TEHCNOLOGY MAPPING OF TIMED ASYNCHRONOUS CIRCUITS

by

Curtis A. Nelson

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

Doctor of Philosophy

in

Electrical Engineering

Electrical and Computer Engineering

The University of Utah

December 2003

THE UNIVERSITY OF UTAH GRADUATE SCHOOL

SUPERVISORY COMMITTEE APPROVAL

of a dissertation submitted by

Curtis A. Nelson

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

Chair: Chris J. Myers

Ken Stevens

Erik Brunvand

Reid R. Harrison

Behrouz

THE UNIVERSITY OF UTAH GRADUATE SCHOOL

FINAL READING APPROVAL

To the Graduate Council of the University of Utah:

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

Date

Chris J. Myers
Chair, Supervisory Committee

Approved for the Major Department

Chris J. Myers
Chair/Dean

Approved for the Graduate Council

David S. Chapman
Dean of The Graduate School

ABSTRACT

This paper presents an efficient method for verifying hazard-freedom in timed asynchronous circuits. Timed circuits are a class of asynchronous circuits that utilize explicit timing information for optimization throughout the entire design process. In asynchronous circuits, correct operation requires that there are no hazards in the circuit implementation. Therefore, when designing an asynchronous circuit, each internal node and output of the circuit must be verified for hazard-freedom to ensure correct operation. Current verification algorithms for timed asynchronous circuits require an explicit state exploration often resulting in state explosion for even modest sized examples. The goal of this work is to abstract the behavior of internal nodes and utilize this information to make a conservative determination of hazard-freedom for each node in the circuit. Experimental results indicate that this approach is substantially more efficient than existing timing verification tools. These results also indicate that this method scales well for large examples in that it is capable of analyzing circuits in less than a second that could not be previously analyzed. While this method is conservative in that some false hazards may be reported, our results indicate that the number of false hazards is small.

To all the insults that have made me who I am

CONTENTS

LIST OF TABLES

LIST OF FIGURES

CHAPTER 1

INTRODUCTION

Synchronous systems are those controlled by a central clock. These systems have dominated logic design since the advent of the integrated circuit and the necessary design styles and tools have become mature and well understood. However, with trends in the integrated circuit industry pushing performance to physical limitations, the timing issues involved with global clock synchronization have become increasingly difficult to resolve. Numerous studies have shown the significant design burden that clock skew and clock line load has placed on the synchronous system designer [11, 19, 35, 56, 58]. As clock speeds increase, the amount of time available per clock period decreases to the point where serious design problems and even outright failure can occur if these issues cannot be resolved. As a result, logic designers are turning to more aggressive design styles to address this particular issue.

Asynchronous design styles remove the need to address clocking issues by removing the global clock itself. However, asynchronous designs are difficult to implement reliably because of the presence of *hazards*. In many cases, the advantages inherent to asynchronous design are wasted due to the overhead required to implement hazard-free circuits. *Timed circuits* are a class of asynchronous design that uses explicit timing information in circuit synthesis [39]. The use of explicit timing information has been shown to potentially outperform synchronous designs and other asynchronous design styles as well. This was demonstrated in the Intel RAPPID project in which an asynchronous instruction length decoder for an x86 processor was designed using timed circuits which was three times faster while using half the power of the comparable synchronous design [53].

Researchers have developed numerous CAD tools to support asynchronous design styles. These tools have focused on the specification and synthesis stages of the design flow. Designs are specified at a high level and synthesis produces a logic description that is guaranteed to be hazard-free. Since logic synthesis is best optimized when it is not constrained by implementation, the resulting logic equations are unlikely to conform directly to available library parts. Currently, very few CAD tools are available to assist the asynchronous designer in implementing the synthesized equations. *Technology mapping*, also called library binding, is the process whereby a *technology independent* logic representation is mapped to a *technology dependent* library. For timed asynchronous circuits, this process is often done by hand as there are currently no automated CAD tools available that address the issue of hazards.

Synchronous designers have complete CAD tool-sets available from several companies and high-level, technology independent design descriptions can be implemented in available libraries with little or no custom work needed by the designer. These tools and the algorithms they employ cannot be directly applied to timed asynchronous designs because they have little or no ability to deal with hazards. Although performance pressure is causing designers to expand their consideration of other design methodologies, the adoption of timed asynchronous circuits is unlikely until the supporting CAD tools are complete and have been tested on industry grade designs. In addition, few designers are aware of these alternative methodologies because they remain primarily in the purview of research labs and universities, and as such, have seen little exposure to market driven pressures.

Our work attempts to bridge the gap between synchronous technology mapping techniques and the technology mapping of timed asynchronous circuits. Our goal is to complete the design flow for timed asynchronous circuits by developing an efficient and verifiable methodology for identifying and eliminating hazards created during technology mapping.

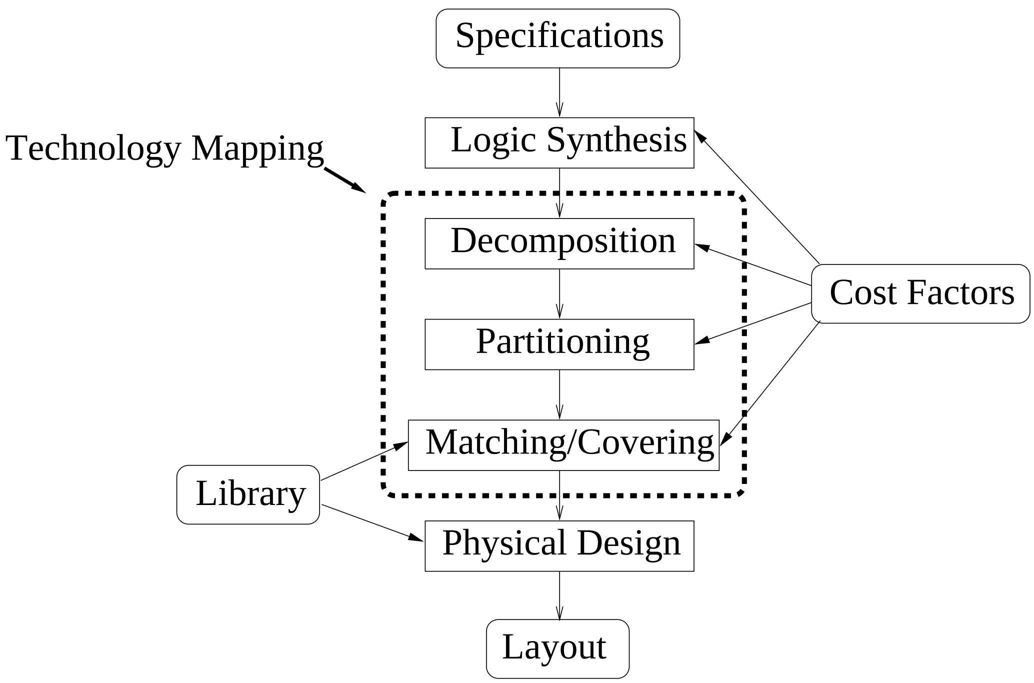

Figure 1.1. Synchronous design flow.

1.1 Synchronous Design Flow

The design flow for synchronous systems is shown in Figure 1.1. The process of automatic synthesis starts with a technology independent description of the design and ends with a technology dependent transistor level implementation.

The subject of fully automated systems for synchronous design has been an active area of research for the past 20 years [6, 7, 13, 22]. The theory and tools are mature and a number of companies such as Synopsis, Cadence, and Mentor Graphics currently produce complete tool sets allowing the designer an automated means of producing fully functional and verifiable designs. The design process typically starts with an abstract high-level behavioral description of the circuit. Over the years, this has migrated from schematics or RTL descriptions to high-level languages such as VHDL [54] or Verilog [55]. This allows for better maintenance, portability, and reuse of existing design efforts.

The logic synthesizer takes this high level description and creates a logical representation of the circuit implementation. This process is often guided by a cost metric such as area, power, or delay and the resulting logic equations reflect this metric. For instance, a one-hot encoding scheme may be used in a synchronous state machine to reduce the next-state logic which in turn reduces the combinational path delay. This allows for a faster clock but at the cost of increased area. The synthesis engine works best with few (or no) constraints and the resulting logic equations are seldom optimized. An optimal set of logic equations can then be created from a logic minimization program such as SIS [49].

1.2 Asynchronous Design Flow

Synchronous design flows cannot be used *as is* for asynchronous circuits because of the possibility of introducing hazards on new internal wires. Nor can timing information be used easily since causality is not expressed directly between outputs and inputs in sequential machines. As a result, the design flow shown in Figure 1.1 must

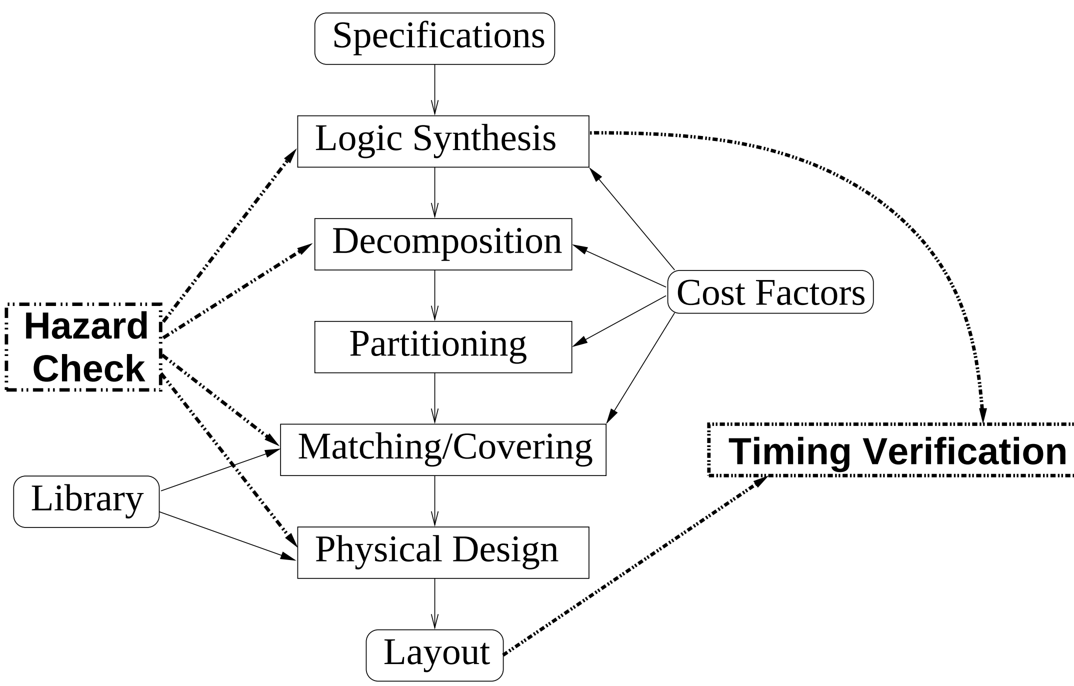

Figure 1.2. Asynchronous design flow.

be modified to accommodate the added restrictions placed on asynchronous design. Figure 1.2 shows the additional complexity associated with these modifications.

Hazard checking must begin in the synthesis stage of the design process. The particular asynchronous design style chosen determines the algorithms used during synthesis to generate a set of hazard-free logic equations. We assume that these logic expressions and supporting circuit descriptions (state graphs, time Petri nets, netlists, etc.) passed to the technology mapper are hazard-free under the constraints specified with the design.

Depending on the particular asynchronous design style, hazards can be introduced during the decomposition stage because the structure of the circuit changes. Each node in the new decomposed netlist must be annotated with hazard properties. This information is then passed along to the matching/covering stage where every attempt is made to eliminate the hazards by affecting the timing properties of the covered netlist or using the *atomic gate assumption* to encapsulate hazardous nodes. An atomic gate is modeled by a single delay element connected to its output and it is assumed that its operation is equivalent to a single-stack cell. Since physical design and layout can affect the timing behavior of the final circuit, timing verification must be performed after the final design is implemented in a given technology to ensure the mapped circuit meets the initial timing constraints. Failure to do so may result in circuits whose behavior is not consistent with the initial specifications.

1.3 Verification

While timed asynchronous circuits offer potential advantages over synchronous circuits such as faster operation and lower power, these advantages are often offset by the expense of the circuit overhead needed to eliminate *h*azards. Hazards are conditions generated by the structure of the circuit or timing relationships between inputs and propagation delays that can cause incorrect behavior. As synthesized hazard-free logic equations are mapped to a given gate library, new internal nodes are introduced in the circuit netlist. Each new internal node as well as the outputs of the circuit must be verified for hazard-freedom to ensure correct operation of the mapped circuit. This verification must be extremely efficient to allow for many alternative designs to be considered during technology mapping. Current timing verification algorithms [46, 5, 43, 36, 32] often suffer from state explosion problems because each node in the circuit netlist is treated as a new state variable, potentially doubling the number of states.

There have been numerous methods developed for the verification of gate-level *speed-independent* asynchronous circuits [16, 17, 4, 27, 21, 44, 45]. In speed-independent circuits, no timing assumptions are made about gates or the environment. In [4], an efficient verification method for determinate speed-independent circuits is proposed. This work reduces the state space explosion problem found in earlier methods by examining individual behavior at each internal node and approximating this behavior for each state in the specification. The hazard-freedom of the circuit is then verified by examining this *cube approximation.* When the number of internal signals is high as compared with the number of primary inputs and outputs (a feature common of many circuit design styles), this cube approximation technique has the potential to substantially reduce the complexity of verification as demonstrated in the results shown in [4].

1.4 Synchronous Technology Mapping

As previously mentioned, *technology mapping* is the process of binding the technology independent design to a dependent technology. The mapper takes as inputs a technology independent set of logic equations and a library of cells, and produces a technology dependent netlist implementing the circuit from cells found in the library. The process is often optimized for a particular cost metric such as area, speed, power, or in our case, hazard-freedom. Solving this problem exactly is intractable [20] so heuristics have been created to break the problem up into smaller parts to get as good an approximation as possible.

An excellent overview of synchronous technology mapping algorithms is given in [14, 23]. Algorithmic mappers [24, 34, 48] generally break the process up into three phases: decomposition, partitioning, and matching/covering.

The synthesized logic expressions from the initial network are represented as a directed acyclic graph (DAG). These expressions are decomposed into a multi-level circuit composed of simple gates called *base functions*. Base functions are typically two-input AND, OR, NAND, or NOR gates and possibly inverters. The implementation library must include these base functions to guarantee a solution.

Decomposition serves two purposes. First, it guarantees that a solution can be found, that is, a netlist of the logic equations can always be created from the library elements. Secondly, the finer granularity of a decomposed network allows for more matching options and a chance of a higher quality solution. This is particularly important since different matchings may be needed depending on the chosen cost metric.

As an example, Figure 1.3 shows two possible decomposition architectures for a 4-input AND gate. Which architecture is chosen depends on the cost metric. For instance, the decomposition shown in Figure 1.3(b) has a longer worst case path delay than the decomposition shown in Figure 1.3(c) so may not be desirable if delay is the primary cost metric. For our work, explicit timing information is available so it may be possible to order the later arriving inputs closer to the output to mitigate the longer delay or at least affect the hazard properties. The decomposition in Figure 1.3(b) may also prove to be attractive because the architecture is consistent for N-inputs whereas the architecture in Figure 1.3(c) will change depending on the number of inputs.

During partitioning [33], the network is split into single-output *cones* of logic where each cone represents a partition of the circuit obtained by creating a cut-set at points of multi-fanout. Each of these sub-circuits is called a *subject graph* which is matched to the various library cells, called *pattern graphs* in the matching/covering step. The purpose of partitioning then, is to divide the size of the circuit to make the covering function more tractable. In it's simplest form, the circuit is split at all points of multi-fanout and at the inputs to sequential elements. Each sub-partition is then covered locally. While this certainly makes the covering problem more tractable, the efficiency obtained by covering across partition boundaries

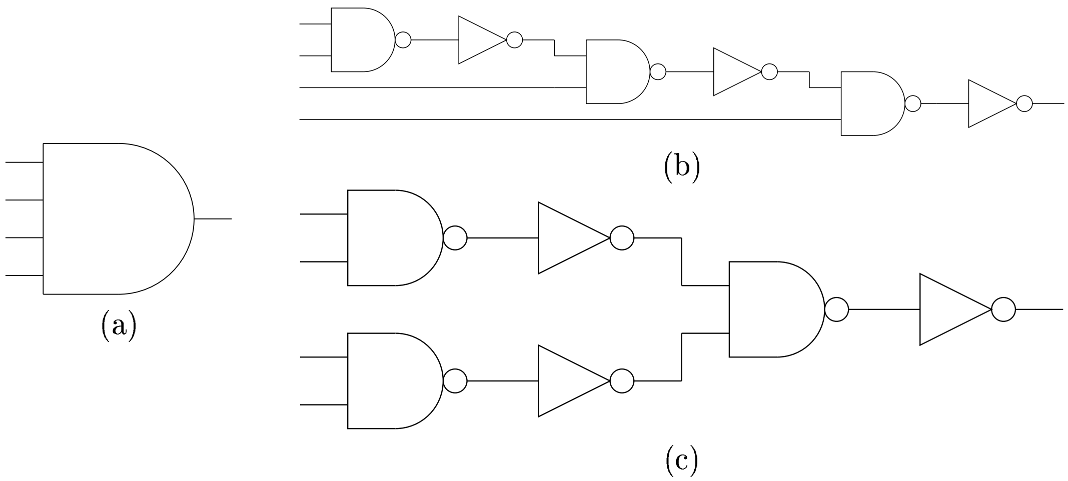

Figure 1.3. (a) 4-input AND gate. (b) A non-symmetric decomposition. (c) A symmetric decomposition.

is completely missed and the result is less optimum. Programs such as MIS [7] optimize the partitioning step to include the possibility of covering across partition boundaries. The final result of this optimization is likely to be a smaller circuit with a corresponding penalty paid in computation time.

The matching/covering step [33] finds all possible matches between the subject graph and pattern graph for each node in the sub-network. A set of these matches is then selected, as best conforms to the cost factors. Matching algorithms are classified as either *structural* or *boolean* and both methods are discussed in more detail below. Both methods fit into the design flow shown in Figure 1.1 and are differentiated primarily by the process which the subject graph, representing the synthesized design, is matched to the pattern graph, representing the available library elements.

Structural mapping is straight-forward and is the simplest to implement [24]. The decomposed functions can be viewed as trees, with roots at the outputs and the primary inputs as leaves. Beginning at the leaves, the decomposed subject graph is then compared node-by-node to the available library gates. Because of the semi-canonical nature of the decomposition, each node may have several variations that need to be examined. If a library gate can implement the tree to that point, the node is annotated with that information in addition to the cost up to that point. The cost includes the cost of generating the inputs and the cost of the gate. For example, in Figure 1.4(b), the cost at node e would be the cost of a 2-input NAND gate where the cost of node g could either be the cost of an inverter in series with a 2-input NAND gate or the cost of a 2-input AND gate.

When the root is reached, the lowest cost for implementing that portion of the circuit is chosen. If there are higher level functions available in the library such as 3-input NAND gates or AND-OR-INVERT gates, it is typically advantageous to enclose as many gates in the subject graph as possible. The complexity of the matching algorithm increases as the base functions used increase in complexity. The additional search space needed for complete matching makes base functions beyond 2-input NAND gates and inverters computationally expensive.

The structural mapping method usually gives good results but the quality can be affected by how the decomposition is done and even by the form of the equations derived to describe the design. Because of this, structural mapping does not find all possible covers for a design.

Boolean mapping uses a process similar to structural matching except the nodes are labeled with the Boolean functions they represent rather than the cost. In [33], Mailhot proposes to use a list of possible matching functions at all nodes. This list represents all combinations of intermediate nodes available. For instance, given the 4-input AND function $f = (abcd)$ shown decomposed into 2-input NAND gates and inverters in Figure 1.4(b), the network consists of the following nodes: $f = \neg j$, $j = \neg(id)$, $i = \neg h$, $h = \neg(gc)$, $g = \neg e$, $e = \neg(ab)$. The library will be exhaustively checked for boolean functions matching each node. For instance, node h would be checked for gates implementing the following functions: $h = \neg(gc)$, $h = \neg(\neg e(c))$, $h = \neg(\neg(\neg(ab))(c))$. Unlike the structural method, these gates here need not match exactly; they need only be equivalent in a Boolean manner. For instance, in boolean matching, the gate $g = a(b + c)$ matches the function $f = (ab + ac)$. A match of this type would not be found by the structural mapper unless both gates were structurally represented in the library. This boolean matching method can check for input permutations, inversions, and use *don't care* conditions to further optimize the solution. This method tends to be computationally prohibitive for larger networks but work by Burch [8] shows implicit methods can be used to produce more reasonable run times.

Looking again at Figures 1.4(b) and (c), we see two different structural decompositions of the 4-input AND gate shown in Figure 1.4(a). A boolean matching procedure at node f would find both of these decompositions equivalent. However, a structural mapper would find both of these a match to a 4-input AND gate only if their unique structural representations were entered as two separate library cells. Thus a library for a structural mapper must contain every possible structural representation of the function represented. Because of this, structural libraries tend to be limited to a smaller number of functions but the structural matching technique is extremely efficient computationally and the choice of which mapping technique to use depends as much on reasonable run times as it does on the quality of the final solution.

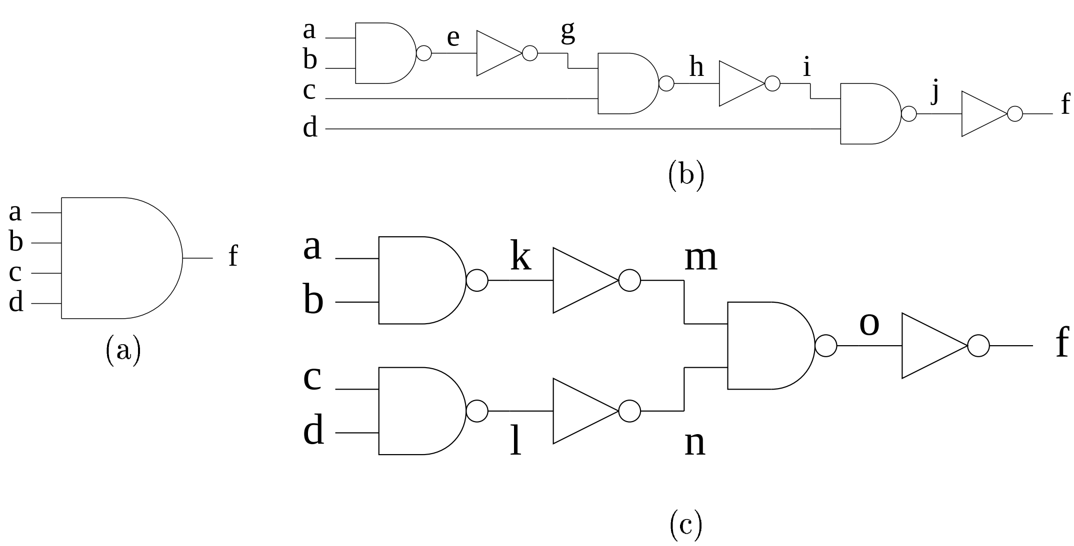

Figure 1.4. (a) 4-input AND gate. (b) A non-symmetric decomposition. (c) A symmetric decomposition.

1.5 Asynchronous Technology Mapping

The particular asynchronous design style chosen determines the types of hazards generated and the algorithms used to synthesize the design. Four main classes of asynchronous design styles avoid or reduce hazards during synthesis and technology mapping by simplifying the timing assumptions.

1.5.1 Delay Insensitive Circuits

Delay-insensitive circuits [47] make no assumptions on the delays of the logic or routing wires. This type of design is quite restrictive and few designs can be implemented by direct synthesis methods. Often, syntax-directed methods are employed which use non-standard libraries [18]. This approach renders technology mapping as unnecessary because the language compiler determines the structure and selects the library elements directly.

1.5.2 Speed Independent Circuits

Speed-Independent (SI) circuits [38] assume that the time skew of output branches is less than that of the logic gate delays. This timing assumption increases the potential for higher circuit performance by exploiting concurrency although not without it's added design challenges. Siegel showed in [50] that the partitioning and matching/covering algorithms that she originally described in [51] as pertaining to burst-mode circuits (discussed in section 1.5.3) can be used without modification as applied to SI technology mapping. For decomposition, she stated that OR functions can be decomposed according to the associative law independent of signal ordering and the resulting network is still hazard-free. However, AND functions must be dealt with in a completely different manner. She suggests adding acknowledgment wire forks defined in [1] and proposed for use during decomposition in [3] for those gates where a hazard-free decomposition does not exist. These cases occur when sequential hazards are generated, which happens whenever a monotonic transition on an intermediate gate is not acknowledged somewhere else in the circuit.

Work addressing the decomposition of speed-independent designs was first addressed by Kimura in [25, 26]. Kimura's work involved analysis of circuit delays when buffers were added to wires. In [29, 31, 30], Lavagno et al. leverages synchronous technology mapping methods by using the atomic gate assumption to produce hazard-free circuits but did so by introducing delay elements where needed to remove hazards. Myers et al. used a decomposition and re-synthesis approach applied to large-fanin generalized C-element gates [40]. A generalized C-element (gC) is a state holding component that also contains some random logic. Some of the most important work in this area is that done by Burns [9] where he addressed the decomposition of gC gates by using implicit methods to create and analyze large numbers of potential decompositions. Finally, Cortadella [12] and Kondratyev [28] more recently expanded the work of Burns to allow for a larger number of decompositions.

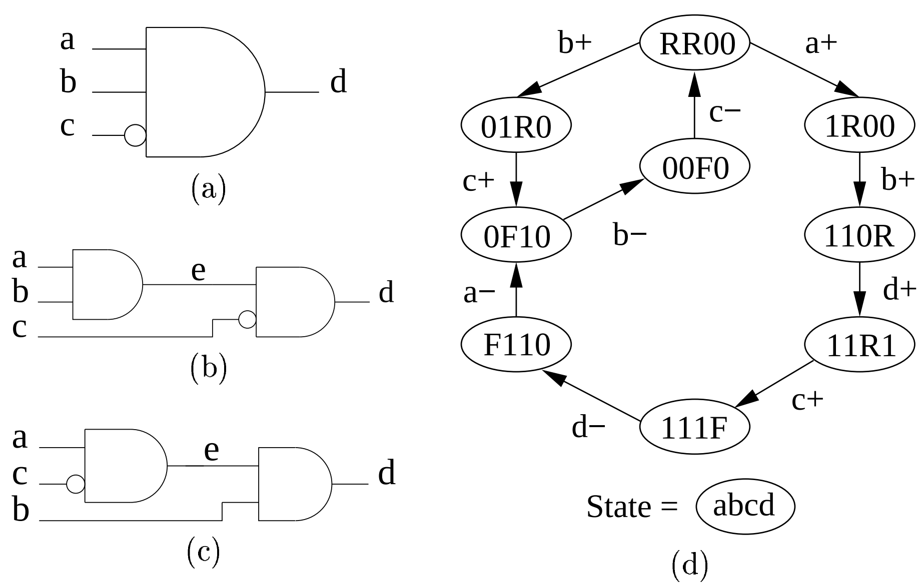

Figure 1.5. (a) 3-input AND function. (b) A circuit netlist that is hazardous under the speed-independent model. (c) A circuit netlist that is hazard-free under the speed-independent model. (d) State graph for 3-input AND function.

Figure 1.5 illustrates how the decomposition of an AND function may or may not be hazard-free under SI assumptions. The example shown is taken directly from [41] and utilizes a state graph (defined formally in chapter 2) shown in (d) to indicate a sequence of transitions. The implementation for output d whose behavior is shown in the state graph, is synthesized using a CAD tool and the results are shown in Figure 1.5(a). Suppose this 3-input AND function is not available in the library. Decomposition must then be done in order for the circuit to be implemented. Figures 1.5(b) and 1.5(c) show two logically equivalent decompositions. The new internal signal e must now be checked for potential hazards introduced during decomposition. For the decomposition in 1.5(b), in state (F110), inputs a, b, c and internal signal e are high while the output d is low. After a falls, we move to state (0F10), and e becomes excited to go low. Assume the AND gate generating e is slow. When b falls, we move to state (00F0). If at this point c falls before e falls, d can become excited to rise prematurely. The result is a hazard on the signal d, and there is a potential for circuit failure.

Next, consider the decomposition shown in Figure 1.5(c), beginning in state (F110). This time e begins low, and it does not become excited to change until after a falls, b falls, c falls, and a rises again. At this point, b is already low which maintains d in its low state until b rises again. There is no sequence of transitions that can cause this circuit to experience a hazard on the output d.

This example illustrates the need to take special care during decomposition of SI circuits. It also hints at how explicit timing information may be used to help determine whether a hazard condition actually manifests or not. This topic is explored fully in Chapter 3.

For most of the work published in the SI domain, the implementation is done completely in the decomposition stage, making partitioning and matching/covering unnecessary. The goal of this work is to eliminate *all* hazards through a decomposition and re-synthesis loop.

1.5.3 Fundamental Mode Circuits

Fundamental mode circuits [57, 42, 59] further constrain the inputs by assuming the environment is too slow to respond to new input changes before the previous change has stabilized. This type of circuit acts much like a synchronous state machine where the timing on the input transitions is like the clock in a synchronous system. This allows the designer to apply basic synchronous techniques. An expansion to the fundamental mode model, called burst-mode, allows a burst of inputs to change rather than a single input. A further extension, termed extended burst-mode (XBM), allows for the use of directed don't cares to specify that an input *may or may not occur* in a given burst. Decomposition can be performed by recursively applying De-Morgan's theorem and the associative boolean law to the network. Both operations have been shown to be hazard preserving [57, 51, 52], that is, if the original circuit was hazard-free, than the decomposed circuit is also hazard-free. More recently, technology mapping techniques for burst mode machines were developed in [10]. While these results were significant for the XBM type of design, they do not take into account possible optimization's due to known timing relationships. Figure 1.6 illustrates how the hazard problem encountered in Figure 1.5(b) for SI circuits cannot occur under fundamental mode assumptions.

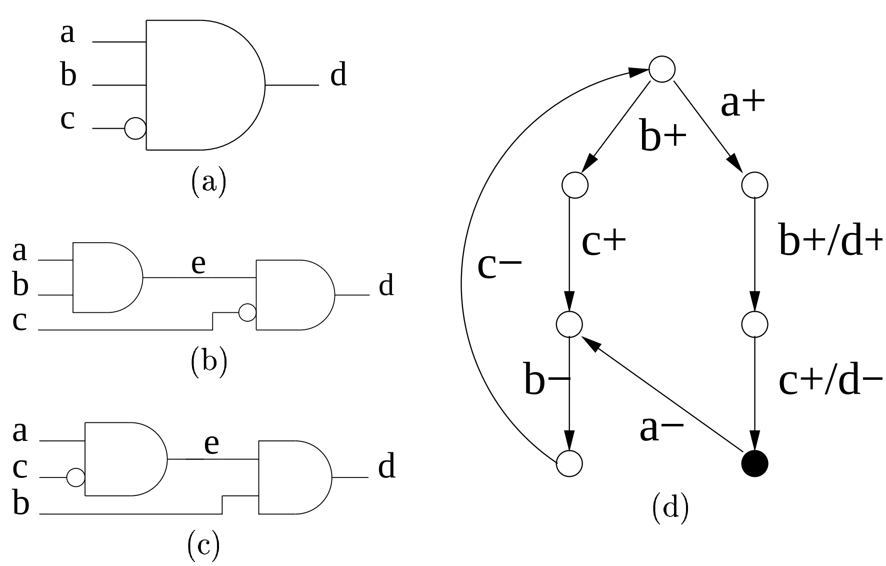

Figure 1.6. (a) 3-input AND function. (b) A circuit netlist that is hazardous under the speed-independent model. (c) A circuit netlist that is hazard-free under the speed-independent model. (d) Petri net for the 3-input AND function.

Figure 1.6(d) shows a burst-mode state machine represented as a Petri net (defined formally in chapter 2) with inputs a, b, c and output d. The output d has again been synthesized with a CAD tool and the results shown in Figure 1.6(a). Starting with the dark circle in Figure 1.6(d), the circuit state is a, b, c, and e all high and d low. At this point, a goes low causing the new internal node e to go low. Then b goes low followed by c going low. The output cannot glitch because e is already low well before c goes low. In fact, there is no sequence of events that can cause a hazard, under fundamental mode assumptions, in either of the decompositions shown in Figures 1.6(b)and 1.6(c). This is a simple example of the hazard-preserving nature of fundamental mode decompositions.

This result should not be surprising; once any input or burst of inputs change, *all* internal signals and outputs must be allowed to evaluate before another burst of inputs can occur. The process for mapping fundamental mode circuits is virtually the same as that for mapping synchronous circuits with the restrictions that decomposition be done using only De-Morgan's theorem and associative laws, and the library elements used during the matching stage be hazard-free. Fundamental mode circuits suffer from the same limitation that synchronous circuits do; the reduced ability to exploit concurrency.

1.5.4 Timed Circuits

Timed circuits add a further restriction by placing two-sided timing constraints on the inputs and gates. There is little published work regarding technology mapping of timed circuits except for that done by Myers et al. in [40] where they used decomposition and re-synthesis to break up high fan-in gC gates.

Considerable work has been done on the synthesis stages of the timed circuit design process [39]. Since exhaustive state space methods quickly become intractable, much effort has gone into developing new algorithms that examine the state space and generate a reduced state graph (RSG) that factors in the initial timing constraints. The RSG is then analyzed to determine a set of valid solutions for the outputs. These solutions are further analyzed, and an optimum solution is chosen based on a set of constraints.

The output of the synthesis stage is a set of Boolean equations that consist of combinational and sequential components. The sequential components contain state-holding elements and the combinational components contain standard logic expressions. These logic expressions have no fanin/fanout constraints and must be decomposed into smaller expressions that can be mapped directly to library cells.

It is often the case that hazard conditions found in SI circuits do not manifest as glitches in the real circuit implementation due to the actual timing behavior. The reason for this is that internal signals once enabled certainly do fire in some finite time. If we can track the time evolution in the state space, then it may be possible to identify the stability of all internal signals. Using this *timed cube approximation*, we can rapidly analyze a gate-level timed circuit for hazards.

The goal of our research is to apply the technology mapping process of Figure 1.2 to timed circuits. We plan to do this by extending the verification work done in [4] to perform efficient verification of our decomposed netlist. Verification will also be done on the final covered netlist to ensure that any timing changes incorporated during the covering process did not introduce new hazards. This new method that we present has the potential to greatly increase the size of circuits that can be technology mapped in an efficient and hazard-free manner.

1.6 Contributions

This dissertation makes five main contributions in the area of technology mapping of timed asynchronous circuits: applying the synchronous technology mapping flow to timed asynchronous circuits, utilizing explicit timing information to assist in the detection and elimination of hazards, driving the design flow to minimize rather than eliminate hazards, using the atomic gate model to encapsulate hazards during covering, and using standard libraries to implement hazard-free timed asynchronous circuits.

The first contribution is to utilize the synchronous approach to technology mapping. This allows us to leverage the existing research that is plentiful and mature and apply it to our design flow for timed asynchronous circuits. The synchronous methods of design decomposition, partitioning, matching, and covering can work in our favor if we can address the issue of hazard elimination. Our approach requires us to use hazard-freedom as our primary cost metric.

The second contribution is the use of explicit timing information to aid the verification process. Not all hazard conditions result in actual glitches due to the finite time that it takes gates to evaluate. The identification and determination of which conditions *do* cause problems is a pivotal piece of the work we are attempting to accomplish.

The third contribution is the development of algorithms that minimize the number of hazards present in the netlist after decomposition rather than eliminate all hazards. Previous work has focused on the removal of *all* hazards prior to mapping the design to a library. Certainly it is easier to map a hazard free netlist to the library but this isn't always practical without numerous iterations or significant computational expense. Our algorithm allows for the existence of hazardous nodes after decomposition with the expectation that the matching/covering algorithm can eliminate the remainder.

The fourth contribution is the development of algorithms that identify the remaining hazardous nodes during the matching step and then eliminates them by encapsulating the hazardous nodes within library elements. Also, in conjunction with the timing algorithms, it may be possible to show that the new netlist timing from the covered circuit has altered the timing of the netlist such that nodes previously determined to be hazardous no longer exhibit hazardous properties.

The fifth and final contribution is the ability to utilize standard libraries to implement the timed circuit. Typical asynchronous approaches often require the use of specialized libraries to do a hazard-free implementation of the design. We are planning to push as far as we can the use of standard libraries to complete a hazard-free implementation. The quality of the final netlist, of course, depends on the available library components, but this is true for other types of asynchronous design as well as for synchronous design.

1.7 Dissertation Overview

Chapter 2 develops the necessary mathematical descriptions used throughout this thesis. Examples are used to help illustrate these descriptions. Formalism is developed to support the design description methods including time Petri nets, state graphs, and netlists.

Chapter 3 presents the bulk of the work focusing on methods to verify hazard freedom in a decomposed netlist. Since the decomposition can result in a large netlist, special attention is given to efficient verification techniques. Each node is checked for hazard behavior independently first using speed independent techniques then using explicit timing information. The netlist is then annotated with any nodes found to be hazardous.

Chapter 4 presents an efficient method for the decomposition of the netlist, particularly as it relates to the synchronous world. Since many of the existing algorithms for asynchronous technology mapping decompose directly to library gates, this chapter follows more of the synchronous approach, discussing enhancements applicable to asynchronous designs and in particular, timed methods. It finishes with a discussion of optimization techniques used to extract a more optimal solution throughout the entire mapping process.

Chapter 5 discusses the methods used to match the decomposed netlist to a general purpose library and then do a covering based upon hazard elimination. The netlist is annotated with information regarding any remaining hazardous properties. A final netlist is then written, containing only elements from the available library. This netlist is re-verified for hazard freedom and iterations are done as needed.

Chapter 6 tabulates and discusses the results from running numerous examples through the supporting software. Particular emphasis is given to the methods used to optimize the final netlists. The results are compared against other technology mapping tools. The cost of using hazard-freedom as the primary cost metric driving the final netlist creation is presented versus using the more traditional metrics of area and/or delay.

Chapter 7 summarizes the results of the work and discusses the successes and limitations of this technology mapping approach. A future work section presents a number of ideas that we think could enhance or extend this work to produce more robust and/or correct circuits.

CHAPTER 2

CIRCUIT SPECIFICATIONS AND SEMANTICS

In this chapter, we expand on the timed asynchronous technology mapping design flow presented in Chapter 1. This expanded design flow is shown in Figure 2.1. We then develop the necessary semantics to support the remainder of this thesis and illustrate with two examples.

Our work begins with the logic equations generated by the synthesis portion of the design flow and ends with a (hopefully) hazard-free netlist of the circuit using only cells available from a standard library.

2.1 Logic Expressions

The logic expressions provided by the synthesis engine are assumed to be of a sum-of-products (SOP) form. Figure 2.2(a) shows the SOP structure that results from synthesis. Although this example shows a limited number of terms, there are an arbitrary number of AND terms and each AND term can have an arbitrary number of literals in the actual implementation. In this form, each output is naturally partitioned into a cone of logic. As a result, there is no partitioning step necessary in this design flow.

In Figure 2.2(a), all AND terms are OR'd together to create the inputs to a C-element. The C-element is a state holding device [41]. When both of it's inputs are '0', its output goes to '0'. Similarly, when both of its inputs are '1' its output goes to '1'. Otherwise, it retains its old value. In boolean form, the logic equation can be written as $c = ab + ac + bc$ where a and b are inputs to the C-element and c is the output. Using the C-element is preferable in asynchronous circuits over an SR latch for state holding circuits because it does not exhibit metastability in any of

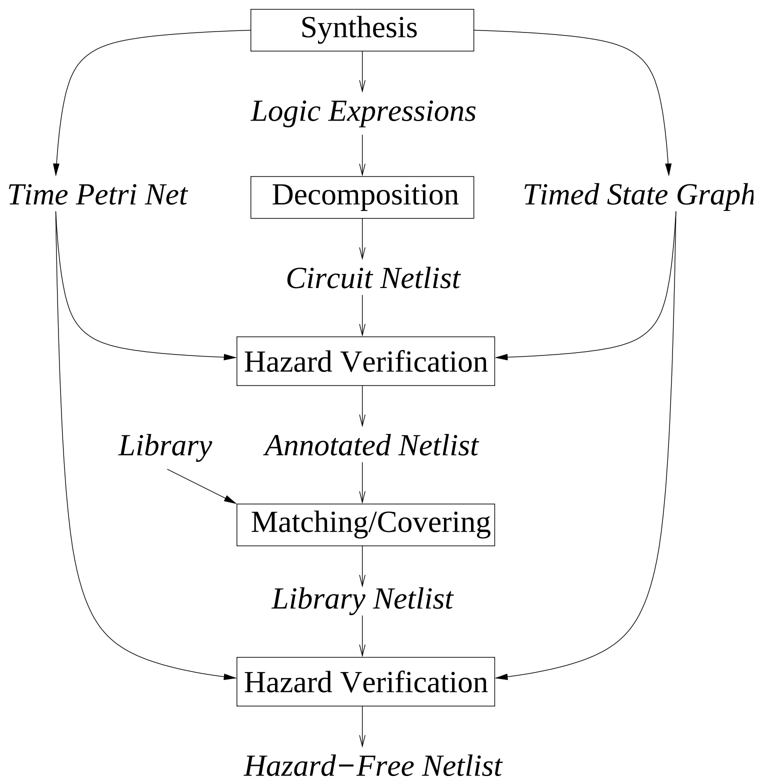

Figure 2.1. Expanded timed asynchronous technology mapping design flow.

its states. The upper sum-of-products represents the logic that sets the output of the C-element high and the lower sum-of-products represents the logic that resets the output. The set and reset cones of logic are naturally de-coupled and can be decomposed separately unless gate sharing is enabled. If there are only set terms present or there are only reset terms present then the output is combinational and the C-element can be removed.

A generalized C-element (gC) is a state holding component that, in addition to containing the C-element, also contains the set and reset logic. Figure 2.2(b) shows a transistor level gC implementation of the logic shown in Figure 2.2(a). This implementation, with weak feedback, is considerably more efficient than a fully static implementation. An attractive feature of gC's is that they reduce the potential for hazards. Static hazards cannot manifest on the output of a gC gate

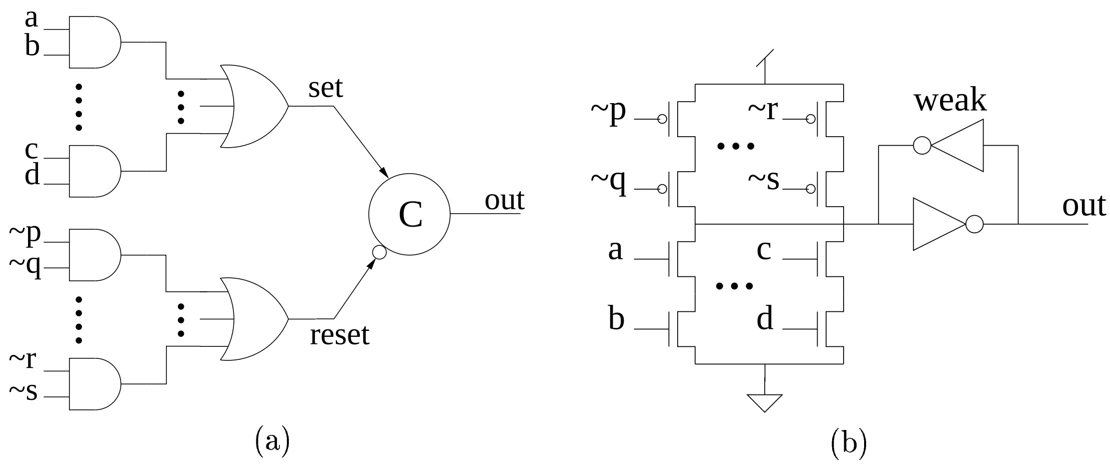

Figure 2.2. (a) POS form of the logic expressions. (b) A CMOS implementation with weak feedback.

by its very structure. However, the synthesis engine must ensure that there is never both a set and reset term active concurrently or a direct short appears in the CMOS stack.

In actuality, the synthesized equations do guarantee that a short circuit will not occur directly in the transistor stack because this feature corresponds directly to avoiding dynamic hazards caused by decomposing an N- or P-stack[41]. However, this is true only if the equations are implemented in a gC element. Once we change the structure of the netlist through decomposition and cover some portions of the network with library cells other than gC elements, we introduce the possibility of small windows of time where a short circuit can occur. When a gC element is chosen from the library as a best match, a dynamic timing analysis must be done to ensure that a short circuit condition is not present.

2.2 Time Petri Nets

As an intermediate form of circuit representation, the synthesis engine creates a *Time Petri Net* (TPN) to model the possible input behaviors and the required output behaviors for timed circuits [37]. Let W be a finite set of wires in a timed circuit. The timed behavior of a circuit is modeled as sequences of rising and falling transitions on W. For any $w \in W$, $w+$ is a rising transition and $w-$ is a falling transition on the wire w. In the following definitions, let $\mathbb{Q}^+$ and $\mathbb{R}^+$ denote the sets of non-negative rational and non-negative real numbers, respectively. A W-labeled one-safe TPN is a directed bipartite digraph described by the tuple $TPN = \langle W, T, P, F, M_0, s_0, l, u, L \rangle$ where:

- $W = I \cup O$ is the set of wires where I is the set of input wires and O is the set of output wires;
- T is the set of transitions;
- P is the set of places;
- $F \subseteq (T \times P) \cup (P \times T)$ is the flow relation;
- $M_0 \subseteq P$ is the initial marking;
- $s_0 \subseteq W$ is the set of wires that are initially high;
- $l : T \to \mathbb{Q}^+$ is the lower timing bound function;
- $u : T \to \mathbb{Q}^+ \cup \{\infty\}$ is the upper timing bound function;
- $L : T \to W$ is the labeling function.

The state of a TPN is a pair $\langle M, D \rangle$ where M is the current marking (i.e., the subset of places that hold tokens) and $D : T \to \mathbb{R}^+$ is a clock assignment function assigning nonnegative real valued ages to transitions. With every transition $t \in T$, its associated *preset* is $\bullet t = \{p \in P \mid (p, t) \in F\}$. The *postset* of a transition is defined as $t\bullet = \{p \in P \mid (t, p) \in F\}$. Note that the preset and postset for places are defined in a similar manner. A transition, t, is *enabled* in a state if the members of its preset form a subset of the places in the marking of the state (i.e., $\bullet t \subseteq M$). A transition, t, is *fireable* in a state if it has been enabled longer than its lower timing bound (i.e., $D(t) \geq l(t)$). A transition, t, *must fire* before it has been enabled longer than its upper timing bound (i.e., $D(t)$ must never exceed $u(t)$).

A TPN for the example presented in Figure 1.5 is shown in Figure 2.3(a). In the initial state, transitions $a+$ and $b+/1$ are enabled, and exactly one of these transitions fires within 2 to 5 time units. If $a+$ fires, then the $b+/2$ transition becomes enabled and fires within 2 to 5 more time units enabling $d+$. The sequence thus continues throughout the TPN.

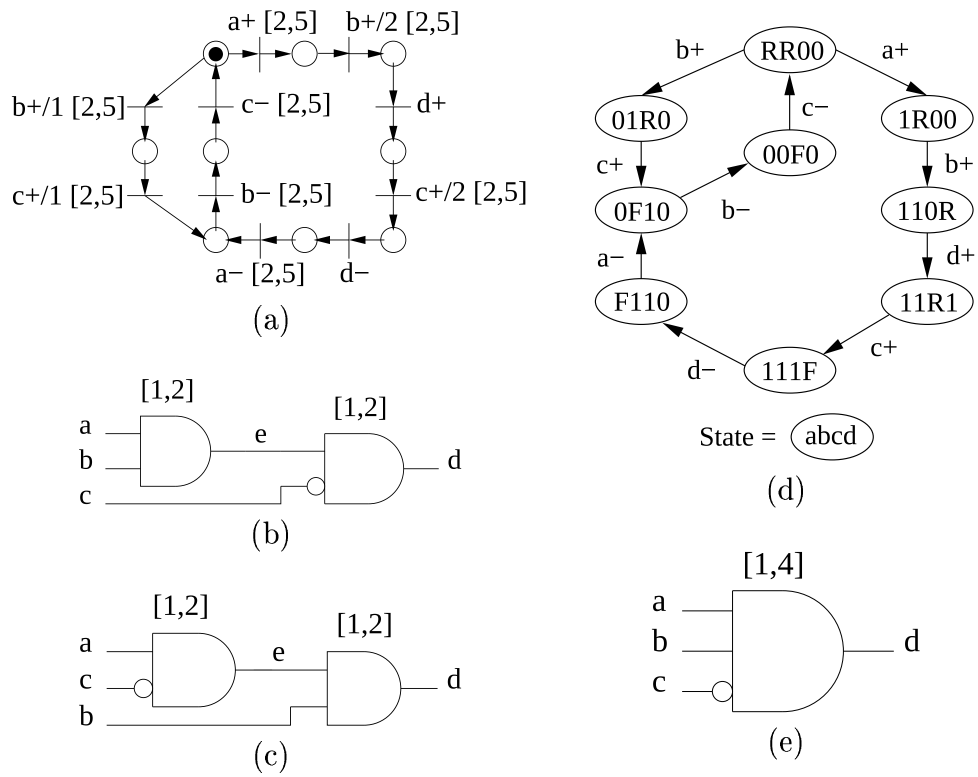

Figure 2.3. (a) An example TPN. (b) A circuit netlist that is hazardous under the speed-independent model. (c) A circuit netlist that is hazard-free under the speed-independent model. (d) A state graph. (e) A complex gate equivalent circuit.

2.3 Netlist

The goal of the verification portion of our work is to verify the correctness of a circuit implementation against a given TPN specification. The circuit to be verified is described using a netlist modeled by a directed graph $NET = \langle V, E \rangle$ where

- $V = I \cup O \cup N$ is the set of vertices in the circuit;
- $E \subseteq (I \cup O \cup N) \times (N \cup O)$ is the edges between vertices.

Each vertex $v \in V$ represents a node in the netlist. This set is composed of both the input wires, I, and output wires, O, from the TPN description as well as new nodes internal to the circuit, N. Each $e \in E$ represents a directed connection in the netlist from one node to another node. The set of *fanins* to a node is denoted by $FI(v)$, and the *fanouts* are denoted by $FO(v)$. Each node which is in $N \cup O$ has an associated gate output function $f_v(v_1, \ldots, v_r)$ where $FI(v) = \{v_1, \ldots, v_r\}$. This gate output function also has an associated minimum, min_v, and maximum, max_v, gate delay.

The netlist for a possible circuit implementation of the signal d in our example is shown in Figure 2.3(b). The set of vertices, V, is $\{a, b, c, d, e\}$, and the set of edges, E, is $\{\ (a, e),\ (b, e),\ (e, d),\ (c, d)\ \}$. The function associated with e is $f_e(a, b) = AND(a, b)$ which has a delay of 1 to 2 time units. An alternative circuit implementation for signal d is shown in Figure 2.3(c).

2.4 State Graphs

In order to check correctness, a verification method typically uses a specification such as a TPN and a representation of the circuit implementation such as a netlist and finds all possible states represented using a state graph. This verification method then checks the state graph (often on the fly as the state graph is being generated) for various correctness properties. We first define the semantics for a state graph and a netlist and then show how the state graph of Figure 2.3(d) can be derived from the TPN of Figure 2.3(a) and the netlist of Figure 2.3(b) or (c).

A SG is a labeled directed graph whose nodes are *states* and edges are *state transitions*. Formally, a SG is modeled by the tuple $SG = \langle\ S,\ \delta\ \rangle$ where

- S is the set of states.
- $\delta \subseteq S \times T \times S$ is the set of state transitions.

Each individual state $s \in S$ is modeled as a tuple $s = \langle \nu, z \rangle$ where

- $\nu \subseteq V$ is the set of wires that are high in the state.
- z is a *zone* representing timing relationships.

Timing information in each state is given using zones which are typically represented using difference bound matrices (DBMs) [15]. These matrices represent time differences between recently fired transitions. Each entry, z_{ij}, in the matrix represents a timing relationship of the form $\tau_{t_i} - \tau_{t_j} \leq z_{ij}$ where τ_{t_i} is the time at which t_i fires. In other words, z_{ij} represents the maximum amount of time in which t_i fires after t_j. An example zone for the point right after $a+$ fires is given below which represents the relationship $2 \leq \tau_{a+} - \tau_{c-} \leq 5$.

$$\begin{array}{c|cc|} & \tau_{c-} & \tau_{a+} \\ \tau_{c-} & 0 & -2 \\ \tau_{a+} & 5 & 0 \end{array}$$

The states are labeled in the state diagram to show the value of all signal wires in the state. Each edge of the state graph is labeled with a signal transition. The input wire set is $\{a, b, c\}$ and the output wire set is $\{d\}$. There are nine states including $RR00$ and $1R00$, and ten state transitions including $(RR00, a+, 1R00)$.

Using a timed state space exploration algorithm such as the ones in [5, 36], it is possible at this point to derive the state graph of Figure 2.3(d) using the TPN to drive the inputs and check the outputs, and a netlist to drive the outputs. This state graph reflects the *complex gate equivalent* version of the netlists of Figure 2.3(b) and (c) and is shown in Figure 2.3(e) where the only signals present are the primary inputs a, b, c and the primary output d. As the complex gate equivalent circuit of Figure 2.3(e) is decomposed, new internal signals are added and a new state graph must be constructed to include these new signals. The state graph for the netlist of Figure 2.3(c) is shown in Figure 2.4. Note the increased number of states. Every new node added to the circuit potentially doubles the number of states in the state graph. In this particular example, the number of states increased from 9 to 12 reflecting the fact that the synthesis engine pruned some unnecessary states from the state graph. This state graph expansion is a function of circuit structure and it's growth can conservatively be estimated as doubling the number of states for

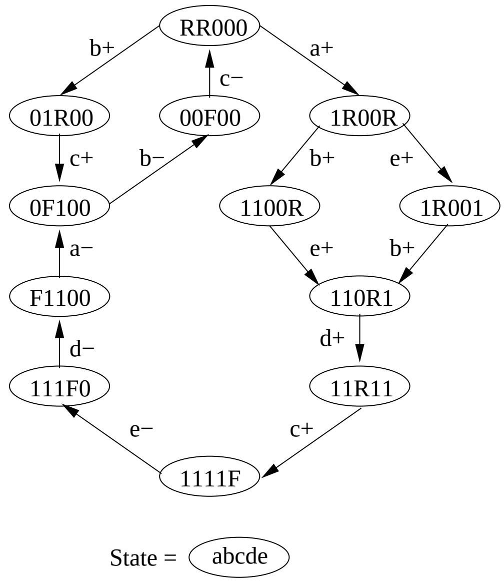

Figure 2.4. State graph for the network of Figure 2.3(c).

each new node added. This growth becomes prohibitively large as a netlist grows in complexity.

The key result of our work is that our method never explicitly derives a new state graph for a decomposed netlist. Instead, we only need to derive a state graph for complex gate equivalent of the network. The details of this procedure are explained in Chapter 3.

2.5 Design Flow Completion

The hazard verification process of Figure 2.1 takes as inputs a *time Petri net* (TPN) defining the circuit and the behavior of the specified environment, a *netlist* representing the decomposed circuit to be verified, and a *state graph* which represents reachable timed states. This process is discussed fully in Chapter 3 where each node in the decomposed netlist is checked for hazard conditions. Each node is then marked as to its hazard properties and this annotated netlist is passed to the matching/covering stage.

The matching/covering stage takes as inputs the annotated netlist from the hazard verification process along with a technology dependent library and creates a new netlist composed exclusively of cells from the library. The matching stage finds the best match at every node from the available library cells. The matcher takes into consideration a primary and secondary cost factor chosen from hazard-freedom, area, or delay. Since a sub-optimized netlist in regards to area and/or delay still functions correctly provided it is hazard-free, our primary cost objective is hazard-freedom. Without a hazard-free circuit, there is no guarantee of correct operation. Thus the matcher makes a determination at each node of the circuit which library cell best matches the circuit at that node and annotates each node with the selected library cell. The matcher takes into account whether or not a hazard exists on the current node being worked on and whether or not it should be encapsulated or left exposed.

There are two types of hazards that we are dealing with in this process. These two hazard types are more formally defined and explained in Chapter 3. For now, the matcher must make every effort to encapsulate *acknowledgment* hazards and leave uncovered *monotonicity* hazards. An acknowledgment hazard occurs when an internal node becomes excited to change to a new value, but its excitation changes value before it can be shown to have stabilized. A monotonicity hazard occurs when an internal or output node is supposed to remain stable but it becomes momentarily excited or it is supposed to make a transition which it makes non-monotonically.

After the circuit is matched, the covering algorithm determines the best cover using library cells and writes a new, reduced netlist composed only of library cells. For two reasons, it is still possible that our reduced netlist may yet contain some hazardous nodes. First, it may not be possible to eliminate all hazardous nodes in the covering. Second, the new covered netlist alters the timing properties and may not verify in a hazard-free manner against the original timing specifications. Although we have not seen results indicating so, it is certainly feasible that the covered circuit may create new hazards due to new timing properties. As a result, we perform another hazard verification on the library netlist.

If this final verification shows no hazards our technology mapping is complete and the library netlist is a hazard-free implementation of the original circuit specification. If not, we must iterate on the design flow until we can produce a hazard-free netlist or prove that one is not possible. This iteration process involves a number of optimizations discussed in more detail in Chapter 4.

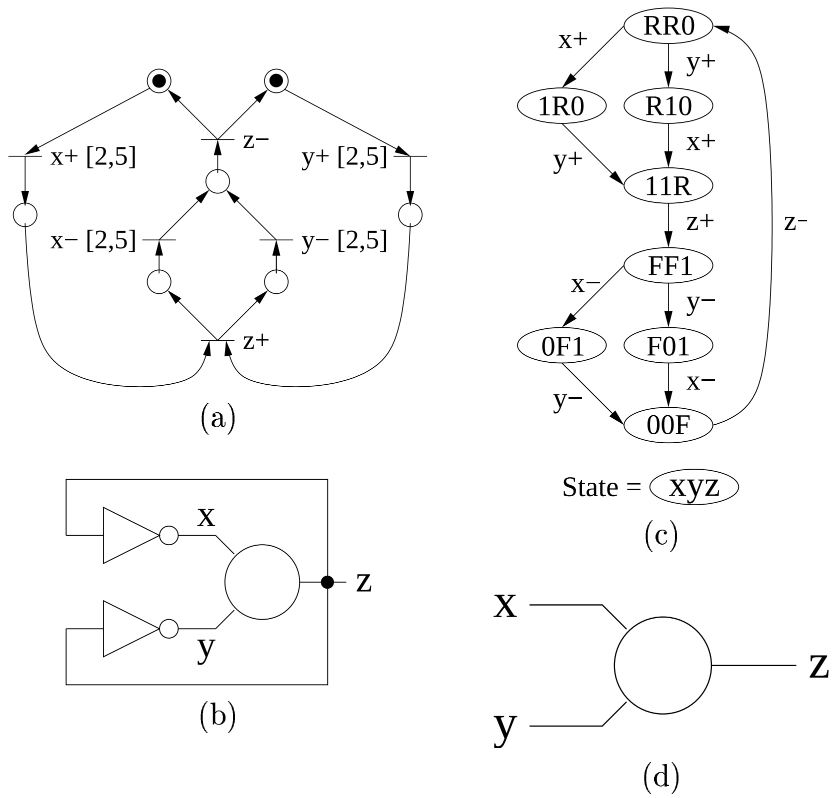

Figure 2.5. (a) A Petri net of a C-element. (b) A netlist of a C-element. (c) A state graph of a C-element. (d) A complex gate equivalent netlist of a C-element.

2.6 Another Example

We conclude this chapter with another example to illustrate the process by which circuit representations are created and how they inter-relate. We begin with a time Petri net and a synthesized netlist, then create a state graph, and finally represent the netlist as a complex gate equivalent circuit.

The example we have chosen is that is that of the basic C-element. An example Petri net for the C-element is shown in Figure 2.5(a). Initially x and y are enabled to change to '1', that is $x+$ and $y+$ are enabled to fire. After both x and y change to '1', z is enabled to change to '1'. Note that which order x and y fire in does not matter. Z cannot fire until both $x+$ and $y+$ have occurred. After z changes to '1', x and y are enabled to change from '1' to '0'. After they have both changed to '0', z can change from a '1' to a '0'. This returns the circuit back to its initial state.

The sequence of events described above can be followed in the schematic netlist shown in Figure 2.5(b). The state graph of Figure 2.5(c) indicates the sequence of events in yet another form where unique state codes have been a assigned to ensure a correct and repeatable sequence of events.

The complex gate equivalent circuit of Figure 2.5 is simply a symbol with inputs x and y on the left and the output z. The behavior of this complex gate equivalent circuit must reflect the behavior specified in the Petri net and also that shown in the state graph.

2.7 Summary

In this chapter, we have laid a foundation using semantics to define time Petri nets, netlists, and state graphs. We have presented a design flow for our work that begins with a timed Petri net and a netlist, derives a state graph, and uses a complex gate equivalent representation during hazard verification. Finally, we have illustrated this work with two examples.

CHAPTER 3

DETERMINATION OF HAZARD-FREEDOM

In this chapter, we present the core material that allows us to do a fast and efficient hazard determination for each node in the decomposed netlist. We began with a formal definition of hazard freedom. We than follow with formal definitions of acknowledgement and monotonicity hazards. We then present a short discussion concerning the generation of the time Petri net and the state graph. While these two circuit representations are assumed to be part of the design information that we began our work with, an understanding of their creation and interrelation is deemed helpful.

At this point, sufficient background exists to began discussion on the core work of this thesis, that being the determination of hazard-freedom at each node in the decomposed netlist. This amoundts to finding stable states using two seperate but related techniques: untimed (speed-independent) stabilization and timed stabilization. Once stable states are found, this stabilization is propagated throughout the state graph. Finally, an examination of the behaviour of each node is made to determine whether or not that particular node is free of acknowledgement and monotonicity hazards. Each node is then annotated with it's hazard properties and this information is passed on to the matching/covering stage.

The verification method described in our work requires that the primary outputs must cut the circuit. In other words, if all primary outputs are removed from the netlist, the netlist would become acyclic. Intuitively, this means there can be no internal cycles in the netlist. Since the goal of this work is to use this verifier as a hazard checker during technology mapping and the technology mapper that we are developing satisfies this restriction, this seems acceptable. However, in the future, we are interested in generalizing this work to the case where there is internal feedback in the netlist.

3.1 Formal Definitions

In [4], the following theorem giving sufficient conditions for correctness of an asynchronous circuit is presented (reworded here to match the semantics of this thesis), and it is later proven in [2].

Theorem 3.1. *(Sufficient conditions for correctness) Let* $NET = \langle V, E \rangle$ *be a circuit implementing the behavior described by* $TPN = \langle\ W, T, P, F, M_0, s_0, l, u, L\ \rangle$*. The* NET *is a correct implementation of the* TPN *if (1) it is complex gate equivalent to the TPN, and (2) it satisfies the acknowledgment and monotonicity properties.*

Our verification algorithm to check these correctness conditions is shown in Figure 3.1. This algorithm takes as input a TPN representing the possible input behavior and the required output behavior and a netlist, NET, representing the circuit to be checked, and it determines if the circuit is correct. When the circuit is not correct, this algorithm reports the locations of the errors that it finds. This algorithm is described in detail in section**??**.

```
verify(TPN,NET) {
  SG = check_equivalence(TPN,NET)
  find_stable_states(TPN,SG,NET)
  check_acknowledgment(SG,NET)
  check_monotonicity(SG,NET)
}
```

Figure 3.1. Top-level algorithm for verification.

3.2 Checking equivalence

The `check_equivalence` function forms a complex gate equivalent netlist, uses this netlist and the given TPN to derive a SG, and checks if the complex gate netlist provides outputs only at specified times.

The first step is to derive a complex gate netlist in which there are no internal signals. In other words, it derives a netlist that has one gate per primary output signal in which the Boolean function is only in terms of the primary inputs and outputs. The delay of this gate is set to the minimum and maximum delay from any input to the complex gate to the primary output. In our example, the complex gate representation for the netlists shown in Figure 2.3(b) and Figure 2.3(c) is shown in Figure 2.3(e). The delay for this gate is set to $[1, 4]$, since in both cases there exists a minimum delay path of 1 time unit and a maximum path of 4 times units.

Using this complex gate netlist and the given TPN, a SG is found using a timed state space exploration algorithm. If, during the course of this state space exploration, an output is fired prematurely such as in the example described earlier when using the circuit shown in Figure 2.3(b), an error is reported to the user. Also, if an output is expected and the circuit does not provide one, an error is reported. In our example, if we use the function $f_d = AND(a, c)$, after $a+$ and $b+$, a $d+$ would be expected, but the circuit would not produce it. This models a progress condition similar to *completeness with respect to specification* [17] and *strong conformance* [21]. When no errors are detected, `check_equivalence` returns a SG.

One last thing to note is that during the state space exploration to derive this SG, our method checks that the given complex gate circuit is equivalent to the desired one. For example, if the complex gate circuit given had been the one in Figure 2.3(b), after $a+$ fires, the netlist could produce a $d+$ when one is not expected in the TPN. This complex gate equivalence failure would then be reported to the user.

```
hazard_check() {
  Check circuit output timings vs. TPN timings();
  Find number of states, edges, and nodes in the netlist;
  Init color and eval vectors();
  Find start state;
```

```
  Stabilize edges and propagate coloring information();
  Check for hazards();
}


Check circuit output timings vs. TPN timings() {
  for each output in netlist {
    for each input in netlist {
      if (exists_path) {
        if ((path_length_min < CGE_min) || (path_length_max > CGE_max)) {
          print timing violation;
        }
      }
    }
  }
}


Init color and eval vectors() {
  for each state s in S {
    for each node in netlist {
      if (bool_eval[s,n] == 1) {
        colorvec[n] = RISING;
        evalvec[n] = 1;
      }
      else {
        colorvec[n] = FALLING;
        evalvec[n] = 0;
      }
    }
```

```
    for each succussor edge e of s {
      colorvec[e] = colorvec[s];
    }
  }
}


Stabilize edges and propagate coloring information() {
  do timed stabilization();
  do untimed stabilization();
}


do timed stabilization() {
  for each node in netlist {
    stabilize node timing wise();
    modified = true;
    while (modified) {
      modified = propagate coloring info();
    }
  }
}


stabilize node timing wise() {
  va_nodeR = clear visited array for RISING region;
  va_nodeF = clear visited array for FALLING region;
  va_path = clear visited array for path;
  find_abs_minmax_times from any input to this node;
  for each edge where eval[s] != eval[s'] for this output {
```

```
update_zone(TPN,NET,z,t_i) {
  z' = extend(z,t_i)
  found_causal = FALSE
  foreach t_j ∈ z' in reverse order
    if (t_j ∈ ••t_i) then
      if (!found_causal) then
        found_causal = TRUE
        if (L(t_i) ∈ I) then
          z'_ij = u(t_i)
        else
          z'_ij = find_max_delay(NET,L(t_i))
      if (L(t_i) ∈ I) then
        z'_ji = (-1) * l(t_i)
      else
        z'_ji = (-1)* find_min_delay(NET,L(t_i))
    else
      z'_ij = ∞
      z'_ji = ∞
  recanonicalize(z')
}
```

```
    for each zone in s {
      z' = update_zone();
      do_timed_recursion();
    }
  }
}
```

```
do untimed stabilization() {
  while (modified_high_level) {
    modified_high_level = false;
    for each node in netlist {
      modified_lower_level = true;
      while (modified_lower_level) {
```

```
do_timed(TPN,SG,NET,n,s,z,t_i,d,visit,path) {
  visit(s) = TRUE
  path(s) = TRUE
  foreach (s,t_j,s') in δ
    z' = update_zone(TPN,NET,z,t_j)
    if (-1*z'_ij > d) then
      if (not visit(s')) then
        stable(s,s',n) = TRUE
    else if (not path(s') &&
            eval(s,n) == eval(s',n)) then
      stable(s,s',n) = FALSE
      do_timed(TPN,SG,NET,n,s',z',t_i,d,visit,path)
  path(s) = FALSE
}
```

```
        modified_lower_level = false;
        clear visited array;
        modified_lower_level = stabilize node untimed wise();
        if (modified_lower_level) {
          modified_high_level = true;
        }
        modified_prop = true;
        while (modified_prop) {
          modified_prop = propagate coloring info();
          if (modified_prop) {
            modified_high_level = true;
          }
        }
      }
    }
  }
}
```

```
propagate coloring info() {
```

```
  clear visited array to FALSE;
  if (color_next_state()) {
    visted array[node] = TRUE;
    propagate();
  }
}


propagate() {
  for each successor of state {
    if (!visted) {
      if (color_next_state()) {
visited = TRUE;
propagate();
      }
    }
  }
}


color_next_state() {

}


Check for hazards() {

}
```

```
Discussion items:

1)  Why do timed before untimed?
2)  Why untimed should be iterated but not timed?
3)  All the iterations scenarios during untimed stabilization.
4)  Discussion of tree structure vs. multiple outputs with reconvergant fanout.
5)  Discussion of RISING and FALLING regions for keeping track of visited
    information during timed stabiliazation.
6)  Discussion of getting the initial zone and it's relationship to
    causal events.
7)  Extending timed stabilization to level-based events.
8)  'unwinding' the zone to find the real causal event.
9)  Detailed discussion ofconcurrency and how it relates to causal
events.
```

3.3 Finding stable states

After the `check_equivalence` step, our method has shown that the circuit is correct at a complex gate level. By hiding the internal signals before finding the state space, we reduce the potential size of the state space from $O(2^{|I|} * 2^{|O|} * 2^{|N|})$ to $O(2^{|I|} * 2^{|O|})$ representing an exponential reduction in the potential size of the state space. When the number of internal signals is large, as is often the case in real designs, this savings can be quite dramatic. However, hazards on internal nodes can still produce incorrect circuit behavior. Therefore, it is now necessary to check that all internal nodes are hazard-free. This is accomplished by determining internal signal behavior implicitly. In particular, our method annotates each state with stability information about each internal signal. The goal of the `find_stable_states` algorithm is to determine in which states and for which state transitions in the complex gate SG that each internal node is stable. This is accomplished by deriving a predicate `stable`(s,n) for each state $s \in S$ and node $n \in N$ and another predicate `stable`(s,s',n) for each state transition $(s, t, s') \in \delta$. This stability information can then be used to determine if there are any hazards in the given netlist.

The algorithm to find the stability information is shown in Figure 3.2. The algorithm begins by first determining the predicate `eval`(s,v) by finding the Boolean evaluation in each state in the SG for each node in the netlist. This can be accomplished by simply fixing the values for each primary input and output in the netlist to the values given in the state and propagating this information through the netlist. From the SG in Figure 2.3(c) and netlist in Figure 2.3(b), `eval`(1100,e) and `eval`(1100,d) are determined to both be 1. For node e, the states in the set {1100, 1101, 1111, 1110} evaluate to 1 while the remaining states evaluate to 0.

```
find_stable_states(TPN,SG,NET) {
  find eval(s,v) for all s ∈ S  and v ∈ V
  initialize stable(s,n) and stable(s,s',n) to FALSE
    for all n ∈ N, s ∈ S, and (s,t,s') ∈ δ
  stabilize_timed(TPN,SG,NET)
  do {
    propagate(SG)
    modified = stabilize_untimed(SG,NET)
  while modified
}
```

Figure 3.2. Algorithm for finding stable states.

The algorithm next initializes the stability predicates to FALSE to indicate that initially we do not know whether the internal signals are stable or changing in each state and during each state transition. The goal of the rest of the algorithm is to determine stability of the internal signals, whenever possible. In the next subsection, we briefly review untimed stabilization which comes from the work in [4]. In the following subsection, we discuss our new contribution which is timed stabilization. The timed stabilization routine does not need to be iterated, so it is executed first. The untimed stabilization routine may require iteration since stabilizations on one node of the network can influence stabilizations on other nodes.

3.3.1 Untimed stabilization

The objective of stabilization is to show that at some points in the SG, the evaluations of some internal node, n, are certain to be stable. The algorithm to determine untimed stability is shown in Figure 3.3. An internal node is considered untimed stable if a change in evaluation on an internal node is acknowledged on a primary output. In other words, for a state transition (s, t, s'), if the transition t could only have occurred if the internal node n is stable at its Boolean evaluation, then we can say that the transition t has acknowledged that the node n is stable.

```
stabilize_untimed(SG,NET) {
  foreach n ∈ N
    foreach (s,t,s') ∈ δ
      if ((exists_path(NET,n,L(t))) &&
          (must_prop(NET,s,n,L(t)))) then
        stable(s,s',n) = TRUE;
}
```

Figure 3.3. Untimed stabilization algorithm.

To determine if an internal node n is acknowledged to be stable by a state transition (s, t, s'), we first must determine if a path exists from n to the output transition under consideration. It must then be determined using the function `must_prop` if the value at n can propagate through any possible path to the output. This is done by ensuring that all functions in the path from n to the output have non-controlling values on the side inputs. Consider the example netlist in Figure 2.3(b) and the state transition (1100,d+,1101). There exists a path between node e and the output d. In state 1100, node e evaluates to 1. This value at e must propagate to the output because d cannot go high until e has gone high. More succinctly, output d switched from low to high as a direct consequence of node e going high and the side input, c being at 0. Therefore, `stable`(1100,1101,e) is set to TRUE. Next, the `propagate` function is used to propagate this stabilization forward in the state graph until a change in evaluation is encountered. In particular, `stable`(1100, 1101, e) implies the following stability conditions are TRUE: `stable`(1101, e), `stable`(1101, 1111,

e), stable(1111, e), stable(1111, 1110, e), stable(1110, e), and stable(1110, 0110, e). The propagation of stability information is halted when it reaches state 0110 because the Boolean evaluation of e in this state changes to 0.

The other transition in the SG that could possibly indicate a stabilization for node e is the state transition (1111, $d-$, 1110). In this case, however, the input c is 1 (a controlling value), prohibiting the propagation of node e to the output d. Thus, no stabilization can be assumed for the falling transition of e. As explained later, this lack of stabilization on the falling transition of e indicates a hazard on node e.

A similar analysis done on the circuit in Figure 2.3(c) shows that the rising transition on node e is acknowledged by $d+$ and the falling transition is acknowledged by $d-$ since b is high (a non-controlling value) when d goes low. Therefore, this circuit is hazard-free under a speed-independent model.

3.3.2 Timed stabilization

When timing information is taken into account, the hazard found for the netlist shown in Figure 2.3(b) may not actually manifest. If this is the case, then node e is hazard-free. This subsection describes our new method to determine stabilization using timing information. Timed stabilization attempts to show further stability in the state graph by calculating the maximum possible time through the network to the node of interest and comparing against the minimum time spent traversing the state graph. When it can be shown that in the worst-case a sufficient amount of time has elapsed, node n can be stabilized.

The algorithm to determine timed stabilization is shown in Figure 3.4. For each node n, the algorithm first measures the longest path delay from any primary input or output to the node n.[1] For our example netlist in Figure 2.3(b), this delay is determined to be 2. Next, the algorithm initializes the `visit` array which is used to let the recursion know when a state has been visited along multiple paths

[1] In order to be safe, the absolute longest path is used and not just the longest path from $L(t_i)$. This must be done because the actual signal that causes n to change evaluation may not be $L(t_i)$ due to differences in path lengths.

when determining stabilization of node n. At this point, the algorithm finds state transitions, (s, t_i, s'), where the Boolean evaluation of n changes.

This indicates locations in the state graph where the node n becomes unstable. The algorithm then takes the zone z associated with state s and updates it to include the transition t_i. The reason this is done rather than taking the zone associated with s' is that t_i may have been pruned from this zone. It is important that t_i is in the zone that we use for timed stabilization as t_i serves as a reference transition as we move forward in the state graph. Finally, the algorithm initializes a `path` array which is used to terminate cycles during the analysis of a path in the SG.

```
stabilize_timed(TPN,SG,NET) {
  foreach (n ∈ N)
    d = find_maximum_delay(NET,n)
    initialize visit(s) for all s ∈ S
    foreach (s,t_i,s') ∈ δ where s=⟨v,z⟩
      if (eval(s,n) ≠ eval(s',n)) then
        z' = update_zone(TPN,NET,z,t_i)
        initialize path(s'') for all s'' ∈ S
        do_timed(TPN,SG,NET,n,s',z',t_i,d,visit,path)
}
```

Figure 3.4. Timed stabilization algorithm.

The `update_zone` algorithm is used to add a new transition to a given zone, and is shown in Figure 3.5. The first step is to extend the zone to include a new row and column for the new transition t_i. Next, it searches the zone starting with the transitions that have been added most recently for transitions that enable t_i (i.e., $t_j \in \bullet \bullet t_i$). The first such transition that it finds is the causal transition for t_i. The upper bound of the firing time for t_i should be set in reference to this transition. The upper bound is either taken from the TPN when t_i is a transition on an input wire or it is taken as the maximum delay in the netlist generating t_i when it is a transition on an output wire. For all transitions that enable t_i, a lower bound must be set between t_j and t_i. For all transitions that do not enable t_i, the

```
update_zone(TPN,NET,z,t_i) {
  z' = extend(z,t_i)
  found_causal = FALSE
  foreach t_j ∈ z' in reverse order
    if (t_j ∈ ••t_i) then
      if (!found_causal) then
        found_causal = TRUE
        if (L(t_i) ∈ I) then
          z'_ij = u(t_i)
        else
          z'_ij = find_max_delay(NET,L(t_i))
      if (L(t_i) ∈ I) then
        z'_ji = (−1) ∗ l(t_i)
      else
        z'_ji = (−1)∗ find_min_delay(NET,L(t_i))
    else
      z'_ij = ∞
      z'_ji = ∞
  recanonicalize(z')
}
```

Figure 3.5. Algorithm to update the zone.

timing relationships are initially set to be unbounded. At this point, the zone is recanonicalized using Floyd's all-pairs shortest path algorithm to tighten any loose inequalities. As an example, the zone found for the state 1110 in our example is shown in Figure 3.6(a). The new zone after adding the transition $a-$ is shown in Figure 3.6(b).

The do_timed algorithm shown in Figure 3.7 is used to recursively explore the SG, attempting to accumulate sufficient time to stabilize a given node n before reaching a termination condition. This algorithm first marks the current state s as visited in the visit and path arrays described earlier. Next, it considers each state transition (s, t_j, s'). First, it adds the transition, t_j, to the zone. Next, it checks the zone to determine if enough time has accumulated from the reference transition t_i to the new transition t_j such that the node of interest n has certainly stabilized. If it has, it must also check that the state s' has not been visited along a different path. It must be the case that the minimum time upon reaching a state

$$\begin{array}{cc} & \tau_{d-} \\ \tau_{d-} & \begin{vmatrix} 0 \end{vmatrix} \end{array}$$

(a)

$$\begin{array}{ccc} & \tau_{d-} & \tau_{a-} \\ \tau_{d-} & 0 & -2 \\ \tau_{a-} & 5 & 0 \end{array}$$

(b)

$$\begin{array}{cccc} & \tau_{d-} & \tau_{a-} & \tau_{b-} \\ \tau_{d-} & 0 & -2 & \infty \\ \tau_{a-} & 5 & 0 & -2 \\ \tau_{b-} & \infty & 5 & 0 \end{array}$$

(c)

$$\begin{array}{cccc} & \tau_{d-} & \tau_{a-} & \tau_{b-} \\ \tau_{d-} & 0 & -2 & -4 \\ \tau_{a-} & 5 & 0 & -2 \\ \tau_{b-} & 10 & 5 & 0 \end{array}$$

(d)

$$\begin{array}{ccccc} & \tau_{d-} & \tau_{a-} & \tau_{b-} & \tau_{c-} \\ \tau_{d-} & 0 & -2 & -4 & \infty \\ \tau_{a-} & 5 & 0 & -2 & \infty \\ \tau_{b-} & 10 & 5 & 0 & -2 \\ \tau_{c-} & \infty & \infty & 5 & 0 \end{array}$$

(e)

$$\begin{array}{ccccc} & \tau_{d-} & \tau_{a-} & \tau_{b-} & \tau_{c-} \\ \tau_{d-} & 0 & -2 & -4 & -6 \\ \tau_{a-} & 5 & 0 & -2 & -4 \\ \tau_{b-} & 10 & 5 & 0 & -2 \\ \tau_{c-} & 15 & 10 & 5 & 0 \end{array}$$

(f)

Figure 3.6. Zone creation and evolution.

along all paths to that state has exceeded the maximum logic delay d. Therefore, if along a different path we encounter this state and did not stabilize, then this state transition cannot stabilize the node n. If the amount of accumulated delay does not exceed the delay d, then the algorithm must determine if it is going to recurse down this state transition. If this state has been seen previously upon this path, the algorithm has encountered a cycle of states and must not recurse. If the Boolean evaluation of the node n has changed, then again the algorithm must not recurse. If this is a new state on this path and the Boolean evaluation is maintained, then the algorithm recursively visits the state s'. Note that this edge may have been found to be stable along a different path, but it is not stable along the path the algorithm is currently working on. Therefore, the algorithm must say this edge is not stable before recursing. Although this algorithm can potentially explore a lot of paths, we have not seen this happen in our experimental results. Typically, the length of the paths explored is very short as the recursion is terminated quickly. If in the future, we find examples where this is not the case, we can limit the path length to improve efficiency at the potential cost of more false negative results.

Let us consider again the example netlist in Figure 2.3(b). A change in evalua-

```
do_timed(TPN,SG,NET,n,s,z,t_i,d,visit,path) {
  visit(s) = TRUE
  path(s) = TRUE
  foreach (s,t_j,s') in δ
    z' = update_zone(TPN,NET,z,t_j)
    if (-1*z'_ij > d) then
      if (not visit(s')) then
        stable(s,s',n) = TRUE
    else if (not path(s') &&
            eval(s,n) == eval(s',n)) then
      stable(s,s',n) = FALSE
      do_timed(TPN,SG,NET,n,s',z',t_i,d,visit,path)
  path(s) = FALSE
}
```

Figure 3.7. Timed stabilization recursion.

tion in node e occurs between states 1110 and 0110. As mentioned previously, the do_timed function is called with the zone shown in Figure 3.6(b). As we traverse the SG, the next transition encountered is $b-$. Since $b-$ fires 2 to 5 time units after $a-$, these entries are entered into the appropriate rows and columns as shown in Figure 3.6(c). The timing of the other non-diagonal entries are set to ∞. The zone is then recanonicalized using Floyd's all-pairs shortest-path algorithm, and the resulting zone is shown in Figure 3.6(d). We are interested in the minimum elapsed time between the last transition entered, $b-$, and the initial transition $a-$ which is 2 in this case. Note that lower bounds appear as negative values in a DBM. Since two time units is insufficient time to say with certainty that node e has stabilized, the algorithm considers recursing on state 0010. Since this state has not yet been explored on this path, and since node e still evaluates to 0 in this state, the algorithm recurses to state 0010. Upon recursion, the algorithm adds transition $c-$ to the zone as shown in Figure 3.6(e) and recanonicalizes to obtain the zone shown in Figure 3.6(f). The new minimum time elapsed from $a-$ till $c-$ is 4 time units. Since this number is larger than the maximum delay of the AND gate (2 time units), the algorithm can mark this edge as stabilized. The propagate function copies this stabilization onto states 0000, 0100, and 1000 and edges $(0000, b+/1, 0100)$,

$(0100, c+/1, 0110)$, $(0000, a+, 1000)$, and $(1000, b+/2, 1100)$. This is significant in that the hazard condition that existed after untimed stabilization cannot manifest because of the timing relationships between the circuit and the SG.

3.4 Checking for hazard-freedom

Hazards can manifest in asynchronous circuits due to violations in the *acknowledgment* or *monotonicity* properties [4]. This section gives an explanation of how our method checks for violations in acknowledgment and monotonicity. This explanation though, is brief, and high-level since it is essentially the same as that described in [4].

An acknowledgment violation occurs when an internal node becomes excited to change to a new value, but its excitation changes value before it can be shown to have stabilized. The algorithm shown in Figure 3.8 uses the stability information found earlier to check for acknowledgment on all excited nodes. The algorithm examines each node n and each state transition (s, t, s') in which n changes Boolean evaluation. If n is not stable before it changes Boolean evaluation, then an acknowledgment hazard is reported. For the netlist shown in Figure 2.3(b) using only untimed stabilization, a hazard is found on node e for the state transition $(1000, b+/2, 1100)$. However, since timed stabilization detects that this edge is stable for node e, this circuit is hazard-free when timing is considered.

```
check_acknowledgment(SG,NET) {
  foreach (n ∈ N)
    foreach (s,t,s') ∈ δ
      if ((eval(s,n) ≠ eval(s',n)) and
        not stable(s,t,s')) then
         report hazard on n for (s,t,s')
}
```

Figure 3.8. Algorithm to check acknowledgment.

A monotonicity violation occurs when an internal or output node is supposed to remain stable but it becomes momentarily excited or it is supposed to make a transition which it makes non-monotonically. This occurs when a gate has a *potential hazard* while there is no stable, *forcing* side-input. For example, a potential hazard exists when the output of an AND gate is supposed to remain stable low or fall, but one input is rising. If a side-input cannot be found that is stable low while the other input is rising, it is possible that the AND gate may momentarily evaluate to 1 causing a glitch on the output of the AND gate. The algorithm to check for monotonicity violations is given in Figure 3.9. The definitions of `potential_hazard` and `forcing` are a bit involved and can be found in [4].

```
check_monotonicity(SG,NET) {
  foreach (n ∈ N)
    foreach (s ∈ S)
      foreach v ∈ FI(n)
        if (potential_hazard(s,n,v)) then
           if (not exist w such that forcing(s,n,w)
              and stable(s,w)) then
              report hazard on n for (s,v)
}
```

Figure 3.9. Algorithm to check monotonicity.

CHAPTER 4

DECOMPOSITION

- Re-order the inputs in the decomposition. Since timing is important and signals arrive at different times, our work has shown that re-ordering the inputs will have an affect on the hazard behavior of both the decomposed netlist and the covered netlist.

- Add inverter pairs. This process involves adding inverter pairs to every wire in the netlist. This optimization increases the options available to the matching/covering step and has been shown to be effective in [23, 14] and also in our own work.

There are several others optimizations that could prove helpful and these are listed in the future work section in Chapter 7.

CHAPTER 5

MATCHING AND COVERING

CHAPTER 6

EXPERIMENTAL RESULTS

The gate-level timing verification method described in this paper has been implemented and tested on numerous examples. Table 6.1 compares our new gate-level timing verification method using standard benchmarks against results for the timed automata tool `KRONOS` [32], a conservative approximation method described in [43], and the `ATACS` explicit state timing verifier [41]. For the `KRONOS` runtimes, an entry with a question mark indicates the amount of time after which the verification ran out of memory. The runtimes for `KRONOS` and Pena's methods are taken from their papers while the runtimes for `ATACS` and our new method are from a 650 MHz Pentium III. For our new method, an entry of n/a indicates that this example has an internal cycle and cannot be analyzed using our new method. For the smaller examples, our method has comparable and usually better runtimes than the other methods. However, for larger examples such as trimos-send, our method is more than two orders of magnitude faster than `KRONOS`, twenty-five times faster than Pena's tool, and nearly three times faster than the explicit state method in `ATACS`.

Since our goal is to determine which gates have hazards on their outputs, we configured the explicit method in `ATACS` to continue after finding one hazard and identify all hazards. It should be noted that `KRONOS` did not check for hazards, but it instead was only checking conformance while Pena's tool halted after a hazard is found. The last column of the table indicates the number of gates that have hazards found by the explicit state method and our new method. Despite being a conservative approximation, our method found the exact number of hazards in most cases. However, in three examples, *rpdft*, *sbuf-ram-write*, and *sbuf-send-pkt2*,

Table 6.1. Comparison of standard benchmarks against other timing verification tools.

Example	Gates	KRONOS CPU Time	PENA CPU Time	ATACS CPU Time	New Method CPU Time	Hazards
alloc-outbound	11	0.09	3	0.37	0.14	0/0
chu133	9	0.63	1	0.14	0.11	1/1
converta	12	0.19	12	0.26	0.09	2/2
dff	6	0.19	3	0.18	n/a	3/?
ebergen	9	0.14	1	0.18	0.11	3/3
half	7	0.41	1	0.11	0.09	1/1
mp-forward-pkt	10	0.24	5	0.19	0.14	0/0
nowick	10	0.05	3	0.22	0.11	0/0
rcv-setup	6	0.22	1	0.22	0.10	0/0
rpdft	8	2.93	2	0.35	0.11	1/2
sbuf-ram-write	17	31.77	415	0.40	0.18	1/2
sbuf-read-ctl	10	0.13	2	0.17	0.12	0/0
sbuf-send-ctl	13	54	0.49	0.87	0.14	1/1
sbuf-send-pkt2	13	0.07	103	0.61	0.16	0/1
vme	12	0.39	30	0.49	n/a	1/?
mr1	16	607.43	317	0.31	n/a	0/?
tsend-bm	12	589.56	46	5.83	n/a	1/?
mmu	22	595.09?	480	0.66	n/a	0/?
mr0	20	593.24?	48	0.62	n/a	0/?
ram-read-sbuf	17	678.48?	550	0.41	0.18	0/0
trimos-send	24	580.33?	127	13.75	5.06	5/5

our new method found one additional false hazard.

The key advantage of our new method is its ability to be able to efficiently verify circuits with a large number of internal signals. In order to demonstrate this, we selected a few of our benchmark circuits derived from a variety of sources, and we derived gate-level circuits for them that use only 2-input NAND gates and inverters. Our results are shown in Table 6.2. In all the examples, our method is still able to check for hazards in less than a second while for the largest examples the explicit state method cannot complete.

Table 6.2. Comparison for decomposed netlists.

Example	Gates	ATACS CPU Time	New Method CPU Time	Hazards
slatch	13	3.28	0.16	0/0
lapbsv	14	0.89	0.12	0/0
scsiSV	14	0.96	0.12	0/0
elatch	17	88.0	0.23	0/0
cnt3	32	40.9	0.17	6/6
srgate	36	6.65	0.26	0/0
selopt	66	>1000	0.70	?/21
cnt11	86	>2000	0.99	?/26

CHAPTER 7

CONCLUSIONS

This paper presents a new method for efficiently checking hazard-freedom in gate-level timed circuits. This method uses a cube approximation of the internal signal behavior in order to avoid generating an explicit state graph representing the switching behavior of the internal signals. Our experimental results show that this new method can be substantially faster than previous gate-level timing verification tools. While this method is conservative and thus can report some incorrect hazards, the number of such false negative results appears to be small. This method has been shown to scale very well in that it can verify examples with more than 80 gates in less than a second while previous methods fail to complete.

The long term goal of this work is to use this hazard analyzer to guide technology mapping of timed circuits. In asynchronous circuits, hazards must be avoided and care must be taken during technology mapping to not introduce hazards in the design. Therefore, an asynchronous technology mapper requires a method to rapidly determine when a transformation of the netlist has introduced a hazard. The hazard analyzer described in this paper addresses this need making efficient technology mapping of timed circuits possible.

7.1 Future Work

What about the idea of allowing wome hazards to exist in the synthesized equations before I even start decomposition? Would this indicate a bad synthesis or allow more flexibility to the whole process?

- Identify context and trigger signals for the cover under consideration and place these signals wisely in the decomposition [40].

- Examine the nature of transitions involved and use the work in [52] to determine a potentially hazard-free decomposition.
- Investigate state signal insertion methods described in [41] to create a hazard-free decomposition.
- Add wire acknowledgment forks described in [52] where necessary.
- Further investigate the work of Kondratyev in [28] and apply his methods, if possible, to timed circuits.

REFERENCES

[1] P. Beerel and T.-Y. Meng. Automatic gate-level synthesis of speed-independent circuits. In *Proc. International Conf. Computer-Aided Design (ICCAD)*, pages 581–587. IEEE Computer Society Press, Nov. 1992.

[2] P. A. Beerel, J. R. Burch, and T. H.-Y. Meng. Checking combinational equivalence of speed-independent circuits. *Formal Methods in System Design*, Mar. 1998.

[3] P. A. Beerel and T. H.-Y. Meng. Logic transformations and observability don't cares in speed-independent circuits. In *Proceedings of TAU 1993*, Sept. 1993. Participant's proceedings.

[4] P. A. Beerel, T. H.-Y. Meng, and J. Burch. Efficient verification of determinate speed-independent circuits. In *Proc. International Conf. Computer-Aided Design (ICCAD)*, pages 261–267. IEEE Computer Society Press, Nov. 1993.

[5] W. Belluomini and C. J. Myers. Timed state space exploration using posets. *IEEE Transactions on Computer-Aided Design*, 19(5):501–520, May 2000.

[6] R. K. Brayton, R. Camposano, G. De Micheli, R. H. J. M. Otten, and J. van Eijndhoven. The yorktown silicon compiler system. In D. Gajski, editor, *Silicon Compilation*. Addison Wesley, 1988.

[7] R. K. Brayton, R. Rudell, Sangiovanni-Vincentelli, and A. R. Eang. Mis: A multiple-level interactive logic optimization system. *IEEE Transactions on Computer-Aided Design*, 6(6), Nov. 1987.

[8] J. R. Burch and D. F. Long. Efficient boolean function matching. In *Proc. International Conf. Computer-Aided Design (ICCAD)*, pages 408–411. IEEE, Nov. 1992.

[9] S. M. Burns. General condition for the decomposition of state holding elements. In *Proc. International Symposium on Advanced Research in Asynchronous Circuits and Systems*. IEEE Computer Society Press, Mar. 1996.

[10] W.-C. Chou, P. A. Beerel, and K. Y. Yun. Average-case technology mapping of asynchronous burst-mode circuits. *IEEE Transactions on Computer-Aided Design*, 18(10):1418–1434, Oct. 1999.

[11] B. Coates, A. Davis, and K. Stevens. The Post Office experience: Designing a large asynchronous chip. *Integration, the VLSI journal*, 15(3):341–366, Oct. 1993.

[12] J. Cortadella, M. Kishinevsky, A. Kondratyev, L. Lavagno, E. Pastor, and A. Yakovlev. Decomposition and technology mapping of speed-independent circuits using Boolean relations. *IEEE Transactions on Computer-Aided Design*, 18(9), Sept. 1999.

[13] J. Darringer, D. Brand, W. Joyner, and L. Trevillyan. LSS: A system for production logic synthesis. *IBM J. Res. Develop.*, Sept. 1984.

[14] G. De Micheli. *Synthesis and Optimization of Digital Circuits.* McGraw-Hill, Inc., New York, New York, 1994.

[15] D. L. Dill. Timing assumptions and verification of finite-state concurrent systems. In *Proceedings of the Workshop on Automatic Verification Methods for Finite-State Systems*, 1989.

[16] D. L. Dill. *Trace Theory for Automatic Hierarchical Verification of Speed-Independent Circuits.* ACM Distinguished Dissertations. MIT Press, 1989.

[17] J. Ebergen and S. Gingras. A verifier for network decompositions of command-based specifications. In *Proc. Hawaii International Conf. System Sciences*, volume I. IEEE Computer Society Press, Jan. 1993.

[18] J. C. Ebergen. *Translating Programs into Delay-Insensitive Circuits*, volume 56 of *CWI Tract.* Centre for Mathematics and Computer Science, 1989.

[19] S. B. Furber, P. Day, J. D. Garside, N. C. Paver, and J. V. Woods. A micropipelined ARM. In *VLSI '93*, 1993.

[20] M. Garey and D. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness.* W.H. Freeman and Company, New York, 1979.

[21] G. Gopalakrishnan, E. Brunvand, N. Michell, and S. Nowick. A correctness criterion for asynchronous circuit validation and optimization. *IEEE Transactions on Computer-Aided Design*, 13(11):1309–1318, Nov. 1994.

[22] D. Gregory, K. Bartlett, A. de Geus, and G. Hachtel. SOCRATES: A system for automatically synthesizing and optimizing combinational logic. In *23rd Design Automation Conference*, pages 79–85. IEEE/ACM, 1986.

[23] G. D. Hachtel and F. Somenzi. *Logic Synthesis and Verification Algorithms.* Kluwer Academic Publishers, Boston, 1996.

[24] K. Keutzer. DAGON: Technology binding and local optimization by DAG matching. In *24th Design Automation Conference*, pages 341–347. IEEE/ACM, 1987.

[25] I. Kimura. Extensions of asynchronous circuits and the delay problem I: Good extensions and the delay problem of the first kind. *Journal of Computer and System Sciences*, 2(3):251–287, Oct. 1968.

[26] I. Kimura. Extensions of asynchronous circuits and the delay problem II: Spike-free extensions and the delay problem of the second kind. *Journal of Computer and System Sciences*, 5(2):129–162, Apr. 1971.

[27] M. Kishinevsky, A. Kondratyev, A. Taubin, and V. Varshavsky. Analysis and identification of speed-independent circuits on an event model. *Formal Methods in System Design*, 4(1):33–75, 1994.

[28] A. Kondratyev, J. Cortadella, M. Kishinevsky, L. Lavagno, and A. Yakovlev. Logic decomposition of speed-independent circuits. *Proceedings of the IEEE*, 87(2):347–362, Feb. 1999.

[29] L. Lavagno. *Synthesis and Testing of Bounded Wire Delay Asynchronous Circuits from Signal Transition Graphs*. PhD thesis, University of California, Berkeley, Nov. 1992.

[30] L. Lavagno, K. Keutzer, and A. Sangiovanni-Vincentelli. Synthesis of hazard-free asynchronous circuits with bounded wire delays. *IEEE Transactions on Computer-Aided Design*, 14(1):61–86, Jan. 1995.

[31] L. Lavagno, N. Shenoy, and A. Sangiovanni-Vincentelli. Linear programming for hazard elimination in asynchronous circuits. *Journal of VLSI Signal Processing*, 7(1/2):137–160, Feb. 1994.

[32] O. M. M. Bozga, H. Jianmin and S. Yovine. Verification of asynchronous circuits using timed automata. In O. M. Eugene Asarin and S. Yovine, editors, *Electronic Notes in Theoretical Computer Science*, volume 65. Elsevier Science Publishers, 2002.

[33] F. Mailhot. *Technology Mapping for VLSI Circuits Exploiting Boolean Properties and Operations*. PhD thesis, Stanford University, Dec. 1991.

[34] F. Mailhot. Algorithms for technology mapping based on binary decision diagrams and on boolean operations. 12(5):599–620, May 1993.

[35] A. Marshall, B. Coates, and P. Siegel. Designing an asynchronous communications chip. *IEEE Design & Test of Computers*, 11(2):8–21, 1994.

[36] E. Mercer, C. Myers, and T. Yoneda. Improved poset timing analysis in timed petri nets. In *The Tenth Workshop on Synthesis and System Integration of MIxed Technologies (SASIMI 2001)*, October 2001.

[37] P. Merlin and D. J. Faber. Recoverability of communication protocols. *IEEE Trans. on Communication*, COM-24(9):1036–1043, 1976.

[38] D. E. Muller and W. S. Bartky. A theory of asynchronous circuits. In *Proceedings of an International Symposium on the Theory of Switching*, pages 204–243. Harvard University Press, Apr. 1959.

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

[40] C. J. Myers, P. A. Beerel, and T. H.-Y. Meng. Technology mapping of timed circuits. In *Asynchronous Design Methodologies*, IFIP Transactions, pages 138–147. Elsevier Science Publishers, May 1995.

[41] C. J. Myers, W. Belluomini, K. Killpack, E. Mercer, E. Peskin, and H. Zheng. Timed circuits: A new paradigm for high-speed design. In *Proc. of Asia and South Pacific Design Automation Conference*, pages 335–340, Feb. 2001.

[42] S. M. Nowick and D. L. Dill. Exact two-level minimization of hazard-free logic with multiple-input changes. *IEEE Transactions on Computer-Aided Design*, 14(8):986–997, Aug. 1995.

[43] M. A. Peña, J. Cortadella, A. Kondratyev, and E. Pastor. Formal verification of safety properties in timed circuits. In *Proc. International Symposium on Advanced Research in Asynchronous Circuits and Systems*, pages 2–11. IEEE Computer Society Press, Apr. 2000.

[44] O. Roig, J. Cortadella, and E. Pastor. Hierarchical gate-level verification of speed-independent circuits. In *Asynchronous Design Methodologies*, pages 129–137. IEEE Computer Society Press, May 1995.

[45] O. Roig, J. Cortadella, and E. Pastor. Verification of asynchronous circuits by BDD-based model checking of Petri nets. In *16th International Conference on the Application and Theory of Petri Nets*, volume 815 of *Lecture Notes in Computer Science*, pages 374–391, June 1995.

[46] T. G. Rokicki and C. J. Myers. Automatic verificaton of timed circuits. In *International Conference on Computer-Aided Verification*, pages 468–480. Springer-Verlag, 1994.

[47] F. U. Rosenberger, C. E. Molnar, T. J. Chaney, and T.-P. Fang. Q-modules: Internally clocked delay-insensitive modules. *IEEE Transactions on Computers*, C-37(9):1005–1018, Sept. 1988.

[48] R. Rudell. *Logic Synthesis for VLSI Design.* PhD thesis, U. C. Berkeley, Apr. 1989.

[49] E. Sentovich, K. Singh, L. Lavagno, C. Moon, R. Murgai, A. Saldanha, H. Savoj, P. Stephan, R. Brayton, and A. Sangiovanni-Vincentelli. SIS: A system for sequential circuit synthesis. Technical Report UCB/ERL M92/41, University of California, Berkeley, May 1992.

[50] P. Siegel and G. D. Micheli. Decomposition methods for library binding of speed-independent asynchronous designs. In *Proc. International Conf. Computer-Aided Design (ICCAD)*, pages 558–565, Nov. 1994.

[51] P. Siegel, G. D. Micheli, and D. Dill. Automatic technology mapping for generalized fundamental-mode asynchronous designs. In *Proc. ACM/IEEE Design Automation Conference*, pages 61–67, June 1993.

[52] P. S. K. Siegel. *Automatic Technology Mapping for Asynchronous Designs.* PhD thesis, Stanford University, Feb. 1995.

[53] K. S. Stevens, S. Rotem, R. Ginosar, P. Beerel, C. J. Myers, K. Y. Yun, R. Koi, C. Dike, and M. Roncken. An asynchronous instruction length decoder. *IEEE Journal of Solid-State Circuits*, 36(2):217–228, Feb. 2001.

[54] D. Subcommittee. Ieee standard vhdl language reference manual. Technical report, University of California, Davis.

[55] D. Thomas and P.R.Moorby. *Verilog Hardware Description Language.* Kluwer, Dordrecht, Netherlands, 1991.

[56] J. A. Tierno, A. J. Martin, D. Borkovic, and T. K. Lee. A 100-MIPS GaAs asynchronous microprocessor. *IEEE Design & Test of Computers*, 11(2):43–49, 1994.

[57] S. H. Unger. *Asynchronous Sequential Switching Circuits.* Wiley-Interscience, John Wiley & Sons, Inc., New York, 1969.

[58] C. K. van Berkel, R. Burgess, J. Kessels, A. Peeters, M. Roncken, and F. Saeijs. A fully-asynchronous low-power error corrector for the digital compact cassette player. In *IEEE International Solid-State Circuits Conference*, 1994.

[59] K. Y. Yun, D. L. Dill, and S. M. Nowick. Synthesis of 3D asynchronous state machines. In *Proc. International Conf. Computer Design (ICCD)*, pages 346–350. IEEE Computer Society Press, Oct. 1992.