International Journal of Foundations of Computer Science

COVERING OF TRANSIENT SIMULATION OF FEEDBACK-FREE CIRCUITS BY BINARY ANALYSIS

YULI YE
Department of Computer Science, University of Toronto
Toronto, Ontario M5S 3G4, Canada

and

JANUSZ BRZOZOWSKI
David R. Cheriton School of Computer Science
University of Waterloo
Waterloo, Ontario N2L 3G1, Canada

Received (received date)
Revised (revised date)
Communicated by Editor's name

ABSTRACT

Transient simulation of a gate circuit is an efficient method of counting signal changes occurring during a transition of the circuit. It is known that this simulation covers the results of classical binary analysis, in the sense that all signal changes appearing in binary analysis are also predicted by the simulation. For feedback-free circuits of 1- and 2-input gates, it had been shown that the converse also holds, if wire delays are taken into account. In this paper we generalize this result. First, we prove that, for any feedback-free circuit N of arbitrary gates, there exists an *expanded circuit* $\hat{N}$, constructed by adding a number of delays to each wire of N, such that binary analysis of $\hat{N}$ covers transient simulation of N. For this result, the number of delays added to a wire is obtained from the transient simulation. Our second result involves adding only one delay per wire, which leads to the *singular circuit* $\dot{N}$ of N. This result is restricted to circuits consisting only of gates realizing functions from the set $\mathcal{H} = \{$IDENTITY, AND, OR, XOR$\}$, functions obtained by complementing any number of inputs and/or the output of a function from $\mathcal{H}$, and FORKs. The numbers of inputs of the AND, OR and XOR gates are arbitrary, and all functions of two variables are included. We show that binary analysis of such a circuit $\dot{N}$ covers transient simulation of N. We also show that this result cannot be extended to arbitrary gates, if we allow only a constant number of delays per wire.

Keywords: analysis, binary, circuit, hazard, simulation, transient.

1. Introduction

Hazard detection is important in circuit design, because unwanted hazard pulses may affect the correctness of a circuit's operation and may increase computation time and energy consumption. Thus, after a circuit is designed, it should be analyzed for hazards to avoid computation errors. One obvious method for finding

hazards is to use classical binary analysis [3, 9, 10]. In such an analysis, one examines all behaviors of the circuit, under all possible delay distributions. Some of these behaviors may be free of hazards, while others may contain them. If we know that no hazards occur in any case, then the circuit is safe. If there are hazards, then either the design needs to be modified to remove them, or a more accurate delay analysis must be performed to prove that the hazards are very unlikely to occur.

The problem with binary analysis is that it is exponential in the number of gates. To overcome this, several multi-valued algebras have been proposed for hazard detection over the years; see [5] for a recent survey on this topic. Brzozowski and Ésik [4] generalized these algebras to an infinite-valued algebra, which we call the *algebra of transients*. Simulation in this algebra, called *transient simulation*, detects all hazards [2], permits us to count the number of signal changes occurring in a circuit under worst-case conditions, and is polynomial in the number of gates. Also, all the previously defined multi-valued algebras are quotients of this algebra [4, 5].

To be sure that a simulation method gives the desired results, it is necessary to prove that it is correct in some sense. The obvious comparison to be made is to binary analysis. We say that an analysis method A *covers* analysis method B if all signal changes predicted by B are also discovered by A. Of the several multi-valued simulations proposed for hazard detection, only Eichelberger's ternary simulation [6] has been completely characterized [3] in terms of binary analysis. Brzozowski and Gheorghiu [2] have proved that transient simulation covers binary analysis; it is easy, however, to find examples of circuits in which transient simulation predicts more changes than binary analysis.

A circuit is modeled by a *network* of gates and wires [3]. It appears that binary analysis *can* cover simulation, if wire delays are taken into account. Gheorghiu [7, 8] proved this for feedback-free networks of 1- and 2-input gates. Here, we first show that, for any feedback-free network N of arbitrary gates, there exists an *expanded network* $\hat{N}$, constructed by adding several delays to each wire of N, such that binary analysis of $\hat{N}$ covers transient simulation of N. Our second result involves adding only one delay per wire to a feedback-free network N (as was done in [7, 8]), thus producing a *singular network* $\dot{N}$, but is restricted to circuits constructed with gates realizing Boolean functions from the set $\mathcal{G} = \mathcal{H} \cup \overline{\mathcal{H}} \cup \{\text{FORK}\}$, where $\mathcal{H} = \{$IDENTITY, AND, OR, XOR$\}$, $\overline{\mathcal{H}}$ is the set of functions obtained by complementing any number of inputs and/or the output of functions from $\mathcal{H}$, and AND, OR, and XOR can have arbitrary numbers of inputs. We show that binary analysis of a singular network $\dot{N}$ of N covers transient simulation of N. Since the set of functions that we can handle includes all 1- and 2-variable functions, our result is a generalization of that of [7, 8]. Thus our results show that, for circuits constructed with common gates, transient simulation is an attractive alternative to binary analysis.

In addition to the positive results above, we show that there exist networks in which binary analysis does not cover simulation, if each wire has a constant number of delays. The counterexample network contains a gate realizing the Boolean function $x_1 * \overline{x_3} + x_2 * x_3$, where $*$ and $+$ are Boolean multiplication and addition, respectively.

We consider only feedback-free circuits in this paper, for several reasons. Circuits with feedback correspond to graphs with cycles, and their simulation in the infinite algebra of transient may not terminate; that is, such circuits may oscillate [3], and this adds to the complexity of the problem. It is possible to simulate general circuits in finite algebras [4], and then the simulation always terminates. But it is then necessary to use of another algorithm, Algorithm B, and the nature of the problem changes. Even in the apparently simple case of feedback-free networks, it is nontrivial to show that binary analysis of a circuit with added delays covers the simulation of the original circuit. Thus our paper is a contribution to a work in progress, and the case of circuits with feedback remains open.

The paper is organized as follows. In Section 2 we introduce the network model of gate circuits. We decribe the classical binary analysis in Section 3. In Section 4, we present the algebra of transients, and in Section 5 we introduce *transient networks* and their *transient simulation* based on this algebra. Section 6 introduces *wire-delay extensions* and two of their variants, *expanded networks* and *singular networks*. We then present in Section 7 some additional properties of the algebra of transients, and in Section 8, we introduce signal changes and their properties in gate circuits. Finally, in Section 9, we prove that transient simulation of N can be covered by binary analysis of some wire-delay extension of N.

2. Gate Circuits and Network Models

Our mathematical model of a gate circuit is based on those in [3, 7, 8], with some small differences. A gate circuit consists of *(external) input terminals* (or simply *inputs*), *input gates* (one for each input), *(external) output terminals* (or simply *outputs*), *gates, forks,* and *wires.* An example is shown in Fig. 1, where 1 and 2 are input terminals, 3 and 4 are input gates, each of which is connected to an input terminal, 10 and 11 are output terminals, 6 is an inverter or NOT gate, 7 is a 3-input AND gate, and 9 is a 2-input OR gate. Input gate 3 is connected to gates 6 and 7; the branching point 5 is called a *fork*. For mathematical convenience, we consider forks to be gates, and we refer to them as FORK gates; all other gates are called *logic* gates. We assume that each logic gate has one or more inputs and exactly one output.[a] In contrast to this, each FORK has exactly one input and two or more outputs. The FORK gates of our example circuit are shown explicitly in Fig. 2. A wire connects two points; multiple connections must be done with forks.

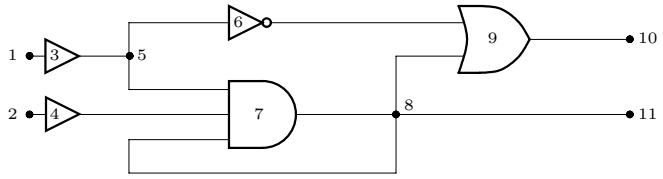


Fig. 1. A gate circuit.

[a]The model can be easily extended to multiple-output gates.

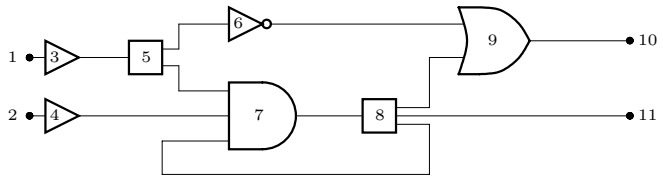


Fig. 2. A circuit with FORK gates.

Our formal model of a circuit is called a *network*. Input gates always correspond to identity functions; hence they are equivalent to delays. In general, we associate some delays with wires, but we never add delays to the wires connecting input terminals to input gates. *From now on, when we talk about* `wires`, *we ignore all the wires connecting external inputs to input gates.* Thus every wire connects a gate output to either a gate input, or to an output terminal. Every wire can have $k \geq 0$ delays, but for now *we assume all wires have zero delays*. If C is a circuit, let $X = \{x_1, \ldots, x_p\}$ be its set of inputs, $Y = \{y_1, \ldots, y_n\}$, its set of gates (including FORKS), and $Z = \{z_1, \ldots, z_q\}$, its set of outputs. Note that X, Y and Z are pairwise disjoint. We also label all the wires by variables from the set $W = \{w_1, \ldots, w_m\}$. Variables in $X \cup Z$ are called *external* variables, whereas those in $Y \cup W$ are *internal*. Figure 3 shows our example circuit with all components labeled as described above.

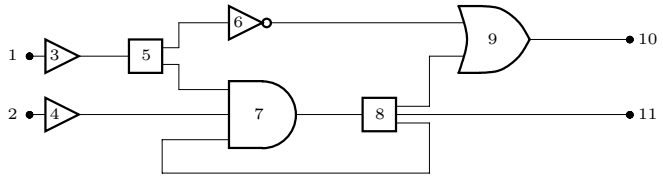


Fig. 3. Circuit variables.

A *directed graph*, or *digraph* [1], $D = (V, E)$, consists of a set V of vertices and a set $E \subseteq V \times V$ of directed edges. We now introduce the *network graph* which describes the connections among the components of a circuit.

Definition 1 *The* `network graph` *of a gate circuit is a digraph $D = (V, E)$, where $V = Y \cup Z$ is the set of vertices, and $E = W$, the set of edges. Associated with each input gate is its corresponding external input. There is exactly one outgoing edge from each input gate to a gate input, and one outgoing edge from each gate output to a gate input or an external output.*

We assume that a network graph is connected; a network that is not connected can be treated as several independent connected networks. In a digraph, a vertex is a *source* if it has in-degree zero, a *sink* if it has out-degree zero, and an *internal vertex* if it is neither a source nor a sink. In a network graph, input gates are sources, outputs are sinks, and other gates are internal vertices. The *distance* of a vertex is the length of the longest directed path from a source to that vertex.

The network graph of our example circuit is shown in Fig. 4, where we have used the labeling of Fig. 3. The external input associated with each input gate is indicated in brackets. The distance of y_5 is 2, and the distance of y_6 is infinity.

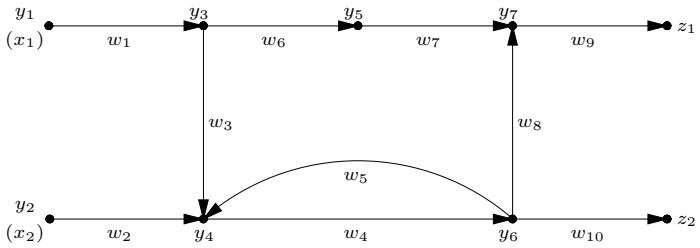


Fig. 4. Network graph.

From now on, we assume that the network graph has been labeled with vertex and edge variables, and we do not distinguish between an edge and its wire variable, or between a vertex and its variable. In any digraph, an edge e from vertex t to vertex h can be represented by the ordered pair $e = (t, h)$. Then t is the *tail* of e, and h is its *head*. We also view a wire as an edge $w = (t(w), h(w))$, and treat $t(w)$ and $h(w)$ as both vertices and vertex variables.

The network graph of a gate circuit captures the connection structure of the circuit, but does not include the gates' functions; these are added next.

Definition 2 *The* `excitation` *and* `output functions` *of a network are defined as follows:*

- *For a* FORK y_i*, the* `excitation` *is the identity function* $Y_i = t(w_j)$*, where* $h(w_j) = y_i$*.*
- *For an input gate* y_i*, the* `excitation` *is the identity function* $Y_i = x_i$*, where* x_i *is the external input associated with* y_i*. Thus an input gate is equivalent to a delay.*
- *For any other gate* y_i *with input wires* $w_{i_1}, \ldots, w_{i_k}$*, the* `excitation` *is* $Y_i = f(t(w_{i_1}), \ldots, t(w_{i_k}))$*, where* f *is the Boolean function of the gate.*
- *The* `output function` *of output* z_i *is the identity function* $z_i = t(w_j)$*, where* $h(w_j) = z_i$*.*

For our example circuit of Fig. 2 and its network graph of Fig. 4, we denote Boolean AND(x_1, x_2) by $x_1 * x_2$, OR(x_1, x_2) by $x_1 + x_2$, and NOT(x) by $\overline{x}$. The excitation and output functions of this network are: $Y_1 = x_1$, $Y_2 = x_2$, $Y_3 = y_1$, $Y_4 = y_2 * y_3 * y_6$, $Y_5 = \overline{y_3}$, $Y_6 = y_4$, $Y_7 = y_5 + y_6$, $z_1 = y_7$, $z_2 = y_6$.

Definition 3 *A* `network` *is a network graph* (V, E) *together with an assignment of excitation functions to all of its gate variables, and output functions to its output terminals, as in Definition 2.*

Variables to which excitations are assigned are the *state variables* of the network.[b] Note that there are no input or output excitations in the network; input values are assumed to be supplied directly by the environment.

[b] In this section, only gates are state variables, and we denote them by y_i; later, however, we add delays, which are also state variables. All the state variables are then denoted by s_i.

Definition 4 *A network is* `feedback-free` *if its network graph is acyclic.*

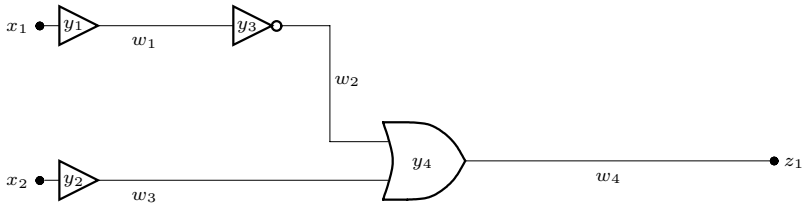


Fig. 5. A feedback-free network.

We consider only feedback-free networks from now on. In such networks, we can arrange the gates and wires in *levels*. The level of a gate is the distance of the vertex of that gate in the network graph. If the network is feedback-free, the level of a gate is always bounded. Every wire is connected to the output of some gate; the level of a wire is the level of that gate.

A simple feedback-free circuit is shown in Fig. 5; for brevity, we omit its network graph. In the corresponding network, gates y_1 and y_2 have level 0, gate y_3 has level 1, and y_4 has level 2. Wires w_1 and w_3 have level 0, w_2 has level 1, and w_4 has level 2.

3. Binary Analysis

Binary analysis is due to Muller [9, 10], but we use the terminology and notation of [3], where additional references can be found. In the sequel, we denote tuples of variables by unsubscripted letters and their components by subscripted letters. Let N be a network, and $B = \{0, 1\}$, the binary domain. For a positive integer r, $[r]$ denotes $\{1, \ldots, r\}$. Let $x = (x_1, x_2, \ldots, x_p)$ be the tuple of input variables of N, and $s = (s_1, s_2, \ldots, s_n)$, the tuple of state variables. An *internal state* of N is an n-tuple b of values from B assigned to state variables $s_1, s_2, \ldots, s_n$. A *total state* is a $(p + n)$-tuple $c = a \cdot b$ of values from B, the p-tuple a being the values of the input variables, and the n-tuple b, the internal state (the " $\cdot$ " is used for convenience to separate the input from the internal state). Each state variable s_i has an excitation S_i, where S_i is a function of some inputs $x_{j_1}, \ldots, x_{j_l}$, and some state variables $s_{i_1}, \ldots, s_{i_k}$, *i.e.*, $S_i = f(x_{j_1}, \ldots, x_{j_l}, s_{i_1}, \ldots, s_{i_k})$, where $f : B^{l+k} \to B$. It is often convenient to treat S_i as a function from B^{p+n} into B. Thus we define $\hat{S}_i : B^{p+n} \to B$ by $\hat{S}_i(a \cdot b) = f(a_{j_1}, \ldots, a_{j_l}, b_{i_1}, \ldots, b_{i_k})$, for any total state $a \cdot b$. From now on we write S_i for $\hat{S}_i$; the meaning is clear from the context.

A state variable is *stable* if its value agrees with its excitation. For any $i \in [n]$, the value of S_i in total state $a \cdot b$ is denoted by $S_i(a \cdot b)$. The tuple $(S_1(a \cdot b), \ldots, S_n(a \cdot b))$ is denoted by $S(a \cdot b)$. For a total state $a \cdot b$, we denote the set of unstable state variables as $U_a(b) = \{s_i \mid b_i \neq S_i(a \cdot b)\}$. Thus, $a \cdot b$ is *stable* if and only if $U_a(b) = \emptyset$, *i.e.*, $S(a \cdot b) = b$. For any $a \cdot b$ and $a \cdot \tilde{b}$, we denote the set of state variables which differ in state b and $\tilde{b}$ as $\Delta_a(b, \tilde{b}) = \{s_i \mid b_i \neq \tilde{b}_i\}$.

Let $a \in B^p$ be a fixed input, and $b \in B^n$, the initial internal state of N. Let $G_a = (V, E)$ be a digraph, where $V = B^n$, and for any $u, v \in V$, $u \neq v$, $(u, v) \in E$ if and only if $\Delta_a(u, v) \subseteq U_a(u)$. If $v \in V$ is stable, there is an edge in E from v to

v; *i.e.*, if $U_a(v) = \emptyset$, then $(v, v) \in E$. For any $u, v \in V$, v is *reachable* from u if and only if there is a directed path from u to v in G_a. The *binary analysis* of N with fixed input a and initial state b is the maximal subgraph $G_a(b)$ of G_a, such that each vertex in $G_a(b)$ is reachable from b.

An informal interpretation of binary analysis is as follows. We consider a circuit's behavior when its inputs are fixed at some binary values (tuple a). The present values at the outputs of all the gates (tuple b) constitute the present internal state. The behavior of each gate is governed by the Boolean function f implemented by that gate. If the present value S_i computed by f, that is, the excitation of the gate, agrees with the output of the gate, that gate is stable; otherwise it is unstable.

If all the gates are stable, then the circuit is in a stable total state, and no changes will take place. This is indicated by a self-loop around state b in the graph $G_a(b)$. Every feedback-free circuit eventually reaches a unique stable total state.

If there are several unstable gates, we assume that any nonempty subset of these gates can change. The new state reached is then a possible successor of $a \cdot b$. Our model is pessimistic in that it considers all such possible successor states. Thus, binary analysis does not require the knowledge of the sizes of the gate delays, and permits us to examine all possible behaviors.

When some gates change, the excitations of other unstable gates can also change, and these gates then become stable; this models the inertial nature of the delays [3].

To simplify notation in examples, we denote binary tuples by binary words, where a *binary word* is any word in B^*.

Example 1 *Let $s = (y_1, y_2, y_3, y_4)$ be the tuple of state variables of the network shown in Fig. 5. The excitation functions of this network are: $Y_1 = x_1$, $Y_2 = x_2$, $Y_3 = \overline{y_1}$, $Y_4 = y_2 + y_3$. Suppose the internal state is $b = 0111$, and the input is fixed at $a = 10$; the binary analysis $G_{10}(0111)$ is shown in Fig. 6, where tuples are shown as words, and unstable variables are underlined.*

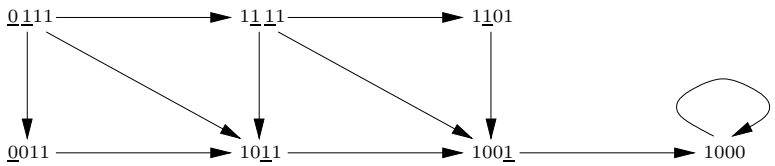


Fig. 6. Binary analysis of a network.

4. Algebra of Transients

This section is based on [4]. A *transient* is a nonempty binary word in which no two consecutive symbols are the same. We use boldface letters to denote transient variables. The set of all transients is $\mathbf{T} = 0(10)^* \cup 1(01)^* \cup 0(10)^*1 \cup 1(01)^*0$. For a transient $\mathbf{t} \in \mathbf{T}$, $\alpha(\mathbf{t})$ and $\omega(\mathbf{t})$ denote the first and last letters of $\mathbf{t}$ respectively, and $l(\mathbf{t})$ is the length of $\mathbf{t}$. A transient can be obtained from any nonempty binary word by *contraction*, *i.e.*, elimination of all duplicates immediately following a symbol (*e.g.*, the contraction of 001000 is 010). For a word s, we denote its contraction by

$\hat{s}$. We denote by $\mathbf{t} \circ \mathbf{t}'$ concatenation followed by contraction, *i.e.*, $\mathbf{t} \circ \mathbf{t}' = \widehat{\mathbf{t}\mathbf{t}'}$, where $\mathbf{t}, \mathbf{t}' \in \mathbf{T}$. The $\circ$ operation is associative, and is extended to tuples component-wise.

We are about to introduce a network model in which gates process transients, instead of binary values. For this purpose, we define extensions of Boolean functions to transients, following [4]. Suppose that $\mathbf{x} = (\mathbf{x}_1, \ldots, \mathbf{x}_r)$ is an r-tuple of input transients of a logic gate implementing a Boolean function f. Define the directed graph $D(\mathbf{x})$ to have as vertices r-tuples $\mathbf{y} = (\mathbf{y}_1, \ldots, \mathbf{y}_r)$, where each $\mathbf{y}_i$ is a prefix of length > 0 of $\mathbf{x}_i$, for each $i \in [r]$. There is an edge from vertex $\mathbf{y} = (\mathbf{y}_1, \ldots, \mathbf{y}_r)$ to vertex $\mathbf{y}' = (\mathbf{y}'_1, \ldots, \mathbf{y}'_r)$ if and only if $\mathbf{y}$ and $\mathbf{y}'$ differ in exactly one coordinate, say i, and $\mathbf{y}'_i = \mathbf{y}_i a$, where $a \in B$. Graph $D(\mathbf{x})$ shows all possible orders in which the r variables can change, while undergoing a transition from the initial values $(\alpha(\mathbf{x}_1), \ldots, \alpha(\mathbf{x}_r))$ to the final values $(\omega(\mathbf{x}_1), \ldots, \omega(\mathbf{x}_r))$. We label each vertex $\mathbf{y} = (\mathbf{y}_1, \ldots, \mathbf{y}_r)$ of $D(\mathbf{x})$ with the value $f(a_1, \ldots, a_r)$, where a_i is the last letter of $\mathbf{y}_i$, i.e., $a_i = \omega(\mathbf{y}_i)$, for each $i \in [r]$. Figure 7 shows the graph $D(01, 101)$ and its labeling for a 2-input OR gate.

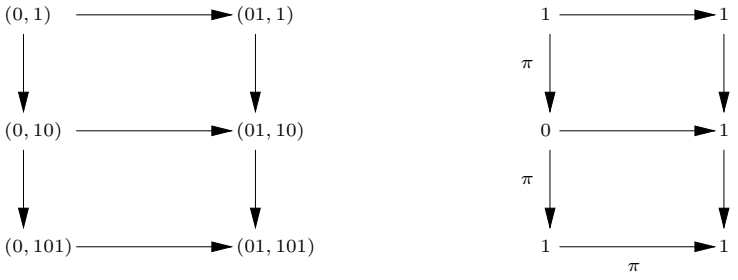


Fig. 7. Graph $D(01, 101)$.

Definition 5 *Given a Boolean function $f : B^r \to B$, we define a function $\mathbf{f} : \mathbf{T}^r \to \mathbf{T}$ as follows. For any r-tuple $(\mathbf{x}_1, \ldots, \mathbf{x}_r)$ of transients, the value of $\mathbf{f}(\mathbf{x}_1, \ldots, \mathbf{x}_r)$ is the contraction of the label sequence of a path in $D(\mathbf{x})$ from $(\alpha(\mathbf{x}_1), \ldots, \alpha(\mathbf{x}_r))$ to $(\mathbf{x}_1, \ldots, \mathbf{x}_r)$ which has the largest number of alternations between 0 and 1. We call $\mathbf{f}$ the* `extension` *of the Boolean function f.*

Thus, when $\mathbf{x}_1, \ldots, \mathbf{x}_r$ are applied to the inputs of a gate performing function f, then $\mathbf{f}(\mathbf{x}_1, \ldots, \mathbf{x}_r)$ is the longest transient at the output of the gate, when all possible orders of variable changes in the input transients are considered. For example, path π in Fig. 7 is a path corresponding to the longest transient. Whether the longest transient actually occurs in a gate depends on the delays in the input wires. By defining the transient extension of f to correspond to the longest transient we are able to find the *worst-case transient behavior* of a gate.

Let $z(\mathbf{t})$ and $u(\mathbf{t})$ denote the number of 0s and the number of 1s in a transient $\mathbf{t}$, respectively. We denote by $\otimes$ and $\oplus$ the extensions of the Boolean AND and OR operations, respectively. It is shown in [4] that for any $\mathbf{w}, \mathbf{w}' \in \mathbf{T}$ of length > 1,

$\mathbf{w} \otimes \mathbf{w}' = \mathbf{t}$, where $\mathbf{t} \in \mathbf{T}$ is such that[c]

$$\alpha(\mathbf{t}) = \alpha(\mathbf{w}) * \alpha(\mathbf{w}'), \quad \omega(\mathbf{t}) = \omega(\mathbf{w}) * \omega(\mathbf{w}'), \quad \text{and} \quad u(\mathbf{t}) = u(\mathbf{w}) + u(\mathbf{w}') - 1.$$

Similarly, $\mathbf{w} \oplus \mathbf{w}' = \mathbf{t}$, where $\mathbf{t} \in \mathbf{T}$ is such that

$$\alpha(\mathbf{t}) = \alpha(\mathbf{w}) + \alpha(\mathbf{w}'), \quad \omega(\mathbf{t}) = \omega(\mathbf{w}) + \omega(\mathbf{w}'), \quad \text{and} \quad z(\mathbf{t}) = z(\mathbf{w}) + z(\mathbf{w}') - 1.$$

If one of the arguments is 0 or 1, the following rules apply:

$$\mathbf{t} \oplus 0 = 0 \oplus \mathbf{t} = \mathbf{t}, \quad \mathbf{t} \oplus 1 = 1 \oplus \mathbf{t} = 1,$$
$$\mathbf{t} \otimes 1 = 1 \otimes \mathbf{t} = \mathbf{t}, \quad \mathbf{t} \otimes 0 = 0 \otimes \mathbf{t} = 0.$$

The complement $\overline{\mathbf{t}}$ of $\mathbf{t} \in \mathbf{T}$ is obtained by complementing each character of $\mathbf{t}$. For example, $\overline{1010} = 0101$.

Algebra $C = (\mathbf{T}, \oplus, \otimes, \overline{\ }, 0, 1)$, is called the *change-counting algebra* [4]. We also refer to C as the *algebra of transients*, following [2, 7, 8].

Let $\pi = s^0, \ldots, s^h$ be a path of length $h \geq 0$ in $G_a(b)$. Recall that each $s^j = (s_1^j, \ldots, s_n^j)$ is an internal state in B^n. For any $i \in [n]$, we denote by $\boldsymbol{\sigma}_i^\pi$ the transient $\widehat{s_i^0 \ldots s_i^h}$, which shows the changes of the i-th state variable along path π. We refer to $\boldsymbol{\sigma}_i^\pi$ as the *history* [2] of variable s_i along the path. The histories of all state variables along π constitute the tuple $\boldsymbol{\sigma}^\pi = (\boldsymbol{\sigma}_1^\pi, \ldots, \boldsymbol{\sigma}_n^\pi)$.

5. Transient Networks and Transient Simulation

We now extend the binary network model to the transient network model by changing its *domain* from the binary domain B to the domain $\mathbf{T}$ of transients.

Definition 6 *A* `transient network` $\mathbf{N}$ *is a network graph* (V, E) *together with an assignment of excitation functions which are extensions of the Boolean excitations.*

In a transient network, gates process transients instead of binary signals. An assignment of transients to the variables of a transient network is a total state of that network. Return to the binary network N of Fig. 5; the excitation functions of its transient equivalent $\mathbf{N}$ are:

$$\mathbf{Y_1} = \mathbf{x_1}, \ \mathbf{Y_2} = \mathbf{x_2}, \ \mathbf{Y_3} = \overline{\mathbf{y_1}}, \ \mathbf{Y_4} = \mathbf{y_3} \oplus \mathbf{y_2}.$$

In a transient network, a gate is *stable* if the extension of its Boolean excitation agrees with the present state of the gate, and a network $\mathbf{N}$ is *stable* if all of its gates are stable. If the total state of the transient network of Fig. 5 is $\mathbf{x}_1 = 10$, $\mathbf{x}_2 = 1010$, $\mathbf{y}_1 = 10$, $\mathbf{y}_2 = 1010$, $\mathbf{y}_3 = 01$, and $\mathbf{y}_4 = 10101$, then the network is stable.

We now describe an efficient simulation algorithm [4], which we call *transient simulation*. Let $\mathbf{N}$ be a transient network with $\mathbf{x} = (\mathbf{x}_1, \ldots, \mathbf{x}_p)$ as the input variables, and $\mathbf{s} = (\mathbf{s}_1, \ldots, \mathbf{s}_n)$ as the state variables. Assume that the network starts in a stable initial state $\tilde{a} \cdot b \in B^{p+n}$, and the input is changed to $a \in B^p$.

[c]The symbol + is used both as OR and addition of integers; the meaning is clear from the context.

Algorithm A
$\mathbf{x} := \tilde{a} \circ a$;
$\mathbf{s}^0 := b$;
$h := 1$;
repeat {
  $\mathbf{s}^h := \mathbf{S}(\mathbf{x} \cdot \mathbf{s}^{h-1})$;
  $h := h + 1$
  }
until $\mathbf{s}^h = \mathbf{s}^{h-1}$;

For a feedback-free network, Algorithm A always terminates. Let the sequence of states produced by Algorithm A be $\mathbf{s}^0, \ldots, \mathbf{s}^H$, and let $\mathbf{s}^H = \{\mathbf{s}_1^H, \ldots, \mathbf{s}_n^H\}$ be the final value after termination. The *transient* of wire w_i in this simulation is the final value of node variable $t(w_i)$ upon termination.

Table 1. Simulation of a network.

$\mathbf{x_1}$	$\mathbf{x_2}$	$\mathbf{y_1}$	$\mathbf{y_2}$	$\mathbf{y_3}$	$\mathbf{y_4}$	state
01	01	0	0	1	1	$\mathbf{s^0}$
01	01	01	01	1	1	$\mathbf{s^1}$
01	01	01	01	10	1	$\mathbf{s^2}$
01	01	01	01	10	101	$\mathbf{s^3}$

Example 2 *For the network of Fig. 5, with extended excitations* $\mathbf{Y}_1 = \mathbf{x}_1$, $\mathbf{Y}_2 = \mathbf{x}_2$ $\mathbf{Y}_3 = \overline{\mathbf{y}_1}$, *and* $\mathbf{Y}_4 = \mathbf{y}_2 \oplus \mathbf{y}_3$, *let* $\mathbf{s} = (\mathbf{y}_1, \mathbf{y}_2, \mathbf{y}_3, \mathbf{y}_4)$, *and let* $b = 0011$ *be the initial internal state. Suppose* $\mathbf{x} = (\mathbf{x}_1, \mathbf{x}_2)$ *changes from* $\tilde{a} = 00$ *to* $a = 11$. *The result of Algorithm A is given in Table 1.*

6. Network Models with Wire Delays

We now define the *expanded* and *singular networks* of any network; these are special cases of networks with wire delays. Let N be a network as in Section 2, and let $W = \{w_1, \ldots, w_m\}$ be the set of wires. Define a function $\mathcal{D}$ that maps each wire variable to a non-negative integer; this is the *wire-delay function* of the network. We add $\mathcal{D}(w_i)$ delays to wire w_i; let $s_{i1}, \ldots, s_{il_i}$ be the outputs of the added delays from right to left, where s_{i1} is the *head*, and s_{il_i}, the *tail*. We modify the *excitation functions* as follows. If a delay variable s_{ij} is not a tail, then its excitation function is $S_{ij} = s_{i(j+1)}$. If it is a tail, then its excitation is $S_{ij} = t(w_i)$. The excitation function of a FORK y_i with input wire w_j is the identity function $Y_i = s_{j1}$, where $h(w_j) = y_i$, and s_{j1} is the head segment of w_j. The excitation function of an input gate remains unchanged. If y_i is a gate performing a Boolean function f, and the incoming edges of y_i are wires $w_{i_1}, \ldots, w_{i_k}$, then the excitation of the gate is $Y_i = f(s_{i_1 1}, \ldots, s_{i_k 1})$. The *output function* of an external output $z_i = h(w_j)$ is the identity function $z_i = s_{j1}$. This modification allows a network to have any number

of delays on each wire. We call the modified network a *wire-delay extension* of N. Note that N is just a special wire-delay extension, where each wire has zero delays.

Definition 7 *An* `expanded network` $\hat{N}$ *with respect to a given transient simulation is a wire-delay extension of* N *with* $\mathcal{D}(w_i) = l(\mathbf{t}_i) - 1$*, for* $i = 1, \ldots, m$*, where* $\mathbf{t}_i$ *is the transient of wire* w_i *in the simulation.*

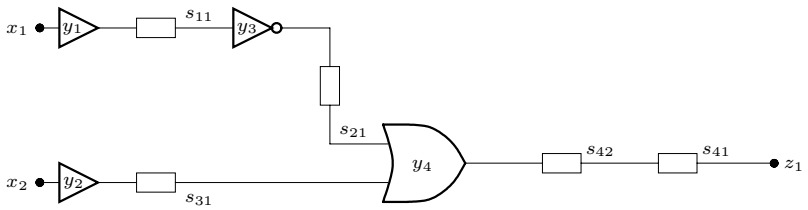


Fig. 8. An expanded network.

Figure 8 shows the expanded network of Fig. 5 with the simulation of Table 1. The excitation and output functions are: $S_{11} = y_1$, $S_{21} = y_3, S_{31} = y_2$, $S_{42} = y_4$, $S_{41} = s_{42}$, $Y_1 = x_1$, $Y_2 = x_2$, $Y_3 = \overline{s_{11}}$, $Y_4 = s_{21} + s_{31}$, $z_1 = s_{41}$.

Definition 8 *The* `singular network` $\dot{N}$ *of a network* N *is a wire-delay extension of* N *with* $\mathcal{D}(w_i) = 1$*, for* $i = 1, \ldots, m$*.*

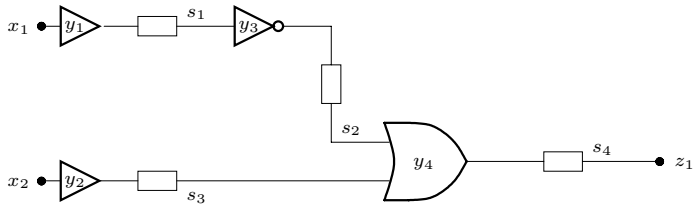


Fig. 9. A singular network.

Figure 9 shows the singular network of the network of Fig. 5. Since there is only one delay per wire, the output of the delay of wire w_i is labeled s_i. The excitation and output functions are: $S_1 = y_1$, $S_2 = y_3$, $S_3 = y_2$, $S_4 = y_4$, $Y_1 = x_1$, $Y_2 = x_2$, $Y_3 = \overline{s_1}$, $Y_4 = s_2 + s_3$, $z_1 = s_4$.

7. More about the Algebra of Transients

For any two binary words t and t', t is a *prefix* (respectively, *suffix*) of t' if there exists a (possible empty) binary word t'' such that $t' = tt''$ (respectively, $t' = t''t$). Let $w = a_1 a_2 \ldots a_l$ be a binary word of length l. For an integer k, $1 \leq k \leq l$, the prefix of w ending at position k is denoted by ${}_k(w) = a_1 \ldots a_k$, and the suffix of w starting at position k is denoted by $(w)_k = a_k \ldots a_l$.

The prefix and suffix relations are partial orders on the set of binary words; the prefix order is denoted by $\leq$. Let $(\mathbf{x}_1, \ldots, \mathbf{x}_r)$ be an r-tuple of transients. It is shown in [4] that extensions of Boolean functions are monotonic with respect to the prefix order, *i.e.*, if $\mathbf{f}$ is the extension of f, then $\mathbf{x}_1 \leq \mathbf{x}'_1, \ldots, \mathbf{x}_r \leq \mathbf{x}'_r \Rightarrow \mathbf{f}(\mathbf{x}_1, \ldots, \mathbf{x}_r) \leq \mathbf{f}(\mathbf{x}'_1, \ldots, \mathbf{x}'_r)$. This also holds for the suffix order.

Let $\mathbf{f}$ be the extension of the r-argument Boolean function AND, and let $\mathbf{y} = \mathbf{f}(\mathbf{x}_1, \ldots, \mathbf{x}_r)$. Consider any $\mathbf{x}_i$, $i \in [r]$. If $\mathbf{x}_i = 0$, then $\mathbf{y} = 0$ irrespective of the values of the other transients. If $\mathbf{x}_i = 1$, then $\mathbf{y}$ is independent of $\mathbf{x}_i$. Now we evaluate $\mathbf{f}(\mathbf{x}_1, \ldots, \mathbf{x}_r)$ under the assumption that all transients $\mathbf{x}_1, \ldots, \mathbf{x}_r$ have length > 1. It is shown in [4] that the result $\mathbf{y} \in \mathbf{T}$ is uniquely determined by the first and last letters and the number of 1s, which are computed as follows:

$$\alpha(\mathbf{y}) = \alpha(\mathbf{x}_1) * \cdots * \alpha(\mathbf{x}_r); \;\; \omega(\mathbf{y}) = \omega(\mathbf{x}_1) * \cdots * \omega(\mathbf{x}_r); \;\; u(\mathbf{y}) = 1 + \sum_{i=1}^{r} (u(\mathbf{x}_i) - 1).$$

Proposition 1 *If there exists an $i \in [r]$ such that $01 \leq \mathbf{x}_i$, then $\mathbf{f}(\mathbf{x}_1, \ldots, \mathbf{x}_r) = 0 \circ \mathbf{f}(\mathbf{x}_1, \ldots, (\mathbf{x}_i)_2, \ldots, \mathbf{x}_r)$.*

Proof. Let $\mathbf{y}' = 0 \circ \mathbf{f}(\mathbf{x}_1, \ldots, (\mathbf{x}_i)_2, \ldots, \mathbf{x}_r)$, and $\mathbf{y}'' = \mathbf{f}(\mathbf{x}_1, \ldots, (\mathbf{x}_i)_2, \ldots, \mathbf{x}_r)$; we prove that $\mathbf{y} = \mathbf{y}'$ by showing that $\mathbf{y}$ and $\mathbf{y}'$ have the same first letter, the same last letter and the same number of 1s.

(i) Since $\alpha(\mathbf{x}_i) = 0$, also $\alpha(\mathbf{y}) = 0 = \alpha(\mathbf{y}')$.

(ii) Since $\omega(\mathbf{x}_i)_2 = \omega(\mathbf{x}_i)$, we have $\omega(\mathbf{y}) = \omega(\mathbf{x}_1) * \cdots * \omega(\mathbf{x}_i) * \cdots * \omega(\mathbf{x}_r) = \omega(\mathbf{x}_1) * \cdots * \omega((\mathbf{x}_i)_2) * \cdots * \omega(\mathbf{x}_r) = \omega(\mathbf{y}'') = \omega(\mathbf{y}')$, where the last equality holds because $\mathbf{y}''$, being a transient, is nonempty.

(iii) Since $u(\mathbf{x}_i) = u((\mathbf{x}_i)_2)$, we have $u(\mathbf{y}) = u(\mathbf{y}')$.

□

Proposition 2 *If $1 \leq \mathbf{x}_j$ for all $j \in [r]$, and there exists an $i \in [r]$ such that $101 \leq \mathbf{x}_i$, then $\mathbf{f}(\mathbf{x}_1, \ldots, \mathbf{x}_r) = 10\mathbf{f}(\mathbf{x}_1, \ldots, (\mathbf{x}_i)_3, \ldots, \mathbf{x}_r)$.*

Proof. Let $\mathbf{y}' = 10\mathbf{f}(\mathbf{x}_1, \ldots, (\mathbf{x}_i)_3, \ldots, \mathbf{x}_r)$; we show that $\mathbf{y} = \mathbf{y}'$.

(i) Since $1 \leq \mathbf{x}_j$ for all $j \in [r]$, we have $\alpha(\mathbf{y}) = 1 = \alpha(\mathbf{y}')$.

(ii) By the arguments used in the proof of Prop. 1, it is clear that $\omega(\mathbf{y}) = \omega(\mathbf{y}')$.

(iii) Since $u(\mathbf{x}_i) = u((\mathbf{x}_i)_3) + 1$, we have $u(\mathbf{y}) = u(\mathbf{y}')$.
Since $\mathbf{f}(\mathbf{x}_1, \ldots, (\mathbf{x}_i)_3, \ldots, \mathbf{x}_r)$ always begins with 1 here, we use concatenation, rather than the $\circ$ operation, in $10\mathbf{f}(\mathbf{x}_1, \ldots, (\mathbf{x}_i)_3, \ldots, \mathbf{x}_r)$.

□

We now give a recursive algorithm which computes $\mathbf{f}$, the extension of AND.

Algorithm AND

if $\exists\ i \in [r]$ such that $\mathbf{x}_i = 0$
  then $\mathbf{f}(\mathbf{x}_1, \ldots, \mathbf{x}_r) = 0$
else if $\exists\ i \in [r]$ such that $01 \leq \mathbf{x}_i$
  then $\mathbf{f}(\mathbf{x}_1, \ldots, \mathbf{x}_i, \ldots, \mathbf{x}_r) = 0 \circ \mathbf{f}(\mathbf{x}_1, \ldots, (\mathbf{x}_i)_2, \ldots, \mathbf{x}_r)$
else if $\{\exists\ i \in [r]$ such that $101 \leq \mathbf{x}_i\}$
  then $\mathbf{f}(\mathbf{x}_1, \ldots, \mathbf{x}_i, \ldots, \mathbf{x}_r) = 10\mathbf{f}(\mathbf{x}_1, \ldots, (\mathbf{x}_i)_3, \ldots, \mathbf{x}_r)$
else if $max\{l(\mathbf{x}_i) \mid i \in [r]\} = 2$
  then $\mathbf{f}(\mathbf{x}_1, \ldots, \mathbf{x}_r) = 10$
else $\mathbf{f}(\mathbf{x}_1, \ldots, \mathbf{x}_r) = 1$;

The algorithm is correct for the following reasons. If there is an $i \in [r]$ such that $\mathbf{x}_i = 0$, then $\mathbf{f}(\mathbf{x}_1, \ldots, \mathbf{x}_r) = 0$. If there is an $i \in [r]$ such that $01 \leq \mathbf{x}_i$, then $\mathbf{f}(\mathbf{x}_1, \ldots, \mathbf{x}_i, \ldots, \mathbf{x}_r) = 0 \circ \mathbf{f}(\mathbf{x}_1, \ldots, (\mathbf{x}_i)_2, \ldots, \mathbf{x}_r)$, by Prop. 1. If neither case above holds, then for all $i \in [r]$, $1 \leq \mathbf{x}_i$.

If there is an $i \in [r]$ such that $101 \leq \mathbf{x}_i$, then by Prop. 2, $\mathbf{f}(\mathbf{x}_1, \ldots, \mathbf{x}_i, \ldots, \mathbf{x}_r) = 10\mathbf{f}(\mathbf{x}_1, \ldots, (\mathbf{x}_i)_3, \ldots, \mathbf{x}_r)$. Otherwise, if the maximal length of all transients is 2, then each transient is either 1 or 10, and at least one transient is 10. Hence $\mathbf{f}(\mathbf{x}_1, \ldots, \mathbf{x}_i, \ldots, \mathbf{x}_r) = 10$. If each transient is of length 1, then each transient is 1, and $\mathbf{f}(\mathbf{x}_1, \ldots, \mathbf{x}_r) = 1$.

Each transient represents a sequence of signal changes, and the extension of the AND function represents the AND gate. Algorithm AND suggests an order in which input signal changes should be processed through the gate to create the longest sequence of output changes. We should first process 0 to 1 changes (transients with prefix 01), then changes from 1 to 0 to 1 (transients with prefix 101), and finally, changes from 1 to 0 (transients with prefix 10). Note that it is possible that during the run of the algorithm there might be more than one choice for the transient to be processed. Algorithm AND does not specify the rules of breaking ties. These rules will be introduced in Section 9.

Example 3 *Let* $\mathbf{f}$ *be the* AND *function and consider* $\mathbf{f}(101, 10, 010)$. *None of the transients is 0, but 010 has prefix 01; hence* $\mathbf{f}(101, 10, 010) = 0 \circ \mathbf{f}(101, 10, 10)$. *Next, 101 has prefix 101 in* $\mathbf{f}(101, 10, 10)$, *and* $\mathbf{f}(101, 10, 010) = 0 \circ \mathbf{f}(101, 10, 10) = 010\mathbf{f}(1, 10, 10)$. *Now all transients are of length 1 or 2 in* $\mathbf{f}(1, 10, 10)$, *and we have* $\mathbf{f}(101, 10, 010) = 0 \circ \mathbf{f}(101, 10, 10) = 010\mathbf{f}(1, 10, 10) = 01010$.

By duality, we also have a recursive algorithm to compute the extension for OR.

Algorithm OR

if $\exists\ i \in [r]$ such that $\mathbf{x}_i = 1$
  then $\mathbf{f}(\mathbf{x}_1, \ldots, \mathbf{x}_r) = 1$
else if $\exists\ i \in [r]$ such that $10 \leq \mathbf{x}_i$
  then $\mathbf{f}(\mathbf{x}_1, \ldots, \mathbf{x}_i, \ldots, \mathbf{x}_r) = 1 \circ \mathbf{f}(\mathbf{x}_1, \ldots, (\mathbf{x}_i)_2, \ldots, \mathbf{x}_r)$
else if $\{\exists\ i \in [r]$ such that $010 \leq \mathbf{x}_i\}$
  then $\mathbf{f}(\mathbf{x}_1, \ldots, \mathbf{x}_i, \ldots, \mathbf{x}_r) = 01\mathbf{f}(\mathbf{x}_1, \ldots, (\mathbf{x}_i)_3, \ldots, \mathbf{x}_r)$
else if $max\{l(\mathbf{x}_i) \mid i \in [r]\} = 2$
  then $\mathbf{f}(\mathbf{x}_1, \ldots, \mathbf{x}_r) = 01$
else $\mathbf{f}(\mathbf{x}_1, \ldots, \mathbf{x}_r) = 0$;

The order in which signal changes should be processed to produce the longest transient for the OR gate is: 10, 010, 01.

The algorithm for XOR is very simple, since the output changes with each single input change. Thus, any order in which the signal changes are processed for the XOR gate results in the same transient output.

Algorithm XOR

if $max\{l(\mathbf{x}_i) \mid i \in [r]\} = 1$ and the number of 1s is even
  then $\mathbf{f}(\mathbf{x}_1, \ldots, \mathbf{x}_r) = 0$
else if $max\{l(\mathbf{x}_i) \mid i \in [r]\} = 1$ and the number of 1s is odd
  then $\mathbf{f}(\mathbf{x}_1, \ldots, \mathbf{x}_r) = 1$
else $\{\exists\ i \in [r]$ such that $l(\mathbf{x}_i) \geq 2\}$
  $\mathbf{f}(\mathbf{x}_1, \ldots, \mathbf{x}_i, \ldots, \mathbf{x}_r) =$
    $\mathbf{f}(_1(\mathbf{x}_1), \ldots, _1(\mathbf{x}_i), \ldots, _1(\mathbf{x}_r))\mathbf{f}(\mathbf{x}_1, \ldots, (\mathbf{x}_i)_2, \ldots, \mathbf{x}_r)$;

It is also easy to give recursive algorithms for NAND, NOR, XNOR, NOT and FORK, since the first three are similar to Algorithms AND, OR, and XOR, and the last two only have a single input. Finally, the algorithm also works for functions obtained from one of the functions above by complementing any number of inputs. For example, if $\mathbf{f}$ has a recursive algorithm as given above, and $\mathbf{g}(\mathbf{x}_1, \ldots, \mathbf{x}_i, \ldots, \mathbf{x}_r) = \mathbf{f}(\mathbf{x}_1, \ldots, \bar{\mathbf{x}}_i, \ldots, \mathbf{x}_r)$, then so does $\mathbf{g}$. Therefore, the extension of Boolean functions like $\mathbf{x}_1 \oplus \bar{\mathbf{x}}_2$ or $\mathbf{x}_1 \otimes \bar{\mathbf{x}}_2$ also have such recursive algorithms. This shows that recursive algorithms exist for all the Boolean functions in the set $\mathcal{G}$, which includes all the functions of two variables.

8. Signal Changes in Gates

Before proceeding we add a comment about terminology. We find it convenient to present our proofs using a somewhat informal framework. Binary analysis assumes that the delays in any network are arbitrary and can change with time. In looking for paths with special properties, we prefer to talk about "scheduling signal changes" to create these paths. Of course, our statements can be rephrased in the standard terminology.

The following example shows the difference in terminologies. Suppose a gate network starts in a stable total state, and some inputs change simultaneously, as is usually assumed in binary analysis [3]. Let's say inputs $x_{i_1}, \ldots, x_{i_k}$ are the changing inputs. In binary analysis, the corresponding input-gate variables $y_{i_1}, \ldots, y_{i_k}$ are unstable after the inputs change. These unstable variables can change in any order, and that includes simultaneous changes. In particular, they can change one at a time in some order. We call such an order an "arrival order" of inputs to the non-input gates of the circuit. Thus, rather than saying that in binary analysis there is a path in graph $G_a(b)$ in which the unstable input gates change in a certain order, we "schedule" the changes in that order.

A *signal change* is a transient of length two, that is, either 01 or 10. A transient of length l consists of $l-1$ signal changes; every two consecutive changes are complementary. For example, transient 010 consists of signal changes 01 and 10. Signal changes occur in both binary analysis and transient simulation. In binary analysis, a change in the input tuple is a set of signal changes, and a history $\boldsymbol{\sigma}_i^{\pi}$ of a state variable s_i along a path π contains the sequence of changes that have occured in s_i along π. In transient simulation, the transient of a variable at the end of simulation

contains all the signal changes occurring in that variable in the worst case. Our goal is to find a path in the binary analysis of some wire-delay extension of N which contains all the signal changes predicted by the transient simulation of N.

In this section we study some properties of single gates. Consider a gate that is part of some network, performs the function $f(x_1, \ldots, x_r)$, and has transient extension $\mathbf{f}$. Suppose that the input wires of this gate have transients $\mathbf{x}_1, \ldots, \mathbf{x}_r$ in the last state of the transient simulation of the circuit. Then the output of the gate has transient $\mathbf{f}(\mathbf{x}_1, \ldots, \mathbf{x}_r)$.

We assume that the initial state of the network is stable, and the initial signal on input wire w_i is a. Suppose that, in binary analysis, w_i receives a sequence $c_{i_1}, \ldots, c_{i_k}$ of signal changes that corresponds precisely to transient $\mathbf{x}_i$, where c_{i_1} is a change from a to $\overline{a}$, c_{i_2}, from $\overline{a}$ to a, etc. For convenience, we introduce the following terminology: change c_{i_1} is the *winner* of the sequence of changes on w_i, and $c_{i_2}, \ldots, c_{i_k}$ are the *losers*. Also, c_{i_2} is the *runner-up* on w_i.

We postulate that the changes on the input wires of a network N arrive in some order, called the *arrival order*. This order can be arbitrary, except that changes on a given wire must appear consecutively, that is, the $(j+1)$st change must appear after the jth change on that wire; every arrival order must be *consistent* in this sense. We also assume that no two changes arrive at the same time, that is, the arrival order is a total order. Our objective is to determine how the arrival order can be modified in a delay-extension of N with the aid of wire delays to produce the longest transients on the outputs of gates. This subproblem is a key step in our proof that binary analysis covers transient simulation.

If an input wire of a gate has delays, then some changes can be postponed till a later time. Figure 10(a) shows a 2-input AND gate in a stable initial state. When the first change arrives, the situation is as shown in Fig. 10(b); because of the delay, the gate does not "see" the change. Before the next change arrives at the input of the delay, the first change needs to be *processed,* as shown in Fig. 10(c), that is, when the new value reaches the gate, the gate must evaluate its new inputs. Then a second change can arrive, and its processing can again be postponed, if needed, as in Fig. 10(d) until it is evaluated as in Fig. 10(e).

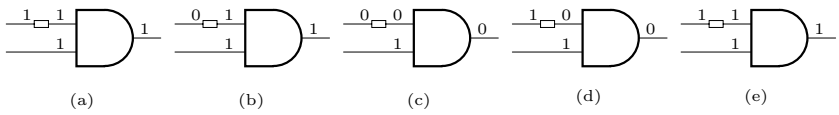


Fig. 10. Delaying signal changes.

We now examine a general AND gate. Assume that the arrival order of the signal changes on the input wires of the gate is given. We postulate that we can insert delays in the input wires and use these delays to change the order to an *evaluation order*, which is the order of changes at the outputs of the wire delays connected directly to the gate inputs. Thus, the arrival order is the order of changes supplied by the environment, while the evaluation order is the order of changes as "seen" by the gate. This approach is consistent with binary analysis. If a delay is unstable, we can change it right away, or postpone the change until some other changes have

taken place. This is possible, because binary analysis permits the delay magnitudes to be arbitrary.

Given an arrival order, we now describe Algorithm ANDI, which is an iterative version of Algorithm AND. There is only one possible order of handling signal changes in ANDI, whereas AND allows choice when handling transients beginning with 01 and 101.

First, ANDI tests if there is a transient that is 0, in which case $\mathbf{f}(\mathbf{x}_1, \ldots, \mathbf{x}_r) = 0$. Next, if there is a transient beginning with 01, then Algorithm AND01 is used to determine the order in which 01 changes should be processed, and results in a vector $(\mathbf{y}_1, \ldots, \mathbf{y}_r)$ of transients all beginning with 1. Vector $(\mathbf{y}_1, \ldots, \mathbf{y}_r)$ is passed to Algorithm AND1, which then evaluates $\mathbf{f}(\mathbf{y}_1, \ldots, \mathbf{y}_r)$, and we have $\mathbf{f}(\mathbf{x}_1, \ldots, \mathbf{x}_r) = 0 \circ \mathbf{f}(\mathbf{y}_1, \ldots, \mathbf{y}_r)$. If there are no transients beginning with 0, Algorithm AND1 determines the order in which transients beginning with 101 should be handled. If there are no such transients, then $\mathbf{f}(\mathbf{x}_1, \ldots, \mathbf{x}_r) = 10$, if there is a transient that is 10, and $\mathbf{f}(\mathbf{x}_1, \ldots, \mathbf{x}_r) = 1$, otherwise.

Algorithm ANDI$(\mathbf{x}_1, \ldots, \mathbf{x}_r)$ {Arrival order is given}

if $\exists i \in [r]$ such that $\mathbf{x}_i = 0$
  then $\mathbf{f}(\mathbf{x}_1, \ldots, \mathbf{x}_r) = 0$
else if $\exists i \in [r]$ such that $01 \leq \mathbf{x}_i$
  then
    Perform Algorithm AND01 with $(\mathbf{x}_1, \ldots, \mathbf{x}_r)$ to find $(\mathbf{y}_1, \ldots, \mathbf{y}_r)$;
    Perform Algorithm AND1 with $(\mathbf{y}_1, \ldots, \mathbf{y}_r)$ to find $\mathbf{f}(\mathbf{y}_1, \ldots, \mathbf{y}_r)$;
    $\mathbf{f}(\mathbf{x}_1, \ldots, \mathbf{x}_r) = 0 \circ \mathbf{f}(\mathbf{y}_1, \ldots, \mathbf{y}_r)$
  else
    Perform Algorithm AND1 to find $\mathbf{f}(\mathbf{x}_1, \ldots, \mathbf{x}_r)$;
return $\mathbf{f}(\mathbf{x}_1, \ldots, \mathbf{x}_r)$;

Algorithm AND01 processes transients that begin with 01 in the order of their arrivals.

Algorithm AND01$(\mathbf{x}_1, \ldots, \mathbf{x}_r)$ $\{\exists i \in [r]$ such that $01 \leq \mathbf{x}_i$ $\}$

Let L be the list of 01 winner changes in arrival order;
$\mathbf{y}_i = \mathbf{x}_i$ $\forall i \in [r]$;
  repeat {
  Let $\mathbf{x}_i$ be that transient beginning with 0_c1 where c is first in L;
  Delete c from L;
  $\mathbf{y}_i = (\mathbf{y}_i)_2$
  }
  until $\alpha(\mathbf{y}_i) = 1$ $\forall i \in [r]$;
return $(\mathbf{y}_1, \ldots, \mathbf{y}_r)$;

Algorithm AND1 applies when all the transients $\mathbf{x}_1, \ldots, \mathbf{x}_r$ begin with 1.

Algorithm AND1($\mathbf{x}_1, \ldots, \mathbf{x}_r$) $\{\forall i \in [r]\ \alpha(\mathbf{x}_i) = 1\ \}$

$\mathbf{y}_i = \mathbf{x}_i\ \forall i \in [r]$;
$k = 0$; $\{k$ is a counter$\}$
if $\exists\ i \in [r]$ such that $101 \leq \mathbf{x}_i$
then
repeat {
Let $\mathbf{y}_i$ be the transient with a 10 winner whose runner-up
is first in arrival order;
$\mathbf{y}_i = (\mathbf{y}_i)_3$;
$k = k + 1$
}
until no $\mathbf{y}_i$ begins with 101;
if $max\{l(\mathbf{y}_i) \mid i \in [r]\} = 2$
then $\mathbf{f}(\mathbf{x}_1, \ldots, \mathbf{x}_r) = (10)^{k+1}$
else $\mathbf{f}(\mathbf{x}_1, \ldots, \mathbf{x}_r) = 1 \circ (10)^k$; {Note: $1 \circ (10)^0 = 1$}
return $\mathbf{f}(\mathbf{x}_1, \ldots, \mathbf{x}_r)$;

Example 4 *Consider the evaluation of* $\mathbf{f}(10101, 0101, 010)$ *by Algorithm ANDI. We number the changes for convenience:* $\mathbf{f}(1_{c_1}0_{c_2}1_{c_3}0_{c_4}1,\ 0_{c_5}1_{c_6}0_{c_7}1,\ 0_{c_8}1_{c_9}0)$. *Arrival order* $c_8, c_9, c_1, c_2, c_3, c_5, c_6, c_4, c_7$ *is consistent; given this order, the evaluation of* $\mathbf{f}$ *is as follows. There is no* i *such that* $\mathbf{x}_i = 0$, *but two of the transients begin with 01. Hence we invoke AND01. List L is* c_8, c_5. *First,* $\mathbf{y}_1 = 10101, \mathbf{y}_2 = 0101, \mathbf{y}_3 = 010$; $\mathbf{x}_i = \mathbf{x}_3$; *The list is now* c_5. *Second,* $\mathbf{y}_1 = 10101, \mathbf{y}_2 = 0101, \mathbf{y}_3 = 10$; $\mathbf{x}_i = \mathbf{x}_2$; *The list is now empty. Third,* $\mathbf{y}_1 = 10101, \mathbf{y}_2 = 101, \mathbf{y}_3 = 10$; *Now all the input transients begin with 1 and AND1 is invoked.* $\mathbf{f}(10101, 0101, 010) = 0 \circ \mathbf{f}(10101, 101, 10)$

There remains to be evaluated the function $\mathbf{f}(1_{c_1}0_{c_2}1_{c_3}0_{c_4}1,\ 1_{c_6}0_{c_7}1,\ 1_{c_9}0)$, *with the arrival order* $c_9, c_1, c_2, c_3, c_6, c_4, c_7$. *We evaluate* $\mathbf{f}$ *by Algorithm AND1. First,* $\mathbf{y}_1 = 1_{c_1}0_{c_2}1_{c_3}0_{c_4}1$, $\mathbf{y}_2 = 1_{c_6}0_{c_7}1$, $\mathbf{y}_3 = 1_{c_9}0$; $k = 0$; *transients* $\mathbf{y}_1$ *and* $\mathbf{y}_2$ *begin with 101,* $\mathbf{y}_i = \mathbf{y}_1$, *since* c_2 *arrives before* c_7; $\mathbf{y}_1 = 1_{c_3}0_{c_4}1$, $\mathbf{y}_2 = 1_{c_6}0_{c_7}1$, $\mathbf{y}_3 = 1_{c_9}0$; $k = 1$. *Second,* $\mathbf{y}_i = \mathbf{y}_1$, *since* c_4 *arrives before* c_7; $\mathbf{y}_1 = 1$, $\mathbf{y}_2 = 1_{c_6}0_{c_7}1$, $\mathbf{y}_3 = 1_{c_9}0$; $k = 2$. *Third,* $\mathbf{y}_i = \mathbf{y}_2$; $\mathbf{y}_1 = 1$, $\mathbf{y}_2 = 1$, $\mathbf{y}_3 = 1_{c_9}0$; $k = 3$. *Now the maximum length of transients is 2, and* $\mathbf{f}(10101, 101, 10) = (10)^4 = 10101010$, *agreeing with the formula in Section 4.*

Algorithm AND1 imposes the evaluation order $c_1, c_2, c_3, c_4, c_6, c_7, c_9$. *For the complete arrival order* $c_8, c_9, c_1, c_2, c_3, c_5, c_6, c_4, c_7$, *the required evaluation order is* $c_8, c_5, c_1, c_2, c_3, c_4, c_6, c_7, c_9$. *We must use wire delays to produce this order. Change* c_8 *can be processed on arrival, and* c_9 *must be delayed until all the other changes have taken place. This can be done with one delay on the third wire. Next,* c_1, c_2, c_3 *must be delayed until* c_5 *arrives; this requires 3 delays on the first wire, etc.* □

Proposition 3 *If each input wire* w_i *of any gate has a transient of length* t_i *and* $t_i - 1$ *delays, then the evaluation order required for the longest output transient can be produced for any arrival order.*

Proof. Since each wire can store its transient, we can allow all the signal changes to arrive without processing any of them. Graph $D(\mathbf{x}_{i_1}, \ldots, \mathbf{x}_{i_r})$ defined in Section 4 then gives an order in which the signal changes should be processed to produce the longest transient. □

In order to use only one delay per wire we need to restrict the arrival orders. An arrival order is *initial* if all the wire winners arrive before any runner-up.

Proposition 4 *If each input wire of an* AND *gate has one delay, then the evaluation order of Algorithm ANDI can be produced for any initial arrival order.*

Proof. If the arrival order is initial, then no runner-up (01) belonging to a transient beginning with 101 can occur until all the winners of transients beginning with 01 have occurred. Thus we only need to delay at most one 10 change on a wire until there are no more changes to be handled by Algorithm AND01. Algorithm ANDI now becomes Algorithm AND1.

Consider the first application of Algorithm AND1. If the 101 rule on wire w_i is used first, the arrival order must have the form $u, (10)_i, v, (01)_i, w$, where $(10)_i$ and $(01)_i$ are the winner and runner-up on w_i, respectively, and u and v contain only winners (10 changes), each on a wire other than w_i, and each involving a unique wire, since 10 changes cannot be consecutive on the same wire. All winners in u are delayed by their individual wire delays. The change $(10)_i$ is performed on arrival, and all winners in v are delayed. The runner-up $(01)_i$ is processed on arrival. The reduced transient on w_i, left after the winner $(10)_i$ and the runner-up $(01)_i$ have been processed and removed, still begins with a 1, and there are no changes stored on the delay of w_i at this time. The delay on any other wire holds at most one winner (10), and this change will be evaluated before any more changes are to be stored on that delay. Hence we can view the new order as u, v, w. Since all the input transients again begin with 1, and the total number of changes has been reduced, we can apply this argument inductively. □

It is clear that the same approach works for NAND gates, except that the output values are complements of those of the corresponding AND gate. A dual approach applies to OR and NOR gates. The case of XOR or XNOR gates is the easiest one. Since every evaluation order leads to the output transient of maximum length, the changes can be processed in the order of arrival. Also, the case of inverters and FORKS is trivial, since they have only one input, and there is only one way to process the arriving changes. Similar approaches work for functions obtained from the functions above by complementing some of the inputs.

Lemma 1 *Let f be any gate from the set $\mathcal{G}$, with the set W_{in} of input wires and the set W_{out} of output wires. Let W be a set of wires such that $W_{in} \subseteq W$ and $W \cap W_{out} = \emptyset$. If the arrival order for W is initial and each wire in W_{in} has one delay, then f can process all the signal changes on W_{in} to produce the longest output transient, in such a way that the arrival order for $(W \setminus W_{in}) \cup W_{out}$ is still initial.*

Proof. Since the arrival order for W is initial and $W_{in} \subseteq W$, the arrival order for W_{in} is also initial. Since each wire in W_{in} has one delay, by the observations above, the initial arrival order for W_{in} can be changed to an evaluation order producing

the longest output transient. We now concentrate on the winner c_1 and runner-up c_2 of an output wire in W_{out}.

In the case of the AND gate, we claim that c_1 can happen immediately after some winner in W_{in}, and before any runner-up in $W \setminus W_{in}$. If all the input transients are 0 or 1, then there are no output changes; hence these cases are trivial. This leaves the following two cases:

- If at least one input transient starts with 0, let c be the last winner of type 01, among all the winners of type 01 on the gate's input wires, to be processed through the gate. Then c_1 is a 01 change on the output of the AND gate, and it can happen right after c, before any runners-up occur on $W \setminus W_{in}$.

- If all the input transients start with 1, the output of the gate is 1. Let c be the first winner in W_{in} to be processed through the gate; this is necessarily a 1 to 0 change. It is clear that winner c_1 can occur immediately after winner c, and before any runner-up in $W \setminus W_{in}$ occurs.

In summary, since the arrival order for W is initial, we can make winner c_1 happen before any runner-up of $(W \setminus W_{in}) \cup W_{out}$.

Next we claim that c_2 can be made to occur after all the winners in $W \setminus W_{in}$. Now we assume that there are at least two output changes. There are two cases:

- If at least one input transient starts with 0, then the output is initially 0, c_1 is a 01 change, and c_2 is a 10 change. Now c_2 can be the result of a winner 10, or of a runner-up 10.

  If it is the result of a winner change c, we wait until all the winner changes arrive. Then we process c through the gate, creating c_2. Thus c_2 can be made to happen after all the winners have occurred.

  If c_2 is the result of a runner-up change c, then c_2 occurs after all the winners, since the arrival order is initial.

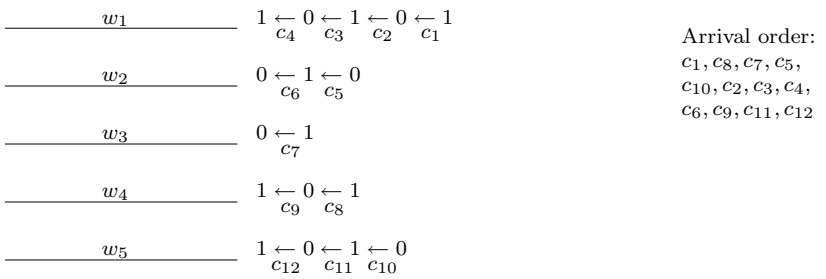
  Since the arrival order for W is initial, we can also make c_2 happen after all the winners of $(W \setminus W_{in}) \cup W_{out}$.

- If all the input transients start with 1, then the output is initially 1. Since there are at least two output changes, at least one input transient must begin with 101. We take that transient t_i beginning with 101 whose runner-up is earliest; let the winner of t_i be c and the runner-up, c'. We first process c, producing c_1, and follow it by c', which results in c_2. Thus c_2 can happen after all winners have arrived.

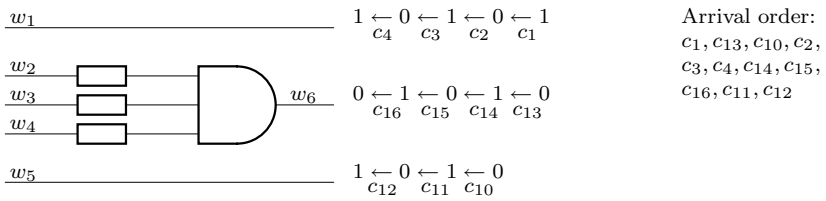
A similar situation occurs if f is any other gate from the set $\mathcal{G}$. Therefore the arrival order for $(W \setminus W_{in}) \cup W_{out}$ can be made initial. □

Example 5 *This is an example to illustrate Lemma 1 in the case of* AND *gate. In Fig. 11(a), let* $W = \{w_1, w_2, w_3, w_4, w_5\}$ *be a set of wires , and let the arrival order for* W *be*

$$c_1, c_8, c_7, c_5, c_{10}, c_2, c_3, c_4, c_6, c_9, c_{11}, c_{12};$$



(a) The arrival order before adding AND gate.



(b) The arrival order after adding AND gate.

Fig. 11. Example to illustrate Lemma 1.

this order is initial. Let $W_{in} = \{w_2, w_3, w_4\}$ *be the set of input wires of an* AND *gate, and let* $W_{out} = \{w_6\}$ *be the (singleton) set of output wires. Note that* $W_{in} \subseteq W$ *and* $W \cap W_{out} = \emptyset$*. According to Lemma 1, the signal changes are processed as follows:*

(i) We begin with $w_2 = 0$*,* $w_3 = 1$*, and* $w_4 = 1$*. Thus we want to use Algorithm AND01, that is, process* c_5 *first. Change* c_1 *arrives and has no effect on the gate.*

(ii) c_8 *arrives and is held by the delay.*

(iii) c_7 *arrives and is held by the delay.*

(iv) c_5 *arrives and propagates through the gate, which creates* c_{13}*. We now have* $w_2 = 1$*,* $w_3 = 1$*, and* $w_4 = 1$*, and so we use Algorithm AND1. More specifically, we want to process* c_8 *and* c_9*, when* c_9 *arrives.*

(v) c_{10}*,* c_2*,* c_3*, and* c_4 *arrive, in that order. They have no effect on the gate.*

(vi) c_6 *arrives and is held by the delay.*

(vii) c_9 *is about to arrive next. Hence* c_8 *propagates through the gate, creating* c_{14}*.*

(viii) c_9 *arrives, propagates through the gate, and creates* c_{15}*. We have* $w_2 = 1$*,* $w_3 = 1$*, and* $w_4 = 1$ *again, and the transients left on the gate inputs are 10 on* w_2 *and* w_3*, and* 1 *on* w_4*.*

(ix) c_7 *propagates through the gate, creating* c_{16}*.*

(x) c_6 *propagates through the gate; nothing changes.*

(xi) c_{11} *and* c_{12} *arrive.*

The winner c_{13} on the output wire arrives immediately after c_5, the winner of w_2. Hence c_{13} arrives before any runner-up. Runner-up c_{14} on the output wire arrives immediately before c_9, which is the runner-up on w_4. Hence c_{14} arrives after all the winners. The arrival order for $(W \setminus W_{in}) \cup W_{out}$ is

$$c_1, c_{13}, c_{10}, c_2, c_3, c_4, c_{14}, c_{15}, c_{16}, c_{11}, c_{12},$$

which is still initial.

9. Covering of Simulation in Wire-Delay Extensions

It is shown in [7] that, if wire delays are not considered, binary analysis of a network may not cover its simulation. Let $\hat{N}$ be the version of a network N expanded with respect to a particular transient simulation. We show that binary analysis of $\hat{N}$ covers this transien t simulation of N. Furthermore, if N is a network consisting only of gates from $\mathcal{G}$, then transient simulation of N is covered by binary analysis of its singular network $\dot{N}$.

Theorem 1 *Let N be a network constructed of arbitrary gates, let* **N** *be its transient netw ork, and $\hat{N}$, its expanded network. Binary analysis of $\hat{N}$ covers transient simulation of* **N**.

Proof. The external input transients of **N** are of length 1 or 2. In the binary analysis, the corresponding sequences of binary signals on the wires leaving the input gates appear as well. Thus all the wires of level 0 have the required binary sequences. Now assume that a gate of level k has the correct binary sequence corresponding to the final transient on each input wire. By Proposition 3, it is possible to produce the correct binary sequence on the gate's output wire. Hence the induction step goes through, and the theorem holds. □

Theorem 1 states that, for a network N and a particular transient simulation of **N**, we can always find an expanded network $\hat{N}$ with respect to this transient simulation such that binary analysis of $\hat{N}$ covers this transient simulation of **N**. This result has two limitations. First, different transient simulations of the same network may result in different expanded networks. Hence there is no 1-1 correspondence between a network and its expanded network. Second, the numbers of wire delays cannot be bounded uniformly and tend to grow fast as the number of levels in the network increases. Our next theorem improves this result using the additional properties of transients given in Section 7. From now on, we only consider networks consisting of gates in $\mathcal{G}$.

Theorem 2 *Let N be a network constructed with gates from $\mathcal{G}$, let* **N** *be its transient network, and $\dot{N}$, its singular network. Binary analysis of $\dot{N}$ covers transient simulation of* **N**.

Proof. In transient simulation, each transient on an external input has length zero or one. Thus, in the corresponding binary analysis, each input also has zero or one changes. As usual, we assume that in binary analysis the inputs change all at once [3]. By using input gates, we can produce any arrival order at the outputs of the input gates. In particular, we can produce some arrival order in which no two

changes arrive at the same time. This arrival order is necessarily initial, since each wire that changes has exactly one change, that is, all changes are winners. Thus, if W_1 is the set of wires of level 1, that is the set of output wires of input gates, the changes in W_1 have an initial arrival order. This provides a basis for a proof by induction on the number of non-input gates. We prove that, if we add a gate, we can produce the longest transient output for that gate, and also cause the arrival order of signal changes on the wires connected to external outputs to be initial.

Suppose N_n is a network with $n \geq 0$ non-input gates, and W_n is the set of wires connected to external outputs. By the inductive hypothesis, we assume that the arrival order of changes on the wires in W_n is initial.

Let y be a new gate (logic gate or FORK) with input wire set W_y and output wire set W^*, where $W_y \subseteq W_n$. By Lemma 1, we can use wire delays in such a way that y produces the longest transient output and the arrival order on $(W_n \setminus W_y) \cup W^*$ can be made initial. Therefore the statement holds for $n+1$ gates. This shows that we can reproduce the transient that occurs at the end of simulation on the output wire of a gate in binary analysis. Consequently, binary analysis covers the simulation. □

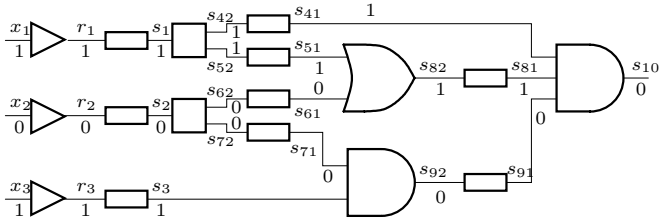


Fig. 12. Example circuit.

Example 6 *The circuit of Fig. 12 is started in the stable total state shown in the figure. The inputs change from 101 to 010. The input transients in the simulation are* $\mathbf{x_1} = 10$, $\mathbf{x_2} = 01$, *and* $\mathbf{x_3} = 10$. *We use square brackets to indicate the new signal changes being added to the list at each step.*

We can pick an arbitrary arrival order by using input delays r_1, r_2, r_3. *We select the order*

$$[(10)_{r_1}, (01)_{r_2}, (10)_{r_3}].$$

We illustrate our algorithm using Table A.1, where new values are shown in boldface. Row 0 shows the initial stable state, and the three changes in the r_i *are in Rows 1–3 of the table. The external outputs of the network considered so far are* r_1, r_2, *and* r_3, *and the set of winners is* $\{(10)_{r_1}, (01)_{r_2}, (10)_{r_3}\}$; *the set of runners-up is empty, and the arrival order is initial.*

We decide next to add the fork with input s_1. *The order for the fork outputs can be selected arbitrarily, and we chose* $(10)_{s_{42}}$ *to be first. Thus we perform the changes* $(10)_{s_1}$, $(10)_{s_{42}}, (10)_{s_{52}}$ *in that order, as shown in Rows 4–6. Since both* $(10)_{s_{42}}$ *and* $(10)_{s_{52}}$ *are winners, we can merge the two lists* $((10)_{r_1}, (01)_{r_2}, (10)_{r_3})$ *and* $((10)_{s_1}, (10)_{s_{42}}, (10)_{s_{52}})$ *freely, as long as* $(10)_{s_1}$ *occurs after* $(10)_{r_1}$. *One extended order is*

$$(10)_{r_1}, (01)_{r_2}, (10)_{r_3}, [(10)_{s_1}, (10)_{s_{42}}, (10)_{s_{52}}].$$

The external outputs of the network considered so far are s_{42}, s_{52}, r_2, and r_3, and the set of winners is $\{(01)_{r_2}, (10)_{r_3}, (10)_{s_{42}}, (10)_{s_{52}}\}$; the set of runners-up is empty. The arrival order is initial.

Now we add the fork with input s_2, as shown in Rows 7–9. This time we choose to change s_{72} first. The order becomes

$$(10)_{r_1}, (01)_{r_2}, (10)_{r_3}, (10)_{s_1}, (10)_{s_{42}}, (10)_{s_{52}}, [(01)_{s_2}, (01)_{s_{72}}, (01)_{s_{62}}].$$

The external outputs of the network considered so far are s_{42}, s_{52}, s_{62}, s_{72}, and r_3, and the set of winners is $\{(10)_{r_3}, (10)_{s_{42}}, (10)_{s_{52}}, (10)_{s_{72}}, (10)_{s_{62}}\}$; the set of runners-up is empty. The arrival order is initial.

We can now add the OR *gate. According to Algorithm OR01 (dual of AND01), we must change s_{51} first, and then s_{61}. Note that, after we change s_{51}, we must change s_{82}. Then we change s_{61}, and change s_{82} again. Thus we perform the changes $(10)_{s_{51}}, (10)_{s_{82}}, (01)_{s_{61}}, (01)_{s_{82}}$ in that order, as shown in Rows 10–11 and 20–21. Note that the change $(01)_{s_{82}}$, being a runner-up, must arrive after all the winners. Change $(10)_{s_{51}}$ must occur after $(10)_{s_{52}}$, and $(01)_{s_{61}}$ must occur after $(01)_{s_{62}}$. Therefore the order becomes*

$$(10)_{r_1}, (01)_{r_2}, (10)_{r_3}, (10)_{s_1}, (10)_{s_{42}}, (10)_{s_{52}}, (01)_{s_2}, (01)_{s_{72}}, (01)_{s_{62}},$$

$$[(10)_{s_{51}}, (10)_{s_{82}}, (01)_{s_{61}}, (01)_{s_{82}}].$$

The external outputs of the network considered so far are s_{42}, s_{82}, s_{72}, and r_3, and the set of winners is $\{(10)_{r_3}, (10)_{s_{42}}, (10)_{s_{72}}, (10)_{s_{82}}\}$; the set of runners-up is $\{(01)_{s_{82}}\}$. The arrival order is initial.

We can now add the AND *gate with input s_3 and s_{71}. According to Algorithm AND01, we perform the changes $(01)_{s_{71}}, (01)_{s_{92}}, (10)_{s_3}, (10)_{s_{92}}$ in that order, as shown in Rows 12–13 and 18–19. Change $(10)_{s_{92}}$, being a runner-up, must arrive after all the winners. The order becomes*

$$(10)_{r_1}, (01)_{r_2}, (10)_{r_3}, (10)_{s_1}, (10)_{s_{42}}, (10)_{s_{52}}, (01)_{s_2}, (01)_{s_{72}}, (01)_{s_{62}},$$

$$(10)_{s_{51}}, (10)_{s_{82}}, [(01)_{s_{71}}, (01)_{s_{92}}, (10)_{s_3}, (10)_{s_{92}}], (01)_{s_{61}}, (01)_{s_{82}}.$$

The external outputs of the network considered so far are s_{42}, s_{82}, and s_{92}. The set of winners is $\{(10)_{s_{42}}, (01)_{s_{92}}, (10)_{s_{82}}\}$; the set of runners-up is $\{(01)_{s_{82}}, (10)_{s_{92}}\}$. The arrival order is initial.

Finally we add the last AND *gate. According to Algorithm AND01, we perform the changes $(01)_{s_{91}}, (01)_{s_{10}}, (10)_{s_{81}}, (10)_{s_{10}}, (01)_{s_{81}}, (01)_{s_{10}}, (10)_{s_{41}}, (10)_{s_{10}}$, $(10)_{s_{91}}$ in that order, as shown in Rows 14–17 and 22–26. The order becomes*

$$(10)_{r_1}, (01)_{r_2}, (10)_{r_3}, (10)_{s_1}, (10)_{s_{42}}, (10)_{s_{52}}, (01)_{s_2}, (01)_{s_{72}}, (01)_{s_{62}},$$

$$(10)_{s_{51}}, (10)_{s_{82}}, (01)_{s_{71}}, (01)_{s_{92}}, [(01)_{s_{91}}, (01)_{s_{10}}, (10)_{s_{81}}, (10)_{s_{10}}],$$

$$(10)_{s_3}, (10)_{s_{92}}, (01)_{s_{61}}, (01)_{s_{82}}, [(01)_{s_{81}}, (01)_{s_{10}}, (10)_{s_{41}}, (10)_{s_{10}}, (10)_{s_{91}}].$$

This is the final arrival order for the whole network. The final external output of the network is s_{10}.

In Theorem 1 we show that, for a general network N, the binary analysis of its expanded network covers the transient simulation of its transient network. The expanded network does not have constant bounds on the numbers of delays on each wire. The question remains whether there exists a network with a constant number of delays on each wire such that its binary analysis covers the simulation. The answer is negative. Consider a 3-input gate A with inputs x_1, x_2 and x_3 realizing the Boolean function f_A: $x_1 * \overline{x_3} + x_2 * x_3$. If $x_3 = 0$, the output is equal to x_1; otherwise, it is x_2. Suppose that x_1 and x_2 both come from a fork, as in Fig. 13, and that the fork input wire has a transient $(10)^k$. Then the transients on x_1 and x_2 are both $(10)^k$. Suppose further that the transient on x_3 is 10. Then the extension of the function f_A has the value $f_A((10)^k, (10)^k, 10) = (10)^{2k}$. In order to get the same number of changes in binary analysis, we require at least $2k - 1$ delays in the input wire of x_1 to hold the signal changes on x_1. Therefore, there is no constant bound on the number of wire delays for this gate.

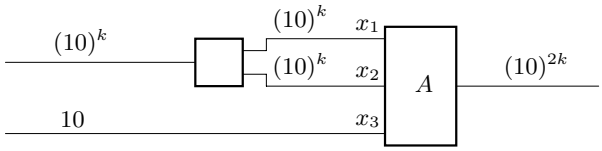


Fig. 13. A counterexample with a 3-input Boolean function.

Acknowledgements

This research was supported by the Natural Sciences and Engineering Research Council of Canada under grant No. OGP0000871. The authors are greatly indebted to Mihaela Gheorghiu for her constructive criticisms of several versions of this paper.

References

1. J. A. Bondy and U. S. R. Murty, *Graph Theory with Applications* (American Elsevier, 1976).
2. J. A. Brzozowski and M. Gheorghiu, "Gate circuits in the algebra of transients," *Theoretical Informatics and Applications,* **39**, pp. 67–91, 2005.
3. J. A. Brzozowski and C.-J. H. Seger, *Asynchronous Circuits* (Springer, 1995).
4. J. A. Brzozowski and Z. Ésik, "Hazard algebras," *Formal Methods in System Design,* **23**, pp. 223–256, 2003.
5. J. A. Brzozowski, Z. Ésik, and Y. Iland, "Algebras for hazard detection," *Beyond Two - Theory and Applications of Multiple-Valued Logic*, M. Fitting, and E. Orłowska, eds., pp. 3–24 (Physica-Verlag, Heidelberg, 2003).
6. E. B. Eichelberger, "Hazard detection in combinational and sequential switching circuits," *IBM J. Research and Development,* **9**, pp. 90–99, 1965.
7. M. Gheorghiu, *Circuit Simulation Using a Hazard Algebra,* MMath Thesis, (Department of Computer Science, University of Waterloo, Waterloo, ON, Canada, 2001)
8. M. Gheorghiu and J. A. Brzozowski, "Simulation of feedback-free circuits in the algebra of transients," *Int. J. of Found. of Computer Science,* **14(6)** pp. 1033–1054, 2003.

9. D. E. Muller, *A Theory of Asynchronous Circuits*. Technical Report 66, (Digital Computer Laboratory, University of Illinois, Urbana-Champaign, Illinois, USA, 1955).

10. D. E. Muller and W. S. Bartky, A theory of asynchronous circuits, *Proceedings of an International Symposium on the Theory of Switching*, pp. 204–243, Annals of the Computation Laboratory of Harvard University, (Harvard University Press, 1959).

Appendix A:

Table A.1. Path in binary analysis.

	r_1	r_2	r_3	s_1	s_2	s_3	s_{42}	s_{41}	s_{52}	s_{51}	s_{62}	s_{61}	s_{72}	s_{71}	s_{82}	s_{81}	s_{92}	s_{91}	s_{10}
0	1	0	1	1	0	1	1	1	1	1	0	0	0	0	1	1	0	0	0
1	**0**	0	1	1	0	1	1	1	1	1	0	0	0	0	1	1	0	0	0
2	0	**1**	1	1	0	1	1	1	1	1	0	0	0	0	1	1	0	0	0
3	0	1	**0**	1	0	1	1	1	1	1	0	0	0	0	1	1	0	0	0
4	0	1	0	**0**	0	1	1	1	1	1	0	0	0	0	1	1	0	0	0
5	0	1	0	0	0	1	**0**	1	1	1	0	0	0	0	1	1	0	0	0
6	0	1	0	0	0	1	0	1	**0**	1	0	0	0	0	1	1	0	0	0
7	0	1	0	0	**1**	1	0	1	0	1	0	0	0	0	1	1	0	0	0
8	0	1	0	0	1	1	0	1	0	1	1	0	**1**	0	1	1	0	0	0
9	0	1	0	0	1	1	0	1	0	1	**1**	0	1	0	1	1	0	0	0
10	0	1	0	0	1	1	0	1	0	**0**	1	0	1	0	1	1	0	0	0
11	0	1	0	0	1	1	0	1	0	0	1	0	1	0	**0**	1	0	0	0
12	0	1	0	0	1	1	0	1	0	0	1	0	1	**1**	0	1	0	0	0
13	0	1	0	0	1	1	0	1	0	0	1	0	1	1	0	1	**1**	0	0
14	0	1	0	0	1	1	0	1	0	0	1	0	1	1	0	1	1	**1**	0
15	0	1	0	0	1	1	0	1	0	0	1	0	1	1	0	1	1	1	**1**
16	0	1	0	0	1	1	0	1	0	0	1	0	1	1	0	**0**	1	1	1
17	0	1	0	0	1	1	0	1	0	0	1	0	1	1	0	0	1	1	**0**
18	0	1	0	0	1	**0**	0	1	0	0	1	0	1	1	0	0	1	1	0
19	0	1	0	0	1	0	0	1	0	0	1	0	1	1	0	0	**0**	1	0
20	0	1	0	0	1	0	0	1	0	0	1	**1**	1	1	0	0	0	1	0
21	0	1	0	0	1	0	0	1	0	0	1	1	1	1	**1**	0	0	1	0
22	0	1	0	0	1	0	0	1	0	0	1	1	1	1	1	**1**	0	1	0
23	0	1	0	0	1	0	0	1	0	0	1	1	1	1	1	1	0	1	**1**
24	0	1	0	0	1	0	0	**0**	0	0	1	1	1	1	1	1	0	1	1
25	0	1	0	0	1	0	0	0	0	0	1	1	1	1	1	1	0	1	**0**
26	0	1	0	0	1	0	0	0	0	0	1	1	1	1	1	1	0	**0**	0