LOGICAL ANALYSIS OF COMBINATIONAL CYCLES

by

Thomas R. Shiple, Robert K. Brayton, Gerard Berry
and Alberto L. Sangiovanni-Vincentelli

Memorandum No. UCB/ERL M02/21

24 June 2002

LOGICAL ANALYSIS OF COMBINATIONAL CYCLES

by

Thomas R. Shiple, Robert K. Brayton, Gerard Berry and
Alberto L. Sangiovanni-Vincentelli

Memorandum No. UCB/ERL M02/21

24 June 2002

ELECTRONICS RESEARCH LABORATORY

College of Engineering
University of California, Berkeley
94720

Logical Analysis of Combinational Cycles

Thomas R. Shiple
Advanced Technology Group
Synopsys, Inc., Mountain View, CA

Gérard Berry
Esterel Technologies
Villeneuve-Loubet, France

Robert K. Brayton
Department of EECS
University of California, Berkeley

Alberto L. Sangiovanni-Vincentelli
Department of EECS
University of California, Berkeley

June 24, 2002

Contents

Abstract

We classify gate level circuits with combinational cycles based on their input-output behavior. We define a formal class of circuits, the output-stable circuits, for which the outputs settle to a unique value, for any input. Since circuits with combinational cycles can exhibit asynchronous behavior, such as oscillations and race conditions, it is crucial to ground their analysis in a formal delay model, which previous work in this area did not do. We root our definition of output-stable circuits in the up-bounded inertial (UIN) delay model. Building on the work of Brzozowski and Seger in asynchronous circuits, we prove that a practical algorithm proposed by Malik decides the class of output-stable circuits. Finally, we extend the analysis to circuits with flip-flops (i.e., finite state machines).

Although proving the correctness of Malik's algorithm provided the impetus for this work, we also prove a more general result that links ternary simulation with the UIN delay model. Namely, we prove that ternary simulation on a binary input, with internal variables initialized to X, computes exactly the steady-state behavior of a circuit under the UIN delay model, starting from an arbitrary initial state. To our knowledge, this is the first proof of this widely-assumed result.

This paper is theoretical in nature, but it has practical implications. Circuits with combinational cycles arise in practice, especially at higher levels of design. Downstream analysis and synthesis tools are notorious for not handling combinational cycles, or handling them haphazardly. This work provides a theoretical basis for classifying such circuits so that tools can determine whether to reject a cyclic circuit. Furthermore, Malik's algorithm provides two important by-products: for circuits that are output-stable, an equivalent circuit with no combinational cycles can be derived, and for circuits that are not, an input vector demonstrating instability can be produced.

1 Introduction

We analyze the logical behavior of circuits described at the gate level. A *combinational cycle* in such a circuit is a structural cycle containing only logic gates. The analysis of circuits without combinational cycles is straightforward. A Boolean function for each node, in terms of the circuit inputs, can be derived by applying functional composition in a topological order. These Boolean functions exactly correspond to the steady-state electrical behavior of the circuit.

On the other hand, circuits with combinational cycles are usually avoided because the presence of cycles can lead to oscillating or unpredictable behavior. However, not all combinational cycles lead to undesirable behavior. Informally, we say that a circuit is *well-behaved* if for every input, the output stabilizes to a unique value within a bounded amount of time. All acyclic circuits are well-behaved in this sense. Also, some cyclic circuits are well-behaved. For example, for the circuit in Figure 1, the output is $z = x$, even though there can be an oscillation at node y when $x = 0$. Other cyclic circuits may not be well-behaved. In Figure 2, on input $x = 0$, there exists an assignment of delay values to the circuit such that the output z will oscillate, even though the output of the AND seems to be forced to 0 (we study this circuit in Section 2.1).

Techniques to analyze combinational cycles are useful because they arise in practical situations. Consider the following:

1. High-level synthesis, where cycles are created to share datapath resources. An example is Figure 3, which computes $z =$ if (c) then $F(G(x))$ else $G(F(x))$. Even though the cycle in the example is false (because c and $\overline{c}$ are mutually exclusive), Stok [20] notes that such cycles are undesirable because downstream tools cannot handle cyclic circuits. He solves this problem by modifying resource sharing to prevent cycles from being created, at the cost of more control circuitry. Our philosophy is to provide rigorous analysis so that cyclic circuits can be handled directly.

2. The composition of Mealy machines. When a single FSM is synthesized within the context of a set of interacting FSMs, the resulting composition may create combinational cycles [21].

3. The specification of reactive programs in synchronous programming languages. A language like Esterel allows the specification of "zero-delay cycles," and it is the task of the compiler to determine if such cycles are false.

4. The design of symmetric protocols or resource access strategies, where cycles arise naturally [6]. However, the cycles are false for all reachable states of the design. Such circuits are discussed in Section 4.

In some cases (1, 3 and 4), combinational cycles are created intentionally, but with the knowledge that the cycles are false (i.e., for every input provided by the operating environment, no event can be propagated around the cycle). In other cases (2 and 3), the cycles may have been created inadvertently, and the circuit may or may not be well-behaved. Regardless of how or why a combinational cycle is created, the only issue is whether the resulting circuit is well-behaved at its outputs.

The goal of this paper is to define a class of well-behaved circuits, called the *output-stable* class, and demonstrate an algorithm that decides this class (i.e., correctly classifies all circuits). Even though we are interested in the *logical* analysis of combinational cycles, the issue of circuit delays cannot be avoided because they can affect the steady-state behavior of a circuit. Thus, before any class of well-behaved circuits can be defined, we must state precisely what the underlying delay model is. We use the up-bounded inertial (UIN) delay model of Brzozowski and Seger [4], because it is simple yet captures a wide range of behaviors. Roughly, for a delay element with bound D, any pulse of width at least D is propagated within D time units, and any pulse of width less than D may or may not be propagated.

We say a circuit with UIN delays is output-stable if for every binary input value, the outputs stabilize to a unique binary value in bounded time, regardless of the initial state[1] of the circuit. This definition

[1]Intuitively, the *state* of a logic circuit is the value of all the circuit nodes, sometimes called the *internal state*; a formal definition is given in Section 2.

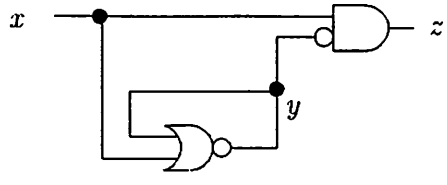

Figure 1: Well-behaved, even though y oscillates when $x = 0$ (Figure 4b from Malik [13]).

permits indefinite oscillations on internal nodes, although these can be prohibited by simply declaring all nodes to be outputs. We show that deciding this class is co-NP-complete. Malik [13] proposed an algorithm to determine if a cyclic circuit is well-behaved, but he did not ground his work in a delay model. We set out to prove that his algorithm is correct for the UIN delay model. In developing this proof, we discovered a more general result that links ternary simulation to the UIN delay model, and which forms a key contribution of this paper.

We define the state vector UIN-SS(N, D, a) roughly as the steady-state behavior of a circuit N with maximum UIN delay bound D, when the input is held at a, and the circuit state evolves in time consistent with the UIN delay model, starting from an arbitrary initial state. In other words, for a given circuit node y, if the value of y always stabilizes in bounded time to 0 on input a, regardless of the initial state and regardless of the UIN delays, then the y component of UIN-SS(N, D, a) is 0. Likewise, if y always stabilizes to 1, then the y component of UIN-SS(N, D, a) is 1. However, if y can stabilize to either 0 or 1 or never stabilizes at all, perhaps because of a different initial state or because of a particular valuation of delays consistent with the UIN delay model, then the y component of UIN-SS(N, D, a) is X. In this terminology, a circuit is output-stable on a if UIN-SS(N, D, a) is not X for any output.

Computing UIN-SS by trying all possible initial states and all possible delay values is infeasible. However, ternary, or 3-valued, simulation can be used. In particular, we show that by initializing all circuit nodes to X and setting the input to a, ternary simulation computes exactly UIN-SS. Since ternary simulation is the essence of Malik's algorithm, showing that his algorithm decides the output-stable class is trivial, once we prove that ternary simulation computes UIN-SS.

Given the semantic gap between the world of UIN delays and the world of ternary simulation, proving the link between these two worlds is not straightforward. Fortunately, we can build on the extensive theory of asynchronous circuits developed by Brzozowski and Seger [4]. They define the possible behaviors of a circuit with UIN delays, and then describe two successively more abstract techniques to analyze the behaviors. First, GMW (greatest multiple winner) analysis is a technique that abstracts away time for all delay assignments to define a state transition graph for the circuit's behaviors under a given input a, from a given initial state. As a direct corollary of their work, we show that GMW analysis can be used to compute UIN-SS, but this is still cumbersome since it involves traversing a state transition graph for each initial state. The second technique Brzozowski and Seger describe is a two part ternary simulation algorithm that computes the steady-state behavior under input a, from a given initial state. Again, given their proof of a tight link to GMW analysis, we show that this two part algorithm can be used to compute UIN-SS. But this still requires a separate computation for each initial state.

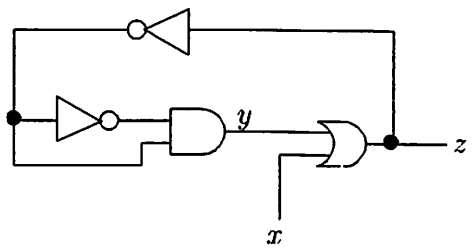

Figure 2: Not well-behaved when $x = 0$ (Figure 6a from Malik [13]).

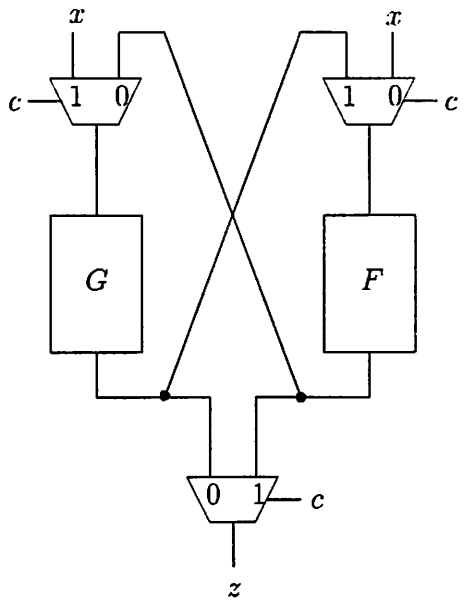

Figure 3: Sharing of resources leads to a false combinational cycle (Figure 2 from Malik [13]).

Malik's algorithm simplifies the analysis further. First, it initializes the state to X, and hence requires just one invocation of the algorithm for each input a. Second, it drastically reduces the number of circuit state variables in the computation by using as variables just a circuit feedback vertex set. We prove that Malik's algorithm computes UIN-SS by showing its relationship to Brzozowski and Seger's two part ternary simulation algorithm. Furthermore, we show that the value of UIN-SS is independent of the delay bounds in the circuit. That is, even though the timing behavior and circuit settling time depend on the bounds, the possible values of the state after the circuit settles do not depend on the bounds.

Note that the link we prove between UIN delays and ternary simulation does not hold when some of the inputs are X; in this case, ternary simulation is conservative. Also note that for acyclic circuits, ternary simulation with binary inputs and X initial state just reduces to binary simulation. In summary, the interesting case for the result we prove is for cyclic circuits with binary inputs.

We extend the definition of output-stable to circuits with flip-flops, to arrive at the class of constructive circuits. Beyond the obvious relevance to hardware design, this extension plays a critical role in compiling programs in Esterel, a language for embedded software. In particular, Berry [1] shows that an Esterel program is well-behaved if and only if the circuit derived from the program is constructive. The algorithm for deciding constructive circuits consists of an initial call to Malik's algorithm, followed by implicit state enumeration over the reachable state space of the flip-flops.

This work has practical applications because many EDA tools, such as cycle-based simulators and formal verification tools, do not accept circuits with combinational cycles, or they handle them in an ad-hoc manner. Malik's algorithm enjoys the feature that if a circuit is output-stable, then a by-product is a functional description for the circuit in the form of a binary decision diagram, whose direct translation to an acyclic circuit can be used for downstream tools. On the other hand, if it is not output-stable, then vectors demonstrating instability can be generated to help debug the circuit.

The outline of this paper is as follows. Section 2 introduces the circuit and delay models, and after describing a sequence of analysis techniques, proves that Malik's algorithm computes UIN-SS. Some proofs are relegated to the appendix. Section 3 defines output-stable circuits, and shows that Malik's algorithm decides this class, and that computing this class is co-NP-complete. Section 4 extends the theory to FSMs. Section 5 reviews related work. Finally, Section 6 provides a summary and discussion of future work.

2 UIN Delay Model and Ternary Simulation

In this section, we show that Malik's algorithm computes UIN-SS, the steady-state behavior under the UIN delay model. Given the large semantic gap between these two notions, we introduce several intermediate analysis methods to help bridge this gap. This section begins by formally defining the circuit model and delay model, upon which we can define the steady-state behavior, or the set of UIN-nontransient states. It is followed by subsections on GMW analysis and ternary simulation, which provide important stepping stones to our final result. The last subsection introduces Malik's ternary simulation algorithm, and shows that it coincides with the UIN delay model. The notation and terminology of this section are largely due to Brzozowski and Seger [4].

2.1 Circuit and Delay Model

The goal of this subsection is to define UIN-SS. Before doing so, we need to formally define our circuit and delay models. Ultimately, the objects we analyze are circuits composed of an arbitrary interconnection of logic gates, with delays inserted at various points in the circuit. The topology of a circuit is given by a *circuit graph*.

Definition 1 [Brzozowski and Seger] A *circuit graph* is a 5-tuple $G = \langle \mathcal{X}, \mathcal{I}, \mathcal{G}, \mathcal{W}, \mathcal{E} \rangle$ where

- $\mathcal{X}$ is a set of *input vertices*, labeled $X_1, X_2, \ldots, X_n$;
- $\mathcal{I}$ is a set of *input-delay vertices*, labeled $x_1, x_2, \ldots, x_n$;
- $\mathcal{G}$ is a set of *gate vertices*, labeled $y_1, y_2, \ldots, y_r$;
- $\mathcal{W}$ is a set of *wire vertices*, labeled $z_1, z_2, \ldots, z_j$; and
- $\mathcal{E} \subseteq (\mathcal{X} \times \mathcal{I}) \cup ((\mathcal{I} \cup \mathcal{G}) \times \mathcal{W}) \cup (\mathcal{W} \times \mathcal{G})$ is a set of directed edges.

■

As we will see, delays can be inserted at the circuit vertices. Input vertices and input-delay vertices are distinct because it is assumed that all external inputs arrive simultaneously. However, the insertion of delays at input-delay vertices allows external inputs arriving at different times to be adequately modeled.

The edge set $\mathcal{E}$ has the following interpretation and restrictions. The indegree of input vertices is 0. Each input-delay vertex is driven by one input vertex, and can drive multiple wire vertices. A wire vertex is driven by exactly one input-delay or gate vertex, and drives exactly one gate vertex. A gate

vertex is driven only by wire vertices. As an example, Figure 4 shows a gate circuit, and Figure 5 gives its corresponding circuit graph. X_1 is an input vertex, x_1 is an input-delay vertex, $y_1, \ldots, y_4$ are gate vertices, and $z_1, \ldots, z_7$ are wire vertices. Note that for each fanout of a gate there is a distinct wire vertex z_i to allow different arrival times at each fanout to be modeled.

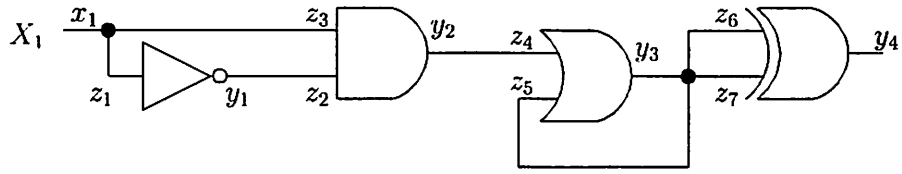

Figure 4: Gate circuit. (Figure 4.5 from Brzozowski and Seger [4].)

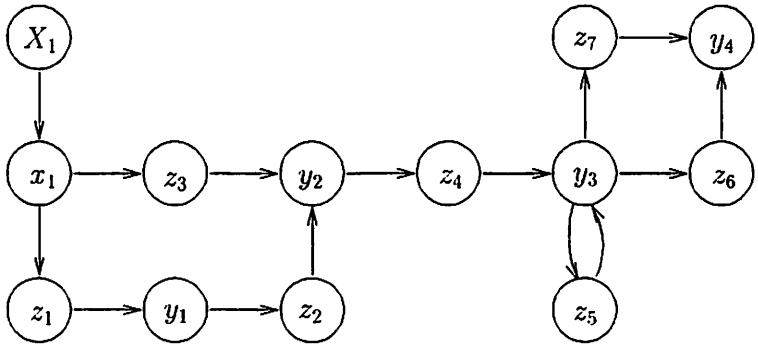

Figure 5: Circuit graph corresponding to the gate circuit in Figure 4 (Figure 4.6 from [4].)

For each gate, wire, and input-delay vertex, there is an associated *vertex function* defined in terms of the immediate fanin of the vertex. For a gate vertex, this is just the Boolean function of the gate. For a wire vertex, it is the value of the input-delay or gate vertex driving the wire vertex. Finally, for an input-delay vertex, the vertex function is X_i, where X_i is the input value provided by the environment. As an example, in Figure 5, the vertex function for y_2 is $Y_2 = z_2 z_3$, for z_1 is $Z_1 = x_1$, and the input-delay vertex function is X_1.

The objects we analyze are networks. A *network* N is derived from a circuit graph by associating delay elements with some subset of input-delay, wire, and gate vertices. The delays are the state holding elements, and hence the vertices with delays are called *state vertices*. The minimum requirement is that each cycle in the circuit graph must contain at least one state vertex, or equivalently, that the set of state vertices forms a feedback vertex set. The input vertices $X_1, X_2, \ldots, X_n$ are always included in a network. Each state vertex has a *state variable* s_i, a *delay bound* D_i, and an *excitation function* S_i, defined as follows.

Definition 2 [Brzozowski and Seger] Start with the vertex function. Then repeatedly remove all dependencies on vertices that are neither state vertices nor input vertices, by using functional composition of the vertex functions. The result of this process is the *excitation function* S_i. ∎

In other words, the excitation functions are the logic functions at each state vertex derived by collapsing all the logic between state vertices. As an example, for the circuit graph of Figure 5,

suppose we select the vertices y_1 and z_5 as state vertices. The corresponding network is depicted in Figure 6 showing the new functional dependencies. There are two set of vertices: the input vertices (in this case just X_1), and the state vertices (y_1 and z_5). The excitation functions of the state vertices are $Y_1 = X_1$ and $Z_5 = z_4 + z_5 = y_1 X_1 + z_5$.

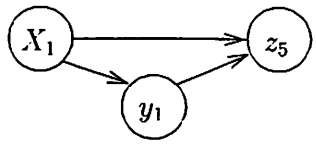

Figure 6: Network corresponding to the circuit in Figure 4 with state vertices y_1 and z_5.

A *total state* $c = (a, b)$ of a network is an $(n+m)$-tuple of binary values, the n-tuple a being the value of the inputs, and the m-tuple b being the values of the state variables $s_1, s_2, \ldots, s_m$. For convenience, we write the total state (a, b) as $a \cdot b$. Note that the excitation functions S_i are defined over the total state.

Thus far we have discussed the structure and function of a circuit. Now we introduce the realm of time. The state vertices are those vertices with delay elements. A delay element has an input $X(t)$, an output $x(t)$, and a delay $\delta(t)$, as depicted in Figure 7. The signals $X(t)$ and $x(t)$ vary with time. They are assumed to be binary and capable of instantaneous changes from 0 to 1 and from 1 to 0. x is *unstable* at time t if $x(t) \neq X(t)$. In network terms, x represents a state variable, and X its excitation function. The total state $a \cdot b$ is *stable* if $b_i = S_i(a \cdot b)$ for each state variable s_i, and is *unstable* otherwise. Thus, $a \cdot b$ is stable if the state b can change only by changing the input a. In the example in Figure 6, the total state $X_1 = 1$, $y_1 = 1$ and $z_5 = 1$ is unstable because $y_1 \neq Y_1 = 0$, but $X_1 = 1$, $y_1 = 0$ and $z_5 = 0$ is stable.

Many delay models have been introduced in the literature, ranging from fixed ideal delay to bi-bounded inertial delay [4]. The delay model we adopt is the *up-bounded inertial* (UIN) delay model [4]. This model captures an infinite range of possible behaviors, which in fact is a superset of the behaviors possible in the fixed ideal delay and bi-bounded inertial delay models. Considering such a wide range of behaviors is conservative, which is appropriate when determining the function of circuits with combinational cycles. Furthermore, as this paper shows, this delay model permits efficient analysis via ternary simulation.

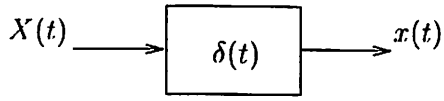

Figure 7: Delay element.

Definition 3 [Brzozowski and Seger] In the *up-bounded inertial* delay model, the delay is bounded from above by a parameter D, $0 < \delta(t) < D$, and the following two properties must be satisfied:

1. *If x changes, then it must have been unstable.*
   Formally, if $x(t)$ changes from α to $\overline{\alpha}$ at time τ, then there exists $\delta > 0$ such that $X(t) = \overline{\alpha}$ for $\tau - \delta \leq t < \tau$.

2. *x cannot be unstable for D units of time without changing.*
   Formally, if $X(t) = \alpha$ for $\tau \leq t < \tau + D$, then there exists a time $\tilde{\tau}, \tau \leq \tilde{\tau} < \tau + D$, such that $x(t) = \alpha$ for $\tilde{\tau} \leq t < \tau + D$. (Note that this property implies that the δ in Property 1 must be less than D.)

■

Intuitively, if an input pulse is at least D units of time, then the output must respond within D. If an input pulse is less than D, then the output may or may not respond. Note that D cannot be zero; this is to avoid zero-delay loops in the network. Also, note that the delay δ is itself a function of time. As an example, Figure 8 shows two possible responses to an input waveform, where $D = 2$.

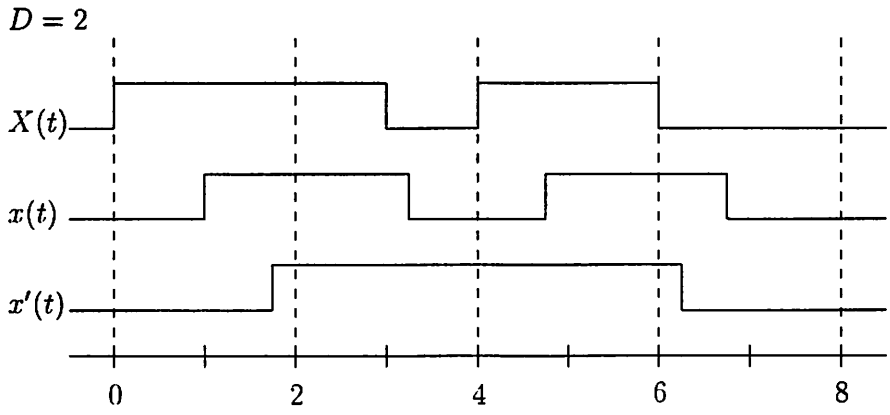

Figure 8: Up-bounded inertial delay waveforms.

Now we move from the behavior of a single delay element to the behavior of a network containing multiple delay elements. A *UIN-history*(N, D, a) captures the behavior of a network as it evolves over time in response to an input a. Since a UIN-history includes the time of every state change (as we will see next), and since δ is a function of time, there are an infinite number of UIN-histories for a given input a. For state variable s_i, $s_i(t)$ is the value of s_i at time t, and $S_i(X(t) \cdot s(t))$ is the value of the corresponding excitation at time t, where $X(t) \cdot s(t)$ gives the value at time t of the total state.

Definition 4 [Brzozowski and Seger] A *UIN-history*(N, D, a) of a network N for some input value a is an ordered triple $\mu = \langle \Theta, X(t), s(t) \rangle$, where[2]

- Θ is a strictly increasing sequence $\Theta = (t_0, t_1, \ldots)$ of real numbers giving the instants at which the state vector s changes.
- $X(t)$ maps the real numbers to B^n, and satisfies $X(t) = a$ for all $t \geq t_0$.
- $s(t)$ maps the real numbers to B^m, and satisfies the properties that:
  1. $s(t)$ is constant during any interval $t_i \leq t < t_{i+1}$, for all $i \geq 0$,

[2] $B = \{0, 1\}$.

2. $s(t_{i-1}) \neq s(t_i)$, for each $i \geq 1$, i.e., $s(t)$ changes at each t_i;
3. if the sequence $(t_0, t_1, \ldots, t_r)$ is finite, then for all $t \geq t_r$ we have $s(t) = b^r$, for some $b^r \in B^m$ such that $a \cdot b^r$ is a stable total state of N, and
4. if the sequence $(t_0, t_1, \ldots)$ is infinite, then only a finite number of state changes occur in any finite time interval (i.e., non-Zeno).

- For each variable s_j, the input/output waveform $S_j(X(t) \cdot s(t))/s_j(t)$ is consistent with the assumption that variable s_j is represented by the delay-free excitation function S_j in series with an up-bounded inertial delay. In other words, the input/output waveform satisfies the two properties of UIN delays in Definition 3.

■

Note that the definition of UIN-history places no restriction on the initial state $s(t_0)$. A UIN-history can be seen as a timed sequence of states. Also, one can associate a corresponding untimed history giving the sequence of states through which the network passes. For a given UIN-history, we say that $s(t_j)$ is *reachable* from $s(t_i)$ if $t_j > t_i$. As an example, consider the network in Figure 9, where $D_1 = 1$, $D_2 = 3$ and $D_3 = 2$. A UIN-history$(N, D, 1)$ is given by $\Theta = (0, 0.5, 3.0, 4.3)$, $X(t) = 1$ for all $t \geq 0$, and $s(t)$ is given by the waveforms s_1, s_2, s_3 in Figure 9. One can verify that these waveforms are consistent with the properties of UIN delays, for the specified delay bounds. For example, s_2 follows the change on s_1 after 2.5 time units, which is less than $D_2 = 3$. The corresponding untimed history for $s_1 s_2 s_3$ is $(001, 101, 111, 110)$.

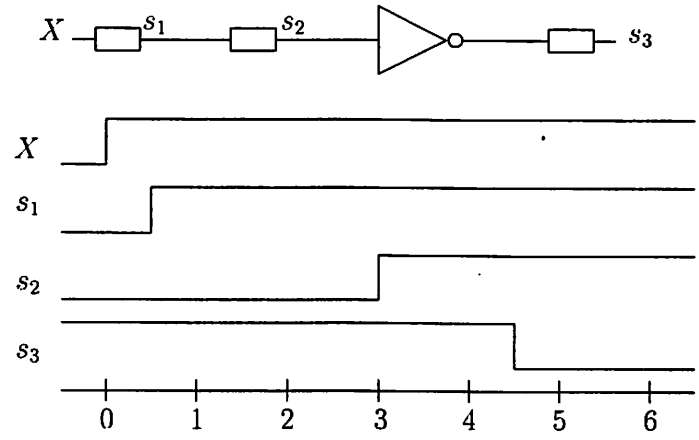

Figure 9: A UIN-history$(N, D, 1)$; $D_1 = 1$, $D_2 = 3$ and $D_3 = 2$. Delay elements are associated with the state vertices s_1, s_2 and s_3.

Ultimately, we are only interested in the steady-state behavior of a network, that is, after the "transients" have died off. Because each inertial delay is up-bounded, the network can remain in the "transient" phase after an input change for only a bounded time, before passing into the "nontransient" phase. The following definition and theorem make this notion precise.

Definition 5 [Brzozowski and Seger] Let N be a network with maximum delay bound D. Let N be started in state b with the input held constant at a. A state b' is said to be *UIN-transient*(N, D, a, b) *with limit* τ for $a \cdot b$ if it is reachable from b and there exists a real number $\tau > 0$ such that in every UIN-history μ, the condition $t \geq \tau$ implies $s(t) \neq b'$. ∎

That is, a state is UIN-transient with limit τ if the network cannot be in that state after time τ.

Theorem 6 (Brzozowski and Seger) *Let N be a network with m up-bounded delays with upper bound D. Suppose N is in state b at time 0 and the input is held constant at a from time 0 until time $t \geq (2^m - 2)D$. Then the state of the network at time t is not a* UIN-transient(N, D, a, b) *state with limit $\tau = (2^m - 2)D$.*

In other words, by waiting at most $(2^m - 2)D$ time, the network has enough time to pass through any transients. We call a state *UIN-nontransient* if it is reachable and not UIN-transient. In the sequel, UIN-nontransient states will always be with limit $(2^m - 2)D$. It is important to note that Theorem 6 follows from the properties of UIN delays, and is not a statement about any particular analysis technique.

Before proceeding further, we briefly introduce the least upper bound, or *lub*, of ternary algebra. This will receive further treatment in Section 2.3, but here we define $\text{lub}\{0,0\} = 0$, $\text{lub}\{1,1\} = 1$, $\text{lub}\{0,1\} = \text{lub}\{1,0\} = \text{X}$. This can be naturally extended to the *lub* of sets of binary values, and the *lub* of sets of binary vectors.

Given the definitions of UIN-nontransient states and of *lub*, we can define UIN-SS, the steady-state behavior of a circuit. This is the key definition of Section 2.

Definition 7 Let N be a network with maximum delay bound D, with the input held constant at a. UIN-SS$(N, D, a) \in \{0, 1, \text{X}\}^m$ is the least upper bound of the states N can be in after $(2^m - 2)D$ time, starting from any initial state b. That is,

$$\text{UIN-SS}(N, D, a) = \text{lub}\{\text{UIN-nontransient}(N, D, a, b) | b \in B^m\}.$$

∎

UIN-SS summarizes the UIN-nontransient states of a network for any UIN-history, starting from any state. At first glance, it would seem impossible to compute UIN-SS since an infinite number of discrete delay values would need to be considered for each delay element. In fact, UIN-SS can be computed in polynomial time using ternary simulation, as shown later.

We will see that UIN-SS is precisely the information needed to decide if a network is output-stable. However, it turns out that the selection of state variables (i.e., the placement of delays) can affect the value of UIN-SS, and hence the value of the circuit outputs, as illustrated by the following example. Consider the output z in Figure 10, with the input $x = 0$. Let N_1 be the network with just one state variable, y_1. The excitation function for y_1 is $S_1 = x + (\overline{y_1} \cdot y_1) = x$. Hence, UIN-nontransient$(N_1, D, 0, b) = 0$, and thus UIN-SS$(N_1, D, 0) = y_1 = 0$, and $z = 0$. Now let N_2 be the network with state variables y_1 and y_2, and corresponding excitation functions $S_1 = x + (\overline{y_1} \cdot y_2)$ and $S_2 = y_1$. Furthermore, let the delay bounds be $D_1 = D_2 = 1.5$, the starting state be $y_1 = 0$ and $y_2 = 1$, and the input be $x = 0$. Then the UIN-histories$(N, D, 0)$ shown in Figure 11 are possible, which implies that both 01 and 10 are UIN-nontransient$(N_2, D, 0, 01)$ states. Thus, UIN-SS$(N_2, D, 0) =$

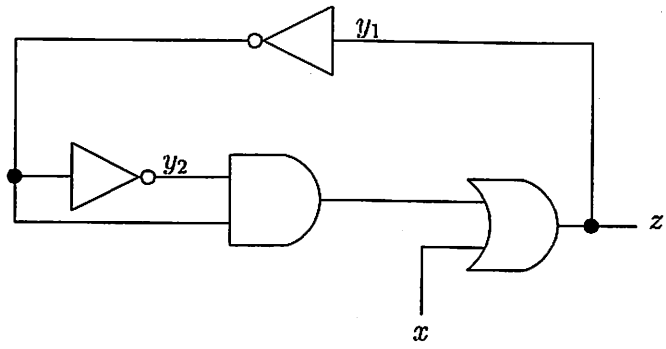

Figure 10: Different placement of delay elements affects output value.

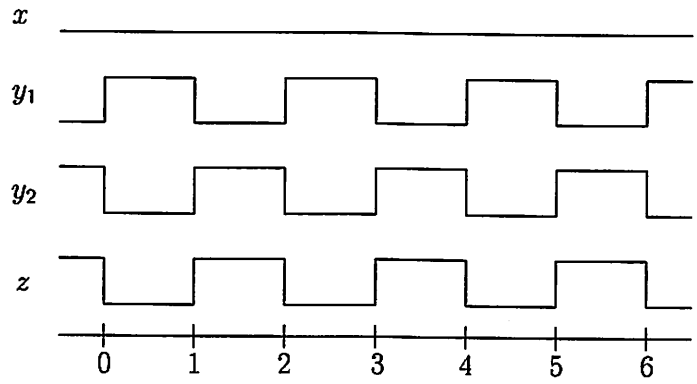

Figure 11: UIN-histories$(N, D, 0)$ for N_2 with $D_1 = D_2 = 1.5$.

$\text{lub}\{01, 10\} = \text{XX}$. Since the output z equals S_1, and $S_1(0{\cdot}01) = 1$ and $S_1(0{\cdot}10) = 0$, the output can be either 0 or 1.

Each choice of where delay elements are placed corresponds to a different network. Because different networks derived from the same circuit can have different output behaviors, we want to identify a particular network for each circuit as the representative network of the circuit. The most conservative choice is to assume that each gate, wire and input-delay vertex can have a delay independent of the others. This is called a *complete network*. This assumption is warranted for two reasons. First, the selection of all circuit vertices as state vertices matches the reality of having independent sources of delay at each and every circuit element. One generally should not make assumptions about the correlation of delays to ensure the functional correctness of a circuit. The second reason is that adopting this conservative model allows us to leverage the power of ternary simulation to simplify the analysis of circuits with up-bounded inertial delays. Henceforth, when we refer to UIN-SS for a given circuit, we are referring equivalently to the corresponding complete network.

2.2 GMW Analysis

We have defined UIN-SS for circuits directly in terms of the UIN delay model as the *lub* of the UIN-nontransient states over all UIN-histories. Our goal is to prove that Malik's algorithm computes UIN-SS; conceptually, this is a big jump. In this subsection, we describe GMW analysis, which helps us bridge this gap by abstracting away time in UIN-histories to yield a state transition graph.

General multiple winner (GMW) analysis is a technique to determine the response of a network to a given input. The technique is called "general" because the relative values of the delays are not specified. The only assumption is that the delays are bounded from above; this is consistent with the UIN delay model. Even though we are ultimately interested in the analysis of complete networks (our representatives for circuits), this subsection continues with arbitrary networks without complicating the discussion.

For a total state $a{\cdot}b$, the set of *unstable state variables* is defined as

$$\mathcal{U}(a{\cdot}b) = \{s_i \mid b_i \neq S_i(a{\cdot}b)\}.$$

That is, a state variable s_i is unstable with respect to state $a{\cdot}b$ if applying the corresponding excitation function S_i to $a{\cdot}b$ yields a value different from the current value b_i. State $a{\cdot}b$ is stable if $\mathcal{U}(a{\cdot}b) = \emptyset$. The GMW relation $R_a \subseteq B^m \times B^m$ describes how the state of network N evolves with the input held constant at a; bR_ab' means that the state may change from b to b'. Intuitively, R_a is the state transition graph for input a.

Definition 8 [Brzozowski and Seger] For any $b \in B^m$,

- bR_ab, if $\mathcal{U}(a{\cdot}b) = \emptyset$, i.e., b transitions to itself if it is stable
- $bR_ab^{\mathcal{K}}$, if $\mathcal{U}(a{\cdot}b) \neq \emptyset$, and $\mathcal{K}$ is any nonempty subset of $\mathcal{U}(a{\cdot}b)$, where $b^{\mathcal{K}}$ means b with all the variables in $\mathcal{K}$ complemented.

No other pairs of states are related by R_a. ∎

The model is called "multiple winner," because in an unstable state, any nonempty subset of unstable state variables can change at the same time. Note that R_a makes no reference to an initial state.
The relation R_a can be depicted as a directed graph, where an edge from b to b' indicates that bR_ab'. A state b may have more than one immediate successor, indicating a race condition. A state b with a self-loop indicates that $a \cdot b$ is stable. $R_a(b)$ denotes the graph R_a restricted to those states reachable from b. As an example, consider the RS-latch in Figure 12, with inputs r and s and state variables q and z. The graph of R_{01} is shown in Figure 13. Each pair of binary values is a state qz. An underlined value (e.g., $\underline{0}$) indicates that the corresponding variable is unstable in that state. Only unstable variables can change. The states 01 and 00 each have one unstable variable, so each has a unique successor state. The state 11 has two unstable variables, so it has three successor states, one for each nonempty subset of unstable variables. State 10 is stable, so it has a self-loop. In this case, the graph for $R_{01}(11)$ (the subgraph induced by those states reachable from 11, holding the input constant at 01) is the same as R_{01}.

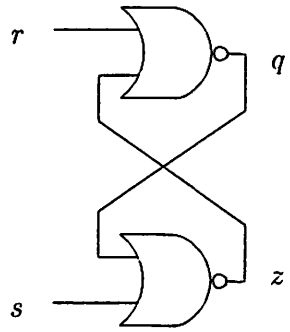

Figure 12: RS-latch.

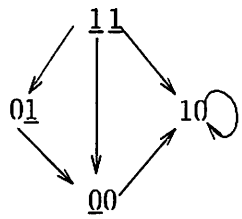

Figure 13: Possible state sequences over qz, for the RS-latch with input $rs = 01$.

For GMW analysis, there is a concept corresponding to the UIN-nontransient states. Consider the fragment of a GMW relation of Figure 14, for some input a (states 011 and 110 and their incident edges are not shown). The cycle $(0\underline{0}\underline{0}, 0\underline{1}\underline{0})$ is called *transient* because there exists a variable (the third one) that is unstable and has the same value (0) in every state of the cycle. Since all delays are up-bounded, the network cannot remain in this cycle indefinitely. Contrast this to the *nontransient* cycle $(\underline{0}\underline{0}1, \underline{1}\underline{1}1)$. The network can remain in this cycle indefinitely because each unstable variable changes value during the cycle. As an aside, the network is not constrained to remain in this cycle, since it can transition to the stable state 101 whenever it is in state 001.
Those states that are in a nontransient cycle, or follow a nontransient cycle, are called *GMW-outcome* states. Formally, Brzozowski and Seger define GMW-outcome(N, a, b), the *outcome states of* b,

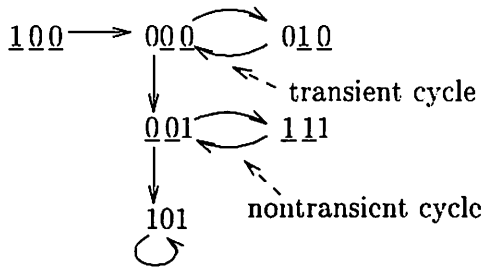

Figure 14: A fragment of a GMW relation.

as that set of states in the graph $R_a(b)$ that are reachable from b via a nontransient cycle. For the graph in Figure 14, GMW-outcome$(N, a, 100)$= $\{001, 111, 101\}$ and GMW-outcome$(N, a, 101)$= $\{101\}$. Note that states of a transient cycle can be GMW-outcome states of b, as long as they are reachable from b via a nontransient cycle. The next result establishes the correspondence between the UIN-nontransient states and the GMW-outcome states. This result is critical because it makes an exact link between the physical world of delay models and the more abstract world of GMW analysis.

Theorem 9 (Brzozowski and Seger) *Let N be a network with maximum delay bound D. Let N be started in state b with input held constant at a. Then*
UIN-nontransient(N, D, a, b) = GMW-outcome(N, a, b).

As an example of GMW-outcome states, for the network N_2 from Figure 10, the GMW relations over y_1y_2 for $x = 0$ and $x = 1$ are shown in Figure 15. When $x = 1$, y_1y_2 stabilizes to 11. However, when $x = 0$ and the starting state is 01, 10 or 11, both 01 and 10 are nontransient states, and hence GMW-outcome states. This corroborates the earlier analysis based on UIN-nontransient states.

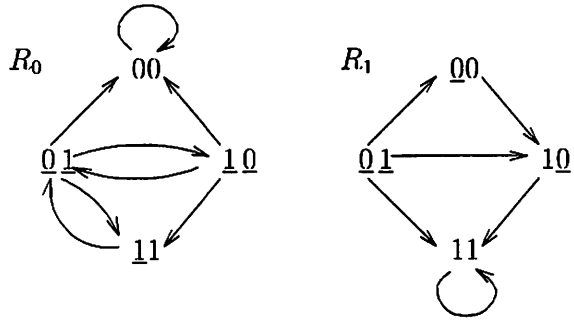

Figure 15: GMW relation over y_1y_2 for network N_2.

The definition of UIN-SS(N, D, a) is based on the UIN-nontransient(N, D, a, b) states of a network, where the network has a specific maximum delay bound D. One might wonder if the set of UIN-nontransient states changes as the delay bounds change. The answer is no: all that matters is that the delays are up-bounded. Since the actual delay of a delay element with bound D_1 can dynamically vary from any value greater than 0 to D_1, any relative ordering of events seen in a network with maximum bound D can also be achieved with maximum bound D', and vice versa.

Theorem 10 *Consider two networks N and N' that are exactly the same, except that N has maximum UIN delay bound D and N' has maximum UIN delay bound D'. Let a be an input and b be an internal state. Then* UIN-nontransient(N, D, a, b) = UIN-nontransient(N, D', a, b).

Proof The definition of the GMW relation R_a is independent of the delay bounds (it only assumes they are finite), and hence the definition of GMW-outcome(N, a, b) is independent of the delay bounds. Given the equality of GMW-outcome(N, a, b) and UIN-nontransient(N, D, a, b) in Theorem 9, the result follows trivially. ∎

The next proposition reduces the computation of UIN-SS to the GMW-outcome states, and provides us a step towards proving the correctness of Malik's algorithm.

Proposition 11 *Let N be a network, and a an input value. Then*

$$\text{lub}\{\text{GMW-outcome}(N, a, b) | b \in B^m\} = \text{UIN-SS}(N, D, a).$$

Proof Trivial, since by Theorem 9, UIN-nontransient(N, D, a, b) = GMW-outcome(N, a, b). ∎

2.3 Ternary Simulation

In theory, GMW analysis could be used to compute the GMW-outcome states of an input change. However in practice, constructing the graph of R_a, which has 2^m vertices, and traversing it is computationally intractable. Ternary simulation is an efficient means to "summarize" the set of GMW-outcome states, and it turns out is sufficient to compute UIN-SS.

Ternary simulation uses a third value, X, to denote an uncertain or changing value on a wire. The set $\{0, 1, \text{X}\}$ is partially ordered on the "uncertainty" relation $\sqsubseteq$ where,

$$0 \sqsubseteq 0, 1 \sqsubseteq 1, \text{X} \sqsubseteq \text{X}, 0 \sqsubseteq \text{X}, \text{ and } 1 \sqsubseteq \text{X}.$$

When $s \sqsubseteq t$, we say that t *covers* s. Likewise, the vector $\langle t_1, t_2, \ldots, t_n \rangle$ covers $\langle s_1, s_2, \ldots, s_n \rangle$ if $s_i \sqsubseteq t_i$, for all i. Any nonempty subset of $\{0, 1, \text{X}\}$ has a *least upper bound*, or *lub*. In particular, lub$\{0\} = 0$, lub$\{1\} = 1$, and the *lub* of every other nonempty subset is equal to X.

A *ternary function*[3] $\mathbf{f}$ is a mapping from $\{0, 1, \text{X}\}^n$ to $\{0, 1, \text{X}\}$. For any Boolean function f there exists a natural *ternary extension*, defined as follows:

$$\mathbf{f}(\mathbf{a}) = \text{lub}\{f(t) \mid t \sqsubseteq \mathbf{a}\}.$$

Figure 16 shows the ternary extension for several Boolean functions. They follow the basic rule that a 0 or 1 output value can be deduced whenever there is sufficient information available at the inputs. For example, a 0 at any input of an AND gate forces the output to 0. An important property of the ternary extension $\mathbf{f}$ of any Boolean function f is *monotonicity*:

$$\mathbf{a} \sqsubseteq \mathbf{b} \text{ implies } \mathbf{f}(\mathbf{a}) \sqsubseteq \mathbf{f}(\mathbf{b}).$$

That is, if $\mathbf{b}$ is at least as uncertain as $\mathbf{a}$, then the output $\mathbf{f}(\mathbf{b})$ is at least as uncertain as $\mathbf{f}(\mathbf{a})$.

[3]Following Brzozowski and Seger's convention, boldface is used to refer to ternary valued functions, relations, and variables.

NOT	
0	1
1	0
X	X

AND	0	1	X
0	0	0	0
1	0	1	X
X	0	X	X

OR	0	1	X
0	0	1	X
1	1	1	1
X	X	1	X

XOR	0	1	X
0	0	1	X
1	1	0	X
X	X	X	X

Figure 16: Ternary extension for the NOT, AND, OR, and XOR functions.

Given a binary network N with n inputs and m state variables, its ternary extension $\mathbf{N}$ is just N with each excitation function $S_i : \{0,1\}^{n+m} \rightarrow \{0,1\}$ replaced by its ternary extension $\mathbf{S}_i : \{0,1,\mathrm{X}\}^{n+m} \rightarrow \{0,1,\mathrm{X}\}$. The vector of ternary excitation functions is denoted by $\mathbf{S}$. This corresponds to the interpretation of the network in Scott's ordered Boolean domain $B_\perp = \{\perp, 0, 1\}$, familiar in other communities [8, 16].

Given the definition of ternary network N, we can now describe Brzozowski and Seger's algorithm for ternary simulation, which they adapted from Eichelberger [7]. The idea is to determine the nontransient behavior of a network starting from a binary valued initial state b, with the input held constant at a. The first part of the algorithm, TernSim-A, takes as input the network $\mathbf{N}$ and the total state $a{\cdot}b$, and propagates maximum uncertainty throughout the network, while leaving the input fixed at a. That is, as the network is simulated on $a{\cdot}b$, for each state vertex, the *lub* of the current value and the next value is taken as the new state value. For example, for a state variable with value 1 at the output of an AND gate, if there is a 0 input, then the state variable changes to X. In fact, all changes are from 0 or 1 to X. This process continues until the total state is ternary stable, i.e., $\mathbf{s} = \mathbf{S}(a{\cdot}\mathbf{s})$.

TernSim-A $(\mathbf{N}, a, b)$
$h := 0$;
$\mathbf{s}^0 := b$;
repeat
    $h := h + 1$;
    $\mathbf{s}^h := \mathrm{lub}\{\mathbf{s}^{h-1}, \mathbf{S}(a{\cdot}\mathbf{s}^{h-1})\}$;
until $\mathbf{s}^h = \mathbf{s}^{h-1}$;

$\mathbf{s}^h$ denotes the ternary vector of state values at each iteration. The final value of $\mathbf{s}^h$ is denoted by TernSim-A$(\mathbf{N}, a, b)$. Due to the monotonicity of the ternary extensions of the excitation functions, it can be shown that TernSim-A converges in at most m steps. The application of TernSim-A to the RS-latch of Figure 12 is illustrated in Figure 17, where the input is $rs = 01$ and the initial state is $qz = 01$, and $rs{\cdot}qz$ is shown at each step.

TernSim-B is the second part of the ternary simulation algorithm. It takes a network , binary input value a and ternary state value $\mathbf{s}$, and removes as much uncertainty as possible from $\mathbf{s}$, while holding the input constant at a. For example, if an AND with output X has a 0 input, then the output will change to 0. In this part, all changes are from X to 0 or 1. Brzozowski and Seger apply TernSim-B to the result of TernSim-A, but we will see that Malik's algorithm is most similar to TernSim-B by itself, applied to the vector X^m.

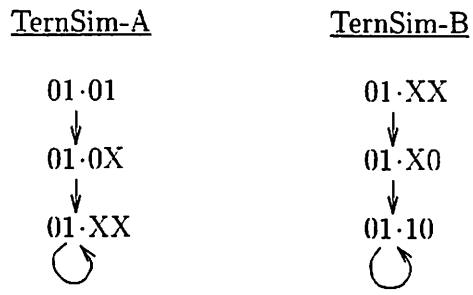

Figure 17: TernSim-A and TernSim-B in operation on the RS-latch, over states $rs{\cdot}qz$.

TernSim-B $(\mathbf{N}, a, \mathbf{s})$
$h := 0$;
$\mathbf{t}^0 := \mathbf{s}$;
repeat
    $h := h + 1$;
    $\mathbf{t}^h := \mathrm{S}(a{\cdot}\mathbf{t}^{h-1})$;
until $\mathbf{t}^h = \mathbf{t}^{h-1}$;

The final value of $\mathbf{t}^h$ is denoted by TernSim-B$(\mathbf{N}, a, \mathbf{s})$. It can be shown that TernSim-B converges in at most m steps. TernSim-B is illustrated in the right hand side of Figure 17, where the input is still $rs = 01$ and the initial state is the final value from TernSim-A, $qz = \mathrm{XX}$.
TernSim-B is computing the greatest fixed point of the excitation functions, over the domain $\{0, 1, \mathrm{X}\}^m$. As such, there are other ways of computing this fixed point [8], but we are interested not in the method, but only in the result, which can be shown computes UIN-SS.

Proposition 12 *Let N be a complete network, and a an input value. Then,*

$$\text{lub}\{\text{TernSim-B}(\mathbf{N}, a, \text{TernSim-A}(\mathbf{N}, a, b))|b \in B^m\} = \text{UIN-SS}(N, D, a).$$

Proof Brzozowski and Seger show that the result of TernSim-B, applied to the result of TernSim-A, is exactly equal to the *lub* of the set of GMW-outcome states starting in the total state $a{\cdot}b$. That is, TernSim-B$(\mathbf{N}, a,$ TernSim-A$(\mathbf{N}, a, b)) =$ lub GMW-outcome(N, a, b).
Next, note that the *lub* of a collection of sets is equal to the *lub* of the *lub* of each set in the collection. Hence, lub$\{$GMW-outcome$(N, a, b)|b \in B^m\} =$ lub$\{$lub GMW-outcome$(N, a, b)|b \in B^m\}$. The result follows then by using the above Brzozowski and Seger result along with Proposition 11. ∎
Proposition 12 provides a stepping stone to our final proof of correctness, but for efficiency, we want to skip TernSim-A and use just TernSim-B. The following theorem shows how to compute UIN-SS using just TernSim-B; this constitutes the first significant result of the paper.

Theorem 13 *Let N be a complete network, and a an input value. Then*

$$\text{TernSim-B}(\mathbf{N}, a, \mathrm{X}^m) = \text{UIN-SS}(N, D, a).$$

The proof of this theorem is presented in Section A.1. Given that the definition of UIN-SS refers to all possible initial states b, it is intuitive that TernSim-B starting from the initial state X^m computes UIN-SS. However, proving this requires the intermediate results on GMW analysis and the combination of TernSim-A and TernSim-B.
Using a complete network would seem to complicate the computation of UIN-SS due to the large number of state variables. Fortunately, we will see in Section 2.4 that it suffices to apply TernSim-B to a network containing only feedback variables.

2.4 Malik's Algorithm

Independently of the work of Brzozowski and Seger, Malik [13] devised a BDD-based algorithm for determining whether or not a network is well-behaved. In this section, we describe the core of his algorithm, which computes the response of a circuit to an input a.
The algorithm proposed by Malik works with symbolic input values; here, we begin by presenting his algorithm for a concrete input a. Before the algorithm is invoked, a vector $\mathbf{y}$ of k gate vertices, constituting a feedback vertex set, is selected to serve as the state variables of the circuit. The algorithm starts with the feedback variables initialized to X (line 2). In each round, the input a and the current values of the feedback variables are propagated through the network to compute the new value at each gate vertex (lines 5-6). At the end of each round, the values of the feedback variables are updated with the new values of the feedback gate vertices (lines 7-8). The algorithm terminates when one complete round fails to change the value of any feedback variable.

Malik's Algorithm
1 $h := 0$;
2 $\mathbf{y}^0 := \mathrm{X}^k$;
3 **repeat**
4 $\quad h := h + 1$;
5 $\quad$ for each gate vertex in topological order
6 $\quad\quad \mathbf{F}_j(a \cdot \mathbf{y}^{h-1}) := \mathbf{V}_j(a \cdot \mathbf{F}_1(a \cdot \mathbf{y}^{h-1}) \cdot \mathbf{F}_2(a \cdot \mathbf{y}^{h-1}) \cdot \ldots \cdot \mathbf{F}_{|\mathcal{G}|}(a \cdot \mathbf{y}^{h-1}))$;
7 $\quad$ for each feedback vertex
8 $\quad\quad \mathbf{y}_i^h := \mathbf{S}_i(a \cdot \mathbf{y}^{h-1})$; /* where $\mathbf{S}_i$ is the excitation equation of $\mathbf{y}_i$ */
9 **until** $\mathbf{y}^h = \mathbf{y}^{h-1}$;

It turns out that this algorithm is essentially TernSim-B, applied to a circuit with delay elements on the feedback edges what Brzozowski and Seger call a *feedback-vertex network*. Theorem 13 states that TernSim-B applied to a complete network computes UIN-SS. What remains to prove is that it is sufficient to apply TernSim-B to a feedback-vertex network. To show this, we need to reason on networks that do not include all vertices as state vertices. First, we define the depth of a vertex in a network as the longest path from an input or state vertex to that vertex.

Definition 14 Consider a vertex v in a circuit graph G, and a network N derived from G. The *depth* of v in N is:

$$depth(v) = \begin{cases} 0 & \text{if } v \text{ is an input or state vertex in } N, \\ 1 + \max\{depth(u) | (u, v) \in \mathcal{E}\} & \text{otherwise.} \end{cases}$$

Since the state vertices are required to form a feedback-vertex set, the depth of v is uniquely defined. ∎

A value for each of the state vertices and inputs uniquely determines the value for each of the remaining vertices of the circuit graph. These values are computed by the set of circuit equations of the network.

Definition 15 Let $a{\cdot}b$ be a total state of network N. Let v be a vertex with vertex function V (assumed to be defined over all circuit vertices). The *circuit equation* F of v is defined inductively on the depth of v.

$$F(a{\cdot}b) = \begin{cases} a_i & \text{if } v \text{ corresponds to input vertex } X_i, \\ b_i & \text{if } v \text{ corresponds to state variable } s_i, \\ V(a, F_1(a{\cdot}b){\cdot}F_2(a{\cdot}b){\cdot}\ldots{\cdot}F_{|\mathcal{I}|+|\mathcal{W}|+|\mathcal{G}|}(a{\cdot}b)) & \text{otherwise.} \end{cases}$$

Remember that input and state vertices have depth 0 by definition. The value of F on $a{\cdot}b$ is uniquely defined because it only depends on the values of vertices with lower depth. ∎

At first glance, the definitions of circuit equations and excitation functions appear similar. However, note that excitation functions are defined only for state vertices, whereas circuit equations are defined for all vertices. Also, for a state vertex, the circuit equation is just the state variable itself, whereas the excitation function is the composition of the circuit equations driving the state vertex. As an example, for the network in Figure 6, the circuit equation for x_1 is X_1, for y_1 is y_1, and for y_2 is y_1X_1. Now we introduce Brzozowski and Seger's concept of a reduced network, which is derived from a network by removing a state variable.

Consider a ternary network $\mathbf{N}$ with state variables $\mathbf{s}_1, \mathbf{s}_2, \ldots, \mathbf{s}_m$. $\mathbf{s}_i$ is a *legal reduction variable* if the corresponding excitation function $\mathbf{S}_i$ does not depend on any input excitation function X_j, nor on the value of $\mathbf{s}_i$ itself. Note that this specifically excludes input-delay variables as legal reduction variables. A reduced network $\dot{\mathbf{N}}$ of $\mathbf{N}$ is created by removing a legal reduction variable, and re-expressing the remaining functions in terms of the remaining variables. Without loss of generality, assume that the variable to be removed is $\mathbf{s}_m$.

Definition 16 [Brzozowski and Seger] Let $\mathbf{N}$ be a ternary network and $\mathbf{s}_m$ a legal reduction variable. Then the *reduced network* $\dot{\mathbf{N}}$ has the state variables $\dot{\mathbf{s}}_1, \dot{\mathbf{s}}_2, \ldots, \dot{\mathbf{s}}_{m-1}$, excitation functions

$$\dot{\mathbf{S}}_i(a{\cdot}\dot{\mathbf{s}}) = \mathbf{S}_i(a{\cdot}\dot{\mathbf{s}}{\cdot}\mathbf{S}_m(a{\cdot}\dot{\mathbf{s}}{\cdot}\mathrm{X})) \text{ for } 1 \le i < m,$$

and circuit equations

$$\dot{\mathbf{F}}_i(a{\cdot}\dot{\mathbf{s}}) = \begin{cases} \mathbf{S}_m(a{\cdot}\dot{\mathbf{s}}{\cdot}\mathrm{X})) & \text{if } i = m, \\ \mathbf{F}_i(a{\cdot}\dot{\mathbf{s}}{\cdot}\mathbf{S}_m(a{\cdot}\dot{\mathbf{s}}{\cdot}\mathrm{X})) & \text{otherwise.} \end{cases}$$

Note that when evaluating $\mathbf{S}_m$, the value of $\mathbf{s}_m$ is immaterial, since $\mathbf{S}_m$ is assumed to be independent of $\mathbf{s}_m$. ∎

Brzozowski and Seger state the following theorem, which says that TernSim-B gives the same result on $\mathbf{N}$ and $\dot{\mathbf{N}}$, with respect to the variables present in $\dot{\mathbf{N}}$, for an arbitrary set of legal reduction variables. Such a set has the property that any variable in the set remains a legal reduction variable even after any subset of other variables in the set has been removed.

Proposition 17 (Brzozowski and Seger) *Let $a{\cdot}\mathbf{s}$ be a total state of* N. *Let $R \subseteq \{1, 2, \ldots, m\}$, where $\forall i \in R$, s_i is a legal reduction variable and s_i is stable on $a{\cdot}\mathbf{s}$ (i.e., $\mathrm{s}_i = \mathrm{S}_i(a{\cdot}\mathbf{s})$) . Let $\dot{\mathrm{N}}$ be the reduced version of* N *with respect to the variables R. Then for every remaining variable s_i*

$$\text{TernSim-B}(\mathrm{N}, a, \mathrm{s})_i = \text{TernSim-B}(\dot{\mathrm{N}}, a, \dot{\mathrm{s}})_i.$$

As a corollary to Proposition 17, TernSim-B gives the same result when applied to N and any feedback-vertex network $\dot{\mathrm{N}}$, with respect to the feedback vertices. The next step is to extend Proposition 17 to the values computed by the circuit equations. The following result is proved by induction on vertex depth, and appears in Section A.2.

Proposition 18 *Let* N *be a complete, ternary network with m state variables. Let $a{\cdot}\mathbf{s}$ be a total state of* N. *Let $\dot{\mathrm{N}}$ be a reduced version of* N *where the eliminated variables are legal reduction variables and are stable on $a{\cdot}\mathbf{s}$. Then for $1 \leq i \leq m$*

$$\text{TernSim-B}(\mathrm{N}, a, \mathrm{s})_i = \dot{\mathbf{F}}_i(a{\cdot}\text{TernSim-B}(\dot{\mathrm{N}}, a, \dot{\mathrm{s}})).$$

That is, the value of state variable s_i found by TernSim-B *on complete network* N *is the same as that computed by the corresponding circuit equation evaluated on the result of* TernSim-B *when applied to the reduced network* $\dot{\mathrm{N}}$.

Now we are ready to state the main theorem of this section, namely that TernSim-B computes UIN-SS when applied to a feedback-vertex network, with state variables initialized to X.

Theorem 19 *Let N be a complete network and $\dot{\mathrm{N}}$ be a ternary, feedback-vertex network of N, where k is the number of feedback vertices and n is the number of inputs. Let $\dot{\mathbf{t}}^B = \text{TernSim-B}(\dot{\mathrm{N}}, a, \mathrm{X}^{k+n})$. Then*

$$\dot{\mathbf{F}}(a{\cdot}\dot{\mathbf{t}}^B) = \text{UIN-SS}(N, D, a),$$

where $\dot{\mathbf{F}}$ represents the vector of circuit equations in $\dot{\mathrm{N}}$.

Proof By Theorem 13,

$$\text{UIN-SS}(N, D, a) = \text{TernSim-B}(\mathrm{N}, a, \mathrm{X}^m).$$

Since each eliminated state variable is ternary stable on total state $a{\cdot}\mathrm{X}^m$, we can apply Proposition 18 to yield

$$\text{TernSim-B}(\mathrm{N}, a, \mathrm{X}^m) = \dot{\mathbf{F}}(a{\cdot}\dot{\mathbf{t}}^B).$$

■

There are still two minor differences to resolve between TernSim-B applied to a feedback-vertex network, the focus of Theorem 19, and Malik's algorithm. The first is that the feedback-vertex network of TernSim-B requires the presence of the n input-delay state variables, since by definition these are not legal reduction variables. However, for the special case where all state variables are initially X, the input-delay variables are not needed since in the first round of TernSim-B, the only variables to change value are the input-delay variables: they change from X to their corresponding input value. Thus, running TernSim-B on a network with the input-delay variables absent is equivalent to starting TernSim-B from the second round with these variables present.

The second difference is that Malik's algorithm gives an explicit procedure for evaluating the non-feedback values (lines 5-6), whereas TernSim-B does not specify a procedure for this. Hence, in this regard Malik's algorithm is a specialization of TernSim-B. In summary, Theorem 19 proves the correctness of Malik's algorithm for concrete input values.
It is useful to pause to review the series of steps needed to reach Theorem 19.

$$
\begin{array}{rcll}
\text{UIN-SS}(N, D, a) & = & \text{lub}\{\text{UIN-nontransient}(N, D, a, b) | b \in B^m\} & \text{Definition 7} \\
& = & \text{lub}\{\text{GMW-outcome}(N, a, b) | b \in B^m\} & \text{Proposition 11} \\
& = & \text{lub}\{\text{TernSim-B}(\mathbf{N}, a, \text{TernSim-A}(\mathbf{N}, a, b)) | b \in B^m\} & \text{Proposition 12} \\
& = & \text{TernSim-B}(\mathbf{N}, a, \mathrm{X}^m) & \text{Theorem 13} \\
& = & \dot{\mathbf{F}}(a \cdot \text{TernSim-B}(\dot{\mathbf{N}}, a, \mathrm{X}^{k+n}))) & \text{Theorem 19}
\end{array}
$$

As presented, Malik's algorithm would have to be executed 2^n times, once for each input combination, to compute UIN-SS(N, D, a) for all a. In fact, the algorithm proposed by Malik works on symbolic input values, using BDDs. In effect, all 2^n cases are handled in parallel, with possible sharing of work among the cases.
The conversion from the explicit algorithm to the symbolic algorithm is straightforward. A ternary valued circuit equation, which is updated on each iteration, is stored at each vertex. These equations are defined over the circuit inputs. Since the inputs are assumed to be binary valued, the functions to be represented are of the form $\mathbf{F} : \{0,1\}^n \rightarrow \{0,1,\mathrm{X}\}$. Such functions are in turn represented by a pair of boolean functions F^1 and F^0, where F^1 (resp. F^0) is the characteristic function of the set of inputs for which $\mathbf{F}$ evaluates to 1 (resp. 0). The set of inputs for which $\mathbf{F}$ evaluates to X is computed as $F^{\mathrm{X}} = \overline{F^1 + F^0}$. The functions F^1 and F^0 are represented by BDDs.
To start the algorithm, each input is initialized to a Boolean symbolic variable, and each circuit equation corresponding to a feedback vertex is initialized to the function X. As before, within each round, the gates are visited in topological order. For each gate, the new circuit equation is computed by combining the circuit equations of lower depth according to the Boolean operation implied by the vertex function V. For example, if V is the Boolean conjunction of two vertices represented by the equations $\mathbf{G}$ and $\mathbf{H}$, then the new circuit equation for $\mathbf{F}$ is given by $F^1 = G^1 \cdot H^1$ and $F^0 = G^0 + H^0$. The algorithm repeats until none of the circuit equations at the feedback vertices change from one round to the next. Convergence is guaranteed within k rounds, where k is the number of feedback vertices. Correctness of the symbolic algorithm follows from the fact that it is just a symbolic implementation of the explicit algorithm.

3 Output-stable Circuits

The ultimate motivation for the previous section is to classify circuits with combinational cycles that are well-behaved. With UIN-SS defined, and a proof that Malik's algorithm computes UIN-SS, it is straightforward to define the class of output-stable circuits, and show that Malik's algorithm decides this class. As a reminder, when we refer to a circuit, we are equivalently referring to its corresponding complete network. Intuitively, a circuit with up-bounded inertial delays is output-stable if for every input value, there exists a unique output value to which the circuit stabilizes in bounded time, regardless of the initial state of the circuit. Or in other words, no output evaluates to X in UIN-SS(N, D, a), for any a.

Definition 20 A circuit with maximum delay bound D is *output-stable* if for every input value a, and every output vertex j, UIN-SS$(N, D, a)_j \neq$ X. ∎

Several points are worth noting. First, as stated, this condition is vacuously true if the circuit has no input vertices. In order to avoid the vacuous case, a dummy input connected to nothing could be added to the circuit. This case will not be discussed in the sequel. Second, we saw in Section 2.2 that the property of output-stability is in fact independent of the maximum delay bound D. Third, an acyclic circuit is trivially output-stable because the outputs are functionally determined by the inputs.

Referring back to Malik's algorithm in Section 2.4, his test for output-stability is that for every output vertex j, $\mathbf{F}_j(a \cdot \mathbf{y}^M) \neq$ X, where M is the final value of h in the algorithm. By Theorem 19, $\mathbf{F}_j(a \cdot \mathbf{y}^M) =$ UIN-SS$(N, D, a)_j$, and hence given the definition of output-stability, Malik's algorithm trivially decides this class.

For the symbolic version of Malik's algorithm, when the algorithm terminates, the circuit equations $\mathbf{F}_1, \mathbf{F}_2, \ldots, \mathbf{F}_p$ of the outputs are examined. If $F_j^{\mathrm{X}} \neq 0$ for some $1 \leq j \leq p$, then any satisfying assignment of F_j^{X} gives an input valuation for which output j is not output-stable. If $F_j^{\mathrm{X}} = 0$ for all $1 \leq j \leq p$, then the circuit is output-stable, and F_j^{1} gives the Boolean function representing output j. Since F_j^{1} is represented as a BDD, which has a trivial transformation to an acyclic, multi-level circuit, then a by-product of the algorithm is an equivalent acyclic implementation of the circuit.

Malik mentions that the test for output-stability can be done with respect to a care set of inputs. Such a set expresses a constraint on the combinations of input values that can occur. If all of the satisfying assignments for F_j^{X}, for $1 \leq j \leq p$, fall outside of the set of care inputs, then the circuit is still output-stable.

Even though the class of output-stable circuits is decidable, it is intrinsically hard due to the inherent need to check stability on all 2^n input values. Malik gave a proof of this within his informal context, but here we provide a proof for our circuit model, using our terminology.

Theorem 21 *Deciding if a circuit is output-stable is co-NP-complete.*

Proof We show that deciding if a circuit is *not* output-stable is NP-complete.

Membership in NP: To show that a circuit is not output-stable, one needs to produce an input a on which an output is unstable. A guess can be verified in time polynomial in the circuit size by applying TernSim-B and examining the result, in accordance with Theorem 19.

NP-hardness: The reduction is from Boolean satisfiability. Let f be a Boolean function that we wish to check for satisfiability. Consider the circuit in Figure 18. Clearly, z is not output-stable if and only if f is satisfiable. ∎

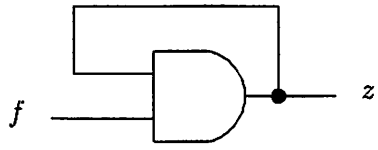

Figure 18: z is not output-stable if and only if f is satisfiable.

As a side note, the work of Kautz [11] is often cited as proving the existence of logic functions whose minimal circuit implementation using 2-input NOR gates must have combinational cycles.

Interestingly, the example circuit he gives actually fails the test for output-stability, under the UIN delay model. In particular, he suggests a class of m-input, m-output logic functions, which for $m = 3$, have the form:

$$\begin{aligned} z_1 &= \overline{x_1}x_2 + \overline{x_1}\,\overline{x_3} \\ z_2 &= \overline{x_1}\,\overline{x_2} + \overline{x_2}x_3 \\ z_3 &= x_1\overline{x_3} + \overline{x_2}\,\overline{x_3} \end{aligned}$$

He claims that the minimum 2-input NOR gate implementation is the circuit in Figure 19, which has a combinational cycle. However, for the input $x_1 = 1$, $x_2 = 1$ and $x_3 = 0$, z_3 is not uniquely determined. In particular, if we select $\{y_3\}$ as the feedback-vertex set, and apply Malik's algorithm, y_3 will be initialized to X, and remain at X, thus forcing z_3 to X. The key point is that the signal y_2 has two paths to the gate at y_3, and these two paths may have different delays; this can cause an oscillation at y_3, and hence at z_3.

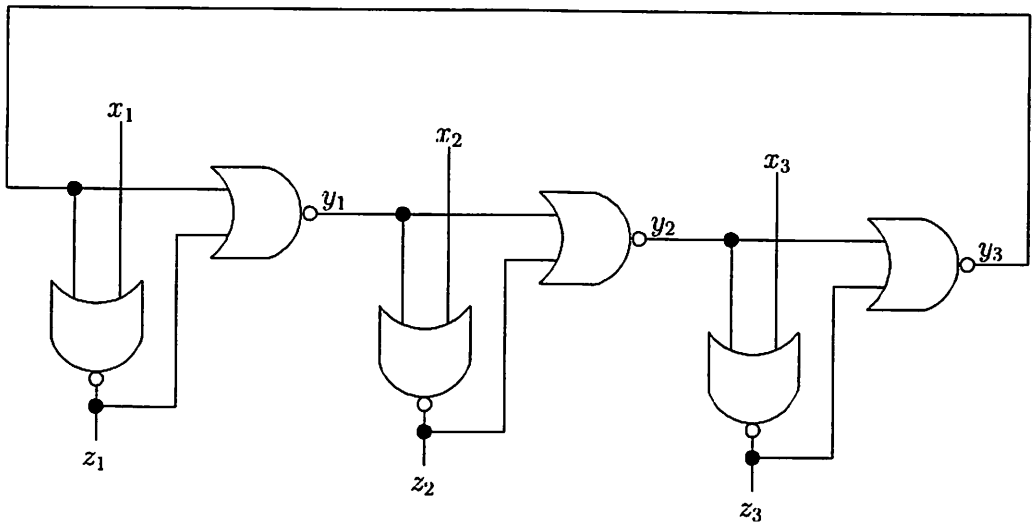

Figure 19: Kautz's circuit with a combinational cycle.

4 Extension to FSMs

The definition of output-stability can be naturally extended to circuits with flip-flops. This leads us to the notion of constructivity, a term coined by Berry to reflect the relationship between constructive logic and stable circuits [1]. Before defining constructivity, we must discuss changes to the circuit model, and the interaction with the environment.

We can view a circuit with flip-flops as depicted in Figure 20. We refer to the union of the primary inputs u and the flip-flop outputs x as the *combinational inputs*, and the union of the primary outputs z and the flip-flop inputs y as the *combinational outputs*. With respect to the definition of total state in Section 2.1, the concatenation of u and x forms the vector a, and the concatenation of w, y and z forms b. We assume that an initial set of valuations for the flip-flops is supplied.

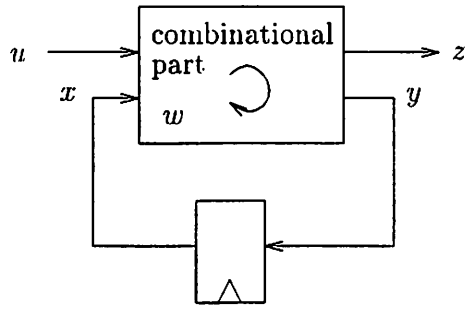

Figure 20: The variables of a circuit.

For the interaction with the environment, we assume that the circuit is driven by a global clock with period greater than $(2^m - 2)D$, so that any combinational output that will eventually stabilize, has time enough to do so. Also, the environment provides inputs and samples outputs at the clock ticks. Furthermore, it is assumed that at each clock tick, all the combinational nodes (those driven by logic gates) "forget" their values from the previous clock cycle, and thus are incapable of storing state. Thus, the state of the circuit from one clock cycle to the next is just the value of the flip-flops. This restriction is not entirely natural for hardware circuits because combinational wires can in fact store state (think of an RS-latch). However, besides being a conservative design principle, it brings the additional benefit that constructive circuits are insensitive to inputs glitching before the clock tick, since these glitches cannot change the value of the flip-flops. Hence, these circuits are closed under cascade composition. That is, if each of N_1 and N_2 is constructive, then N_1 driving N_2, or N_2 driving N_1, is also constructive. However, connecting N_1 and N_2 in a cycle is not guaranteed to preserve constructivity because new combinational cycles can be created.

We are now ready to define the class of constructive circuits. A circuit is *constructive* if for every input sequence, starting from an initial state of the flip-flops, there exists a unique combinational output sequence. That is, both the latch inputs and primary outputs must be unique. The corresponding decision problem is PSPACE-hard, as shown by a reduction from single state reachability [17].

Another way of describing constructivity is that the combinational part of the FSM is output-stable for every primary input value and reachable state of the flip-flops. Since constructivity takes state reachability into account, it is easy to see that constructivity is more permissive than requiring output-stability for *all* combinational input values. In particular, consider an output that is not stable for a given valuation of the flip-flops (i.e., a state): even if this state is not reachable, the circuit is not output-stable. However, it may still be constructive. For example, consider the circuit of Figure 21 with initial state 10. The circuit is not output-stable because when $x_1x_2 = 11$, then y_2 is unstable. On the other hand, the circuit is constructive, because starting from $y_1y_2 = 10$, the external state $x_1x_2 = 11$ cannot be reached after the first clock tick.

The motivation for constructivity comes from software, in particular from the synchronous language Esterel. The condition that combinational wires forget their state is natural in this domain because the combinational wires represent the automatic variables, which are initialized on each invocation they do not remember their previous value. Also, in this domain the flip-flops represent static variables, whose assignment should be unique — corresponding to the condition that the flip-flop inputs are unique at each clock cycle. This application is thoroughly explored in [1], where Berry

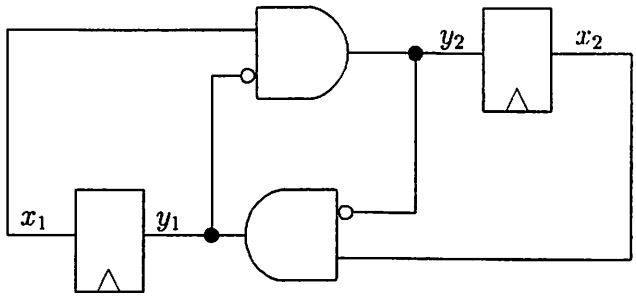

Figure 21: Constructive, but not output-stable.

defines the *constructive semantics* of pure Esterel. He shows that an Esterel program is constructive (in the sense of his semantics) if and only if the circuit derived from the program is constructive (in the sense defined here). This "full abstraction theorem" is very powerful because it provides a means of automatically classifying Esterel programs as legal (i.e., constructive) or illegal. The fact that the theorem connects the abstract world of programs to the concrete world of circuits with delays also points to the universality of constructivity.

Next, we describe a BDD-based algorithm that decides the class of constructive circuits. Roughly, in the first step, we calculate all the combinational input values that cause an unstable combinational output. Then in the second step, we perform FSM reachability to determine if any of these "bad" combinational inputs are possible. In more detail, in the first step, we use Malik's algorithm to implicitly test output-stability for each of the combinational input values. A by-product of this algorithm is a pair of Boolean functions $F_i^1(u,x)$ and $F_i^0(u,x)$ for each combinational output i. Then we compute

$$\mathit{unstableDomain}(u,x) = \sum_i F_i^X(u,x) = \sum_i \overline{(F_i^1(u,x) + F_i^0(u,x))}.$$

The set *unstableStates* is just the projection of *unstableDomain* onto the flip-flop inputs:

$$\mathit{unstableStates}(x) = \exists u\ \mathit{unstableDomain}(u,x).$$

The next step is to perform symbolic reachability analysis. We would like to derive a next state *function* for each flip-flop, so that we have the flexibility of employing reachability methods that exploit the determinism of functions (as opposed to the nondeterminism of relations). However, in general, the next states are not functionally determined. For example, in Figure 21, state 11 has four possible next states. Nonetheless, there is a way around this contradiction: as long as we limit ourselves to the *stableStates* (the complement of *unstableStates*), then $F^1(u,x)$ gives the correct value for the next state. Thus, we use the function $F^1(u,x)$ corresponding to each flip-flop as the next state function.

Reachability then works as follows. Before each BFS step of reachability, the set of states to be explored is intersected with the set of *unstableStates*. If this intersection is non-empty, then "not constructive" is returned. Otherwise, it is safe to perform the next reachability step. If the fixed point is reached, then "constructive" is returned. In this case, the BDDs for the functions $F_i^1(u,x)$ can be used directly to derive an acyclic combinational part of the circuit.

The above algorithm first appeared in [18]. More recently, Namjoshi and Kurshan proposed a more efficient algorithm based on satisfiability [14].

5 Related Work

We divide related work into three categories.

1. Circuit analysis: These works provide techniques to analyze the behavior of circuits, without attempting to classify well-behaved circuits.
2. Circuit classification: These works classify circuits based on their well-behavedness.
3. FSM extraction: These works provide algorithms to extract finite state machines from transistor-level netlists. Their classification of well-behaved circuits is implicit in the result of their algorithms.

5.1 Circuit Analysis

Brzozowski and Seger [4] study asynchronous circuits under various delay models. In particular, for the up-bounded inertial delay model, they present two methods to analyze the behavior of a circuit, as discussed in Section 2. This work does not classify well-behaved circuits. However, it is pivotal to our research because it provides the delay model and techniques upon which we define and analyze well-behaved circuits.

Burch *et al.* [5] model logic gates by ternary-valued relations, where the third value, $\perp$, represents an oscillating or intermediate voltage. By using $\perp$, oscillating behaviors caused by combinational cycles are preserved when gates are composed (by taking the intersection of their corresponding ternary-valued relations). This is in contrast to the use of Boolean relations to model gates, where oscillating behaviors "disappear" when gates are composed. Burch uses the ternary model to solve various substitution and rectification problems for gate-level circuits. However, they do classify well-behaved circuits, or relate the ternary model to a delay model.

Maler and Pnueli [12] provide an elegant method to translate asynchronous circuits, described at the gate-level, into timed automata. They use a delay model equivalent to the bi-bounded inertial delay model of Brzozowski and Seger; this is more general than the up-bounded inertial delay model we use because it allows the specification of a lower bound on the delay.
For each gate in the circuit, they introduce a delay element with an associated timer (or clock). They prove that the resulting timed automaton has the same I/O behavior over time as the original circuit, for the given delay model. Using the timed automaton, they are able to perform state reachability and solve several synthesis problems. However, they do not address the well-behavedness problem, nor is it immediate how this problem can be solved within their framework.
The use of timers complicates the analysis considerably. We show for the up-bounded inertial delay model that output-stability is independent of the delay bounds in the circuit, and hence translating to timed automata is unnecessary. Whether this is necessary for the bi-bounded inertial delay model remains an open question.

5.2 Circuit Classification

Malik [13] inspired our research. He notes that combinational cycles do arise in practice, but that no method for rigorously analyzing such circuits had ever been proposed. To remedy this situation, he introduced the class of "combinational" circuits to capture well-behavedness, and proposed using ternary simulation to decide whether or not a circuit is combinational. However, his work is not based on a delay model. We show that his procedure classifies circuits under the UIN delay model.

Halbwachs and Maraninchi [9] define a class of well-behaved circuits called *consistent* circuits. Basically, they view a circuit as a system of Boolean equations (one equation for each gate), and consider the solutions of this system. For a given input valuation, if the system has at least one solution, and the output has the same value for all solutions, then the circuit is deemed *weakly consistent.* As a special case, if there is exactly one solution, then the circuit is *strongly consistent.* This class is not comparable to our class of output-stable circuits. The circuit in Figure 1 is output-stable, but it is not weakly consistent because there is no consistent assignment to variable y when $x = 0$. On the other hand, the circuit in Figure 2 is strongly consistent, but not output-stable. It is strongly consistent because 0 is the only consistent value for y in the system of equations. It is not output-stable because when $x = 0$, y can in fact oscillate.

5.3 FSM Extraction

FSM extraction techniques in the literature are less formal. First, they do not address the underlying delay model. They simply accept as input to their own tools the output of a circuit extraction tool, like TRANALYZE [3] or ANAMOS [2], without formal regard to how the tool does the extraction. Second, they do not formally classify those circuits that can be represented by an FSM (i.e., are well-behaved), and those that cannot. Instead, they implicitly define well-behavedness by the result of their extraction algorithms: if the algorithm is successful in extracting an FSM, then the circuit can be considered well-behaved, otherwise not. Third, there is no proof that, when the algorithm is able to extract an FSM, that this FSM has the same behavior as the original circuit.
Shiple's thesis [17] provides detailed descriptions of the works of Singh and Subrahmanyam [19], Pandey *et al.* [15], and Kam and Subrahmanyam [10]. In the interest of space, here we only provide a critique of the paper by Singh and Subrahmanyam. They propose a method to extract FSMs, at the Boolean function level, from transistor netlists. They employ TRANALYZE as a preprocessor, which generates a network of zero-delay logic blocks and unit-delay elements, from a transistor netlist. The unit-delay elements are introduced by TRANALYZE to break feedback loops and to model charge storage nodes. An assignment of values to the unit-delay elements is called a *configuration*; this is the state of the network. TRANALYZE uses a 4-valued algebra in deriving logic gates from transistors. Ironically though, it uses a 2-valued algebra to simplify the logic gates that compose a zero-delay logic block. Hence, the expression $y \cdot \overline{y}$ is simplified to 0. For this reason, the circuit in Figure 2 is simplified by TRANALYZE to $z = x$, and thus Singh classifies it as well-behaved. This is counter to our classification.
This work also suffers from two other weaknesses. The first is embodied in the critical assumption that if there exists a stable binary configuration corresponding to a given input, then the circuit will settle in that configuration when the input is applied. This assumption ignores the possibility that

the circuit may settle into an indefinite, race-free oscillation. The second weakness is that for a given input and current configuration, only one next configuration is possible. This is inherent in the fact that TRANALYZE uses functions, and not relations, to model the outputs of zero-delay logic blocks. Thus, different next configurations arising from critical races cannot be modeled.

6 Summary and Future Work

We have defined a formal class of gate level circuits whose outputs settle to a unique value, for every input. This definition is grounded in the up-bounded inertial delay model. We have shown that Malik's algorithm of BDD-based ternary simulation decides this class. This result follows easily once we proved that Malik's algorithm exactly computes the steady-state behavior of a circuit with UIN delays. Given the large semantic gap between the definition of UIN delays and ternary simulation, we have turned to Brzozowski and Seger's extensive theory on asynchronous circuits to bridge this gap.

For circuits that are indeed output-stable, an important by-product of the decision procedure is an acyclic circuit having the same input/output behavior. This is an important feature, as many high-level EDA tools do not accept circuits with combinational cycles.

We extended the theory of output-stability to circuits containing flip-flops. The resulting class of constructive circuits distinguishes between behavior in the reachable and unreachable state space, but does not permit combinational wires to hold state. However, due to this last restriction, the inputs can have glitches, and consequently, this class is closed under cascade composition. This class exactly coincides with the class of constructive Esterel programs.

There are several directions for future work. The first is to extend the analysis to allow transistors and tri-state devices. The second direction, for output-stable circuits, is to derive acyclic implementations that preserve as much structure of the original circuit as possible, so that the changes are not too drastic. The third direction is to explore decision procedures for output-stability under different delay models, such as bi-bounded inertial delay and ideal delay. Given the exact relationship we have shown between the UIN delay model and ternary simulation, if a different delay model yields different steady-state behavior than UIN delays, then ternary simulation definitely cannot be used. Intuitively, there needs to be a match between the choice of delay model and the algorithm used for classification. For ternary simulation, the outcome is independent of the order in which the gates are evaluated. In practice, this means that an event can be "lost". For example, an event can occur at a gate input that would make that gate output unstable. However, that gate may not be evaluated immediately, and in the meantime, the evaluation of other gates could cause the event to disappear, thus making the gate stable again. This situation is analogous to what happens in the UIN delay model, when an event can be lost because it is shorter than the inertial delay.

Contrast this situation to using an ideal delay model, where every event, regardless of its duration, must be propagated. To keep track of the events to be propagated, a count must be maintained for each variable, giving the number of pending events. Clearly ternary simulation is not compatible with the concept of pending events.

Acknowledgements

The authors would like to thank Stephen Edwards for his mnay valuable suggestions to enhance the presentation of this work.

A Proofs

A.1 Proof of Theorem 13

We first prove the following two lemmas.

Lemma 22 *Let N be a complete network. Then $\forall a$, $\exists b$ such that* TernSim-A$(\mathbf{N}, a, b) = \mathrm{X}^m$.

Proof Let $a \in B^n$. Recall in TernSim-A that

$$\begin{aligned} \mathbf{s}_i^0 &:= b_i, \text{ and} \\ \mathbf{s}_i^h &:= \mathrm{lub}\{\mathbf{s}_i^{h-1}, \mathbf{S}_i(a \cdot \mathbf{s}^{h-1})\}. \end{aligned}$$

Construct b as follows. Consider first an input-delay vertex labeled s_i that is driven by input X_j. Let $b_i = \overline{a_j}$. Since $\mathbf{S}_i = X_j = a_j$, then:

$$\begin{aligned} \mathbf{s}_i^0 &= b_i = \overline{a_j}, \text{ and} \\ \mathbf{s}_i^1 &= \mathrm{lub}\{\mathbf{s}_i^0, \mathbf{S}_i(a \cdot \mathbf{s}^0)\} = \mathrm{lub}\{\overline{a_j}, a_j\} = \mathrm{X}. \end{aligned}$$

Hence, after the first iteration, each input-delay variable is set to X.
Next consider a gate variable s_i driven by wire variables $w_1, w_2, \ldots, w_k$. Choose an arbitrary initial assignment for $w_1, w_2, \ldots, w_k$, say $b_1, b_2, \ldots, b_k$. (Note that w_j, for $1 \leq j \leq k$, only fans out to gate s_i since N is complete, so no other gate is constraining the value of w_j.) If $\mathbf{S}_i(a \cdot b) = \alpha$, then let $b_i = \overline{\alpha}$. Then,

$$\begin{aligned} \mathbf{s}_i^0 &= b_i = \overline{\alpha}, \text{ and} \\ \mathbf{s}_i^1 &= \mathrm{lub}\{\mathbf{s}_i^0, \mathbf{S}_i(a \cdot \mathbf{s}^0)\} = \mathrm{lub}\{\overline{\alpha}, \alpha\} = \mathrm{X}. \end{aligned}$$

Hence, after the first iteration, each gate variable is set to X.
At this point, we have constructed the initial value b_i for each variable, but we have not shown that the wire variables are forced to X in TernSim-A. Consider a wire variable s_i driven by gate variable or input-delay variable s_j. Then,

$$\mathbf{s}_i^2 = \mathrm{lub}\{\mathbf{s}_i^1, \mathbf{S}_i(a \cdot \mathbf{s}^1)\} = \mathrm{lub}\{\mathbf{s}_i^1, \mathbf{s}_j^1\} = \mathrm{lub}\{\mathbf{s}_i^1, \mathrm{X}\} = \mathrm{X}.$$

That is, after iteration 1, every gate is driven to X, so that after iteration 2, every wire variable is guaranteed to be driven to X. ∎

Lemma 23 *In* TernSim-B, *let $\mathbf{t}_1^0$ and $\mathbf{t}_2^0$ be two different starting points. If $\mathbf{t}_1^0 \sqsubseteq \mathbf{t}_2^0$, then $\mathbf{t}_1^h \sqsubseteq \mathbf{t}_2^h$, for all $h \geq 0$, where h is the iteration number in* TernSim-B. *That is,* TernSim-B *is monotonic.*

Proof By induction on h. The basis is provided by the hypothesis. Suppose $\mathbf{t}_1^h \sqsubseteq \mathbf{t}_2^h$. Then $\mathbf{t}_1^{h+1} = \mathbf{S}(a \cdot \mathbf{t}_1^h) \sqsubseteq \mathbf{S}(a \cdot \mathbf{t}_2^h) = \mathbf{t}_2^{h+1}$, where the inequality follows by the monotonicity of S. (Note that TernSim-B is not guaranteed to converge for an arbitrary $\mathbf{t}_1^0$ or $\mathbf{t}_2^0$, because neither $\mathbf{t}_1^0$ nor $\mathbf{t}_2^0$ is assumed to be ternary stable.) ∎

To prove Theorem 13, we use Proposition 12 for the definition of UIN-SS(N, D, a).

Proposition 24 *Let N be a complete network, and s_i a state vertex. Then, for all i,*

$$\text{TernSim-B}(\mathbf{N}, a, \mathrm{X}^m)_i = \text{lub}\{\text{TernSim-B}(\mathbf{N}, a, \text{TernSim-A}(\mathbf{N}, a, b)) | b \in B^m\}_i.$$

Proof The proof is by case on the value $\mathbf{s}_i$. First, suppose $\text{TernSim-B}(\mathbf{N}, a, \mathrm{X}^m)_i = \mathrm{X}$. Combining this with Lemma 22, we have

$$\begin{aligned}
&\text{lub}\{\text{TernSim-B}(\mathbf{N}, a, \text{TernSim-A}(\mathbf{N}, a, b)) | b \in B^m\}_i \\
&= \text{lub}\{\text{TernSim-B}(\mathbf{N}, a, \mathrm{X}^m), \{\text{TernSim-B}(\mathbf{N}, a, \text{TernSim-A}(\mathbf{N}, a, b)) | b \in B^m\}\}_i \\
&= \text{lub}\{\mathrm{X}, \{\text{TernSim-B}(\mathbf{N}, a, \text{TernSim-A}(\mathbf{N}, a, b)) | b \in B^m\}\}_i \\
&= \mathrm{X} \\
&= \text{TernSim-B}(\mathbf{N}, a, \mathrm{X}^m)_i.
\end{aligned}$$

Next, suppose $\text{TernSim-B}(\mathbf{N}, a, \mathrm{X}^m)_i = d \in \{0, 1\}$. Since for any b, $\text{TernSim-A}(\mathbf{N}, a, b) \sqsubseteq \mathrm{X}^m$, then by Lemma 23,

$$\text{TernSim-B}(\mathbf{N}, a, \text{TernSim-A}(\mathbf{N}, a, b))_i \sqsubseteq \text{TernSim-B}(\mathbf{N}, a, \mathrm{X}^m)_i = d.$$

Since d is binary, then $\text{TernSim-B}(\mathbf{N}, a, \text{TernSim-A}(\mathbf{N}, a, b))_i = d$, and since b was arbitrary, then

$$\text{lub}\{\text{TernSim-B}(\mathbf{N}, a, \text{TernSim-A}(\mathbf{N}, a, b)) | b \in B^m\}_i = d.$$

∎

A.2 Proof of Proposition 18

Proof Let $\mathbf{t}_i^B = \text{TernSim-B}(\mathbf{N}, a, \mathbf{s})_i$ and $\dot{\mathbf{t}}_i^B = \text{TernSim-B}(\dot{\mathbf{N}}, a, \dot{\mathbf{s}})_i$. Consider a vertex v_i of the circuit graph. The proof is by induction on the depth of v_i in $\dot{\mathbf{N}}$.

Base: $depth(v_i) = 0$: Since the depth of v_i is zero, it has a corresponding state variable, say $\mathbf{s}_i$, in $\dot{\mathbf{N}}$. By Proposition 17, $\mathbf{t}_i^B = \dot{\mathbf{t}}_i^B$. By Definition 15, for a vertex of depth 0, $\dot{\mathbf{F}}_i(a \cdot \dot{\mathbf{t}}^B) = \dot{\mathbf{t}}_i^B$.

I.H.: For all $j < k$, where $depth(v_i) = j$, $\mathbf{t}_i^B = \dot{\mathbf{F}}_i(a \cdot \dot{\mathbf{t}}^B)$.

I.S.: Suppose vertex v_i has depth k, with vertex function $\mathbf{V}_i$, and corresponding state variable $\mathbf{s}_i$ in $\mathbf{N}$. Since the result of TernSim-B is ternary stable,

$$\mathbf{t}_i^B = \mathbf{S}_i(a \cdot \mathbf{t}^B).$$

In a complete network, the excitation function and vertex function are the same for any vertex. Hence,

$$\begin{aligned}
\mathbf{t}_i^B &= \mathbf{V}_i(a \cdot \mathbf{t}^B) \\
&= \mathbf{V}_i(a \cdot \mathbf{t}_1^B \cdot \mathbf{t}_2^B \cdot \ldots \cdot \mathbf{t}_m^B).
\end{aligned}$$

By the induction hypothesis,

$$\mathbf{t}_i^B = \mathbf{V}_i(a \cdot \dot{\mathbf{F}}_1(a \cdot \dot{\mathbf{t}}^B) \cdot \dot{\mathbf{F}}_2(a \cdot \dot{\mathbf{t}}^B) \cdot \ldots \cdot \dot{\mathbf{F}}_m(a \cdot \dot{\mathbf{t}}^B)).$$

Finally, by the definition of circuit equation,

$$\mathbf{t}_i^B = \dot{\mathbf{F}}_i(a \cdot \dot{\mathbf{t}}^B).$$

■

References

[1] G. Berry. *The Constructive Semantics of Esterel.* Draft, version 3.0, available at www.esterel.org, July 1999.

[2] R. E. Bryant. Boolean analysis of MOS circuits. *IEEE Trans. Computer-Aided Design*, 6(4):634 649, July 1987.

[3] R. E. Bryant. Extraction of gate level models from transistor circuits by four-valued symbolic analysis. In *Proc. Int'l Conf. on Computer-Aided Design*, pages 350 353, Nov. 1991.

[4] J. A. Brzozowski and C.-J. H. Seger. *Asynchronous Circuits.* Springer-Verlag, New York, 1995.

[5] J. R. Burch, D. Dill, E. Wolf, and G. D. Micheli. Modeling hierarchical combinational circuits. In *Proc. Int'l Conf. on Computer-Aided Design*, pages 612–617, Nov. 1993.

[6] R. de Simone. Note: A small hardware bus arbiter specification leading naturally to correct cyclic description. Internal note: http://www-sop.inria.fr/meije/verification/esterel/doc.html, 1996.

[7] E. B. Eichelberger. Hazard detection in combinational and sequential switching circuits. *IBM Journal of Research and Development*, 9(2):90–99, March 1965.

[8] M. J. C. Gordon. *The Denotational Description of Programming Languages.* Springer-Verlag, New York, 1979.

[9] N. Halbwachs and F. Maraninchi. On the symbolic analysis of combinational loops in circuits and synchronous programs. In *Euromicro'95*, September 1995. Como, Italy.

[10] T. Kam and P. A. Subrahmanyam. Comparing layouts with HDL models: A formal verification technique. *IEEE Trans. Computer-Aided Design*, 14(4):503 509, Apr. 1995.

[11] W. H. Kautz. The necessity of closed circuit loops in minimal combinational circuits. *IEEE Trans. Comput.*, 19(2):162 164, Feb. 1970.

[12] O. Maler and A. Pnueli. Timing analysis of asynchronous circuits using timed automata. In P. E. Camurati and H. Eveking, editors, *Proceedings of the Conference on Correct Hardware Design and Verification Methods*, volume 987 of *LNCS*, pages 189–205, Frankfurt/Main, Germany, Oct. 1995. Springer-Verlag.

[13] S. Malik. Analysis of cyclic combinational circuits. *IEEE Trans. Computer-Aided Design*, 13(7):950–956, July 1994.

[14] K. J. Namjoshi and R. P. Kurshan. Efficient analysis of cyclic definitions. In N. Halbwachs and D. Peled, editors, *Proc. Computer Aided Verification*, LNCS, pages 394–405, Trento, Italy, July 1999. Springer-Verlag.

[15] M. Pandey, A. Jain, R. E. Bryant, D. Beatty, G. York, and S. Jain. Extraction of finite state machines from transistor netlists by symbolic simulation. In *Proc. Int'l Conf. on Computer Design*, pages 596–601, Oct. 1995.

[16] G. D. Plotkin. LCF as a programming language. *Theoretical Computer Science*, 5(3):223 256, 1977.

[17] T. R. Shiple. *Formal Analysis of Synchronous Circuits.* PhD thesis, U.C. Berkeley, Electronics Research Laboratory, College of Engineering, University of California, Berkeley, CA 94720, Oct. 1996. Memorandum No. UCB/ERL M96/76.

[18] T. R. Shiple, G. Berry, and H. Touati. Constructive analysis of cyclic circuits. In *Proc. European Design and Test Conference*, pages 328–333, Mar. 1996.

[19] K. J. Singh and P. A. Subrahmanyam. Extracting RTL models from transistor netlists. In *Proc. Int'l Conf. on Computer-Aided Design*, pages 11–15, Nov. 1995.

[20] L. Stok. False loops through resource sharing. In *Proc. Int'l Conf. on Computer-Aided Design*, pages 345 348, Nov. 1992.

[21] Y. Watanabe and R. K. Brayton. The maximum set of permissible behaviors for FSM networks. In *Proc. Int'l Conf. on Computer-Aided Design*, pages 316 320, Nov. 1993.