2	560	14.4
sbuf-send-pkt2	320	4.4	416	7.2	384	9.4	432	3.6
sendr-done	96	3.0	112	1.2	104	7.8	104	0.0
qr42	216	3.2	216	0.0	320	14.4	320	0.0
rpdft	168	5.2	232	4.8	240	15.2	288	3.6
trimos-send	576	4.6	576	0.0	600	8.0	600	0.0
vbe10b	688	7.0	784	7.2	856	11.8	1048	14.4
vbe5b	208	4.0	208	0.0	256	15.2	304	3.6
vbe5c	160	3.0	176	1.2	216	9.0	216	0.0
wrdatab	664	4.8	680	1.2	800	15.2	896	7.2
total	7946	112.2	8346	30.0	10056	310.2	11272	91.2

Table 5.2: Effect of different libraries on hazard elimination

Table 5.3 compares an asynchronous and a synchronous implementation. This table is not meant to be a "fair" comparison between the synchronous and asynchronous design styles, because the specifications were designed with an asynchronous implementation in mind. It is meant to show that an asynchronous implementation does not automatically imply a loss in area and/or performance.

Table 5.4 shows a comparison of the optimum hazard elimination Procedure 5.4.2 with the greedy hazard elimination Procedure 5.4.1, assuming that the minimum environment delay is 0. The columns labeled "Pad" give the amount of delay padding required by Procedure 5.4.1 and by Procedure 5.4.2 respectively. The columns labeled "CPU" give the CPU time (on a DECstation 5000/125) for STG synthesis, hazard analysis and linear program solution in both cases. The column labeled "ineq" gives the total number of inequalities (excluding those that were trivially satisfied). The column labeled "var" shows the total number of variables in each linear program.

example	Asynchronous				Synchronous	
	With hazards		Without hazards		Without hazards	
	Area	Delay	Area	Pad	Area	Delay
chu133	224	4.4	224	0.0	216	4.4
chu150	200	4.6	200	0.0	160	3.2
chu172	104	1.6	104	0.0	128	2.8
converta	368	5.0	368	0.0	304	4.4
ebergen	216	3.2	216	0.0	160	3.0
full	192	4.6	208	1.2	176	4.0
hazard	200	4.2	200	0.0	112	2.8
hybridf	242	5.0	258	1.2	256	4.8
nowick	232	4.6	232	0.0	256	8.6
alloc-outbound	272	4.0	272	0.0	272	4.0
mp-forward-pkt	232	3.4	232	0.0	240	4.8
nak-pa	256	3.4	256	0.0	248	3.4
pe-rcv-ifc	752	6.6	784	2.4	656	6.8
pe-send-ifc	912	7.4	1056	10.8	824	6.2
ram-read-sbuf	384	4.2	416	2.4	352	4.2
rcv-setup	128	2.8	128	0.0	120	4.2
sbuf-ram-write	296	4.0	296	0.0	304	4.0
sbuf-read-ctl	272	4.2	272	0.0	256	3.2
sbuf-send-ctl	280	3.2	280	0.0	272	3.2
sbuf-send-pkt2	320	4.4	416	7.2	312	4.4
sendr-done	96	3.0	112	1.2	80	2.8
qr42	216	3.2	216	0.0	160	3.0
rpdft	168	5.2	232	4.8	168	5.2
trimos-send	576	4.6	576	0.0	480	4.6
vbe10b	688	7.0	784	7.2	608	7.0
vbe5b	208	4.0	208	0.0	200	4.4
vbe5c	160	3.0	176	1.2	176	3.0
wrdatab	664	4.8	680	1.2	592	7.0
total	8858	119.6	9402	40.8	8088	123.4

Table 5.3: Asynchronous versus synchronous implementation

name	greedy				optimum					
	Area	Del	Pad	CPU	Area	Del	Pad	CPU	Ineq	Var
chu133	272	4.4	3.6	8.9	224	4.4	0.0	9.5	0	7
chu150	200	4.6	0.0	8.1	200	4.6	0.0	8.4	0	6
chu172	104	1.6	0.0	7.3	104	1.6	0.0	7.1	0	6
converta	368	5.0	0.0	10.9	368	5.0	0.0	14.9	0	5
ebergen	264	3.2	3.6	9.1	216	3.2	0.0	9.3	0	5
full	208	4.6	1.2	7.6	208	4.6	1.2	7.8	6	8
hazard	216	4.2	1.2	8.4	200	4.2	0.0	8.7	0	4
hybridf	274	5.0	2.4	10.8	258	5.0	1.2	11.0	9	14
nowick	264	4.6	2.4	8.1	232	4.6	0.0	8.2	0	6
alloc-outbound	288	4.0	1.2	8.9	272	4.0	0.0	9.0	0	9
mp-forward-pkt	280	3.4	3.6	9.0	232	3.4	0.0	9.1	0	8
nak-pa	256	3.4	0.0	9.7	256	3.4	0.0	10.2	0	10
pe-rcv-ifc	880	6.6	9.6	42.7	784	6.6	2.4	53.6	86	80
pe-send-ifc	1232	7.4	24.0	132.3	1056	7.4	10.8	210.9	381	264
ram-read-sbuf	448	4.2	4.8	13.0	416	4.2	2.4	12.7	3	13
rcv-setup	128	2.8	0.0	7.4	128	2.8	0.0	7.6	9	11
sbuf-ram-write	376	4.0	6.0	12.9	296	4.0	0.0	13.0	0	12
sbuf-read-ctl	288	4.2	1.2	10.7	272	4.2	0.0	9.6	0	8
sbuf-send-ctl	328	3.2	3.6	10.9	280	3.2	0.0	11.3	0	8
sbuf-send-pkt2	400	4.4	6.0	11.2	416	4.4	7.2	11.0	18	21
sendr-done	144	3.0	3.6	6.8	112	3.0	1.2	6.7	3	6
qr42	264	3.2	3.6	8.9	216	3.2	0.0	9.2	0	5
rpdft	248	5.2	6.0	8.0	232	5.2	4.8	7.7	19	18
trimos-send	720	4.6	10.8	36.0	576	4.6	0.0	39.2	0	9
vbe10b	944	7.0	19.2	42.9	784	7.0	7.2	35.3	42	39
vbe5b	256	4.0	3.6	9.1	208	4.0	0.0	9.0	0	6
vbe5c	176	3.0	1.2	8.0	176	3.0	1.2	7.9	3	8
wrdatab	776	4.8	8.4	26.9	680	4.8	1.2	31.1	15	20
total	21204	119.6	130.8	494.5	18804	119.6	40.8	589.0	594	616

Table 5.4: Greedy versus optimum hazard elimination

Chapter 6

The Design for Testability Methodology

The synthesis algorithm described in Chapter 5 produces a circuit implementation that is guaranteed to be hazard-free if and only if a set of inequalities among path delays inside the circuit is satisfied. So using a suitable delay model during the synthesis process it is possible to guarantee hazard-freeness in the absence of delay faults. Now our goal is to test those path delays, and be sure that the above mentioned inequalities are satisfied in *each manufactured circuit.*

Moreover, as asynchronous interface circuits typically have absolute delay requirements, it is highly desirable to know that the synthesized circuit is able to operate with the required timing constraints. Also, because asynchronous circuits are often used as interfaces in systems where very high reliability is required, a stringent manufacture test of the interface circuitry is desirable. Unfortunately current testing procedures do not even reliably provide comprehensive stuck-at-fault testing of asynchronous circuits.

Our goal is to synthesize hazard-free asynchronous circuits that are testable in the hazard-free robust path delay fault model. Producing circuits satisfying two very stringent requirements, namely, hazard-free operation and Hazard-Free Robust Path Delay Fault Testability (HFRPDFT), poses an especially exciting challenge. In this chapter we present techniques which *guarantee* both hazard-free operation and HFRPDFT, at the expense of possibly adding test inputs, and give a set of heuristics which can improve HFRPDFT testability without requiring such inputs. We also present a procedure that guarantees testability in the less stringent Robust Gate Delay Fault Testability model (RGDFT).

The testing scenario that we envision uses **full scan testing** techniques to make the test generation process manageable. In this scenario every flip-flop (*SR*, *C*-element, ...) can be scanned, to increase both the observability of its inputs and the controllability of its output. This is also necessary because some of the flip-flop types, for example *SR* flip-flops, are sensitive only to input transitions in one direction, rising or falling. So we *cannot* test the delay of the transitions in the other direction without resorting to scan techniques.

For the purposes of this chapter, we will assume that

1. each flip-flop is implemented so that it has a testing mode, in which an appropriate value can be scanned in and out [41].

2. each designer-specified signal that is not implemented with a flip-flop is either a primary input or a primary output of the integrated circuit or is otherwise made accessible (for example by insertion of a scan D-latch normally held in transparent mode).

If these assumptions are satisfied, then every cycle in the circuit is broken by at least one scan memory element. So we shall consider flip-flop inputs as primary outputs, and flip-flop outputs as primary inputs of the combinational logic circuit that we are testing.

This chapter is organized as follows: Section 6.1 introduces some necessary terminology. Section 6.2 gives a procedure that is guaranteed to transform the initial two-level circuit produced according to Section 5.2.1 in an HFRPDFT circuit. Section 6.3 gives a variety of heuristics that are likely to increase the HFRPDFT testability of a circuit. Section 6.4 gives another synthesis for testability algorithm for producing an RGDFT circuit. Section 6.5 summarizes how the procedures given in the previous sections can be used together to yield a testable circuit with minimal overhead. Section 6.6 finally shows experimental results on applying these techniques.

6.1 Definitions and Notation

A multi-level circuit is obtained by **algebraic factorization** from a cover of a function f if it is obtained from the cover by applying only the distributive and associative properties of Boolean sum and product (i.e. ignoring that $a \cdot \overline{a} = 0$, $a + \overline{a} = 1$ and so on).

A **combinational circuit path** is a sequence of nodes, $\pi = \{z_0, \ldots z_n\}$, where z_0 is a primary input, z_n is a primary output, all other nodes are gates and each z_i is an input of z_{i+1}.

A **two-vector test** (v_1, v_2) is a pair of vectors of input values for the combinational circuit under test. The test proceeds as follows:

1. v_1 is applied for a time sufficient for the circuit to stabilize.

2. v_2 is applied.

3. the combinational logic output is sampled.

. This delay-testing model implies that the scan flip-flops must have the following capabilities:

- select which of the flip-flop inputs (e.g. S or R for an SR flip-flop) is latched by the test clock and subsequently scanned out.
- memorize two values to be applied in sequence for the application of a two-vector test.

An **event** is a transition $0 \rightarrow 1$ or $1 \rightarrow 0$ at a node. Consider a sequence of transitions, $\{z_0^*, z_1^*, ..., z_n^*\}$ occurring at nodes $\{z_0, z_1, ..., z_n\}$ along a path, such that z_i^* occurs as a result of event z_{i-1}^*. The event z_0^* is said to propagate along the path. If there exists an input vector pair such that under *appropriate* delays in the circuit, an event could propagate along a path, then the path is said to be **event sensitizable**. If there exists an input vector pair such that under *arbitrary* delays in the circuit, an event propagates along a path, then the path is said to be **robustly event sensitizable.**

A circuit has a **gate delay fault** if there is one gate in the circuit such that the output of the circuit is slow to make a $0 \rightarrow 1$ (or $1 \rightarrow 0$) transition when one or more of the gate inputs change values. Each single gate delay fault is assumed to be so catastrophic as to cause a delay along *any path* through the gate to any output, i.e. all the inputs of the gate are assumed to be affected by the fault. This assumption is often considered to be *unrealistic*, thus requiring to use a more stringent delay fault model based on *paths* rather than *gates*.

A circuit has a **path delay fault** if there exists a path from a primary input to a primary output via a set of gates and nets such that a primary input event is slow to propagate *along the path* to the primary output.

A two-vector test (v_1, v_2) is said to be a **robust path delay fault test** for a path π, if and only if, when π is faulty and the test (v_1, v_2) is applied, the circuit output is different from the expected output at sampling time, independent of the delays along wires not on π ([74]). Similar definitions hold for the **robust gate delay fault test**. A more stringent model is the hazard-free robust delay fault model[1], treated in [98, 33]. A robust path delay fault test is said to be a **hazard-free robust path delay fault test** if no hazards can occur on the tested path during the application of the test, regardless of the delay values. Note that an HFRPDFT test is valid for *both transitions* along a path π, the second test being obtained by just reversing the order of application of the vectors.

[1]The hazard free robust path delay fault model is simply called the robust path delay fault model in [34, 33].

It may be useful, at this point, to contrast the requirements of hazard-free operation and HFRPDFT:

- For an asynchronous circuit to be **hazard-free in operation** it is necessary that for*for all legal* input sequences, i.e. corresponding to valid firing sequences of the STG specification, no static hazard occurs in the circuit. Thus hazard-free operation is a global property governing the normal operation of the circuit.
- For an asynchronous circuit to be **hazard-free** robustly path delay fault **testable** (HFRPDFT) it is necessary that *there exists* a vector pair that detects each path delay fault in a robust and hazard-free manner. Such a vector pair (v_1, v_2) might not be a *legal* input sequence, i.e. there might not exist a valid firing sequence of the STG specification bringing the inputs of the circuit from v_1 to v_2. This means also that it might be impossible to apply v_1 and v_2 without using scan techniques, because the circuitry surrounding the path under test (also called the *environment*), which was designed to implement the STG specification, might not be able to produce those vectors in that sequence.

Despite the apparent similarity between the two properties, neither property implies the other. Moreover, optimizing a circuit for one property can diminish or eliminate the other. See for example the case, described in Section 6.2, where making a circuit HFRPDFT introduces hazards during normal operation. Conversely, in the very same example, a cube that appears in the initial two-level cover in order to eliminate a hazard during operation makes the circuit not HFRPDFT.

6.2 A Procedure Guaranteed to Generate an HFRPDFT Circuit

This section describes a technique to implement the two-level initial on-set cover obtained by Procedure 5.2.1 as a fully hazard-free robustly path delay fault testable circuit that has exactly the same hazard properties as the initial two-level cover, so that the algorithms described in Section 5.4 can be used to make it hazard-free in operation and fully HFRPDFT.

Kundu *et al.* ([67]) first gave a procedure, based on *Shannon decomposition*, to make a combinational circuit HFRPDFT. The essence of this procedure is to choose a binate variable, call it x, in a given two-level representation, call it F, of a logic function f, decompose F into $x \cdot F_x + \overline{x} \cdot F_{\overline{x}}$, such that the variable x does not appear in either F_x and $F_{\overline{x}}$. Unfortunately, while this procedure results in an HFRPDFT implementation it may not result in an implementation that is hazard-free during normal operation, as is required.

Take for example the STG reported in Figure 2.23. The on-set and off-set covers for signal A_i, as obtained by Procedure 5.2.1, are respectively $F_{A_i} = D\overline{L} + D\overline{R_i} + L\overline{R_i}$ and $R_{A_i} = \overline{D}\,\overline{L} + \overline{D}R_i + LR_i$. Both covers are *redundant* (cube $D\overline{R_i}$ can be removed from F_{A_i} and cube $\overline{D}R_i$ can be removed from R_{A_i}). If we choose any variable for the above decomposition, we introduce a hazard whenever that variable changes and A_i must remain constant. Note that in this case there is also no hope to obtain an HFRPDFT implementation with any of the heuristics described in Section 6.3, which cannot remove a redundancy in both the on-set and off-set covers.

We now present a procedure which is guaranteed to produce an HFRPDFT circuit. This procedure may require the addition of test inputs. At present we know of no procedure that is guaranteed to produce HFRPDFT circuits, that are also hazard-free in operation, that does not also add test inputs.

6.2.1 Algebraic Decomposition

Our procedure is similar to the one outlined above in some respects: first a variable x at the beginning of an untestable path is identified in a given two-level representation F of a logic function f. F is then algebraically decomposed into $x \cdot G + \overline{x} \cdot H + R$, so that the variable x does not appear in any one of G, H and R. The difference between this procedure and that of [67] is the ability to partition out a remainder R. This results in a more area efficient implementation but more importantly the factoring out of the remainder ensures *hazard-free operation* as we will show in Section 6.2.2 below. We now give the procedure in detail. It takes as input a *prime*, but possibly *redundant*, two-level representation F of a logic function f, and it returns a multi-level implementation of f with exactly the same hazard properties of F:

Procedure 6.2.1

1. *If the combinational circuit F is* HFRPDFT *then return F.*
2. *Otherwise:*
   (a) *Choose (with an appropriate heuristic) an input variable x such that a path beginning from x is untestable in F.*
   (b) *Let G be the two-level cover obtained by collecting all the cubes in F that depend on x, cofactored against x.*

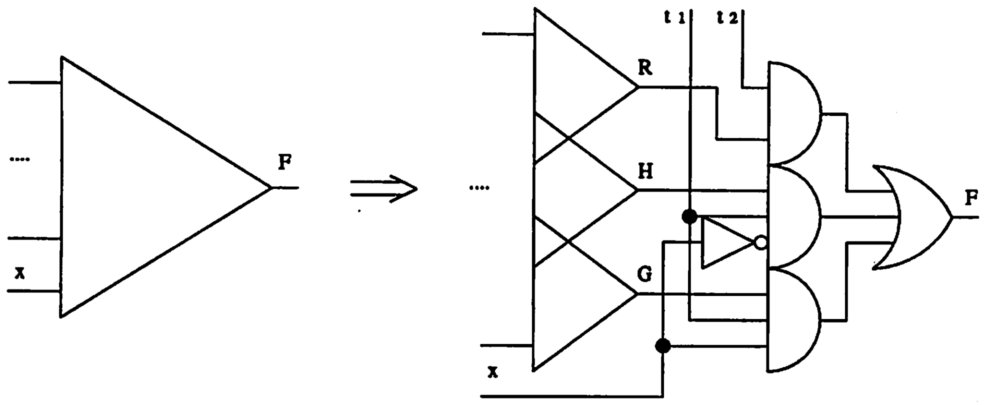


Figure 6.1: Application of Procedure 6.2.1

(c) *Let H be the two-level cover obtained by collecting all the cubes in F that depend on $\overline{x}$, cofactored against $\overline{x}$.*

(d) *Let R be the two-level cover obtained by collecting all the cubes in F that do not depend on either x or $\overline{x}$.*

(e) *Let G', H' and R' be the circuits obtained by a recursive application to G, H and R respectively.*

(f) *Let t_1 and t_2 be two new variables, not in the support of f.*

(g) *Return the circuit*

$$t_1 \cdot x \cdot G' + t_1 \cdot \overline{x} \cdot H' + t_2 \cdot R'$$

Figure 6.1 shows the result of one step of Procedure 6.2.1.

We now proceed to prove that this procedure results in an HFRPDFT circuit. The key result is the following:

Theorem 6.2.1 *Let f be a logic function, let F be a combinational circuit implementing f and let $G \cdot x + H \cdot \overline{x} + R$ be an algebraic factorization of F such that:*

1. *G, H and R are each individually* HFRPDFT *circuits,*

2. *the on-set of the logic function implemented by G contains a vertex not contained in the on-set of the logic function implemented by H,*

3. *the on-set of the logic function implemented by H contains a vertex not contained in the on-set of the logic function implemented by G, and*

4. *t_1 and t_2 are two variables not in the support of f.*

Then $F' = t_1 \cdot G \cdot x + t_1 \cdot H \cdot \overline{x} + t_2 \cdot R$ is an HFRPDFT *circuit.*

Proof We assume that G, H and R are non-empty. If they are, then the analysis is further simplified.

Each of G, H, and R is assumed to be HFRPDFT. The proof strategy used to detect delay faults in these sub-circuits in F' is to augment the vectors which test those paths in isolation with the appropriate values of t_1, t_2, and x.

Consider a circuit path π in G. By supposition G in isolation is HFRPDFT so we have a vector pair (v_1, v_2) that tests π. To test π in F' set $t_1 = 1, t_2 = 0, x = 1$ and apply (v_1, v_2).

Consider a path π in H. By supposition H in isolation is HFRPDFT so we have a vector pair (v_1, v_2) that tests π. To test π in F' set $t_1 = 1, t_2 = 0, x = 0$ and apply (v_1, v_2).

Consider a path π in R. By supposition R in isolation is HFRPDFT so we have a vector pair (v_1, v_2) that tests π. To test π in F' set $t_1 = 0, t_2 = 1, x = 0$ and apply (v_1, v_2).

The paths associated with x and $\overline{x}$ are equally straightforward. Consider the path π associated with x. By supposition the on-set of G contains a vertex not contained in H. Call that vertex v. To test π set $t_1 = 1, t_2 = 0$, give the values of vertex v to the other primary inputs and let x rise $0 \rightarrow 1$.

Consider the path π associated with $\overline{x}$. By supposition the on-set of H contains a vertex not contained in G. Call that vertex v. To test π set $t_1 = 1, t_2 = 0$, give the values of vertex v to the other primary inputs and let x fall $1 \rightarrow 0$.

The testing inputs do not need to run at speed but it may be simpler to test them than to treat them as a special case. The vectors are created as follows.

Consider the path π associated with the testing input t_1 in $x \cdot t_1 \cdot G$. By supposition the on-set of G contains a vertex not contained in H. Call that vertex v. To test π set $t_2 = 0, x = 1$, give the values of vertex v to the other primary inputs and let t_1 rise $0 \rightarrow 1$.

Consider the path π associated with the testing input t_1 in $\overline{x} \cdot t_1 \cdot H$. By supposition the on-set of H contains a vertex not contained in G. Call that vertex v. To test π set $t_2 = 0, x = 0$, give the values of vertex v to the other primary inputs and let t_1 rise $0 \rightarrow 1$.

Consider the path π associated with the testing input t_2. To test π first find any vertex v such that $R = 1$. Then set $t_1 = 0$, give the values of vertex v to the other primary inputs, and let t_2 rise $0 \rightarrow 1$.

Thus every path in F' is HFRPDFT. ■

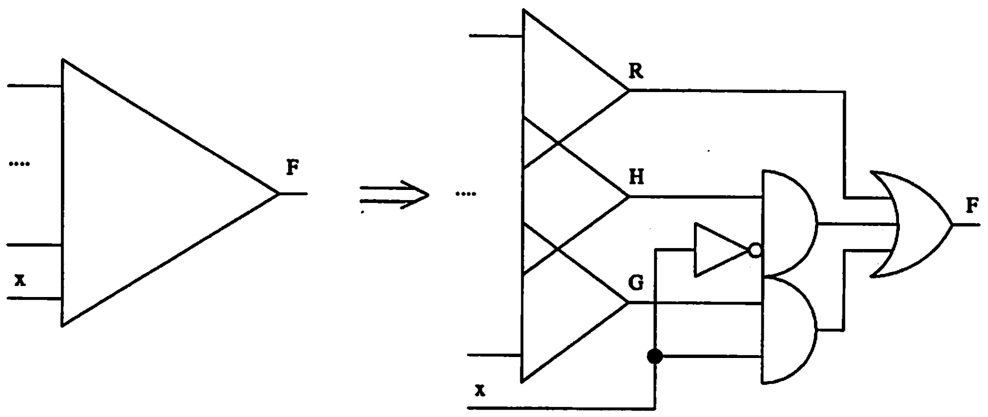


Figure 6.2: Application of Procedure 6.2.1 without test inputs

Note also that a similar argument allows us to conclude that if both $G + R$ and $H + R$ are HFRPDFT circuits, then we do not have to introduce the testing inputs t_1 and t_2, resulting in the circuit structure described in Figure 6.2.

We prove that we retain primality in the recursion step of Procedure 6.2.1 in the following Lemma.

Lemma 6.2.2 *Let F be a prime on-set cover of a logic function f. Let x be a variable in the support of f. Let G be the set of cubes containing x cofactored against x, H be the set of cubes containing $\overline{x}$ cofactored against $\overline{x}$ and R be the set of cubes containing neither x or $\overline{x}$.*

Then each of G, H and R is a prime cover.

Proof Suppose some cube c in G (with corresponding cube $x \cdot c$ in F) is not prime. Then we could remove a literal, say y, from it, and the resulting cube, call it c_y, would still be an implicant of $B(G)$ (i.e. of the logic function denoted by G). So also $c_y \cdot \overline{y}$ would be an implicant of $B(G)$, and $x \cdot c_y \cdot \overline{y}$ would be an implicant of f. Then we could expand $x \cdot c$ to $x \cdot c_y$, contradicting the hypothesis of primality of F.

A similar argument can be used for H. As R is unchanged from F, R remains prime. ■

To prove that Procedure 6.2.1 is correct, we use induction and apply Theorem 6.2.1 at the induction step.

Theorem 6.2.3 *Let F be a prime on-set cover of a logic function f. Then Procedure 6.2.1 results in an* HFRPDFT *combinational circuit F' of the form $t_1 \cdot x \cdot G' + t_1 \cdot \overline{x} \cdot H' + t_2 \cdot R'$ implementing f.*

Proof

- Basis: let f be a logic function of two variables. By inspection we can show that for any function of two variables there exists an HFRPDFT implementation using Procedure 6.2.1.
- Induction step: Suppose that any function of n variables can be made HFRPDFT by Procedure 6.2.1. Let f be a function of $n+1$ variables. Suppose F is not HFRPDFT. Let x be a variable such that a path beginning at x is not testable in F.

  By Lemma 6.2.2, each of G, H and R (supposed not empty, as in the proof of Theorem 6.2.1) is a prime cover of a function of n variables, and therefore, by the induction hypothesis, each has an HFRPDFT implementation that can be arrived at through Procedure 6.2.1. Let us call these HFRPDFT implementations G', H' and R' respectively.

  Now suppose that H' is not empty, and that it contains no vertex not contained in G'. Then, also H is not empty and it contains no vertex not contained in G. Let c be any implicant of F that is assigned to H. Since $c_{\overline{x}}$ contains no vertex that is not contained in G, then we could have removed x from c leaving it an implicant of f. But this contradicts the primality of F.

  By a similar argument we can show that G' contains some vertex not contained in H'.

  We have now shown that:

  1. G', H' and R' are each individually HFRPDFT circuits
  2. The on-set of G' contains a vertex not contained in H' and
  3. The on-set of H' contains a vertex not contained in G'.

  Let t_1 and t_2 be two variables not in the support of f. Then by Theorem 6.2.1

  $$t_1 \cdot x \cdot G' + t_1 \cdot \overline{x} \cdot H' + t_2 \cdot R'$$

  is an HFRPDFT circuit.

  Thus Procedure 6.2.1 produces an HFRPDFT circuit using additional test inputs.

■

6.2.2 Retaining Hazard Free Operation

We wish to show that the techniques used to make the circuit hazard-free robustly path delay fault testable do not destroy the possibility to make the circuit hazard-free in operation using the algorithm described in Section 5.4.

To see that the circuit F' has the same hazards as F, observe that the inputs t_1 and t_2 do not change during normal operation, therefore their introduction does not create any hazards. Thus in normal operation F' operates as $G \cdot x + H \cdot \overline{x} + R$. The factorization of the initial circuit F into $G \cdot x + H \cdot \overline{x} + R$ can be accomplished simply using associativity and distributivity. Thus, according to Theorems 5.1.3, no hazards are introduced in this decomposition and the resulting circuit F' can be made hazard-free in operation.

6.3 Heuristic Procedures to Improve HFRPDFT Testability

In this section we present a number of heuristics which, while not guaranteeing to produce an HFRPDFT implementation of a circuit, may be expected to improve the HFRPDFT testability of a given circuit, without introducing new hazards in its operation.

6.3.1 Algebraic Factorization

In addition to being a useful technique for circuit optimization, *algebraic factorization* can be used as a heuristic for improving the delay fault testability of a circuit. Given a cover $F \cdot G + F \cdot H$, if the paths associated with F were HFRPDFT either in G *or* in H (but not necessarily HFRPDFT in both), then if the cover is algebraically factored to produce $F \cdot (G + H)$ then F becomes *fully* HFRPDFT. This is due to the collapsing of paths that is a natural by-product of algebraic factorization. As a simple illustration of this consider the following example drawn from [35]: let C^T be the two-level circuit $a\overline{b} + \overline{b}c + b\overline{c}$ and let C^M be its multi-level algebraic factorization $(a + c)\overline{b} + b\overline{c}$. The path associated with literal $\overline{b}$ in cube $a\overline{b}$ of C^T, call it π_1, is not HFRPDFT, but the path associated with $\overline{b}$ in cube $\overline{b}c$ of C^T, call it π_2, is HFRPDFT. After C^T is factored into C^M there is a many-to-one reduction from paths π_1 and π_2 to a single path π associated with literal $\overline{b}$ in the factor $(a + c)\overline{b}$, and the testability of π_2 alone is sufficient to ensure the testability of π. As a result C^M is completely HFRPDFT while C^T is not.

As algebraic factorization is essentially an iterative application of the associative law, it retains the hazard properties of the initial two-level implementation, so that again the algorithms of Section 5.4 can be applied to make the circuit hazard-free in operation.

6.3.2 Complementation

When the *on-set* cover of the next-state function of a signal is not HFRPDFT, then it may be the case that its companion *off-set* cover is more easily made HFRPDFT. Consider the function f implemented in the circuit $F = \overline{a}\,\overline{c} + a\overline{b} + \overline{a}b + c\overline{d} + \overline{c}d$. While this is one of a few prime and irredundant implementations of this function, the paths associated with $\overline{a}$ and $\overline{c}$ in $\overline{a}\,\overline{c}$ are not HFRPDFT. Furthermore all prime and irredundant implementations of this function share this problem. Algebraically factoring out $\overline{a}$ produces the circuit $\overline{a}(\overline{c} + b) + a\overline{b} + c\overline{d} + \overline{c}d$. While the path associated with $\overline{a}$ has become HFRPDFT, the path associated with $\overline{c}$ in $(\overline{c} + b)$ is still not HFRPDFT. Furthermore, no other application of algebraic factorization will make this path HFRPDFT. Thus algebraic factorization alone cannot be used to make this circuit HFRPDFT.

An alternative way of getting a fully HFRPDFT implementation of f is to implement an *off-set* cover of f, rather than an on-set cover, and complement the output. The off-set cover R of f is $R = \overline{a}\overline{b}cd + ab\overline{c}\overline{d} + abcd$. This implementation is HFRPDFT. Note that in this case the final prime selection step in Procedure 5.2.1 must be done, obviously, using the cubes of R' rather than the cubes of F'.

Another case in which this can be useful is when the on-set cover of f obtained by Procedure 5.2.1 is redundant, while the corresponding off-set cover is not.

Having implemented R, we can apply Theorem 5.1.3, that states that introducing an inverter at the output of a circuit does not introduce or remove hazards. Thus complementing the output of R to produce f retains the hazard properties of R, and the resulting circuit can be made hazard-free.

6.4 A Procedure Guaranteed to Generate an RGDFT Circuit

In Section 6.2 we presented an algorithm that is guaranteed to produce a hazard-free robustly *path delay fault* testable implementation of a circuit. This algorithm has the potential disadvantage that test inputs may be required. In this section we present a technique that also requires the introduction of test inputs but may require fewer test inputs, producing an implementation that is robustly *gate delay fault* testable. This delay fault testability model is less stringent than HFRPDFT, but it can still be sufficient to determine with the desired accuracy whether the delays in the manufactured circuit lie within the bounds assumed during synthesis, thus ensuring hazard-free operation.

6.4.1 Making the Circuit Robustly Gate Delay Fault Testable

The initial two-level cover produced by the techniques in Section 5.2.1 is prime but may not be irredundant. The first step required is to make each cube *irredundant* through the introduction of test inputs. For each redundant cube in the cover we add one test input.

Starting now with an irredundant two-level cover we proceed to make the cover RGDFT through the introduction of a single test input. This procedure is modeled after the procedure given in [99] to improve stuck-open fault testability. To motivate this procedure we will employ the following Lemma from [33, 36].

Lemma 6.4.1 : *Let C be a two-level single-output circuit and let z_i be a gate in C. If a path π through z_i is* HFRPDFT *then z_i is* RGDFT.

It is likely that after applying the heuristic transformations described in Section 6.3 that *at least* one path per gate is HFRPDFT. If this is not the case then the following procedure will make the circuit RGDFT. As usual, let F be a two-level *prime* and *irredundant* combinational circuit that we want to make RGDFT.

Procedure 6.4.1

1. *If F is not* RGDFT *then:*
   - (a) *Let $C = \{c_1, c_2, \ldots, c_n\}$ be the set of cubes in F that are not* RGDFT.
   - (b) *Let t be an input not in the support of F.*
   - (c) *Add t to each cube in C. For example $c_i' = t \cdot c_i$.*

We now proceed to prove that the resulting circuit F is RGDFT.

Proof If a cube c is not in C, then it was already RGDFT, and the introduction of t does not change this property.

Suppose c is in C. The circuit F is prime and irredundant both before and after Procedure 6.4.1. Let v be a relatively essential vertex of c. Consider $t \cdot v$. Clearly, $t \cdot c$ covers $t \cdot v$. Furthermore no other cube in F covers v otherwise v would not be a relatively essential vertex of c. So $t \cdot v$ is a relatively essential vertex of $t \cdot c$. Consider the vector pair $(\bar{t} \cdot v, t \cdot v)$. The vector $t \cdot v$ is a relatively essential vertex of $t \cdot c$ and it must be the case that the vector $\bar{t} \cdot v$ is in the off-set of F. Therefore by the necessary and sufficient conditions for HFRPDFT in two-level networks given in [36, 33] the path associated with t in $t \cdot c$, call it π is HFRPDFT. Now by Lemma 6.4.1 given above, because π is HFRPDFT then the *and* gate associated with cube $t \cdot v$ must also be RGDFT. Furthermore, because π is HFRPDFT, the output *or* gate of F must also be RGDFT, by the reasoning of the same Lemma. Thus Procedure 6.4.1 creates a two-level circuit that is completely RGDFT. ■

To create an RGDFT multilevel circuit from this two-level circuit then the constrained factorization techniques of [34, 35] may be applied.

To see that the output of Procedure 6.4.1 can be made *hazard-free*, observe that the input t does not change during normal operation, therefore its introduction does not create any hazards. Thus in normal operation the new circuit has exactly the same hazard properties as the old one.

6.5 Design for Delay Testability Methodology

So far we have described the effective procedures of Sections 6.2 and 6.4 and the heuristic techniques of Section 6.3 as independent procedures, but in fact the most effective use of these techniques involves their integrated application. If the initial two-level circuit produced in Section 5.2.1 is HFRPDFT, then algebraic factorization can be freely applied and HFRPDFT will be retained.

If the initial two-level circuit is not HFRPDFT, then the best course is to iteratively apply the heuristics of Section 6.3. If we reach a point at which further applications of algebraic factorization alone will not improve the testability of the circuit, then we have two courses.

1. We can examine the RGDFT of the current circuit. If the fault coverage in this model is high (as would be expected) and this fault model is acceptable, then we can terminate at this point. We can also achieve complete RGDFT using the techniques of Section 6.4.

2. Alternatively, if complete fault coverage in the HFRPDFT model is desired, then this can be achieved at this point using the techniques of Section 6.2.

Through the integrated application of these techniques we aim to achieve the desired fault coverage with the least penalty in area overhead and the fewest additional test inputs.

At this point, we have obtained an HFRPDFT (or RGDFT) implementation of the STG specification that has exactly the same hazard properties as the initial two-level cover. So we can now apply the techniques described in Section 5.4 in order to obtain a circuit that is hazard-free in operation. These techniques simply change the delays of some signals, so both HFRPDFT and RGDFT are maintained in the final hazard-free implementation.

6.6 Experimental Results

In this section we present our experimental results on the set of STG examples used also in the previous chapters.

Table 6.1 gives the area and delay (using a standard cell implementation) of:

- a hazard-free, but not completely testable, optimized implementation (column "Untestable").
- a hazard-free HFRPDFT optimized implementation (column "Testable").

The area, delay and padding columns are as in Section 5.6.

Table 6.2 summarizes the cases when a decomposition was required to make the circuit testable. The column labeled "Decomp." gives the number of times Procedure 6.2.1 had to decompose the circuit in order to make it HFRPDFT, while the column labeled "Test inp." gives the number of added test inputs for each circuit.

example	Untestable				Testable			
	With hazards		Without Hazards		With hazards		Without Hazards	
	Area	Delay	Area	Pad	Area	Delay	Area	Pad
chu133	224	4.4	224	0.0	224	4.4	224	0.0
chu150	200	4.6	200	0.0	216	4.2	248	2.4
chu172	104	1.6	104	0.0	104	1.6	104	0.0
converta	368	5.0	368	0.0	448	6.2	448	0.0
ebergen	216	3.2	216	0.0	232	4.8	248	1.2
full	192	4.6	208	1.2	192	4.6	208	1.2
hazard	200	4.2	200	0.0	216	4.4	248	2.4
hybridf	242	5.0	258	1.2	242	5.0	258	1.2
nowick	232	4.6	232	0.0	232	4.6	232	0.0
alloc-outbound	272	4.0	272	0.0	272	4.0	272	0.0
mp-forward-pkt	232	3.4	232	0.0	232	3.4	232	0.0
nak-pa	256	3.4	256	0.0	256	3.4	256	0.0
pe-rcv-ifc	752	6.6	784	2.4	752	6.6	784	2.4
pe-send-ifc	912	7.4	1056	10.8	912	7.4	1056	10.8
ram-read-sbuf	384	4.2	416	2.4	384	4.2	416	2.4
rcv-setup	128	2.8	128	0.0	128	2.8	128	0.0
sbuf-ram-write	296	4.0	296	0.0	296	4.0	296	0.0
sbuf-read-ctl	272	4.2	272	0.0	272	4.2	272	0.0
sbuf-send-ctl	280	3.2	280	0.0	280	3.2	280	0.0
sbuf-send-pkt2	320	4.4	416	7.2	320	4.4	416	7.2
sendr-done	96	3.0	112	1.2	96	3.0	112	1.2
qr42	216	3.2	216	0.0	232	4.8	248	1.2
rpdft	168	5.2	232	4.8	160	4.0	192	2.4
trimos-send	576	4.6	576	0.0	576	4.6	576	0.0
vbe10b	688	7.0	784	7.2	688	7.0	784	7.2
vbe5b	208	4.0	208	0.0	208	3.2	208	0.0
vbe5c	160	3.0	176	1.2	160	3.0	176	1.2
wrdatab	664	4.8	680	1.2	672	7.2	688	1.2
total	8858	119.6	9402	40.8	9002	124.2	9610	45.6

Table 6.1: Area and delay cost of robust path delay fault testability

example	Testable					
	With hazards		Without Hazards			
	Area	Delay	Area	Pad	Decomp.	Test inp.
chu133	224	4.4	224	0.0	0	0
chu150	216	4.2	248	2.4	1	2
chu172	104	1.6	104	0.0	0	0
converta	448	6.2	448	0.0	1	2
ebergen	232	4.8	248	1.2	2	2
full	192	4.6	208	1.2	0	0
hazard	216	4.4	248	2.4	2	2
hybridf	242	5.0	258	1.2	0	0
nowick	232	4.6	232	0.0	0	0
alloc-outbound	272	4.0	272	0.0	0	0
mp-forward-pkt	232	3.4	232	0.0	0	0
nak-pa	256	3.4	256	0.0	0	0
pe-rcv-ifc	752	6.6	784	2.4	0	0
pe-send-ifc	912	7.4	1056	10.8	0	0
ram-read-sbuf	384	4.2	416	2.4	0	0
rcv-setup	128	2.8	128	0.0	0	0
sbuf-ram-write	296	4.0	296	0.0	0	0
sbuf-read-ctl	272	4.2	272	0.0	0	0
sbuf-send-ctl	280	3.2	280	0.0	0	0
sbuf-send-pkt2	320	4.4	416	7.2	0	0
sendr-done	96	3.0	112	1.2	0	0
qr42	232	4.8	248	1.2	2	2
rpdft	160	4.0	192	2.4	1	2
trimos-send	576	4.6	576	0.0	0	0
vbe10b	688	7.0	784	7.2	0	0
vbe5b	208	3.2	208	0.0	0	0
vbe5c	160	3.0	176	1.2	0	0
wrdatab	672	7.2	688	1.2	1	2
total	9002	124.2	9610	45.6	10	14

Table 6.2: Cases when test inputs must be added

Chapter 7

Conclusions

This work had many ambitious objectives.

Our first aim was to show that the claim (due to A. Turing) that asynchronous circuits are hard to design is valid, but that this difficulty can be overcome by providing the designer with *automated aids* that eliminate the most time-consuming and error-prone parts of the design process.

Despite reasonable objections to a shift from the well-established synchronous design methodology to the relatively new self-timing-based asynchronous methodology, asynchronous designs are going to become a key component of many digital systems of the future. The main reason is that some of the major problems facing today's designers, such as clock skew, scalability, modularity, and power consumption, can be alleviated, if not overcome, with a clock-less design methodology. This methodology must be at least as *efficient* and *effective* as existing synchronous methodologies, in terms of design effort and quality of the final result.

Highly critical applications, such as avionics, can generally afford to be more expensive than average consumer products (even when the latter require a large amount of data processing, as in the case of high-definition TV or of personal communications systems). So a realistic design methodology must allow the designer to trade off reliability against cost, exactly as in existing synchronous design techniques.

Finally, any approach to integrated circuit design automation must have a sound theoretical basis, and its practical significance must be demonstrated with a set of complete experimental results.

We believe that the complete asynchronous circuit design methodology developed in this thesis satisfies the requirements listed above. The proposed methodology contains several phases, from system specification to implementation and testing.

The system specification phase deals with issues such as the dependency on component delays, the robustness with respect to environmental variations, the use of a specific physical design methodology, and the existence of standard specification or of pre-designed parts that need to be re-used. The formal tools required to tackle these problems are described in Chapter 3, which, for example, defines the characteristics of the specification that lead to a design whose behavior does not depend on gate or interconnection delays. The precise characterization of the implementation behavior is obtained with a circuit model that permits the description of a pre-existing component at the desired level of detail or abstraction, using non-determinism.

The circuit behavior is specified using the Signal Transition Graph (STG), a formalization of timing diagrams. The STG is suitable for directly modeling all major asynchronous behavior paradigms, such as *choice*, *concurrency*, *causality*. *Synchronization* and *meta-stability* issues, while not handled directly with the STG, can be cleanly separated from the rest of the design and solved using the appropriate analog design techniques. The components dealing with these synchronization issues can then be uniformly incorporated in the overall circuit model. Chapter 1 contains a realistic example of a *VMEbus* interface. It is specified directly from a timing diagram as a set of two Signal Transition Graphs and a synchronization element. Alternately, the circuit behavior can be given as a Finite State Machine description which is automatically translated, under suitable assumptions on the environment behavior, into an STG.

Even though we do not deal directly with *formal verification* issues, the specification itself is formally defined, hence lending itself to existing automated verification techniques. These techniques can be used to prove that the described behavior satisfies a set of properties that define its correctness (*design verification*), and that the circuit implements the specification (*implementation verification*). The latter verification step is required to ensure that algorithm implementation errors do not produce incorrect designs. Chapter 2 contains a summary of such techniques that can be applied for both the verification problems.

Chapters 4 and 5 contain an automated design procedure that produces a range of implementations of a given STG specification which depend on user-specified cost functions that take into account area, delay, throughput and target implementation technology. The procedure is correct-by-construction, and produces an optimized hazard-free circuit using information on the gate, wire and environment delays. Looser bounds on such delays produce a more robust but possibly more expensive design, leaving the trade-off to the designer.

Finally, since an integrated circuit cannot be used unless it is thoroughly tested for possible manufacturing defects, we give a methodology to test the circuit using the most stringent delay fault model used in practice, the hazard-free robust path delay fault model. The need to test each manufactured circuit for path delay faults also arises because the synthesized circuit is hazard-free only if the delay bounds used in the synthesis phase are actually met.

Our design methodology is aimed at the efficient solution of *low-level* logic design problems, so it has a potentially wider range of applications than as a stand-alone tool. In fact, it can be used as a component in a higher level synthesis system for asynchronous circuits that would rely on our procedures to perform the logic level design.

We hope that the algorithms described in this work, which has been tested experimentally on a large set of example specifications taken from both industry and academic publications, will gain acceptance in the digital circuit design community. We hope that this methodology will enable designers to overcome the problems of the next generations of Ultra-High-Performance, Ultra-Large-Scale-of-Integration digital integrated circuits, since they thus far are intractable using current techniques. Its potential range of application spans the entire spectrum of design problems, from high-end super-computers to low-power, low-budget consumer electronics and appliances.

Bibliography

[1] T. Agerwala. Putting Petri Nets to work. *Computer*, pages 85–93, December 1979.

[2] F. Aghdasi and M. Bolton. Self-clocked asynchronous finite state machine design with PAL22IP6 device. *Microprocessors and Microsystems*, February 1991.

[3] V. Akella and G. Gopalakrishnan. SHILPA: a high-level synthesis system for self-timed circuits. In *Proceedings of the International Conference on Computer-Aided Design*, November 1992.

[4] R. Alur, C. Courcoubetis, and D. Dill. Model-Checking for Real-Time Systems. In *Proceedings of the Fifth Annual IEEE Symposium on Logic in Computer Science*, pages 414–425, June 1990.

[5] D. B. Armstrong, A. D. Friedman, and P. R. Menon. Design of asynchronous circuits assuming unbounded gate delays. *IEEE Transactions on Computers*, C-18(12):1110–1120, December 1969.

[6] P. A. Beerel and T. H-Y. Meng. Semi-modularity and self-diagnostic asynchronous control circuits. In *Proceedings of the Conference on Advanced Research in VLSI*, March 1991.

[7] P. A. Beerel and T. H-Y. Meng. Automatic gate-level synthesis of speed-independent circuits. In *Proceedings of the International Conference on Computer-Aided Design*, November 1992.

[8] P. A. Beerel and T. H-Y. Meng. Gate-level synthesis of speed-independent asynchronous control circuits. In *Proceedings of the ACM International Workshop on Timing Issues in the Specification and Synthesis of Digital Systems (TAU)*, March 1992.

[9] C. Berthet and E. Cerny. Synthesis of speed-independent circuits using set-memory elements. In G. Saucier, editor, *Proceedings of the International Workshop Logic and Architectural Synthesis for Silicon Compilers*. Grenoble, France, May 1988.

[10] E. Best. Structural theory of Petri Nets: the free-choice hiatus. *Lecture Notes in Computer Science*, 254:168–206, 1987.

[11] E. Best and J. Esparza. Model checking of persistent Petri Nets. In *Computer Science Logic 91 (LNCS)*, 1991. Also appeared as Hildesheimer Informatik Fachbericht 11/91.

[12] E. Best and K. Voss. Free choice systems have home states. *Acta Informatica*, 21:89–100, 1984.

[13] D. L. Black. On the existence of delay-insensitive fair arbiters: Trace theory and its limitations. *Distributed Computing*, 1:205–225, 1986.

[14] G. Borriello. *A New Interface Specification Methodology and its Application to Transducer Synthesis*. PhD thesis, U.C. Berkeley, May 1988.

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

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

[17] E. Brunvand and R. F. Sproull. Translating concurrent programs into delay-insensitive circuits. In *Proceedings of the International Conference on Computer-Aided Design*, pages 262–265, November 1989.

[18] R. Bryant. Graph-Based Algorithms for Boolean Function Manipulation. *IEEE Transactions on Computers*, C-35:677–691, August 1986.

[19] J. A. Brzozowski and C-J. Seger. Advances in asynchronous circuit theory – part I: Gate and unbounded inertial delay models. *Bulletin of the European Association of Theoretical Computer Science*, October 1990.

[20] J. A. Brzozowski and C-J. Seger. Advances in asynchronous circuit theory – part II: Bounded inertial delay models, mos circuits, design techniques. *Bulletin of the European Association of Theoretical Computer Science*, March 1991.

[21] J. R. Burch. *Automatic Symbolic Verification of Real-Time Concurrent Systems*. PhD thesis, Carnegie Mellon University, August 1992.

[22] J. R. Burch. Delay models for verifying speed-dependent asynchronous circuits. In *Proceedings of the ACM International Workshop on Timing Issues in the Specification and Synthesis of Digital Systems (TAU)*, March 1992.

[23] S. Burns and A. Martin. A synthesis method for self-timed VLSI circuits. In *Proceedings of the International Conference on Computer Design*, 1987.

[24] S. M. Burns. *Performance Analysis and Optimization of Asynchronous Circuits*. PhD thesis, California Institute of Technology, December 1990.

[25] T.-A. Chu. On the models for designing VLSI asynchronous digital systems. *Integration: the VLSI journal*, 4:99–113, 1986.

[26] T.-A. Chu. Synthesis of self-timed control circuits from graphs: an example. In *Proceedings of the International Conference on Computer Design*, pages 565–571, 1986.

[27] T.-A. Chu. *Synthesis of Self-timed VLSI Circuits from Graph-theoretic Specifications*. PhD thesis, MIT, June 1987.

[28] T.-A. Chu. Synthesis of hazard-free control circuits from asynchronous finite state machine specifications. In *Proceedings of the ACM International Workshop on Timing Issues in the Specification and Synthesis of Digital Systems (TAU)*, 1992.

[29] H. Y.H. Chuang and S. Das. Synthesis of multiple-input change asynchronous machines using controlled excitation and flip-flops. *IEEE Transactions on Computers*, pages 1103–1109, December 1973.

[30] F. Commoner, A. W. Holt, S. Even, and A. Pnueli. Marked directed graphs. *Journal of Computer and System Sciences*, 5:511–523, 1971.

[31] M. E. Dean, T. Williams, and D. Dill. Efficient self-timing with level-encoded 2-phase dual rail (LEDR). In *Proceedings of the Conference on Advanced Research in VLSI*, March 1991.

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

[33] S. Devadas and K. Keutzer. Necessary and Sufficient Conditions for Robust Delay-Fault Testability of Logic Circuits. In *Proceedings of the Sixth MIT Conference on Advanced Research on VLSI*, pages 221–238, April 1990.

[34] S. Devadas and K. Keutzer. Synthesis and Optimization Procedures for Robustly Delay-Fault Testable Logic Circuits. In *Proceedings of the* 27^{th} *Design Automation Conference*, pages 221–227, June 1990.

[35] S. Devadas and K. Keutzer. Synthesis of Robust Delay-fault Testable Circuits: Practice. *IEEE Transactions on Computer-Aided Design of Integrate Circuits and Systems*, October 1991.

[36] S. Devadas and K. Keutzer. Synthesis of Robust Delay-fault Testable Circuits: Theory. *IEEE Transactions on Computer-Aided Design of Integrate Circuits and Systems*, September 1991.

[37] S. Devadas, K. Keutzer, S. Malik, and A. Wang. Verification of asynchronous interface circuits with bounded wire delays. In *Proceedings of the International Conference on Computer-Aided Design*, pages 188–195, November 1992.

[38] D. L. Dill. Timing assumptions and verification of finite-state concurrent systems. In *Proceedings of the LCNS Workshop on Automatic Verification Methods for Finite State Systems*, 1989.

[39] D.L. Dill. *Trace Theory for Automatic Hierarchical Verification of Speed-Independent Circuits*. The MIT Press, Cambridge, Mass., 1988. An ACM Distinguished Dissertation 1988.

[40] J. C. Ebergen. *Translating programs into delay-insensitive circuits*. Centrum voor Wiskunde en Informatica, Amsterdam, 1989.

[41] E. Eichelberger and T. W. Williams. A logical design structure for LSI testing. In *Proceedings of the* 14^{th} *Design Automation Conference*, pages 462–468, June 1977.

[42] E. B. Eichelberger. Hazard detection in combinational and sequential switching circuits. *IBM Journal of Research and Development*, 9, March 1965.

[43] S. Eilenberg. *Automata, Languages and Machines*, volume A. Academic press, 1974.

[44] M. R. Garey and D. S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-completeness*. W. H. Freeman and Company, 1979.

[45] J. Gimpel. A reduction technique for prime implicant tables. *IRE Transactions on Electronic Computers*, August 1965.

[46] A. Grasselli and F. Luccio. A method for minimizing the number of internal states in incompletely specified sequential networks. *IRE Transactions on Electronic Computers*, EC-14(3):350–359, June 1965.

[47] J. Gunawardena and M. Nielsen. Muller unfoldings. (in preparation), 1992.

[48] M. Hack. Analysis of production schemata by Petri Nets. Technical Report TR 94, Project MAC, MIT, 1972.

[49] S. Heath. *VMEbus user's handbook.* CRC press, 1988.

[50] L. G. Heller, W. R. Griffin, J. W. Davis, and N. G. Thoma. Cascode voltage switch logic: A differential CMOS logic family. In *IEEE International Solid State Circuits Conference*, 1984.

[51] C. A. R. Hoare. Communicating Sequential Processes. In *Communications of the ACM*, pages 666–677, August 1978.

[52] L. A. Hollaar. Direct implementation of asynchronous control units. *IEEE Transactions on Computers*, C-31(12):1133–1141, December 1982.

[53] D. A. Huffman. The synthesis of sequential switching circuits. *J. Franklin Institute*, 257:161–190,275–303, March 1954.

[54] IEEE Computer Society, New York, N.Y. *Futurebus+P896.1: Logical Layer Specifications (Draft 8.2)*, January 1990.

[55] N. Ishiura, M. Takahashi, and S. Yajima. Time-symbolic simulation for accurate timing verification. In *Proceedings of the Design Automation Conference*, pages 497–502, June 1989.

[56] N. Ishiura, M. Takahashi, and S. Yajima. Coded time-symbolic simulation using shared binary decision diagrams. In *Proceedings of the Design Automation Conference*, pages 130–135, June 1990.

[57] M. B. Josephs and J. T. Udding. Delay-insensitive circuits: An algebraic approach to their design. In J. C. M. Baeten and J. W. Klop, editors, *CONCUR '90, Theories of Concurrency: Unification and Extension*, volume 458 of *Lecture Notes in Computer Science*, pages 342–366. Springer-Verlag, August 1990.

[58] T. Kam. Multi-valued decision diagrams. Master's thesis, U.C. Berkeley, 1990.

[59] R.M. Keller. A fundamental theorem of asynchronous parallel computation. *Lecture Notes in Computer Science*, 24:103–112, 1975.

[60] M. A. Kishinevsky, A. Y. Kondratyev, and A. R. Taubin. Formal method for self-timed design. In *Proceedings of the European Design Automation Conference*, 1991.

[61] M. A. Kishinevsky, A. Y. Kondratyev, A. R. Taubin, and V. I. Varshavsky. Analysis and identification of self-timed circuits. In *Proceedings of IFIP* 2^{nd} *Workshop on Designing Correct Circuits*, pages 275–287, January 1992.

[62] M. A. Kishinevsky, A. Y. Kondratyev, A. R. Taubin, and V. I. Varshavsky. On self-timed behavior verification. In *Proceedings of the ACM International Workshop on Timing Issues in the Specification and Synthesis of Digital Systems (TAU)*, 1992.

[63] M. A. Kishinevsky, A. Y. Kondratyev, A. R. Taubin, and V. I. Varshavsky. *Concurrent Hardware. The Theory and Practice of Self-Timed Design.* John Wiley and Sons Ltd., February 1993. To appear.

[64] I. Kohavi and Z. Kohavi. Detection of multiple faults in combinational logic networks. *IEEE Transactions on Computers*, C21(6):556–568, June 1972.

[65] Z. Kohavi. *Switching and Finite Automata Theory*. Computer Science Press, 1978.

[66] A. Y. Kondratyev, L. Y. Rosenblum, and A. V. Yakovlev. Signal graphs: a model for designing concurrent logic. In *Proceedings of the 1988 International Conference on Parallel Processing*. The Pennsylvania State University Press, 1988.

[67] S. Kundu and S. M. Reddy. On the Design of Robust Testable CMOS Combinational Logic Circuits. In *Proceedings of the Fault Tolerant Computing Symposium*, pages 220–225, 1988.

[68] D. Kung. Hazard-non-increasing gate-level optimization algorithms. In *Proceedings of the International Conference on Computer-Aided Design*, November 1992.

[69] M. Ladd and W. P. Birmingham. Synthesis of multiple-input change asynchronous finite state machines. In *Proceedings of the Design Automation Conference*, pages 309–314, June 1991.

[70] T. Larrabee. Test pattern generation using boolean satisfiability. *IEEE Transactions on Computer-Aided Design*, 11(8), January 1992.

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

[72] H. R. Lewis. Finite-state analysis of asynchronous circuits with bounded temporal uncertainty. Technical Report TR-15-89, Harvard University, 1989.

[73] B. Lin and F. Somenzi. Minimization of symbolic relations. In *Proceedings of the International Conference on Computer-Aided Design*, November 1990.

[74] C. J. Lin and S. M. Reddy. On Delay Fault Testing in Logic Circuits. *IEEE Transactions on Computer-Aided Design*, pages 694–703, September 1987.

[75] S. Malik. *Combinational Logic Optimization Techniques in Sequential Logic Synthesis*. PhD thesis, U.C. Berkeley, November 1990.

[76] A. Martin. Formal program transformations for VLSI synthesis. In E. W. Dijkstra, editor, *Formal Development of Programs and Proofs*, The UT Year of Programming Series. Addison-Wesley, 1990.

[77] A. Martin. The limitations to delay-insensitivity in asynchronous circuits. In *Proceedings of the Conference on Advanced Research in VLSI*, April 1990.

[78] A. Martin. Programming in VLSI: From communicating processes to delay-insensitive circuits. In C. A. R. Hoare, editor, *Developments in Concurrency and Communications*, The UT Year of Programming Series. Addison-Wesley, 1990.

[79] A. Martin. Synthesis of asynchronous VLSI circuits. In J. Staunstrup, editor, *Formal Methods for VLSI Design*. North-Holland, 1990.

[80] A. Martin, S. Burns, T. Lee, D. Borkovic, and P. Hazewindus. The design of an asynchronous microprocessor. In *Decennial Caltech Conference on VLSI*, pages 351–373. MIT press, 1987.

[81] T. Meng. *Asynchronous Design for Digital Signal Processing Architectures*. PhD thesis, U.C. Berkeley, November 1988.

[82] K. Mc Millan and D. Dill. Algorithms for interface timing verification. In *Proceedings of the ACM International Workshop on Timing Issues in the Specification and Synthesis of Digital Systems (TAU)*, March 1992.

[83] R. E. Miller. *Switching theory*, volume 2, chapter 10, pages 192–244. Wiley and Sons, 1965.

[84] D. Misunas. Petri Nets and speed-independent design. *Communications of the ACM*, pages 474–481, August 1973.

[85] C. E. Molnar, T.-P. Fang, and F. U. Rosenberger. Synthesis of delay-insensitive modules. In *Chapel Hill Conference on VLSI*, pages 67–86, May 1985.

[86] C. W. Moon. On synthesizing logic from signal transition graphs. Personal communication, 1990.

[87] C. W. Moon, P. R. Stephan, and R. K. Brayton. Synthesis of hazard-free asynchronous circuits from graphical specifications. In *Proceedings of the International Conference on Computer-Aided Design*, November 1991.

[88] D. E. Muller and W. C. Bartky. A theory of asynchronous circuits. In *Annals of Computing Laboratory of Harvard University*, pages 204–243, 1959.

[89] T. Murata. Petri Nets: Properties, analysis and applications. *Proceedings of the IEEE*, pages 541–580, April 1989.

[90] S. M. Nowick and D. L. Dill. Automatic synthesis of locally-clocked asynchronous state machines. In *Proceedings of the International Conference on Computer-Aided Design*, November 1991.

[91] S. M. Nowick and D. L. Dill. Exact two-level minimization of hazard-free logic with multiple-input changes. In *Proceedings of the International Conference on Computer-Aided Design*, November 1992.

[92] C. H. Papadimitriou and K. Steiglitz. *Combinatorial Optimization: Algorithms and Complexity*. Prentice-Hall, Englewood Cliffs, N. J., 1982.

[93] S. S. Patil. An asynchronous logic array. Technical Report MAC technical memorandum 62, MIT, 1975.

[94] S. S. Patil and J. B. Dennis. Speed independent asynchronous circuits. In *Proceedings of the Hawaii International Conference on System Sciences*, pages 55–58, 1971.

[95] S. S. Patil and J. B. Dennis. The description and realization of digital systems. In *Proceedings of the IEEE COMPCON*, pages 223–226, 1972.

[96] J. L. Peterson. *Petri Nets*, volume 9. ACM Computing Surveys, No. 3, September 1977.

[97] C. A. Petri. *Kommunikation mit Automaten*. PhD thesis, Bonn, Institut für Instrumentelle Mathematik, 1962.

[98] A. Pramanick and S. Reddy. On The Design of Path Delay Fault Testable Combinational Circuits. In *Proceedings of the* 20th *Fault Tolerant Computing Symposium*, pages 374–381, June 1990.

[99] S. M. Reddy and M. K. Reddy. Testable Realization for FET Stuck-Open Faults in CMOS Combinational Logic Circuits. In *IEEE Transactions on Computers*, volume C-35, pages 742–754, August 1986.

[100] M. Rem, J. L. A. van de Snepscheut, and J. T. Udding. Trace theory and the definition of hierarchical components. In *Proceedings of Third CalTech Conference on VLSI*, pages 225–239. Computer Science Press, Inc., 1983.

[101] F. U. Rosenberger, C. E. Molnar, T. J. Chaney, and T.-P. Fang. Q-modules: Internally clocked delay-insensitive modules. *IEEE Transactions on Computers*, 37:1005–1018, 1988.

[102] L. Y. Rosenblum and A. V. Yakovlev. Signal graphs: from self-timed to timed ones. In *International Workshop on Timed Petri Nets, Torino, Italy*, 1985.

[103] R. Rudell. Logic Synthesis for VLSI Design. Technical Report UCB/ERL M89/49, U. C. Berkeley, April 1989.

[104] A. Saldanha, T. Villa, R. K. Brayton, and A. L. Sangiovanni-Vincentelli. A framework for satisfying input and output encoding constraints. In *Proceedings of the* 28^{th} *Design Automation Conference*, pages 170–175, June 1991.

[105] C. J. Seger. *Models and Algorithms for Race Analysis in Asynchronous Circuits*. PhD thesis, University of Waterloo, Ontario, Canada, May 1988.

[106] C. L. Seitz. Chapter 7. In C. Mead and L. Conway, editors, *Introduction to VLSI Systems*. Addison Wesley, 1981.

[107] E. M. Sentovich, K. J. Singh, L. Lavagno, C. Moon, R. Murgai, A. Saldanha, H. Savoj, P. R. Stephan, R. K. Brayton, and A. Sangiovanni-Vincentelli. SIS: A system for sequential circuit synthesis. Technical Report UCB/ERL M92/41, U.C. Berkeley, May 1992.

[108] E.M. Sentovich, K.J. Singh, C. Moon, H. Savoj, R.K. Brayton, and A.L. Sangiovanni-Vincentelli. Sequential circuit design using synthesis and optimization. In *Proceedings of the International Conference on Computer Design*, October 1992.

[109] K. Shankar and D. Lee. Build a VMEbus interface with PAL devices. *Electronic Design*, 37(15):55–62, July 1989.

[110] K. J. Singh. *Performance Optimization of Digital Circuits*. PhD thesis, U.C. Berkeley, November 1992.

[111] K. S. Stevens, S. V. Robinson, and A. L. Davis. The post office – communication support for distributed ensemble architectures. In *Sixth International Conference on Distributed Computing Systems*, 1986.

[112] J. Sun and R. W. Brodersen. Design of system interface modules. In *Proceedings of the International Conference on Computer-Aided Design*, pages 478–481, November 1992.

[113] I. E. Sutherland. Micropipelines. *Communications of the ACM*, June 1989. Turing Award Lecture.

[114] W. Thomas. Automata on infinite objects. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science*. Elsevier, 1990.

[115] J. H. Tracey. Internal state assignments for asynchronous sequential machines. *IEEE Transactions on Electronic Computers*, EC-15(4):551–560, August 1966.

[116] J. T. Udding. A formal model for defining and classifying delay-insensitive circuits and systems. *Distributed Computing*, 1:197–204, 1986.

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

[118] K. van Berkel. Beware the isochronic fork. *Integration, the VLSI journal*, 13(2):103–128, June 1992.

[119] J. L. A. van de Snepscheut. *Trace Theory and VLSI Design*. PhD thesis, Eindhoven University of Technology, 1983.

[120] J. L. A. van de Snepscheut. *Trace Theory and VLSI Design*, volume 200 of *Lecture Notes in Computer Science*. Springer Verlag, Berlin, 1985.

[121] P. Vanbekbergen. Optimized synthesis of asynchronous control circuits from graph-theoretic specifications. In *Proceedings of the International Conference on Computer-Aided Design*, pages 184–187, November 1990.

[122] P. Vanbekbergen, G. Goossens, and H. De Man. A local optimization technique for asynchronous control circuits. In *Proceedings of the International Workshop on Logic Synthesis*, May 1991.

[123] P. Vanbekbergen, G. Goossens, and H. De Man. Specification and analysis of timing constraints in signal transition graphs. In *Proceedings of the European Design Automation Conference*, pages 302–306, 1992.

[124] P. Vanbekbergen, B. Lin, G. Goossens, and H. De Man. A generalized state assignment theory for transformations on signal transition graphs. In *Proceedings of the International Conference on Computer-Aided Design*, pages 112–117, November 1992.

[125] V. I. Varshavsky, M. A. Kishinevsky, A. Y. Kondratyev, L. Y. Rosenblyum, and A. R. Taubin. Models for specification and analysis of processes in asynchronous circuits. *Izvestiia Akademii nauk SSSR, Tekhnicheskaya Kibernetika*, pages 171–190, 1988. English translation: Soviet Journal of Computer and Systems Sciences.

[126] V. I. Varshavsky, M. A. Kishinevsky, V. B. Marakhovsky, V. A. Peschansky, L. Y. Rosenblum, A. R. Taubin, and B. S. Tzirlin. *Self-timed Control of Concurrent Processes.* Kluwer Academic Publisher, 1990. (Russian edition: 1986).

[127] T. Villa. A heuristic incompletely specified finite state machine minimizer. Personal communication, 1985.

[128] T. E. Williams. Analyzing and improving latency and throughput in self-timed pipelines and rings. In *Proceedings of the ACM International Workshop on Timing Issues in the Specification and Synthesis of Digital Systems (TAU)*, March 1992.

[129] D. Wong, G. De Micheli, and M Flynn. Inserting active delay elements to achieve wave pipelining. In *Proceedings of the International Conference on Computer-Aided Design*, pages 270–273, November 1989.

[130] A. V. Yakovlev. On limitations and extensions of STG model for designing asynchronous control circuits. In *Proceedings of the International Conference on Computer Design*, pages 396–400, October 1992.

[131] A. V. Yakovlev. A structural technique for fault-protection in asynchronous interfaces. In *Proceedings of* 22^{nd} *International Symposium on Fault-tolerant Computing (FTCS)*, pages 288–295, July 1992.

[132] A. V. Yakovlev. Analysis of concurrent systems through lattices. *Theoretical Computer Science*, Submitted for publication.

[133] A. V. Yakovlev and A. Petrov. Petri nets and parallel bus controller design. In *International Conference on Application and Theory of Petri Nets, Paris, France.* IEEE Computer Society, June 1990.

[134] S. Yang and M. Ciesielski. Optimum and suboptimum algorithms for input encoding and its relationship to logic minimization. *IEEE Transactions on Computer-Aided Design*, 10(1):4–12, January 1991.

[135] O. Yenersoy. Synthesis of asynchronous machines using mixed-operation mode. *IEEE Transactions on Computers*, pages 325–329, April 1979.

[136] K. Y. Yun and D. L. Dill. Automatic synthesis of 3D asynchronous state machines. In *Proceedings of the International Conference on Computer-Aided Design*, November 1992.

Index