SPECIFICATION, SYNTHESIS AND VERIFICATION OF HAZARD-FREE ASYNCHRONOUS CIRCUITS

by

Cho W. Moon, Paul R. Stephan, and Robert K. Brayton

Memorandum No. UCB/ERL M91/67

5 August 1991

SPECIFICATION, SYNTHESIS AND VERIFICATION OF HAZARD-FREE ASYNCHRONOUS CIRCUITS

by

Cho W. Moon, Paul R. Stephan, and Robert K. Brayton

Memorandum No. UCB/ERL M91/67

5 August 1991

ELECTRONICS RESEARCH LABORATORY

College of Engineering
University of California, Berkeley
94720

SPECIFICATION, SYNTHESIS AND VERIFICATION OF HAZARD-FREE ASYNCHRONOUS CIRCUITS

by

Cho W. Moon, Paul R. Stephan, and Robert K. Brayton

Memorandum No. UCB/ERL M91/67

5 August 1991

ELECTRONICS RESEARCH LABORATORY

College of Engineering
University of California, Berkeley
94720

Specification, Synthesis and Verification of Hazard-free Asynchronous Circuits

Cho W. Moon, Paul R. Stephan and Robert K. Brayton

Department of Electrical Engineering and Computer Sciences
University of California
Berkeley, California 94720

Abstract

We propose some syntactic and semantic extensions to a graphical specification called the signal transition graph (STG). Our extensions allow for a more natural and compact specification of asynchronous behavior. We show that syntactic constraints on STGs are not sufficient to guarantee a hazard-free implementation, and present techniques to synthesize hazard-free circuits under both single input change and multiple input change conditions.

We also show that behavior containment test using the event coordination model [13] is a powerful tool for the formal verification of asynchronous circuits. Our implementations have been verified to be speed-independent under fundamental mode using this behavior containment test.

1 Introduction

Despite many advantages of asynchronous circuits [23] [17] [11], they have not been widely used except in the restricted domain of interface circuits, where the use of a global clock is impossible. Even in this restricted domain, asynchronous design has been considered time-consuming and difficult because of the lack of a good formal specification and the lack of good synthesis and verification tools. Formal specification of asynchronous circuits is difficult because behavior such as concurrency, sequencing, conflict, timing constraints and data-dependency is difficult to specify in a way that is both natural for designers and easy for formal analysis or verification tools. The synthesis and verification of asynchronous circuits is difficult because of the presence of *hazards*. All hazards must be eliminated because they can cause a circuit to malfunction by, for example, signaling an erroneous initiation or completion event. Hazards complicate synthesis and verification by forcing designers to consider not only static but also dynamic behavior. This can be very complicated in the absence of controlled storage elements and even more so in the presence of gate delay variations.

In this paper, we consider the specification, synthesis and verification problems of asynchronous circuits. The highlights of this paper are

- extensions of a method of graphical specification called STGs which increase expressiveness and extend their applicability (we call these extended STGs simply STGs and the old STGs STG/NCs in this paper)
- techniques to synthesize a hazard-free asynchronous circuit from an STG under different operating conditions (we consider both single signal change and multiple signal change conditions)
- techniques to formally prove that the synthesized circuit is a hazard-free implementation of the specification using the unbounded gate delay model under fundamental mode

The synthesized circuit is guaranteed to be *speed-independent*; it is hazard-free under all possible gate delay variations, assuming that gates have arbitrary delays but wires have no delays and that the circuit operates in fundamental mode. We know of no method which can produce from even an STG/NC a *delay-insensitive* circuit, which is hazard-free under both gate and wire delay variations.

This paper is organized as follows. Section 2 discusses some previous work on asynchronous design and defines the terms used in this paper. Section 3 describes some extensions to STGs. Section 4 describes the hazard-free synthesis techniques. Section 5 describes how we perform design and implementation verification. Section 6 concludes this paper and outlines future work.

2 Preliminaries

2.1 Previous Work

The earliest work on asynchronous design used FSMs with no intervening storage element in the feedback paths [10]. Hazards were removed by performing careful state assignment[25] and/or by adjusting the feedback delays[26]. Although the FSM model is quite general, it is not an ideal form of specification for control-intensive asynchronous circuits, which tend to have a lot of concurrency. A FSM, by definition, needs to be in a single state at all times, and this property makes it difficult to explicitly specify concurrency. Also, it is difficult to reason with FSMs about functional properties such as the latency of the circuit at the specification level.

Another design method for asynchronous circuits relies on rules[16][2]. The behavior is specified with some program or waveform diagram, which is later transformed into a netlist of primitive elements by applying a set of transformations. Although some good results have been obtained with the rule-based approach[15], the scope of optimization tends to be local and it is not clear if rules can be used to eliminate all the hazards under different operating conditions.

In [3], Chu described a graph-theoretic approach to asynchronous design using signal transition graphs (STG). The STG is based on a type of Petri net called the *free-choice net*, which is expressive enough for specifying concurrency and conflict but yet simple enough for analysis [8]. Concurrency can be easily specified with STGs, and its natural timing-diagram-like specification makes it easy to analyze the I/O behavior. Most of the early work on STGs [27][17] focused on syntactic constraints on STGs because a hazard-free circuit was believed to be produced if such constraints were satisfied[3]. In this paper, we show that such constraints do not guarantee hazard-freeness at the gate level, and present techniques to produce hazard-free circuits under single input change and multiple input change conditions using the unbounded gate delay model. We verify that the synthesized circuits are free of hazards by using the event coordination model [13].

2.2 Definitions

2.2.1 Signal Transition Graph

In [3], Chu defined the STG as an interpreted free-choice Petri net suitable for specifying self-timed control circuits. The most general form of STG is the STG/NC, or STG with noninput-choice places. Although STG/NC can be used to specify concurrency, sequencing and choice, there is some behavior which is either impossible or very awkward to specify with STG/NC. To remedy the problems that we have encountered, we propose the following definition of an STG which extends the STG/NC in several ways.

Since an STG is an interpreted Petri net where the net transitions are interpreted as signal transitions, we first define signal types and transitions, and then Petri nets before giving the definition of an STG.

Definition 1 *Let S be a set of signals for a sequential control module. Then S can be partitioned into three sets, input signals S_I, output signals S_O, and internal signals S_N. Output signals are those whose behavior is observable outside the module while internal signals are only for maintaining internal state. The union of output and internal signals, called noninput signals or S_{NI}, are the signals which need to be synthesized. Transitions of a signal t are described by $t \times \{+, -, \sim, \#\}$, where $t+$ means t rises from low to high, $t-$ means t falls from high to low, $t\sim$ means t toggles to the opposite of its current value, and $t\#$ means t may or may not change value.*

Definition 2 *A Petri net is a four-tuple $N = (P, T, F, M_0)$ where P, T and F form a bipartite graph, and M_0 is called the initial marking. In the graph, the vertices P are called places, vertices T are the net transitions, and the directed edges F give the flow relation $F \in (P \times T) \cup (T \times P)$. A marking of the net is an assignment of nonnegative integers to each place $p \in P$, and M_0 defines the initial marking or state of the net.*

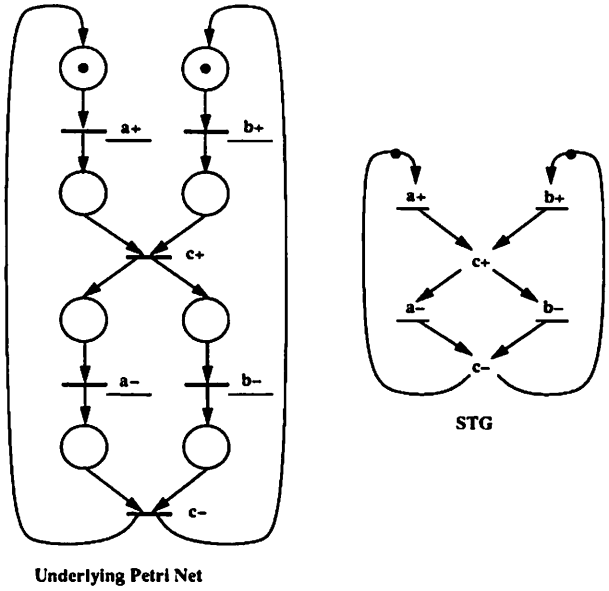

Figure 1: Free-choice Net and STG

A transition t is called the *fanin transition* of a place p if $(t, p) \in F$, and the set of all fanin transitions of p is denoted by $\cdot p$. Fanout transitions, fanin places and fanout places are similarly defined. A good review of other basic Petri net concepts is given in [20].

The complete lack of restrictions on Petri nets makes them difficult to analyze so STGs are based on a subset called free-choice (FC) nets. FC nets allow both concurrency and choice to be specified but are easier to analyze [8].

Definition 3 *Free-choice nets are a subset of Petri nets with the condition:*

$$\forall p \in P :\mid p\cdot \mid > 1 \wedge (p,t) \in F \Rightarrow \mid \cdot t \mid = 1$$

In other words if a place has more than one output transition then it is the unique input place for its fanout transitions.

Definition 4 *A place with more than one fanout transition is called a* **free-choice** *(or* **input-choice***) place if all the fanout transitions belong to S_I. The* **choice** *as to which transition will be enabled is made by the environment. Similarly, in [3] Chu called a place with more than one fanout transitions (including the dummy transition ε) all belonging to S_{NI} a* **controlled-choice** *(or* **noninput-choice***) place, and labeled each fanout edge of a controlled-choice place with a single Boolean literal. Thus, the fanout transition corresponding to the edge label whose Boolean literal evaluates to true is enabled. Although many self-timed circuits can be described with the above constraints on control-choice places, we have found that they are too restrictive. Our extensions allow controlled-choice places to have fanout transitions of input signals and allow arbitrary Boolean predicates on S as edge labels.*

Definition 5 *An* STG *on a set of signals S is a free-choice net N with the net transitions T labeled as signal transitions $S \times \{+, -, \sim, \#\}$ or the null transition ε. The initial marking M_0 of an* STG *can only assign a 0 or 1 token to each place.*

The graphical notation for STGs is illustrated in Fig. 1 along with the underlying Petri net for this STG. Places with only one fanin and fanout transition are omitted, and tokens on these places are drawn directly as dots on the STG arcs. Input signal transitions are underlined.

In this paper, the term STG/NC refer to Chu's version and STG means the definition above. The most noticeable enhancement is the addition of the two new signal transition labels, ~ and #. These will be explained in detail in section 3. Section 3 also gives new analysis techniques to check the consistency of the extended STG.

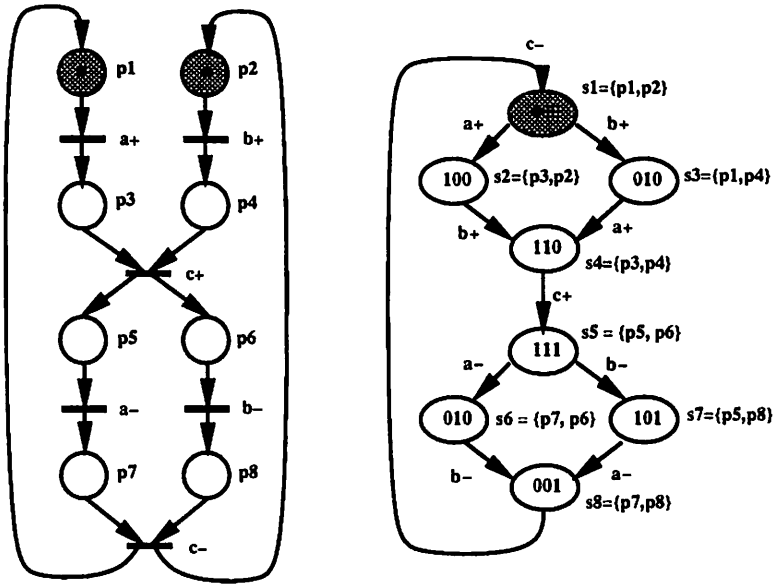

Figure 2: STG and State Graph

2.2.2 State Graph

The state graph (SG) is a finite automaton obtained by "executing" the STG. An STG is executed by examining all the markings reachable from M_0. The SG captures all the possible transition sequences from the STG, and can be derived by the deterministic procedure given in [3]. Formally, SG is a 2-tuple, $\langle V, F \rangle$, where V is a set of states and F a set of edges $\subseteq V \times V$. If a state $v1$ is connected to a state $v2$ by an edge e, then there exists a marking $M2$ (corresponding to state $v2$) which is reached from marking $M1$ (corresponding to state $v1$) by firing a single transition t. The edge e is labeled with t.

In order to implement the SG, each state must be assigned a binary code. In [3], Chu proposed using the signal values S to define a binary labeling of each state. The label is simply the set of values of all signals S that the circuit may have in the given state. With this labeling, only output signals are used as feedback signals. An STG and its SG are shown in Fig. 2. The initial token marking and the corresponding state are shaded. The binary codes are shown inside each state. The order of variables used in the state encoding is *a b c*.

Definition 6 *A state graph is said to satisfy a unique state coding (USC) property [27] if no two different states are assigned the same binary code.*

A variation of the USC property is called the complete state coding (CSC) property which allows two states to have the same code if the environment can distinguish them. This property was proposed by Chu in [3]. When a state graph does not satisfy the CSC property, Chu simply said that the state graph has a "state assignment problem."

Definition 7 *A state graph is said to satisfy a complete state coding (CSC) property if*

- *it satisfies the USC property or*
- *when the same binary code is assigned for two different states, the transitions of the noninput signals, S_{NI}, enabled in two states are identical.*

Thus, only the input transitions enabled in two states are different, and it is assumed that the environment knows how to distinguish them. The state coding is *complete* in the sense that input and output variables completely define the state codes; no extra state variables are needed.

2.2.3 Syntactic Constraints

Syntactic constraints are often imposed on an STG to remove undesirable behavior. The following constraints were proposed by Chu[3]:

- Safeness constraint imposes a bound on the number of tokens that a place may have at any given marking. A free-choice net is called *safe* if the number of tokens in each place does not exceed one for any marking reachable from the initial marking M_0. This safeness constraint is necessary for causality and choice. If a free-choice net is safe, then a transition cannot fire twice in a row without firing some other transition. Otherwise, the causality is violated because the two consecutive firing of a transition, say $t+$, is not possible without firing $t-$ in between. Also, safeness is necessary for specifying choice[3]. Consider a free-choice place with two fanout transitions, t_1 and t_2. If the place has two tokens, then we can no longer say that the place represents a choice between t_1 and t_2 because t_1 can still fire after t_2 and vice versa.

- Liveness guarantees that no deadlock occurs in the circuit. An STG is live if it is strongly connected and if every structural component called a *state machine* has exactly one token. This is a more stringent definition of liveness than the one used in the literature [4], but it makes the behavior of a live and safe STG independent of the initial marking. Thus, the specification of the initial marking becomes optional.

- CSC is necessary because only input and output variables are used for state assignment. Violation of the CSC property must be corrected by adding either extra signals or constraints to the STG.

- Persistency is a fourth constraint which Chu introduced. In section 2.2.4 we show that this constraint is subsumed by the CSC property and should not be used for synthesis from STGs.

Persistency is also confusing since the term has three distinct meanings in the literature. In this paper we distinguish these three meanings with the following terminology.

Persistency A Petri net is said to be persistent if, for any two enabled transitions, the firing of one transition will not disable the other[20]. Thus a marked graph is always persistent, but a free-choice net with free-choice places is not because the firing of one output transition from a free-choice place disables the firing of the rest of the output transitions.

Semi-modularity Using an informal definition from [18], in a semi-modular state graph, if a signal transition $t+$ or $t-$ is excited in state a but signal t does not change value when the circuit goes to a new state b, then t must still be excited in state b. In [18] it is shown that semi-modularity is a sufficient condition for speed-independence with respect to all the signals which appear in the state graph.

STG persistency The term STG persistency refers to the property of the STG, whereas the term semi-modularity refers to the property of the state graph. In [3], Chu proposed a syntactic condition on an STG/NC to make its corresponding state graph semi-modular. This condition will be called STG persistency. In section 2.2.4 we show that this syntactic condition is neither necessary nor sufficient for a semi-modular state graph.

We will discuss the exact implications of the above constraints and show that the syntactic constraints alone are not sufficient to obtain a hazard-free circuit under various delay and operating conditions.

2.2.4 STG Persistency versus Complete State Coding

The characteristic of a state graph which allows a speed independent implementation is *semi-modularity* [18]. In an attempt to characterize speed-independence at the STG level, Chu established the equivalence between an STG and the state graph derived by interpreting the signal values at each marking as a state code. With this equivalence, Chu introduced two syntactic constraints: STG *persistency* and CSC.

According to Chu, an STG is called persistent if all of its noninput transitions are persistent. A transition u is nonpersistent if: transition t enables u but u and $\overline{t}$, the complementary transition of t, are concurrent. To correct this problem for STG/NCs, a persistency constraint from u to $\overline{t}$ can be added so that u must occur before $\overline{t}$ can.

The CSC problem is characterized in terms of complementary sets in the STG. An algorithm for solving the CSC problem for marked graph STGs was developed in [27].

Although some relationship between STG persistency and CSC has been suspected [3][27], it turns out that the CSC property alone is sufficient to ensure a semi-modular state graph, and the STG persistency is only a special case of solving the state assignment problem. Informally, the extracted state graph of an STG is guaranteed semi-modular because of the persistency of noninput signal transitions in the STG. Since the only nonpersistent transitions in an STG are the output transitions of a input-choice place, and these are restricted to signal transitions of input signals, the only possible problem is in the mapping from the STG markings to the states (signal value tuples) which may not be unique. To state this more formally, we propose the following theorem.

Theorem 1 *An* STG *satisfies the CSC property iff its extracted state graph is semi-modular.*

Proof ($\Rightarrow$) Assume the STG satisfies the CSC property but some noninput transition t^* (t^* denotes any transition $\{+, -, \sim, \#\}$ on t) is not semi-modular in the sense that there is a state s_0 where t^* is enabled and after firing some sequence Y of signal transitions, $t^* \notin Y$, state s_1 is reached where t^* is not enabled. Consider the STG marking which corresponds to s_0, which is unique by assumption. Now fire the sequence of net transitions corresponding to Y to reach the marking corresponding to s_1. But since an STG is a free-choice net and such is persistent [in the Petri net sense] for noninput signals, there is no way that this could remove the token(s) which are enabling $t+$ or $t-$. Thus t^* must still be enabled in s_1.

($\Leftarrow$) Assume the state graph is semi-modular but does not satisfy the CSC property. Then there are two markings which have the same label l_0 but enable different noninput signal transitions. Consider one such signal transition t^*. Thus, t^* is both enabled and not enabled in label l_0 which means that the state graph is not even well defined (let alone semi-modular) since the set of enabled transitions in l_0 is not well defined. ■

Since the STG persistency condition is also not sufficient to insure semi-modularity, there is no reason to consider this syntactic condition for synthesis from STGs. In particular, the algorithm presented in [17] may add more constraints than are necessary for state assignment, as observed in [27], but these additional constraints were only an artifact of the incorrect formulation of the conditions for semi-modularity. They are not necessary for a speed-independent implementation.

2.2.5 Logic Functions

A logic function f of n input variables is a mapping $f : \{0,1\}^n \rightarrow \{0,1,2\}$. Each element $\{0,1\}^n$ is called a vertex in the n-dimensional Boolean cube.

The *on-set* of f is defined as the set of vertices x such that $f(x) = 1$; the *off-set*, the set of vertices such that $f(x) = 0$; the *don't-care set*, the set of vertices such that $f(x) = 2$.

A *literal* is a variable or its complement. A *cube* is a set of literals. A cube is *expanded* by removing some literals. When expanded, cubes cover more vertices. For example, when a cube $a\bar{b}$ is expanded to a by removing literal $\bar{b}$, the expanded cube covers both ab and $a\bar{b}$.

A set of cubes is said to be a cover of f if each cube is an *implicant* of f (meaning that it covers some on-set vertex and no off-set vertex) and if every on-set vertex is covered by a cube in the set. A cover is a sum-of-product (SOP) implementation of function f. A cube in the cover can be expanded against the offset by removing some literals without covering any off-set vertices. If a cube cannot be expanded further, it is called a *prime implicant*. A cover consisting only of prime implicants is called a *prime cover*. A cover which ceases to be a cover if any cube is removed is called *irredundant*.

A cover of f is positive (negative) unate in variable x if x appears only in positive (negative) phase in every cube of the cover.

2.2.6 Hazards

A hazard is a possible deviation of the output from expected behavior with respect to some input change. Combinational hazards can be classified into two categories: static and dynamic hazards. A static hazard refers to a 0-1-0 (or 1-0-1) transition when the expected behavior is a static 0 (or a static 1). The former type of hazard is a called a static 0-hazard and the latter a static 1-hazard. Dynamic hazards occur when the expected behavior is a single transition from 0 to 1 (or 1 to 0) but the possible transition becomes 0-1-0-1, 0-1-0-1-0-1, etc. (or 1-0-1-0, 1-0-1-0-1-0, etc.) When two or more feedback lines change at the same, the circuit is said to have a sequential hazard or a *race*. If the output or the state of the circuit depends on the outcome of the race, the race is called *critical*. The manifestation of a hazard is called a *glitch*.

All static and dynamic hazards and critical races should be avoided because they can cause a circuit to malfunction. From now on, we use the term *hazard* to mean both hazards and critical races.

2.2.7 Modes of Operation

A circuit is said to operate in *fundamental mode* if new inputs are applied only after the circuit has assimilated the previous input change. A gate whose output changes only after it has assimilated the previous input is called *atomic*.

If only one input is allowed to change at a time, then it is called the *SIC* condition. If multiple inputs (outputs) are allowed to change at the same time, then it is called the *MIC* condition.

3 STG Extensions

Although the STG/NC is quite powerful for describing both concurrent and sequential behavior, as originally developed in [3] it has several shortcomings in describing general behavior which need to be corrected. Most continuing and recent work has focused on the subset of marked graph STG/NCs which cannot express choices [17][27] and where these shortcomings are not apparent. Since we are interested in using STGs for describing general asynchronous behavior involving both concurrency and sequencing, we propose the following extensions to the original STG/NC:

1. Both input and noninput signal transitions can be used as fanout transitions of a controlled-choice place.
2. There are no restrictions on where the dummy transition ε can be used.
3. The set of predicates labeling the output arcs of a controlled-choice place can be arbitrary expressions on S and need not be disjoint (allowing nondeterminism for input signals).
4. A transition of an input signal does not have to be enabled by the transition of an output signal but may be enabled by the transition of another input signal.
5. More than one net transition can be labeled with the same signal transition.

These extensions to the STG/NC are summarized in Table 1 and will be called simply STGs in the remainder of the paper. The significance of these extensions is illustrated in the remainder of this section.

3.1 Input Signal Controlled-Choices

In an STG/NC, the fanout transitions of a controlled-choice place are restricted to transitions of noninput signals. The example in Fig. 3(a) can only be described using either multiple occurrences of $f+$ and $f-$, or a controlled-choice of the input signal t. Thus we remove the restriction on controlled-choices and allow input as well as output controlled-choice places.

Combining this extension with the null transition, ε, we can specify an input signal which changes during a defined time interval but in an undefined manner. This is similar to the cross-hatched segments denoting don't-care conditions in timing diagrams. Fig. 3(b) shows the construction and Fig. 3(c) an abbreviated notation by extending net transition labels to $S \times \{+, -, \#\}$. The hatch transition $t\#$ means the signal t may or may not change value. Because a $t\#$ transition is nondeterministic, it can only be used for input signals.

Feature	STG/NC	STG
controlled choices	noninput	input & noninput
ε (null) transition	restricted	unrestricted
predicates	disjoint signals	nondisjoint predicates
input transitions	enabled by output only	no restriction
signal transitions	one occurance	multiple occurances

Table 1: Summary of STG extensions to STG/NCs.

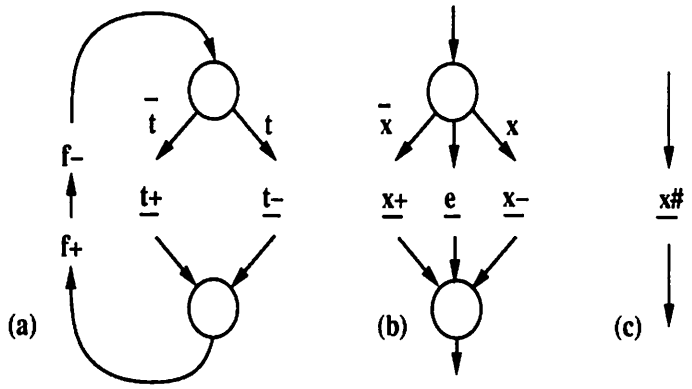

Figure 3: Input choices (a) STG requiring controlled-choice of input signal (b) Common controlled-choice configuration (c) The *hatch* notation for (b)

For a D-latch, the data input D is an example of a signal which changes in an unspecified manner but in a specified time interval (between clocks). Fig. 4(b) (among other things) shows an STG for a D-latch using the *hatch* notation for the data input. This could not be specified by an STG/NC since it requires either multiple occurrences of a transition or controlled input choices.

3.2 Transition Signaling

Binary signal handshaking protocols are divided into two categories: four phase and two phase handshaking. With two phase handshaking, each transition of a signal has the same meaning regardless of whether the transition is from low to high or from high to low. Thus it is also referred to as transition signaling [24].

In an STG/NC, it is not possible to fully represent transition signaling because the phase relationships between signals must be specified. A simple example is shown in Fig. 5. The STG/NC in Fig. 5a and Fig. 5b both meet the specification that each transition of signal a is followed by a transition of signal b. However there is no way to specify one STG/NC which allows both of these possible synthesis choices. The phase relationship between a and b must be fixed in advance.

Using controlled-choice of input signals as in Fig. 5 (c) is one way to avoid this problem. Alternatively the 2-phase nature of a signal can be represented directly by extending the net transitions to $T = S \times \{+, -, \#, \sim\}$, where $j\sim$ means the signal j changes to the opposite of its current value or *toggles*.

As shown in Fig. 5(d), the toggles notation is very concise. The extensions allow the phase relationships to be decided in the synthesis step. A disadvantage is that the concept of a consistent STG must be modified appropriately, as described later. Also note that this changes the relationship between the net state (token marking) and logic state (signal values) since a single marking can represent more than one logic state.

The use of the toggles notation is also illustrated for the output signal Q in Fig. 4(b).

3.3 Controlled-Choice Predicates

Although a free-choice net expresses concurrency very compactly, it can still be tedious to use because similar but not identical sequential behaviors must be specified separately. The D-latch in Fig. 4(a) is an example where two similar sequences differ only in the behavior phase of the output signal Q but otherwise are identical. Extending the arc predicates from single literals to general Boolean expressions allows a more powerful way of collapsing sequential constructs into more compact STGs.

Continuing the D-latch example, Fig. 4(b) shows an STG which is equivalent to the STG in Fig. 4 (a). Essentially a controlled-choice allows a place to represent several different possible states. If B is the Boolean space over the set of signals S, then the number of different states is equal to the number of parts in the partition of B induced by the predicates. In this example, the two predicates (edge labels) $(Q\overline{D} + \overline{Q}D)$ and $(QD + \overline{Q}\,\overline{D})$ partition the space of $S = (Q, D, CLK)$ into two parts so the controlled place represents two distinct states. This corresponds to the two

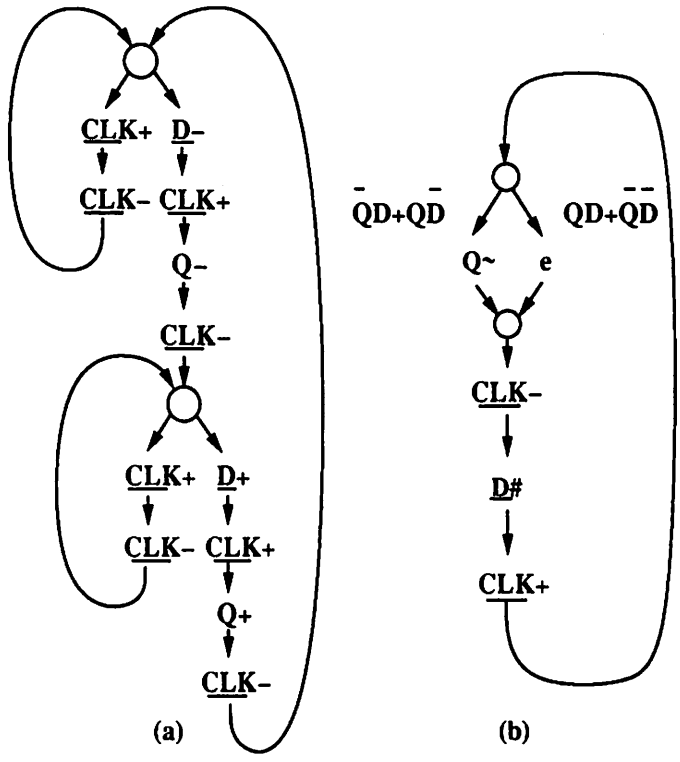

Figure 4: D-latch examples (a) STG using only multiple occurrences of transitions (b) STG using new extensions

free-choice places in the STG of Fig. 4(a). If the two predicates overlapped, the space B could be partitioned into three or more parts which would correspond to three or more different states.

In addition to allowing arbitrarily complex predicates, we also allow the predicates to intersect so that more than one can be true at a time. This form of nondeterminism was already assumed when the *hatch* notation was introduced as shown in Fig. 3. The predicates can only intersect when the choice is for input signals because nondeterminism is only allowed in the environment (the inputs) but not in the implementation. In order for the net to be live or deadlock free, it is necessary that at least one predicate evaluates to true for all states that are possible when the place is marked.

3.4 Consistent STG

With the new STG extensions, the conditions for a consistent STG must be modified. To insure the physical realizability of the STG specification, the signal transitions must always alternate between rising and falling transitions. For example, it is not possible to have an $x+$ followed by an $x+$ without an intervening $x-$, $x\sim$ or $x\#$.

To verify that a Petri net is a free-choice net simply requires a local syntactic check on its structure. However there are no local conditions which will insure that the signal transitions are globally consistent for all possible firing sequences. Thus the consistency of each signal must be checked against the entire STG.

In the STG/NC, consistency for signal x was guaranteed if $x+$ and $x-$ were contained in some simple cycle. When only one occurance of each transition is allowed, this simple condition insures that the signal transitions will alternate. With the extensions, it is much harder to state conditions for consistency. Instead the following procedure is given to check one signal x. For any state machine component of the STG [8] which contains all transitions of x, it uses a DFS traversal to set a phase $\phi_p(x)$ indicating the allowed phase of x at each place p.

1. Initialize the phase ϕ_p to null for all places. At each step of the DFS the current phase of x along the path being searched is maintained in the predicate ϕ.

2. Find an initial place to start the DFS: if an $x+$ or $x-$ transition exists, start at any fanout place and set $\phi = x$ or $\phi = \overline{x}$ respectively; otherwise start at any output place of a $x\#$ or $x\sim$ and arbitrarily set $\phi = \overline{x}$.

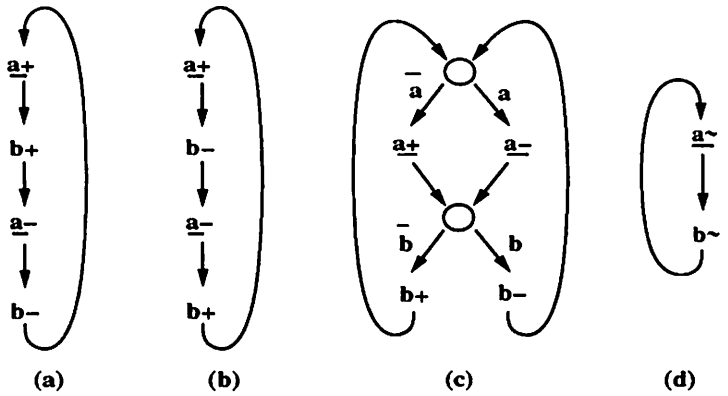

Figure 5: Transition Signaling. (a) Buffer STG. (b) Inverter STG. (c) Using controlled-choice. (d) Using toggles notation.

3. Traverse forward through the STG in DFS manner:
   (a) For each arc predicate encountered, intersect ϕ with that predicate and continue the DFS along that arc if the result is not null.
   (b) At each transition of x, update ϕ as follows:
      i. $x+$: if $\phi = \overline{x}$ set $\phi = x$, otherwise report an inconsistency (x cannot rise if it already is high).
      ii. $x-$: if $\phi = x$ set $\phi = \overline{x}$, otherwise report an inconsistency.
      iii. $x\sim$: set $\phi = \overline{\phi}$.
      iv. $x\#$: set $\phi = 1$ (i.e. x may have either value after this transition).
   (c) At each place, if $\phi \subseteq \phi_p$ then terminate the DFS; otherwise update ϕ_p by $\phi_p = \phi_p \cup \phi$ and continue.
4. If any ϕ_p is null, the STG is inconsistent, i.e. there is no valid phase for signal x at place p of the STG. Also, if a fanout transition is labeled $x\sim$ but ϕ_p is x or $\overline{x}$ then the transition is consistent but could have been labeled just $x-$ or $x+$ respectively.

At the end of this procedure, each ϕ_p indicates the possible values x can have when p is marked with a token. From the termination condition in step (3c), each vertex can be visited at most twice, once with $\phi = x$ and once with $\phi = \overline{x}$. Thus the time to check one signal is linear in the size of the STG. The procedure is repeated for all the signals $x \in S$ resulting in overall quadratic complexity.

4 Hazard-free Logic Implementation

4.1 Deriving Logic From a State Graph

Assuming that the STG is consistent and that it satisfies the CSC property, the state assignment of the section 2.2.2 can be used. After this output logic can be derived from the *implied values* of the outputs at each state. The implied value of an output z in some state s is determined as follows: if a transition belonging to z (i.e., $z+$ or $z-$) is enabled in state s, then the implied value of z is the complement of the present value of z as seen in the binary code of s; otherwise, it is taken to be the present value of z [3]. Thus, each state code becomes either an on-set or an off-set vertex in an n-dimensional Boolean space (where n is the total number of signals S) with respect to each output. It is easy to see that the state assignment of section 2.2.2 produces an implementation where only output variables are used as feedback variables (no internal feedback variables). Fig. 6 illustrates how a logic function is derived from a state graph. The shaded states in the SG become off-set vertices with respect to output c, and are represented by dark circles on the three-dimensional Boolean cube.

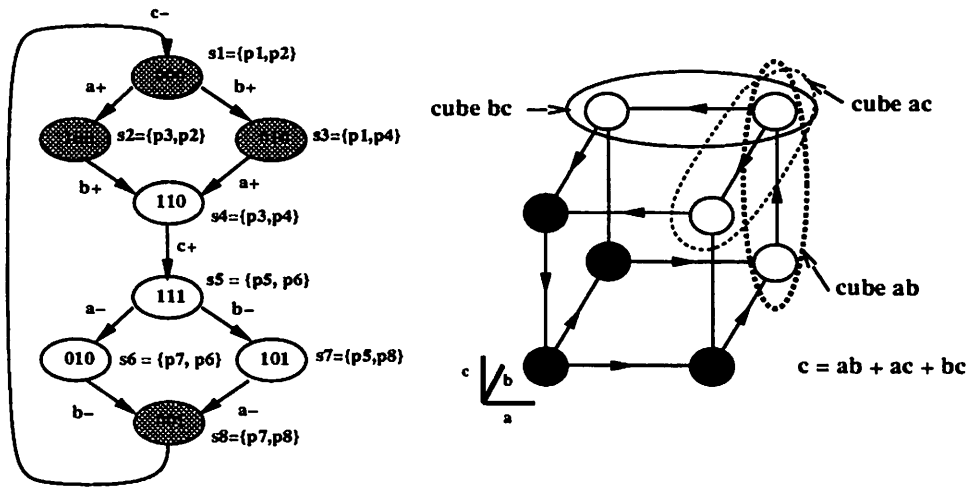

Figure 6: Derivation of Logic from State Graph

4.2 Hazard-free Logic Synthesis

After a set of on-set vertices and a set of off-set vertices are derived from the state graph, logic minimization is applied. For a two-level sum-of-product implementation, the goal is to minimize the number of product terms while making each term depend on as few literals as possible **without introducing any hazards**. This implementation step has not received much attention because a hazard-free implementation was thought to be guaranteed by imposing the syntactic constraints defined in section 2.2.3. Unfortunately, such syntactic constraints only remove undesirable behavior at a functional level and do not automatically produce a hazard-free implementation. We give the exact implications of the syntactic constraints in the following theorem[19][14]:

Theorem 2 *If the given* STG *is live, safe and satisfies the CSC property, then each output variable is a monotone increasing (positive unate) function of itself.*

Proof Assume that an output variable x_i is not positive unate in itself. Then, there must be an on-set vertex

$$v^{on} = (x_1, x_2, \ldots, x_i = 0, \ldots, x_n)$$

and an off-set vertex

$$v^{off} = (x_1, x_2, \ldots, x_i = 1, \ldots, x_n).$$

Since the vertex v^{on} belongs to the on-set, there must exist a state s_0 in the SG corresponding to v^{on} in which the transition x_i+ is enabled. Similarly, there must be a state s_1 corresponding to v^{off} in which the transition x_i- is enabled. Since s_1 is reached from s_0 by firing x_i+, the complementary transition x_i- can fire immediately after x_i+ fires without firing any other transition. Thus the state s_2 reached by firing x_i- from s_1 has the same binary code as s_1 but does not enable x_i+, which violates the CSC assumption. ■

The implications of the above theorem are two-fold:

1. There is no inherent oscillation due to feedback because no output variable depends on the negative phase of itself.

2. Every output Q can be expressed in the form of

$$S + MQ,$$

where S and M are arbitrary Boolean expressions which do not depend on Q. Thus, the following latches can be used to implement the output (or next-state) logic:

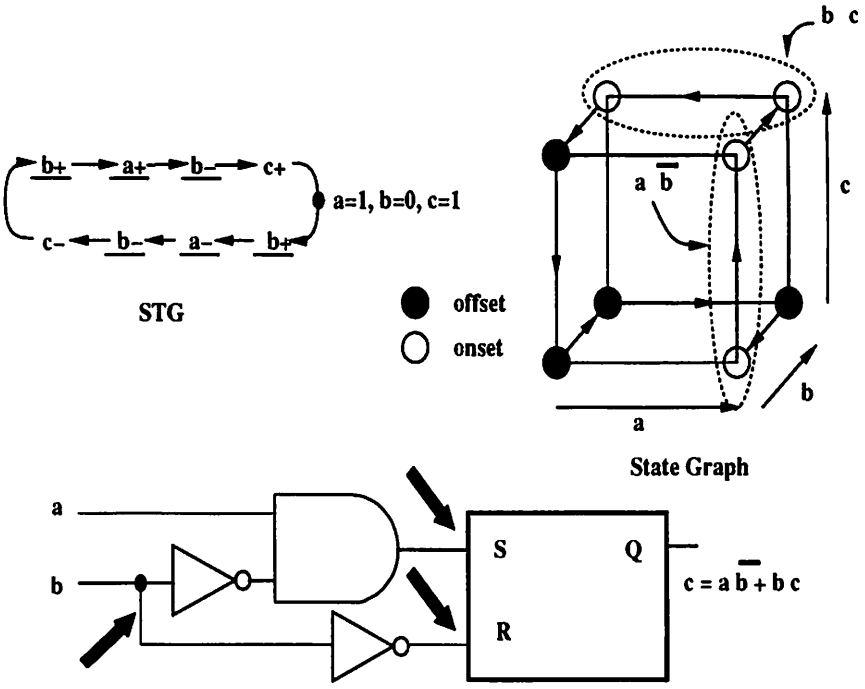

Figure 7: Problem with two-level logic and SR latch implementation

- SM latch : $Q = S + MQ$ [1]
- SR latch : $Q = S + \overline{R}Q$, where $M = \overline{R}$.
- C-element : $Q = AB + (A + B)Q$ [23] where $S = AB$ and $M = A + B$.

Although the above implications are desirable, they imply nothing about hazards in the implementation. We now consider both the single input change (SIC) and the multiple input change (MIC) conditions and give synthesis procedures under both conditions of operation, assuming that the circuit operates in fundamental mode.

4.3 Hazard-free Implementation Under SIC Condition

A two-level SOP implementation has only one type of hazard, namely the 1-hazard, once it is minimized[26]. This property, along with the fact that set-dominant[1] latches implemented with cross-coupled NOR gates are immune to 1-hazards, was used to implement logic in some previous designs [2].

The problem with the above implementation is that a race condition can occur when the set and the reset inputs fall almost simultaneously after both have been asserted high. The expected behavior of the SR latch output is that it should remain high. However, if the set input falls sooner then the reset input, then the output can become low and locked at low. No syntactic constraint can prevent such near simultaneous falling of set and reset inputs because they are internal signals (created during the logic synthesis procedure) whose behavior was not specified by the STG.

Fig. 7 illustrates this race problem. The given STG is live, safe and satisfies the CSC property. Initially the values of inputs a and b are 1 and 0, respectively. The set input, the reset input, and the output c of the SR latch are all 1. When b changes from 0 to 1, it causes both the set and the reset inputs to fall. Because we assume that each gate can have an arbitrary delay, the set input may fall sooner, causing the output to be locked at 0; a clear deviation from the expected behavior! The above problem is a special case of the general logic hazard problem which occurs when an irredundant cover is used. This problem is easily solved by adding some redundant cubes in the cover [7]. This is done by examining all the adjacent state pairs in the state graph and adding some consensus terms if necessary. Fig 8 gives a simple algorithm which removes all logic hazards. The complexity of this algorithm is $O(n)$, where n is the number of states in the state graph.

[1] The set-dominant SR latch should be used to obtain a correct output of 1 when both set and reset inputs are asserted 1.

```
/* F = on-set cover consisting only of minterms; */
/* R = off-set cover; */
hazard_free_minimize(F, R) {
   for (all adjacent states s1 and s2 in state graph) {
      m1 = onset_minterm(s1);
      m2 = onset_minterm(s2);
      if (m1 ≠ NULL and m2 ≠ NULL) {
         c = consensus(m1, m2);
         F = F ∪ c;
      }
   }
   F = single_cube_contain(F); /* remove cubes contained in other cubes */
   F = expand(F, R); /* expand F against R */
}

onset_minterm(s) {
   if (there exists a on-set minterm m corresponding to state s) {
      return m;
   }
   return NULL; /* The implied value of every output in s is 0 */
}
```

Figure 8: Algorithm for Minimizing Logic Under SIC Condition

Fig. 9 shows the application of the above algorithm. A new prime implicant ac is added to eliminate the hazard. In the new implementation the reset input remains at 0 and only the set input falls. Note that with the above algorithm, the latch type is no longer restricted to an SR latch. Any latch which matches the given Boolean expression can be used.

4.4 Hazard-free Implementation Under MIC condition

Concurrent transitions which occur simultaneously can also cause hazards. An example is a 0-hazard produced at the output of an AND gate by two inputs which change simultaneously in opposite directions. Under the MIC condition, a SOP realization is no longer free of 0-hazards, so we must identify cubes (product terms) which can potentially produce glitches and eliminate them. Such cubes can be identified by considering the following four output transitions: $0 \Rightarrow 0$, $0 \Rightarrow 1$, $1 \Rightarrow 0$ and $1 \Rightarrow 1$. For a $0 \Rightarrow 0$ transition, no cube can make a rising or a falling transition. However, cubes with more than two literals changing in opposite directions can produce a glitch which can propagate to an output (0-hazard) and thus must be identified as problematic. Likewise, for a $0 \Rightarrow 1$ transition and a $1 \Rightarrow 0$ transition, glitching cubes need to be identified because they can cause dynamic hazards. For a $1 \Rightarrow 1$ transition, we are guaranteed by the procedure in section 4.3 to have a constant 1 cube; other cubes can make arbitrary transitions. Once all the problematic cubes are identified, we eliminate the glitches by adding to each such cube some literal which remains at 0 while the concurrent transitions take place. Such literals can always be found because it is not possible to have all the transitions in an STG fire simultaneously. However, we need to check that the above cube reduction is legal. We have a legal cube reduction if

$$F_{new} \cup D \supseteq F,$$

where F_{new} is a new on-set cover ($F - c \cup c'$), c a problematic cube, c' the reduced cube and D the external dont-care set. Also, we need to ensure that this reduction does not introduce any logic hazards, which were removed by the previous step. When there are several candidates for reduction, the literal which is least likely to cause further glitches

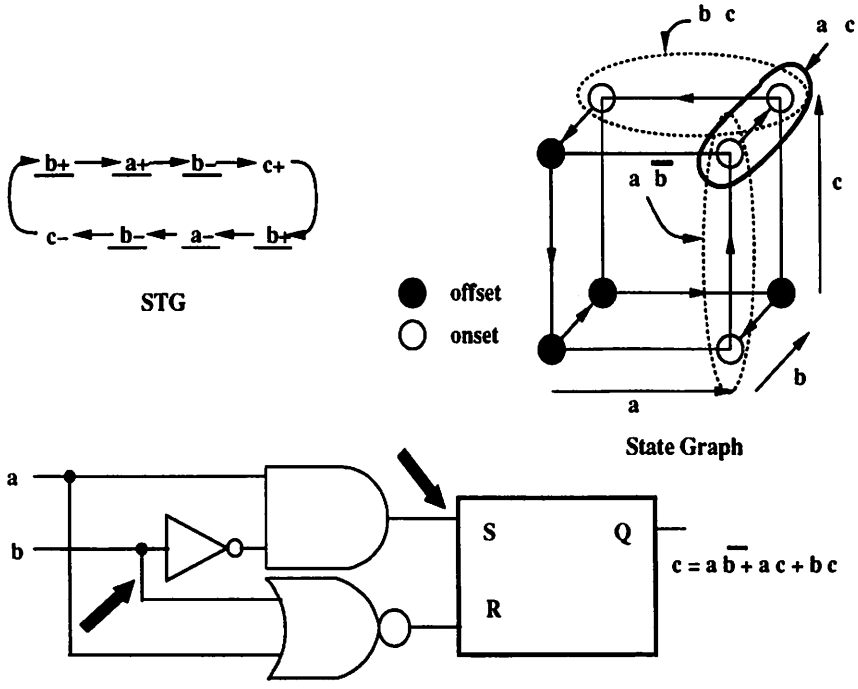

Figure 9: Solution to the hazard problem under SIC

is chosen. On over 40 circuits that we have tried we were able to remove all MIC-related hazards this way. However, if no valid reduction exists, then the circuit is not hazard-free under the concurrent occurrences of some transitions. After such transitions are identified, the STG can be modified (possibly by adding arcs) to sequentialize the concurrency in a manner which does not cause MIC hazards (this can always be done because all the SIC hazards were removed).

Fig. 10 gives an example of a logic which is hazard-free under the SIC condition but not under the MIC condition. The cube which can glitch during the simultaneous occurrences of $a-$ and $d-$ is $\overline{a}d$. In this case, the 0-hazard can be removed by adding literal c to $\overline{a}d$ because it is effectively a constant 0 literal while $a-$ and $d-$ fire. This cube reduction is valid because the on-set vertice(s) (in this case, $\overline{a}\,\overline{c}\,d\,x$) covered by $\overline{a}\,\overline{c}\,d$ are also covered by the implicant $d\,x$.

5 Verification

Formal verification needs to address two aspects of correctness: one is the correctness of the implementation against the specification (*implementation verification*) and the other is the correctness of the design with respect to some desirable properties such as liveness and fairness(*design verification*). Both aspects of correctness are crucial for most circuits but are more so for asynchronous circuits because it is computationally infeasible to perform exhaustive simulation over all possible input patterns and delay variations.

Most of the early work on asynchronous verification did not address both aspects of correctness together because of the lack of a suitable specification and/or underlying hardware model. The ternary simulation technique proposed by Eichelberger in [7] and later refined by Seger in [22] can be used to detect combinational hazards and critical races, but does not address the problem of verifying the correctness of the implementation against the specification. Implementation verification was addressed in [5] using trace theory[21]. The trace-theoretic verification is hierarchical and has been used to detect problems in some published circuits[6]. However, due to the simple underlying model (a deterministic finite automaton with failure states) the useful application of the trace theory has been limited to the verification of safety properties only. Extending the model to check for liveness seems difficult.

We present techniques to perform both design and implementation verification of asynchronous circuits. Our technique is based on the event coordination model of Kurshan [13]. The basic verification approach is similar to

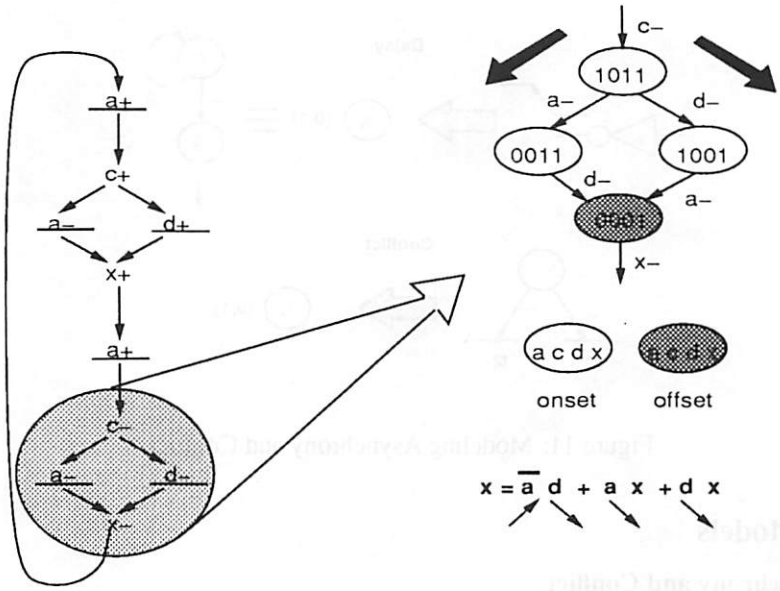

Figure 10: An implementation which is not hazard-free under MIC condition

the trace-theoretic approach of Dill[5] in that both methods check to see if the behavior of the implementation is contained in the behavior of the specification. The differences are that the event coordination model is based on a non-deterministic finite-state model with two types of acceptance conditions (which are very good for liveness checking) and that it supports homomorphic reduction, which is the basis for handling large systems. We show that the event coordination model is a suitable model for modeling concurrency, conflict, and asynchrony of asynchronous circuits. We have used the event coordination model to successfully check for liveness and safeness properties, and to verify that our gate-level implementation is functionally correct and free of hazards with respect to the STG specification.

5.1 Implementation Verification and Design Verification

5.1.1 Implementation Verification

We perform implementation verification as follows. We first transform both the STG specification and the gate-level implementation into a set of interacting FSMs. Then, we feed both of them to *COSPAN*, an AT&T verification engine [9]. *COSPAN* checks if the behavior of the implementation is contained in the behavior of the specification. Any functional incorrectness such as a hazard in the implementation will show up as some extraneous behavior not contained in the specification, and will cause a failure during the behavior containment test. The behavior containment test is done by checking for acceptance conditions on a Cartesian product of two FSMs (one for the implementation and the other for the specification). No complement automaton is computed. The complexity of this verification step is linearly dependent on the size of the product machine, so we have tried to use as few states as possible in our FSM models.

5.1.2 Design Verification

Two properties that we want to check during design verification are liveness and safeness. Liveness can be checked on both the STG and on the gate-level circuits. Safeness is only defined on the STG. Again, we model either the STG or the gate-level circuit with a set of interacting FSMs, and have *COSPAN* perform either a depth-first search (DFS) or a breadth-first search (BFS) of all the reachable states from the initial state. If the specification is live, we should not have any "dead" states: that is, states with no next states. This is checked in *COSPAN* by a special run time option which tells it to abort on deadlocks. Non-safeness in the STG can be detected simply by monitoring the number of tokens in each place and each edge in the STG, and can be checked by *COSPAN* using DFS or BFS enumeration or by directly executing the STG without *COSPAN*.

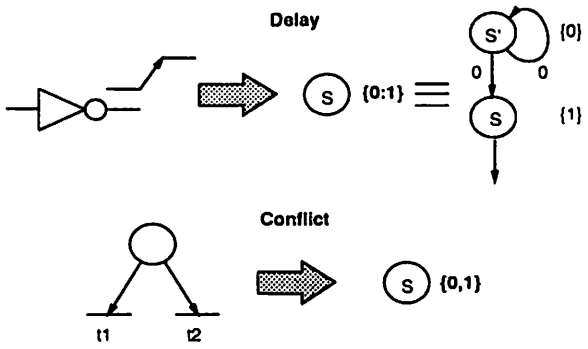

Figure 11: Modeling Asynchrony and Conflict

5.2 Finite-state Models

5.2.1 Modeling Asynchrony and Conflict

Non-determinism is used to model both asynchronous delay and conflict. Asynchronous delay is modeled with a state with two *ordered* outputs[2]. When the state is reached, its first output is selected initially; afterwards the first output may be selected again for any arbitrary number of times but eventually the second output is selected. After the second output is selected, the first output may not be selected again unless the state is re-visited. The model of asynchrony is illustrated in Fig. 11.

Conflict or choice is also modeled by nondeterminism. For example, if there is an input-choice place with two fanout transitions in the STG, it can be modeled with a state with two *unordered* outputs[3]: either output can be selected at any time. It is also straightforward to model the control-choice places. The model of conflict is also illustrated in Fig. 11.

5.2.2 Modeling STGs

The STG model has three basic components: places, transitions and flip-flops. A place is modeled by a deterministic FSM. For example, a place is in state 0 if it has no token. If any of its input transition fires, then it moves to state 1 (has one token). Upon firing of any output transition, the place moves back to state 0, and so on. Transitions are not explicitly represented but rather modeled like ports which connect places. The delay between the time a transition is enabled and the time it actually fires is modeled by the nondeterminism mechanism used for modeling conflict as shown in Fig. 11. Thus, an enabled transition can fire only if some non-deterministic FSM outputs a 1. Concurrent signal transitions can fire simultaneously (MIC condition) because the finite-state model of STG allows enabled transitions to fire totally independently of the other enabled transitions. Flip-flops are necessary to interpret each net transition in the STG as a change in some signal value. For each signal we use a two-state FSM whose state transitions are controlled by the rising/falling transitions of that signal. Fig 12 illustrates the STG model.

5.2.3 Modeling Gate-level Circuits

A gate can have arbitrary delays, and is uniformly modeled by a four-state FSM. (Trace theory uses different models for each gate.) We use four states for two reasons. One is to capture the exact instances of rising and falling transitions. Such transition instances are coordinated with the transitions in the STG during implementation verification. The other reason is to model glitch phenomenon at the gate level. For example, consider the inverter model in Fig. 13. Assume that we are initially in state **OFF**. Now input a becomes low and we move to state **RISE**. If input a changes its value before state transition to **ON** occurs, then we have a glitch. Upon encountering a glitch, we can make a transition to some failure state.

[2]To be precise, outputs should be called *selections* because there is no global distinction between inputs and outputs in the event coordination model.

[3]This type of nondeterminism can also be used to model delay.

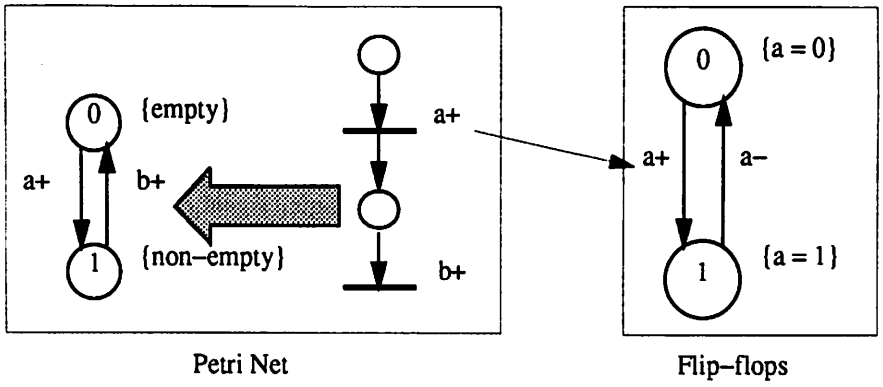

Figure 12: STG Model

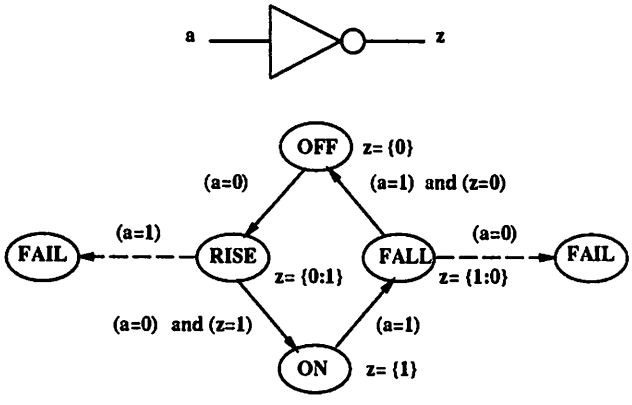

Figure 13: Gate Model

For implementation verification we need to have a complete behavior of the gate-level implementation, i.e., both the system and the environment. To obtain a closed system, we synthesize an environment circuit from the STG. Each gate in the environment circuit is assumed to be atomic and is modeled by the same four-state FSM as shown in Fig. 13. By having the environment circuit monitor all the internal signals in the system circuit, we can allow the environment circuit to change only after all the internal signals in the system have settled down. This is how the fundamental mode of operation is modeled.

5.3 Experimental Results

Our synthesis/verification algorithms are implemented as a package called **ASTG** within the Berkeley Sequential Interactive System (SIS).

Tables 2 and 3 give some experimental results obtained from our synthesis and verification system. The STG examples were obtained from published literature, industry and our digital signal processing group and include both concurrency and conflict. Table 2 gives the result of design verification under the MIC condition. The design verification process checks if an STG or a gate-level implementation is live. The **stg** column is obtained by using *COSPAN* to perform a reachability analysis on STG specifications. All five STGs were found to be live. The number of states and the number of transition conditions are given. The number of transition conditions represents the number of edges out of any state. For example, if there are 3 binary-valued variable, then there are 8 transition conditions (each labeled with some minterm). For the gate-level implementation, the environment circuit is synthesized from the STG specification and the liveness checking is performed on the closed circuit. The two-level implementation refers to the SOP implementation using SM latches. The **old** two-level implementation was obtained by running ESPRESSO. The **old** implementation of *fifo1.g* circuit has a hazard, which is removed by our new technique. In this case, the hazard resulted in a deadlock due to our glitch-catching gate-level model.

Although performing design verification alone on gate-level implementation may detect many hazards, design verification alone does not guarantee that we have a functionally correct implementation. We must perform implementation verification to check for functional correctness. An inverter implementation for a buffer specification is free of any deadlocks but is functionally incorrect (although this type of incorrectness is not called a hazard in the literature, the output clearly deviates from the expected behavior). Table 3 gives the result of implementation verification under the MIC condition. The **STG** column is obtained by verifying a given STG against itself. The **Gate** column is obtained by verifying the gate-level implementation against the STG specification. The **new** implementations of the five circuits were all found to be hazard-free and functionally correct (i.e., their behavior is contained in the behavior of the specification). The reported CPU times give the total time for I/O, synthesis and verification on a DECsystem 5000/200.

The very large number of transition conditions seems to be the bottleneck of verification at this moment. Replacing the core computation inside *COSPAN* with binary decision diagrams (BDD) should help avoid this bottleneck.

6 Conclusions and Future Work

We have proposed some syntactic and semantic extensions to STG. Our extensions allow for a more general specification of asynchronous behavior in a more natural and compact manner. We have shown that syntactic constraints on STGs are not sufficient to guarantee a hazard-free implementation, and presented techniques to synthesize hazard-free SOP implementations under both SIC and MIC conditions. The synthesized circuits are *speed-independent*: they are hazard-free, independent of the gate delay variations, assuming that the circuit operates in fundamental mode.

Behavior containment testing based on the event coordination model has been shown to be a powerful tool for the formal verification of asynchronous circuits. Our implementations have been verified to be speed-independent under the unbounded gate delay model using this behavior containment test. The bottleneck of the present verification comes from the very large number of transition conditions, and we are looking at ways to avoid this bottleneck by using BDD computations. Also, we are looking at automatic reduction techniques for faster verification[12].

Acknowledgment

We gratefully acknowledge the many helpful lectures and discussions by Dr. Robert Kurshan on formal verification. We also thank Peter Vanbekbergen, Luciano Lavagno and Tam-Anh Chu for many interesting discussions. This

Circuit	STG	Gate	
		Two-level (New)	Two-level (Old)
	#states/#trans_conds [CPU time]	#states/#trans_conds (#gates) [CPU time]	
c-elem.g	8/512 [4.8s]	81/134 (5) [3.9s]	81/134 (5) [3.9s]
half.g	14/3584 [7.6s]	197/331 (8) [5.4s]	197/331 (8) [5.3s]
full.g	16/4096 [7.3s]	383/654 (10) [7.4s]	383/654 (10) [6.6s]
fifo1.g	8/512 [4.6s]	63/98 (6) [4.8s]	deadlocked at 48/67 (5) [3.6s]
hybridf.g	80/5242880 [2004.8s]	5789/10897 (17) [17.4s]	5789/10897 (17) [19.3s]

Table 2: Design Verification

Circuit	STG	Gate
	#states/#trans_conds [CPU time]	
c-elem.g	8/512 [9.4s]	105/216 [10.5s]
half.g	14/3584 [7.6s]	251/507 [12.3s]
full.g	16/4096 [13.4s]	473/994 [14.3s]
fifo1.g	8/512 [8.4s]	76/130 [11.0s]
hybridf.g	80/5242880 [13847.7s]	7067/21239 [116.5s]

Table 3: Implementation Verification

work is supported by Defense Advanced Research Projects Agency under contract number N00039-87-C-0182 and by Semiconductor Research Corporation under contract number SRC-91-DC-008, and by various grants from DEC, IBM, Intel, Motorola, AT&T and BNR.

References

[1] C. Berthet and E. Cerny. "Synthesis of Speed-independent Circuits Using Set-Memory Elements". In G. Saucier, editor, *Proc. Int'l. Workshop Logic and Arch. Synthesis for Silicon Compilers*. Grenoble, France, May 1988.

[2] G. Borriello. *"A New Interface Specification Methodology and Its Application to Transducer Synthesis"*. PhD thesis, U.C. Berkeley, 1988.

[3] Tam-Anh Chu. *"Synthesis of Self-timed VLSI Circuits from Graph-theoretic Specifications"*. PhD thesis, MIT, June 1987.

[4] F. Commoner, A. W. Holt, S. Even, and A. Pnueli. "Marked Directed Graphs". *Journal of Computer and System Sciences*, 5:511–523, 1971.

[5] David L. Dill. *"Trace Theory for Automatic Hierarchical Verification of Speed-independent Circuits"*. PhD thesis, Carnegie Mellon University, 1988.

[6] David L. Dill. "Trace Theory for Automatic Hierarchical Verification of Speed-independent Circuits". In *Proc. Fifth MIT Conference*, pages 51 – 65, March 1988.

[7] E. B. Eichelberger. "Hazard Detection in Combinational and Sequential Switching Circuits". *IBM Journal of Research and Development*, pages 90 – 99, March 1965.

[8] Michel H. Hack. "Analysis of Production Schemata by Petri Nets". Master's thesis, M.I.T., February 1972. (Project MAC TR-94).

[9] Z. Har'El and R. P. Kurshan. "Software for Analysis of Coordination". In *Proc. International Conf. Syst. Sci.*, pages 382 – 385, 1988.

[10] D. A. Huffman. "The Synthesis of Sequential Switching Circuits". *J. Franklin Institute*, 257:161 – 190, 275–303, March 1954.

[11] Gordon M. Jacobs. *"Self-timed Integrated Circuits for Digital Signal Processing"*. PhD thesis, U.C. Berkeley, 1989. (UCB/ERL M89/128).

[12] R. P. Kurshan. "Private Communications". 1990.

[13] R.P. Kurshan and K.L. McMillan. "Analysis of Digital Circuits Through Symbolic Reduction". *IEEE Tran. Computer-Aided Design*, 1990. (Submitted for Publication).

[14] L. Lavagno, K. Keutzer, and A. Sangiovanni-Vincentelli. "Synthesis of Verifiably Hazard-free Asynchronous Control Circuits". Technical Report UCB/ERL M90/99, UC Berkeley, November 1990.

[15] A. J. Martin, S. M. Burns, and T. K. Lee. "The First Asynchronous Microprocessor: The Test Results". *Computer Architecture News*, 17(4):95 – 98, June 1989.

[16] Alain J. Martin. "The Design of a Self-timed Circuit for Distributed Mutual Exculsion". In *Chapel Hill Conference on VLSI*, pages 245 – 260, 1985.

[17] T. Meng, R. W. Brodersen, and D. G. Messerschmitt. "Automatic Synthesis of Asynchronous Circuits from High-Level Specifications". *IEEE Tran. Computer-Aided Design*, 8(11):1185 – 1205, November 1989.

[18] R. E. Miller. *"Switching Theory"*, volume II, chapter 10. John Wiley and Sons, 1965.

[19] Cho W. Moon. "On Synthesizing Logic from STGs". April, 1990 (Unpublished Manuscript).

[20] Tadao Murata. "Petri Nets: Properties, Analysis and Applications". *Proceedings of the IEEE*, 77(4):541 – 580, April 1989.

[21] Martin Rem, Jan L.A. van de Snepscheut, and Jan Tijmen Udding. "Trace Theory and the Definition of Hierachical Components". In R. Bryant, editor, *Third CalTech Conference on VLSI*, pages 225 – 239. Computer Science Press, Inc., 1983.

[22] C.-J. Seger. *"Models and Algorithms for Race Analysis in Asynchronous Circuits"*. PhD thesis, University of Waterloo, 1988. (Research Report CS-88-22).

[23] Charles L. Seitz. "System Timing". In C.A. Mead and L.A. Conway, editors, *Introduction to VLSI Systems*, chapter 7. Addison-Wesley, 1980.

[24] Ivan E. Sutherland. "Micropipelines". *Communications of the ACM*, 32(6):720 – 738, June 1989.

[25] James H. Tracey. "Internal State Assignments for Asynchronous Sequential Machines". *IEEE Tran. Electronic Computers*, EC-15(4):551 – 560, August 1966.

[26] Stephen H. Unger. *"Asynchronous Sequential Switching Circuits"*. Wiley-Interscience, 1969.

[27] P. Vanbekbergen, F. Catthoor, G. Goossens, and H. De Man. "Optimized Synthesis of Asynchronous Control Circuits from Graph-theoretic Specifications". In *Proc. Int'l. Conf. Computer-Aided Design*, pages 184 – 187, 1990.

[28] P. Vanbekbergen, F. Catthoor, J.V. Meerbergen, and H. De Man. "Race-free Time-optimised Synthesis of Asynchronous Interface Circuits". In *Proc. Int'l. Workshop on Logic Synthesis*, May 1989.