Theoretical Computer Science 245 (2000) 3–25

Theoretical Computer Science

www.elsevier.com/locate/tcs

Delay-insensitivity and ternary simulation [☆]

J.A. Brzozowski

Department of Computer Science, University of Waterloo, Waterloo, Ont., Canada N2L 3G1

Abstract

Consider a network N constructed from a set of modules interconnected by wires. Suppose that there is a formal specification Σ for N, and that the behavior of N satisfies this specification. Let $\hat{N}$ consist of the same modules, but suppose that these modules and the interconnecting wires have arbitrary delays. We say that N is delay-insensitive if the behavior of any network $\hat{N}$ defined as above still satisfies Σ. An important problem in asynchronous circuits is to determine, given a specification Σ and a set $\mathscr{T}$ of module types, whether there exists a delay-insensitive network of modules from $\mathscr{T}$ with a behavior satisfying Σ. If such a network exists, we say that it implements Σ delay-insensitively. In the case where the components are logic gates, it is known that very few specifications have delay-insensitive implementations. The proofs of several such results involve "ternary simulation" – an analysis method based on ternary algebra – and rely on a key theorem linking binary analysis and ternary simulation. In this paper we survey the known results concerning delay-insensitivity, and outline one proof that a simple specification cannot be implemented. © 2000 Elsevier Science B.V. All rights reserved.

Keywords: Asynchronous; Circuit; Delay-insensitive; Fundamental mode; General multiple-winner; Input/output mode; Network; Semi-modular; Speed-independent; Ternary algebra; Ternary simulation

1. Introduction

Although most currently used digital circuits are synchronous, that is, operate under the control of a clock signal, there is considerable interest in asynchronous circuits [9]. Clock-free design offers some advantages, like the potential for lower-energy consumption, higher speed, and avoidance of clock distribution problems.

Among asynchronous designs, the class of so-called delay-insensitive networks is receiving special attention. Roughly speaking, a network is delay-insensitive if it con-

[☆] This research was supported by the Natural Sciences and Engineering Research Council of Canada under grant No. OGP0000871.

E-mail address: brzozo@uwaterloo.ca (J.A. Brzozowski).

tinues to operate correctly, even if the delays in its components and wires change arbitrarily. When such networks are designed in a modular fashion, it is possible to replace their components by faster or slower ones, without changing the correctness of the network operation, although, of course, performance may be affected.

The following discussion attempts to treat delay-insensitivity in a very general fashion. For this reason, we do not define the concepts formally, but appeal to the reader's intuition. More precise statements are presented later.

A module is a basic element, like a gate or an arbiter. A network consists of a set of modules interconnected by wires. With each network we associate its behavior. For example, a behavior may be specified by a trace structure (a set of sequences of possible actions), or by a finite automaton, or by a directed graph. We also assume that there is a notion of "implementation" of one behavior by another. Now suppose a network N is constructed from a set of modules, and that its behavior implements some specification Σ. Let $\hat{N}$ consist of the same modules and interconnecting wires, but suppose that these modules and wires have arbitrary delays. We say that N is *delay-insensitive* if the behavior of any network $\hat{N}$ defined as above still implements Σ.

An important problem in asynchronous circuits is to determine, given a specification Σ of an asynchronous behavior and a set $\mathscr{T}$ of module types, whether there exists a delay-insensitive network of modules from $\mathscr{T}$ that implements Σ. In the case where the components are logic gates, it is known that very few specifications have delay-insensitive implementations. In this paper we state several such results; their proofs involve "ternary simulation" – an analysis based on ternary algebra – and rely on a key theorem linking binary analysis and ternary simulation. To illustrate the applicability of ternary simulation, we discuss a proof that a very simple behavior has no delay-insensitive implementation by a gate circuit.

The paper is structured as follows. In Section 2 we briefly survey the previous results concerning delay-insensitivity, concentrating on the various definitions of delay-insensitivity. We then define our model of gate networks in Section 3, and describe the binary model of network behavior in Section 4. The ternary analysis method known as "ternary simulation" is next presented in Section 5, and the main results relating ternary simulation and binary analysis are summarized. The limitations of gate networks operated in fundamental mode are discussed in Section 6 and Section 7 treats circuits operated in input/output mode. Section 8 concludes the paper.

2. Delay-insensitivity

We now briefly survey the literature on delay-insensitivity, concentrating on the formal definitions of the concept, rather than on design methods resulting in delay-insensitive circuits. Most of these definitions involve such concepts as "a circuit B satisfies a specification A", or "a circuit with added delays (B) behaves like the original circuit A", or "a circuit B implements a behavior A". Because of space limitations, we are unable to make these concepts much more precise. We point out, however, that many different approaches have been taken in the past to formalize these concepts.

The formalizations range from requiring that "A and B go to the same final stable state", to "A and B have the same trace structures", to "A and B have the same trace structures and deadlock is not possible in B", etc. Because of this diversity, we refer the interested reader to the original sources.

A very early (1955–1959) work concerned with delays in circuits is that of Muller on "speed-independent" circuits. This work was described in a number of technical reports from the University of Illinois (see pp. 242–243 in [30]) and also in [34], but is perhaps more easily available in the book by Miller [30]. Muller considered *autonomous* circuits, i.e., circuits without external inputs. He assumed that components could have arbitrary delays, but that wire delays were negligible. In Muller's approach, the behavior of a circuit is described by *allowed sequences* of states, which specify the order in which the state variables may change, but not the times of change. Only changes of state are recorded; thus, no two consecutive states in an allowed sequence are equal. An allowed sequence may be finite – in which case it must end in a state which has no successors – or infinite. Any given state may have a number of possible successor states; the choice of successor depends on the relative sizes of circuit delays.

Two states b and c of a circuit are *equivalent* if there is an allowed sequence in which b follows c and also c follows b. The relation "follows" induces a partial order on the equivalence classes of a circuit. Consequently, every allowed sequence has a unique *terminal* class. A circuit is said to be *speed-independent with respect to a state q*, if all allowed sequences starting with q have the same terminal class. In case the terminal classes consist of single states, this condition implies that, starting in q, the network can only end up in a unique stable state. Otherwise, if the terminal class has more than one state, then it corresponds to a cycle of states, and all the paths starting from q must end up in the same cycle. Because of differences in component delays, there may be many different paths from q. If the circuit is speed-independent, however, the final destination (state or cycle) of any path from a given state is uniquely determined by that state.

Muller defined another important concept related to proper circuit operation, namely the property of semi-modularity. A circuit is *semi-modular with respect to state q*, if the following condition is satisfied by every state b reachable from q: No transition from one state to another can affect the excitation of any unstable variable that does not change during that transition. More precisely, if b is a state in which s_i is unstable and c is an immediate successor state of b, then either s_i changes to its excitation or s_i is still unstable in state c. Semi-modular circuits are a proper subclass of speed-independent circuits, that is, every circuit that is semi-modular with respect to q is also speed-independent with respect to q, but the converse is false.

An early (1959) result due to Unger deals with limitations of delay-insensitive circuits [44, 45], although Unger does not use the term "delay-insensitive", since it had not yet been introduced. Unger has shown that an asynchronous behavior that contains a so-called essential hazard is not implementable by a gate network with arbitrary gate and wire delays. We return to this result in Section 6.

The term *delay-insensitive circuit* originated from the Macromodules project, which was carried out at Washington University in St. Louis, Missouri around 1970 (see [32] for a short description of the project). This term was used informally. C.E. Molnar introduced the "foam-rubber wrapper" postulate to describe delay-insensitive specifications of network modules [32]. A module has a number of input terminals on which it receives signals from its environment, and a number of output terminals on which it sends signals to the environment. Because of wire delays, however, signals produced by the environment are not seen by the module right away, but only after some delay. Similarly, signals produced on the module outputs are observed by the environment only after some delay. The interface delays can be represented as a "foam-rubber wrapper" surrounding the module. The inner surface of the wrapper corresponds to the module interface, whereas the outer surface defines the environment interface. The foam-rubber analogy suggests that the distance between the inner and outer surfaces along one wire may be different from that along another wire, representing different (and possibly time-varying) delays.

The communications between a component and its environment are specified by "traces", which are sequences of events at the input and output terminals of the component. Formally, a *trace structure* [15, 41, 47] T is a triple $T = \langle \mathbf{i}T, \mathbf{o}T, \mathbf{t}T \rangle$, where $\mathbf{i}T$ is the *input alphabet*, $\mathbf{o}T$ is the *output alphabet*, and $\mathbf{t}T$ is the *trace set*. The input and output alphabets are two disjoint finite sets corresponding to the input and output terminals of the module. An element x of $\mathbf{i}T$ (respectively, an element z of $\mathbf{o}T$) is interpreted as a signal at the input terminal x (respectively, output terminal z). The set $\mathbf{a}T = \mathbf{i}T \cup \mathbf{o}T$ is called the *alphabet* of T. A word w in $(\mathbf{a}T)^*$ is a *trace*. Since a trace represents a history of possible signals, any prefix of a trace must itself be a trace. Thus, $\mathbf{t}T$ is *prefix-closed* in this sense, and always contains the empty word ε that represents the initial state in which no signals have yet been sent. In trace terms, the foam-rubber wrapper postulate requires that the set of traces of a module/environment specification at the inner surface of the wrapper must be the same as the set of traces at the outer surface. For example, suppose a specification (at the inner surface) requires that a module produce output z_1 and then output z_2, in that order. Trace z_1z_2 would then be in its trace set, but z_2z_1 would not be in the set. Because we assume that the delays in the output wires are unknown and arbitrary, z_2 might appear before z_1 at the outer surface, or the correct order might be preserved. Thus, the trace set at the outer surface contains both z_1z_2 and z_2z_1. Since the inner and outer trace sets differ, the module specification is not delay-insensitive.

An important contribution to the design of asynchronous circuits is the work of Seitz (1980) on "self-timed" systems (Chapter 7 in [29]). This work contains an informal description of a design methodology of a class of asynchronous circuits. A *self-timed* system consists of modules performing computations. A computation by a module is initiated by signals at its inputs, and its completion is indicated by signals at its outputs. The modules occupy a small enough area on a VLSI chip to justify the assumption that wire delays are negligible. Such areas are called *equipotential regions*. Thus, the modules themselves are delay-sensitive. A self-timed network of such modules, however, should operate correctly even in the presence of arbitrary wire delays. Inside an equipotential region the designer, knowing some bounds on the various delays, may cause one signal to arrive before another, and this order of signals will be preserved. Such order assumptions cannot be made about communication protocols among equipotential regions. The system must be so designed that a module will not be called upon to initiate a new computation until it has completed its present computation, and has signalled this completion to its environment.

The first formal definition of delay-insensitivity was made by Udding in 1984 [41, 43]. Udding suggests that a delay-insensitive specification should satisfy the following two conditions. First, there should be no *computation interference,* meaning informally that no signal should be sent to a module if that module is not ready to receive it. Similarly, the module should not send a signal to the environment, if the latter is not ready to receive it. Second, there should be no *transmission interference,* meaning informally that two consecutive signals cannot be sent along any wire. Udding then defines a trace structure $T = \langle \mathbf{i}T, \mathbf{o}T, \mathbf{t}T \rangle$ of a module to be delay-insensitive if it satisfies the four rules stated below. Two symbols a and b in $\mathbf{a}T$ are said to be of the *same type* if they are both inputs or both outputs; otherwise, they are of *different types*. The rules are:

1. There should never be two consecutive signals along the same wire. Formally, for any $w \in \mathbf{t}T$ and $a \in \mathbf{a}T$, $waa \notin \mathbf{t}T$.
2. There should not be any ordering between input signals of a module, and the same holds for output signals. Formally, for any $u, v \in \mathbf{t}T$, and a and b of the same type, $uabv \in \mathbf{t}T$ if and only if $ubav \in \mathbf{t}T$.
3. Suppose signals a and b are of different types, and c is of the same type as a. If a and b can occur in either order at some point, and if c can occur when the order is ab, then it can also occur when the order is ba. Formally, for any $u, v \in \mathbf{t}T$, if $uabv \in \mathbf{t}T$ and $ubav \in \mathbf{t}T$, then $uabvc \in \mathbf{t}T$ implies $ubavc \in \mathbf{t}T$.
4. If a and b are signals of different types, and both are enabled to occur after some trace w, then the occurence of one signal should not disable the occurrence of the other signal. Formally, for any $w \in \mathbf{t}T$ and a and b of different types, if $wa \in \mathbf{t}T$ and $wb \in \mathbf{t}T$, then $wab \in \mathbf{t}T$.

Udding has shown that, if a delay-insensitive trace structure and its environment satisfy the four rules above [41], then there is no computation or transmission interference. He has also studied several variations of his rules and the corresponding classes of trace structures.

A different formalization of the foam-rubber wrapper postulate was developed by Schols in 1985, in terms of a certain composition operation on trace structures [35]. This formalization does not include the absence of transmission interference, since there exist trace structures that satisfy the foam-rubber wrapper postulate of Schols, but fail Udding's Rule 1.

Delay-insensitivity of arbiters satisfying fairness conditions was studied by Martin [27] Black [2] and Udding [42, 43] The conclusions about the existence or non-existence of delay-insensitive arbiters depend on the definition of fairness and delay-insensitivity.

In 1987 Ebergen studied decompositions of asynchronous components (behaviors) specified by trace structures into delay-insensitive connections of basic components [15, 16]. A component $S.0$ can be *decomposed* into components $S.i$, $i = 1,\dots,n$ if the following conditions (stated only informally here) are met:

1. The system $\{S.i \mid i = 0,\dots,n\}$ is closed, in the sense that every output of every component is connected to an input of another component, and the same is true if "input" and "output" are interchanged.
2. No two outputs are connected together.
3. The connection behaves as specified by $S.0$. (The environment of the connection of the n components is assumed to be $S.0$.)
4. There is no computation interference.

Ebergen's work can be viewed as another formalization of the foam-rubber wrapper postulate. Informally, a decomposition is delay-insensitive, if it is still a decomposition (in the sense given above) after wire delays have been added. A component S is said to be delay-insensitive, if it operates correctly in the presence of wire delays. Ebergen derives properties of decompositions that permit the decomposition of a specification in a hierarchical way, and that enable us to decompose parts of a specification separately. He proves that, if the basic components are delay-insensitive, then the concept of decomposition coincides with the concept of delay-insensitive decomposition. Ebergen proves that his definition of delay-insensitive component is equivalent to Udding's definition. Thus, a component is delay-insensitive in Ebergen's sense if and only if it satisfies Udding's four rules. Ebergen also develops systematic methods for decomposing a specification into a network of basic components.

Chu developed (1987) methods of designing speed-independent circuits from *signal transition graphs,* which are a form of interpreted Petri nets [13]. A condition closely related to semi-modularity is called *persistence* by Chu. Signal transition methods have been widely used; see, for example, [25].

In 1988, Dill developed methods for the specification and automatic verification of asynchronous circuits [14]. He used the term "speed-independent" in an informal sense, but he did introduce a formal definition of delay-insensitivity. Dill uses a modified trace theory for behavior specifications. Dill's trace structure is a 4-tuple $T = \langle \mathbf{i}T, \mathbf{o}T, \mathbf{s}T, \mathbf{d}T \rangle$, where $\mathbf{i}T$ and $\mathbf{o}T$ are finite disjoint sets of *inputs* and *outputs*, $\mathbf{s}T$ is a set of *successes* (successful traces), and $\mathbf{d}T$ is a set of *divergences* (called *failures* by Dill). The sets $\mathbf{s}T$ and $\mathbf{d}T$ are (not necessarily disjoint) prefix-closed, regular sets of traces over the alphabet $\mathbf{a}T = \mathbf{i}T \cup \mathbf{o}T$. The set $\mathbf{p}T = \mathbf{s}T \cup \mathbf{d}T$ is the set of *possible* traces. This set is nonempty, since the empty trace is always possible. The traces in $(\mathbf{a}T)^* - \mathbf{p}T$ represents the set of *impossible* traces. Trace structures must be *receptive*, meaning that $(\mathbf{p}T)(\mathbf{i}T) \subseteq \mathbf{p}T$. Thus, if a trace w is possible, and x is a module input, then trace wx is possible because the module cannot prevent the environment from sending a signal.

We illustrate this model for a logic gate. If the gate is unstable, and the input changes, making the gate stable, the corresponding trace is a divergence (failure), because it corresponds to a hazard. (In Udding's terminology, this represents computation interference.) However, an input may change successfully if the gate is unstable before and after the change.

Dill defines the concept "trace structure T conforms to trace structure S" as follows. Suppose that S forms part of a network N that operates correctly in the sense that the failure set of its trace structure is empty. Then *T conforms to S in N* if the failure set of the network in which S has been replaced by T is still empty. Finally, *T conforms to S* if it conforms to S in all correct networks. Next, Dill defines the operator DI on trace structures. Intuitively, DI adds delays to all the input and output terminals of a trace structure, hides the original terminals, and renames the new inputs and outputs to the original names. This corresponds to adding a foam-rubber wrapper to the module. A trace structure T is said to be delay-insensitive if and only if $DI(T)$ conforms to T.

In 1988 Seger gave a new proof of Unger's theorem on essential hazards [9, 37, 38]. This proof is discussed further in Section 6.

In 1989 Brunvand and Sproull [4] developed a module-based approach to the design of delay-insensitive circuits. They used a subset of the language OCCAM to specify behaviors, and designed delay-insensitive modules for each language construct (for example, a *while* loop). This permits automatic compilation of programs to networks of modules. See also [5].

In 1990 He, Josephs, and Hoare proposed a mathematical model for dataflow communication [19]. The model is similar to CSP [3, 20] in that it consists of sets of traces, failures and divergences, but, unlike CSP, it uses directed events (i.e., either input or output events), as in trace theory. In this CSP-like model, an axiomatic framework for asynchronous processes was developed. This mathematical model was used to provide the semantic underpinning for a new process algebra.

Udding and Verhoeff noticed the relationship between Udding's model of delay-insensitive communication and the above model of dataflow communication. The main differences are as follows: (1) He et al. [19] consider both safety and progress, whereas Udding [43] considers only safety, (2) Udding's rules are of a convex-closure type because he works with safe traces, whereas the structure of the dataflow model gives rise to a simpler reordering rule, and (3) in dataflow communication one does not need to consider transmission interference.

DI-Algebra [22, 23] was then created in 1990 by modifying the process algebra to include the treatment of transmission interference. Josephs subsequently defined a hierarchy of models [21]: the most general (receptive process) is suitable for modelling and analyzing speed-independent circuits, the next, incorporating the reordering rule, is applicable to dataflow, and the third, incorporating the transmission interference rule, is applicable to delay-insensitivity. Later Lucassen [26] proved the laws of DI-Algebra sound with respect to the third model.

Van Berkel [46] investigated an even more constrained model for "handshaking" communication. He restricted components to those that require only that their environment should respect the handshaking protocol on communication ports. The alphabets of *handshake processes* have more structure than those of trace structures.

However, for every handshake process, one can define a corresponding trace structure. A handshake process is then considered delay-insensitive if its corresponding trace structure is delay-insensitive. Van Berkel proves that his handshake processes are delay-insensitive.

In 1990 Martin essentially used semi-modularity, called *stability* by him, as a *definition* of delay-insensitivity [28]. (Brzozowski and Zhang have proved recently that this assumption can, indeed, be justifiable [11]. This is briefly discussed later.) In [28], Martin considers limitations of a restricted class of delay-insensitive circuits. The modules used by him include not only logic gates, but also some sequential modules such as C-elements. The networks Martin considers are autonomous.

In 1992 Brzozowski and Ebergen showed that very few behaviors are implementable delay-insensitively, if the circuits are operated in the so-called input/output mode [7]. These results were somewhat generalized in [9], and are discussed in Section 7.

Shintel and Yoeli specified (1992) behaviors by Petri nets [40]. Given a network N of modules they defined the *delay-extension* of N as the network $\hat{N}$ obtained from N by adding delays to the input and output wires of each module. Roughly speaking, they define a network N to be an implementation of a behavior $\mathscr{B}$ if the trace set of N is the same as that of $\mathscr{B}$, and also N does not deadlock, in the sense that it fails to produce an output when one is expected according to $\mathscr{B}$. A network is considered to be a delay-insensitive implementation of a behavior $\mathscr{B}$ if the delay-extension of the network is an implementation of $\mathscr{B}$.

Schols studied (1992) delay-insensitive communication [36]. He proposed another definition of delay-insensitivity in a somewhat different formalism, and argued that his definition is equivalent to those of Udding and Ebergen.

Lavagno and Sangiovanni-Vincentelli developed (1993) a methodology for asynchronous circuit design [25]. They specify behaviors by signal transition graphs, and assume that wire delays are bounded. The book [25] contains an extensive survey of previous work on asynchronous circuits. Signal transition graphs are classified “speed-independent with inertial delays”, “speed-independent with pure delay”, or delay-insensi -tive in Udding’s sense. Semi-modularity and related properties are also considered.

In 1994 Verhoeff extended Uddings rules to include progress as a correctness concern, in addition to absence of interference [49]. In Verhoeff’s terminology, a *process* is identified with a trace structure. An (*extended*) *process* is a quintuple $T = \langle \mathbf{i}T, \mathbf{o}T, \nabla T, \square T, \triangle T \rangle$, where $\mathbf{i}T$ and $\mathbf{o}T$ are disjoint input and output alphabets as before, and ∇T, $\square T$, and $\triangle T$ are pairwise disjoint trace sets with the properties given below. Traces in ∇T are *transient* traces; the upside down triangle suggests that the process state corresponding to a transient trace is unstable (will eventually topple) and will be followed by an output. For each transient trace w, there exists an output symbol $z \in \mathbf{o}T$ such that wz is in $\mathbf{t}T$, where $\mathbf{t}T = \nabla T \cup \square T \cup \triangle T$ is the set of traces of T. Traces in $\triangle T$ are *input-demanding* traces. For each input-demanding trace there exists an input symbol $x \in \mathbf{o}T$ such that wx is in $\mathbf{t}T$. States corresponding to input-demanding traces can persist indefinitely, if the environment fails to supply an input (the base of the triangle is stable). However, the environment is expected to produce an input for an input-demanding trace. Traces in $\Box T$ are *indifferent*; after such a state, neither the process nor the environment are obliged to make progress.

Verhoeff generalized Udding's rules to extended processes. Rules 1 and 4 remain unchanged; Rules 2, and 3 require modifications. Rule 2 becomes: For any $u, v \in \mathbf{t}T$, and a and b of the same type,

- $uabv \in \mathbf{t}T$ if and only if $ubav \in \mathbf{t}T$,
- $uabv \in \nabla T$ if and only if $ubav \in \nabla T$,
- $uabv \in \triangle T$ if and only if $ubav \in \triangle T$.

Rule 3 requires a similar modification.

Verhoeff has also shown [48, 49] that Udding's Rules 2–4 are equivalent to a single rule, which he calls the "neighbor-swap rule". Space limitations prevent us from providing further details; we refer the reader to [48, 49], where several other interesting characterizations of delay-insensitivity are given.

Kishinevsky, Kondratyev, Taubin, and Varshavsky published a book (1994) on an asynchronous design approach based on *change diagrams*, which are graphs similar to signal transition graphs [24]. For semi-modular circuits, the two models have similar modeling power.

In 1995 a survey of several asynchronous design methodologies was written by Hauck [18]; a revised and expanded version by Brzozowski, Hauck, and Seger appeared as Chapter 15 of [9].

In 1996 Bush and Josephs studied speed-independent circuits [12] in the formalism of receptive processes [21]. They use trace structures $T = \langle \mathbf{i}T, \mathbf{o}T, \mathbf{t}T \rangle$ with additional constraints. The set $\mathbf{f}T \subseteq \mathbf{t}T$ consists of *failures* that may take the circuit to an undesirable state or to a quiescent state from which no output is issued. The set $\mathbf{d}T \subseteq \mathbf{f}T$ consists only of traces leading to undesirable states, and is called the set of *divergences* of T. Dill's receptivity condition, as described above, is also added. The authors formulate five properties that hold for logic gates and also for other common basic elements. In all the items below, a is an input, b is either an input or an output, c is an output, and u and v are traces. The properties are

1. An enabled event should remain enabled after the occurrence of an input. Formally, if $ub \in \mathbf{t}T$ then $uab \in \mathbf{t}T$.
2. If $uba \in \mathbf{d}T$, then $ubav \in \mathbf{f}T$ implies $uabv \in \mathbf{f}T$.
3. If an input is unsafe after an output, then it is also unsafe before that output. Formally, if $uca \in \mathbf{d}T$, then $ua \in \mathbf{d}T$.
4. If $uv \in \mathbf{f}T$ then $uaav \in \mathbf{f}T$.
5. If $uaa \in \mathbf{d}T$, then $uv \in \mathbf{f}T$ if and only if $uaav \in \mathbf{f}T$.

The authors show that, if all the components used in the construction of a speed-independent circuit satisfy Property X, then so does the circuit, if X is 1, 2, 4, or 5, but there are speed-independent circuits constructed with components satisfying Property 3 that do not satisfy this property.

In 1997 Brzozowski and Zhang [11] studied networks of very general sequential modules, namely nondeterministic Moore type sequential machines [33]. Such modules include, for example, delays, logic gates, forks, latches, counters, C-elements, and arbiters. A single multi-valued state variable is used to describe the internal state of a module; this permits hiding the details of the internal structure of the module.

A network N of modules is *strongly delay-insensitive* if the behavior of any network $\hat{N}$ derived from N by the addition of delays is bisimilar [31] to the behavior of N. In effect, if N is strongly delay-insensitive, N and $\hat{N}$ can always simulate each other, if one looks only on the state variables of N. Note that this condition is stronger than conditions based on standard trace theory; for example, it permits the treatment of deadlock.

To permit the handling of nondeterministic modules, like arbiters, the concept of semi-modularity has been generalized. Suppose a network is in a state $s=(s_1,\dots,s_n)$, where s_i is the state of module i. Suppose further that modules i and j, $i\neq j$, are both unstable and module j can undergo a transition to state t_j. State s is *semi-modular* if, whenever module i changes in state s, causing a transition from state s to state s', module j is still able to undergo the transition to state t_j from s'. A network behavior is semi-modular if every state of the behavior reachable from the initial state is semi-modular.

Assuming that wire delays are included, it has been proved [11] that a network is strongly delay-insensitive if its behavior is semi-modular. Furthermore, it has been shown [50] that, if a network is strongly delay-insensitive, then its modules must obey Udding's rules. However, the converse result does not hold.

3. Gate networks

We now introduce our model of gate networks. This model is based on that of [11], but is much simpler (since we only consider combinational logic gates), and contains external inputs. We introduce the model informally first. Normally, by a (generalized) "gate" we mean a device with some number $m>0$ of inputs, where with each input we associate a binary input variable x_j, taking its values from the set $\{0,1\}$. The gate has an output s, which also takes its values from $\{0,1\}$, according to the excitation function S associated with the gate, where $S:\{0,1\}^m\to\{0,1\}$ is a function that maps every binary input m-tuple to 0 or 1. The value of s follows the value of S after some delay – the delay of the gate.

In the usual models, the output s of the gate can be fanned out to several other terminals, say, $z_1,\dots,z_p$. Thus, in effect, each gate output can be followed by a p-way fork, for some $p>0$. For mathematical convenience, we prefer to include such a fork in the definition of the gate. In this new light, the variable s becomes an internal state variable, and the fork outputs become the gate outputs. Note, however, that the value of each output is always equal to the value of the state variable, that is, we neglect delays in the output wires. This allows us to include delay-free forking. Since such forking assumptions are often made in practice, the model is convenient. On the other hand, if we do want to consider delays in output wires, we can do so in our network model described below.

Formally, a *combinational module* is a 4-tuple $M = \langle \mathscr{X}, s, \mathscr{Z}, S \rangle$, where $\mathscr{X}$, s, and $\mathscr{Z}$ are pairwise disjoint sets, and

- $\mathscr{X} = \{x_1, \dots, x_m\}$, $m \geqslant 0$, is the set of *module input variables*; also, $x = (x_1, \dots, x_m)$ is the ordered m-tuple of module input variables;
- s is the *module internal-state variable*;
- $\mathscr{Z} = \{z_1, \dots, z_p\}$, $p \geqslant 0$, is the set of *module output variables*; also, $z = (z_1, \dots, z_p)$ is the ordered p-tuple of module output variables;
- S is the *module excitation function*, $S : \{0,1\}^m \to \{0,1\}$.

It is convenient to associate with each module a directed graph, the *module graph* $G = \langle \mathscr{V}, \mathscr{E} \rangle$, where

- $\mathscr{V} = \mathscr{V}_{\mathscr{X}} \cup \mathscr{V}_s \cup \mathscr{V}_{\mathscr{Z}}$ is the set of *module vertices*, where
  - $\mathscr{V}_{\mathscr{X}}$ is the set of *module input vertices*, each with indegree 0 and outdegree 1; there is one such vertex for each input variable;
  - v_s is the *module internal-state vertex* with indegree m and outdegree p;
  - $\mathscr{V}_{\mathscr{Z}}$ is the set of *module output vertices*, each with indegree 1 and outdegree 0; there is one such vertex for each output variable;
- $\mathscr{E} = (\mathscr{V}_{\mathscr{X}} \times \{v_s\}) \cup (\{v_s\} \times \mathscr{V}_{\mathscr{Z}})$ is the set of *module edges*.

The module graph simply identifies the input and output terminals of the gate, and shows the dependence of the internal-state variable on the input variables, and the dependence of the output variables on the internal-state variable. This graph will be very convenient when we connect several gates to form a network of gates. □

Combinational modules include delays (or wires with delays), forks, and logic gates performing arbitrary Boolean functions with forked outputs. We illustrate our definition with three examples.

- *Delay*: $M = \langle \{x_1\}, s, \{z_1\}, S \rangle$, where $S(x_1) = x_1$, and $z_1 = s$.
- *3-output fork*: $M = \langle \{x_1\}, s, \{z_1, z_2, z_3\} \rangle$, where $S(x_1) = x_1$, and $z_1 = z_2 = z_3 = s$. Note that there is a delay from the fork input to the internal state variable s, but no delays from s to the outputs.
- *2-output majority gate*: $M = \langle \{x_1, x_2, x_3\}, s, \{z_1, z_2\}, S \rangle$, where S is given by the Boolean expression $S(x_1, x_2, x_3) = x_1x_2 + x_2x_3 + x_3x_1$, and $z_1 = z_2 = s$.

We now consider networks of modules. When several different modules are involved, we will distinguish them with superscripts. A network has some number $m > 0$ of input terminals with associated variables $x_1, \dots, x_m$; it has m input forks, so that each input may be distributed to several modules; and it has some number n of network modules. An input fork is a module, but we will refer to it simply as “input fork” reserving the term “module” for the network components. Each external input terminal is also an input terminal of an input fork. Note that an input fork may have just one output, and that it always has a delay. Thus every external input to the network has a delay; this delay is introduced as a mathematical convenience. Every output of an input fork is connected to exactly one module input terminal, every module output terminal is connected to exactly one module input terminal, and every module input terminal is connected to exactly one fork output terminal, or to exactly one module output terminal. Thus, there are no “dangling” terminals, except for the external input terminals.

It is convenient to think about a network as a directed graph containing input fork vertices and module vertices. Formally, a *network* is an ordered 4-tuple $N = \langle \mathscr{X}, \mathscr{F}, \mathscr{M}, G \rangle$, where

- $\mathscr{X} = \{x_1, \ldots, x_m\}$ is the set of *network input variables*;
- $\mathscr{F} = \{F_1, \ldots, F_m\}$ is the set of *input forks*, where fork F_i has input X_i and p_i outputs, $p_i \geqslant 1$;
- $\mathscr{M} = \{M^1, \ldots, M^n\}$, $n \geqslant 0$, is a set of modules;
- $G = \langle \mathscr{V}, \mathscr{E} \rangle$ is a directed graph, where
  - $\mathscr{V} = \mathscr{V}_{\mathscr{X}} \cup \mathscr{V}_{\mathscr{F}} \cup \mathscr{V}_{\mathscr{I}}$ is the set of *network vertices*, where $\mathscr{V}_{\mathscr{X}}$, $\mathscr{V}_{\mathscr{F}}$ and $\mathscr{V}_{\mathscr{I}}$ are pairwise disjoint, and
    - ○ $\mathscr{V}_{\mathscr{X}}$ is the set of *network-input vertices*, each with indegree 0 and outdegree 1, there being one vertex for each input variable, which is also an input to an input fork;
    - ○ $\mathscr{V}_{\mathscr{F}}$ is the set of *input-fork output vertices*, there being p_i outputs for fork F_i;
    - ○ $\mathscr{V}_{\mathscr{I}} = \bigcup_{i=1}^{n} \mathscr{V}^i$ is the set of *network internal vertices*, where $\mathscr{V}^i$ is the set of vertices in the graph of module M^i, $\mathscr{V}^i = \mathscr{V}_{\mathscr{X}^i} \cup \mathscr{V}_{\mathscr{Z}^i}$, where $\mathscr{V}_{\mathscr{X}^i}$ and $\mathscr{V}_{\mathscr{Z}^i}$ are the sets of input and output vertices of module M^i, respectively.
  - $\mathscr{E}$ is the set of edges, $\mathscr{E} = \bigcup_{j=1}^{m} \mathscr{E}^j \cup \bigcup_{i=1}^{n} \mathscr{E}^i \cup K$, where $\mathscr{E}^j$ is the set of edges of the input fork connected to X_j, $\mathscr{E}^i$ is the set of edges in the graph of module M^i, and

$$\mathscr{K} \subseteq \left(\mathscr{V}_{\mathscr{F}} \cup \bigcup_{i=1}^{n} \mathscr{V}_{\mathscr{Z}^i} \right) \times \left(\bigcup_{i=1}^{n} \mathscr{V}_{\mathscr{X}^i} \right)$$

    is the set of *connections* of N. Furthermore, each vertex in $\mathscr{V}_{\mathscr{F}}$ and in each $\mathscr{V}_{\mathscr{Z}^i}$ has exactly one outgoing edge, and each vertex in each $\mathscr{V}_{\mathscr{X}^i}$ has exactly one incoming edge.

The set $\mathscr{S} = \{s^1, \ldots, s^n\}$ of module internal-state variables is now considered as the set of *network state variables*. The set $\delta = (\delta_1, \ldots, \delta_n)$, is the *network excitation vector*, where δ_i is the excitation function S^i of module M^i with arguments changed as follows. If $(v_{z_h^j}, v_{x_k^i})$ is a connection, then the kth argument of δ_i is s^j.

To take into account wire delays, we can insert a delay module in any connection of a network as follows. Suppose $e = (v_i, v_j)$ is a connection edge in which we wish to insert a delay module M^{d} with module graph vertices $v_{x_1^{\mathrm{d}}}$, $v_{s^{\mathrm{d}}}$, and $v_{z_1^{\mathrm{d}}}$. We add M^{d} to the set of modules of N. We add vertices $v_{x_1^{\mathrm{d}}}$, $v_{s^{\mathrm{d}}}$, and $v_{z_1^{\mathrm{d}}}$ to the set of internal vertices of N. We remove edge e, and add edges $(v_i, v_{x_1^{\mathrm{d}}})$ and $(v_{z_1^{\mathrm{d}}}, v_j)$, as well as the edges of M^{d}. Finally, we add the excitation δ^{d} to the vector of excitations.

We illustrate our network model by the circuit of Fig. 1. The network has one external input x, and hence one input fork, with input x, internal state variable s_0, and two outputs. The OR gate is a 2-input, 2-output module, the inverter is a 1-input, 1-output module, and the NAND gate is a 3-input, 1-output module. There are 6 connections altogether, one for each output of the input fork and one for each module output: from the input fork to the OR gate and the NAND gate, from the OR gate to itself and the inverter, from the inverter to the NAND gate, and from the NAND gate to itself. The

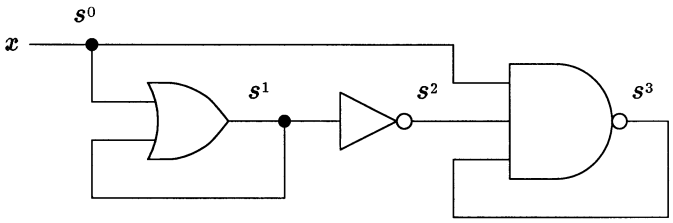


Fig. 1. A gate network.

state of network is given by the vector of values of (s^0, s^1, s^2, s^3). The excitations are as follows: $\delta_0 = x$, $\delta_1 = s^0 + s^1$, $\delta_2 = \overline{s^1}$, and $\delta_3 = \overline{s^0 s^2 s^3}$.

4. Autonomous behaviors of gate networks

In this section we analyze the behavior of a gate network in the binary general multiple-winner (GMW) model, assuming that the network input tuple has some fixed binary value. This material is based on the work of Muller and Bartky [34], which is also described in [30]. However, notationally, our treatment is closer to [9, 10].

An assignment of a binary m-tuple a to the m-tuple x of network input variables is an *input state*. An assignment of a binary n-tuple $b \in \{0,1\}^n$ to the n-tuple s of state variables is an *internal state*. The pair (a,b) is a *total state*. The *excitation* of N in total state (a,b) is the n-tuple $B = B_1, \dots, B_n = \delta_1(a,b), \dots, \delta_n(a,b)$. A state b is *stable* if $B_i = b_i$ for all i; otherwise, b is *unstable*. The set

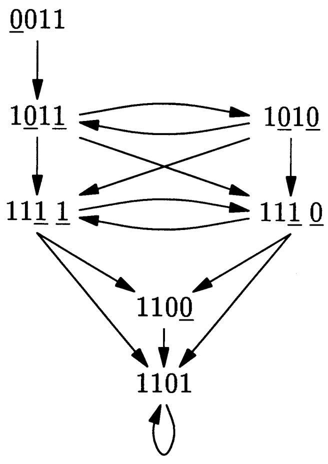
$$\mathscr{U}(b) = \{s^i \in \mathscr{S} \mid B_i \neq b_i\}$$

is the set of unstable variables in state b.

Assume that the input m-tuple is fixed at $x = a$. The *GMW behavior* of a network N started in state q and having fixed input a is an initialized directed graph $\mathscr{B}_a(q) = \langle q, \mathscr{Q}, R_a \rangle$, where $q \in \{0,1\}^n$ is the *initial state*, $\mathscr{Q} = reach(\mathscr{B}_a(q))$ is the set of states reachable from q by using the GMW relation R_a, where R_a is defined by bR_ac if b is stable and $b = c$, or c is obtained from b by complementing all the variables in some nonempty set of unstable variables. The state variables that are unstable in a given state are in a *race* to change. Allowing several variables to change corresponds to multiple winners of the race. Since this model makes no assumptions about the relative sizes of delays, it is the "general multiple-winner" model.

Since the behavior graph is finite, every path from q must eventually reach a cycle. A cycle in $\mathscr{B}$ is *transient* if there exists a state variable s^i that has the same value in all the states of the cycle and is unstable in each state of the cycle. A behavior cannot remain in a transient cycle longer than the delay of the variable s^i. Let the set of *cyclic states* reachable from q in the behavior graph of $\mathscr{B}_a(q)$ be

$$cycl(\mathscr{B}_a(q)) = \{b \in \{0,1\}^m \mid qR_a^* b \text{ and } bR_a^+ b\},$$



$$\begin{tikzcd}
\underline{0}011 \arrow[d] & & \\
1\underline{0}1\underline{1} \arrow[rr, bend left] \arrow[d] \arrow[rrd] & & 1\underline{0}1\underline{0} \arrow[ll, bend left] \arrow[d] \arrow[lld] \\
11\underline{1}\,\underline{1} \arrow[rr, bend left] \arrow[rd] \arrow[rdd] & & 11\underline{1}\,\underline{0} \arrow[ll, bend left] \arrow[ld] \arrow[ldd] \\
 & 110\underline{0} \arrow[d] & \\
 & 1101 \arrow[loop below] &
\end{tikzcd}$$

Fig. 2. Network behavior.

where R_a^+ is the transitive closure of R_a, and R_a^* is the reflexive and transitive closure of R_a. Next, define the set of nontransient cyclic states to be

$$cycl\_nontrans(\mathscr{B}_a(q)) = \{b \mid b \text{ appears in a nontransient cycle}\}.$$

The *outcome* of the transition from q under input a is the set of all the states that are reachable from a state in a nontransient cycle. Mathematically, we have

$$out\,(\mathscr{B}_a(q)) = \{b \mid qR_a^*c \text{ and } cR_a^*b, \text{ where } c \in cycl\_nontrans(\mathscr{B}_a(q))\}.$$

Like Muller's terminal class [30, 34], the outcome consists of all the states that represent the "final destination" of the behavior. In the presence of arbitrary, but bounded, gate and wire delays, after a sufficiently long time the network must be in one of the states in the outcome [9].

To illustrate these concepts let us return to the network of Fig. 1. Suppose the network is started in state $(s^1, s^2, s^3, s^4) = 0011$, and the input is fixed at $x = 1$. Then the GMW behavior graph is as shown is Fig. 2, where unstable state components are underlined. Cycles $(1011, 1010)$, and $(1111, 1110)$ are transient, whereas the cycle (1101) is nontransient. The outcome of the transition is the singleton set $\{1101\}$.

5. Ternary simulation

This section is based on [8, 9, 17, 39]. Ternary simulation, originally introduced by Eichelberger, is an analysis method based on three logic values, 0, 1, and Φ. The three values are partially ordered by the *uncertainty partial order* $\sqsubseteq$, defined as

$$0 \sqsubseteq 0, \quad 1 \sqsubseteq 1, \quad \Phi \sqsubseteq \Phi, \quad 0 \sqsubseteq \Phi \text{ and } 1 \sqsubseteq \Phi$$

and no other pairs are related by $\sqsubseteq$. The values 0 and 1 are thought of as *certain*, whereas Φ is the *uncertain* value. The partial order $\sqsubseteq$ is extended to $\{0,\Phi,1\}^n$ in the natural way. Also, the concept of least upper bound *lub* is defined for this partial order as usual.

Instead of using the Boolean module excitation functions δ_i, we use their ternary extensions $\boldsymbol{\delta}_i$. For a Boolean function $f:\{0,1\}^n \to \{0,1\}$, its *ternary extension* $\mathbf{f}: \{0,\Phi,1\}^n \to \{0,\Phi,1\}$ is

$$\mathbf{f}(\mathbf{a}) = lub\{f(b) \mid b \in \{0,1\}^n \text{ and } b \sqsubseteq \mathbf{a}\}$$

for all $\mathbf{a} \in \{0,\Phi,1\}^n$. Note that any Boolean function f agrees with its ternary extension $\mathbf{f}$ when the argument $\mathbf{a}$ is binary.

A network using the binary domain will be denoted by N, whereas its ternary counterpart will be $\mathbf{N}$.

Ternary simulation consists of two algorithms, A and B. The network is assumed to be started in total state (a,q). In the first algorithm, the variables of unstable modules are assigned the value Φ. This introduces some uncertain values into the network, and this uncertainty may then spread to other parts of the network. Algorithm A is formally defined as follows:

```
**Algorithm A**
$h := 0$;
$\mathbf{s}^0 := q$;
**repeat**
  $h := h+1$;
  $\mathbf{s}^h := lub\{\mathbf{s}^{h-1}, \boldsymbol{\delta}(a,\mathbf{s}^{h-1})\}$;
**until** $\mathbf{s}^h = \mathbf{s}^{h-1}$;
```

Proposition 1. *Algorithm* A *generates a finite sequence* $\mathbf{s}^0,\ldots,\mathbf{s}^{\mathrm{A}}$ *of states, where* $A \leqslant n$. *Furthermore, this sequence is monotonically increasing, i.e.,*

$$\mathbf{s}^h \sqsubset \mathbf{s}^{h+1} \quad \textit{for } 0 \leqslant h < A.$$

The final outcome of Algorithm A is the ternary state $\mathbf{s}^{\mathrm{A}}$.

In Algorithm B, some of the uncertainty introduced by Algorithm A may be removed; we start the network in the state generated by Algorithm A.

```
**Algorithm B**
$h := 0$;
$\mathbf{t}^0 := \mathbf{s}^{\mathrm{A}}$;
**repeat**
  $h := h+1$;
  $\mathbf{t}^h := \boldsymbol{\delta}(a,\mathbf{t}^{h-1})$;
**until** $\mathbf{t}^h = \mathbf{t}^{h-1}$;
```

Proposition 2. *Algorithm* B *generates a finite sequence* $\mathbf{t}^0, \ldots, \mathbf{t}^B$ *of states, where* $B \leqslant n$. *Furthermore, this sequence is monotonically decreasing, i.e.,*

$$\mathbf{t}^h \sqsupseteq \mathbf{t}^{h+1} \quad \textit{for } 0 \leqslant h < B.$$

We illustrate ternary simulation using the network of Fig. 1, started in state 0011, with input fixed at $x = 1$. Algorithm A yields the following sequence of states:

$$0011 \to \Phi011 \to \Phi\Phi1\Phi \to \Phi\Phi\Phi\Phi$$

and Algorithm B yields

$$\Phi\Phi\Phi\Phi \to 1\Phi\Phi\Phi \to 11\Phi\Phi \to 110\Phi \to 1101.$$

The following two theorems are the fundamental results relating GMW analysis and ternary simulation. A network is said to be *complete* if a delay is included for each network connection wire.

The two theorems characterize the results of Algorithms A and B for delay-complete networks. Basically, Algorithm A detects whether a variable changes at all in the GMW analysis. More precisely, if an internal state variable s^i does not change in any possible sequence of states reachable from the initial state (a, q) in the GMW analysis of the network, then the initial value q_i is assigned to the corresponding variable $\mathbf{s}_i^{\mathrm{A}}$ in the result of Algorithm A. If a variable does change (one or more times), then it is assigned the value Φ.

Algorithm B detects whether a variable may keep changing after the outcome of the GMW analysis is reached. In other words, Algorithm B ignores the transients and concentrates on the eventual "terminal classes" of states. More specifically, if a variable has the same value in all the states of the outcome of the GMW analysis (though it may have changed before the outcome is reached), than that (binary) value is assigned to that variable. But, if a variable can oscillate between 0 and 1 in the outcome, or if it has the value 0 in one stable state and the value 1 in another stable state, then that variable is assigned the value Φ by Algorithm B. To put it another way, if the result of Algorithm B has a binary value for a variable, then that variable is guaranteed to have that binary value in all the states of the outcome of the GMW analysis. Otherwise, it has the value Φ, indicating uncertainty.

Theorem 1. *Let* $N = \langle \mathscr{M}, G \rangle$ *be a complete binary network, and let* $\mathbf{N} = \langle \mathbf{M}, G \rangle$ *be its ternary counterpart. If* N *and* $\mathbf{N}$ *are started in total state* (a, q), *then the result* $\mathbf{s}^{\mathrm{A}}$ *of Algorithm* A *for* $\mathbf{N}$ *is equal to the lub of the set* $\mathscr{Q}$ *of all the states reachable from the initial state in the GMW behavior of* N, *i.e.,*

$$\mathbf{s}^{\mathrm{A}} = \textit{lub}\, \mathscr{Q}.$$

Theorem 2. *Let* $N = \langle \mathscr{M}, G \rangle$ *be a complete binary network, and let* $\mathbf{N} = \langle \mathbf{M}, G \rangle$ *be its ternary counterpart. If* N *is started in total state* (a, q) *and* $\mathbf{N}$ *in total state* $(a, \mathbf{s}^{\mathrm{A}})$, *then the result* $\mathbf{t}^{\mathrm{B}}$ *of Algorithm* B *is equal to the lub of the outcome of the GMW analysis, i.e.,*

$$\mathbf{t}^{\mathrm{B}} = lub\ out\ (\mathscr{B}_a(q)).$$

6. Delay-insensitivity in fundamental mode

This section and the next one discuss some limitations of delay-insensitive networks constructed with gates as basic components. We remind the reader that a gate is an m-input, one-output device capable of performing an arbitrary Boolean function of m variables. The multiple-outputs provided in our modules are simply forks. If more complex components are used, then the results below do not apply.

For many applications, the desired behavior of an asynchronous network consists of a series of transitions among stable states. To be more precise, we assume that the network starts in some initial state (a^0, b^0). The environment then changes the input to a^1, say, and waits for the circuit to reach a stable state (a^1, b^1). The environment may then change the input to a^2, etc. In general, a network is said to be operated in *fundamental mode* if its inputs are permitted to change only if the entire network is stable. Note that two assumptions are requireed for this definition. First, it is assumed that the network does, indeed, reach a unique stable state after every input change. Thus, a behavior in which two different stable states could be reached, depending on the relative sizes of the network delays, is considered erroneous. Also, oscillations are considered undesirable. Second, one has to have an estimate of the time required for a network to reach a stable state after any input change. The environment of the network then waits for that time to elapse, before sending new inputs.

Formally, the analysis of a network operated in fundamental mode consists of a series of computations of the outcome using GMW analysis. A transition of a network N, from a stable state $(\hat{a}, b)$ under new input vector a, is said to be *delay-insensitive in fundamental mode* if and only if $out\,(\mathscr{B}_a(b))$ contains a single state, where $out\,(\mathscr{B}_a(b))$ is the outcome of the GMW analysis in the complete network model. Note that this definition involves only a rather crude correctness criterion. We are interested only in the final state reached, and not in any intermediate states. But, even with this rough criterion, delay-insensitive behaviors are very limited. In particular, an old result due to Unger [44, 45] states that a behavior is not delay-insensitively implementable by a gate network if it contains a so-called essential hazard. Suppose a behavior is in a stable state $(\hat{a}, b)$ and the input changes first to a, then back to $\hat{a}$, and again to a. The behavior has an essential hazard if the state reached after the first input change is different from that reached after the third input change. A new proof of this result in a formal model has been given by Seger [9, 37, 38]. This proof is based on Theorems 1 and 2.

Theorem 3. *Let N be any network and let $(\hat{a}, \hat{b})$ be a stable state of N. If $(\hat{a}, \hat{b})$ to (a, b), and (a, b) to $(\hat{a}, \tilde{b})$ are delay-insensitive transitions of N, then so is the*

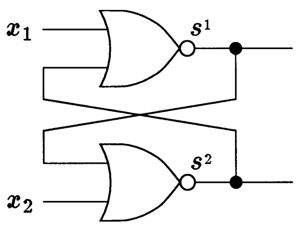


Fig. 3. NOR latch.

transition from $(\hat{a}, \tilde{b})$ *under input* a, *and the result of this transition is again the state* (a, b).

It follows from this theorem that common behaviors such as those of a JK flip-flop or a modulo-2 counter cannot be realized delay-insensitively by any gate circuit. The next result shows that this is also true for arbiters. This result was first proved in [1] in a totally different formalism.

A simple arbiter can be implemented by the NOR latch of Fig. 3. For $i = 1, 2$, a 1 on input x_i indicates no request, and a 0 indicates a request. A 0 on an output s_i indicates that output i has not been granted its request, and a 1 indicates that the request has been granted. The case where there are no requests is represented by both inputs being 1; hence, both outputs are 0. If x_1 becomes 0, while x_2 remains 1, output s_1 becomes 1, while s_2 remains 0. Thus a request on input 1 results in a grant on output 1. Similarly, a request on input 2 results in a grant on output 2. So far, the latch behaves properly like an arbiter. Suppose, however, that two requests arrive simultaneously; then x_1 and x_2 both become 0, while s_1 and s_2 are also both 0. Now both outputs are trying to change to 1. In such a situation, a physical latch circuit enters a metastable state, in which both outputs take on voltage values that are intermediate between those corresponding to logical 0 and logical 1. There is then a danger that a device using these outputs as inputs may interpret both outputs as 1, indicating grants on both lines, which violates mutual exclusion. Eventually, one of the outputs "wins" and becomes 1, while the other becomes 0. However, it is not possible to put an upper bound on the time during which the latch is metastable. Thus a simple latch is not a reliable implementation of an arbiter.

In our formal model, if we were to use a "single-winner" model, then the "race" between the output variables can be won by one output or the other. Thus, when both inputs become 0, we can only have the transitions $00 \to 10$ and $00 \to 01$. The outcome of the transition is the set $\{10, 01\}$, and mutual exclusion is not violated. But the "multiple-winner" GMW model permits also an oscillation $00 \to 11 \to 00 \to 11 \cdots$. This is a very crude model of metastability. Here the outcome is $\{00, 01, 10, 11\}$, which violates mutual exclusion.

Although we have seen that the latch is not a good implementation of an arbiter, the question still remains whether there exists any gate circuit in which the outcome can be limited to 01 and 10. The theorem below provides a negative answer to this question. It is proved using a construction from the proof of Theorem 2 [9, 39]:

Theorem 4. *No arbiter behavior has a delay-insensitive implementation by a gate circuit operated in fundamental mode.*

7. Delay-insensitivity in input/output mode

Informally, a network is operated in *input/output mode* [6, 7, 9] if the environment is allowed to change the network's input only after receiving an appropriate output signal from the network. Unfortunately, there is no general formal definition of input/output mode, and this area needs further work. For our purposes, we will only require some rather simple examples. In fact, it has been shown [7, 9] that the simple behavior given below has no delay-insensitive input/output mode implementation. We use the concept of *whole state* to represent asynchronous behaviors. A whole state is a total state together with the output; since the output is determined by total state, this is redundant, but we use it for convenience. We denote a whole state (a, b, c) by $a{\cdot}b{\cdot}c$.

Consider a behavior $\mathscr{B}_0$ with one binary input x, one binary output z, and two internal states 0 and 1. The output is always equal to the internal state.

$$0{\cdot}0{\cdot}0 \xrightarrow{\{x\}} 1{\cdot}0{\cdot}0 \xrightarrow{\{z\}} 1{\cdot}1{\cdot}1 \xrightarrow{\{x\}} 0{\cdot}1{\cdot}1.$$

The behavior starts in stable whole state $0{\cdot}0{\cdot}0$. When the input changes to 1, the behavior should respond by changing its output to 1. After that, the environment can change the input back to 0, and the behavior should never again change its output. Note that this behavior is delay-insensitively implemented in fundamental mode by an OR gate with output z and inputs x and z. The situation changes, however, if input/output mode is used. The following result was first proved in [7], and presented in a slightly more general framework in [9]:

Theorem 5. *Behavior $\mathscr{B}_0$ has no delay-insensitive implementation by a gate circuit operated in input/output mode.*

In spite of the simplicity of the behavior, it is not easy to prove this theorem. The problem, of course, is that we must show that no matter how we construct a gate network, it can never behave like $\mathscr{B}_0$. Here again the ternary simulation theorems come to the rescue.

First, we need to establish some general properties of the proposed solution. Suppose that a network N implementing $\mathscr{B}_0$ exists. Then this network must have one binary input x, one binary output z, and some number of internal states. There must be at least one internal state, say b, that represents the initial state, and the output in that state must be 0. Thus, we can assume that our network N starts in whole state $0{\cdot}b{\cdot}0$. We are not allowed to assume that this state is stable. In fact, N could be in an oscillation; as long as the output is 0, the behavior is correct. To cover this situation, we introduce a typical state $0{\cdot}c{\cdot}0$ to which a transition from $0{\cdot}b{\cdot}0$ is allowed. Now, in any state like $0{\cdot}c{\cdot}0$, the environment is permitted to change the input to 1; this must result in some

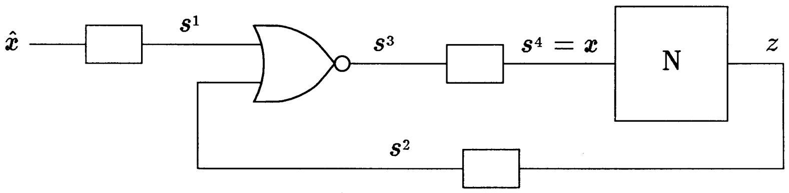


Fig. 4. Network $\hat{N}$.

state $1{\cdot}d{\cdot}0$. Note that, since we know nothing about the internal state of N, we allow it to change at the same time as the input changes. We do know, however, that this last state $1{\cdot}d{\cdot}0$ cannot be stable, because an output change must be produced. Some state (or states) of the form $1{\cdot}e{\cdot}0$ might be entered, but a state of the form $1{\cdot}f{\cdot}1$ must eventually be reached. In summary, our network N must have the following type of behavior corresponding to the first input change:

$$0{\cdot}b{\cdot}0 \to 0{\cdot}c{\cdot}0 \xrightarrow{\{x\}} 1{\cdot}d{\cdot}0 \to 1{\cdot}e{\cdot}0 \xrightarrow{\{z\}} 1{\cdot}f{\cdot}1.$$

Once the output changes, the environment is allowed to change x back to 0, resulting in some state $0{\cdot}h{\cdot}1$. After that, the internal state of N may change again, but never its output. Thus, we may have another typical state $0{\cdot}i{\cdot}1$. The possible state sequence is

$$0{\cdot}b{\cdot}0 \to 0{\cdot}c{\cdot}0 \xrightarrow{\{x\}} 1{\cdot}d{\cdot}0 \to 1{\cdot}e{\cdot}0 \xrightarrow{\{z\}} 1{\cdot}f{\cdot}1 \to 1{\cdot}g{\cdot}1 \xrightarrow{\{x\}} 0{\cdot}h{\cdot}1 \to 0{\cdot}i{\cdot}1.$$

Note that many other internal states may occur; we only show what might be typical.

Assuming that N exists, we construct a network $\hat{N}$ using N, as shown in Fig. 4. First, we find the GMW behavior of $\hat{N}$. Let the total state of $\hat{N}$ be $\hat{x}{\cdot}s^1s^2s^3s^4y{\cdot}z$, where $s_4 = x$ is the input to N, y is its internal state, and z is its output. Note that the output of N must be derived from some delay, since we are using the complete network model for N; thus, the output is part of the state. (Formally, we should add the input fork delay between $\hat{x}$ and the s_1 input of the NOR gate, but this would not change the argument.) Network $\hat{N}$ will be operated with a constant input. It is started in state $0{\cdot}1000b{\cdot}0$, that is, the input delay is unstable, and N is in its initial state $0{\cdot}b{\cdot}0$. It is clear that $\hat{N}$ must reach a state of the form $0{\cdot}0011d{\cdot}0$. According to the behavior of N, we must eventually reach a state of the form $1{\cdot}f{\cdot}1$; hence, $\hat{N}$ reaches $0{\cdot}0011f{\cdot}1$. Next, delay s_2 will change to 1, changing s_3 and s_4 to 0. According to the behavior of N, there will be no observable changes after that. Hence, the outcome of this transition of $\hat{N}$ must be a state (or states) of the form $0{\cdot}0100i{\cdot}1$. The important point to notice is that GMW analysis predicts 1 as the final value of z.

Suppose now that we analyze $\hat{N}$ using ternary simulation. The state sequence for Algorithm A is

$$0{\cdot}1000b{\cdot}0 \to 0{\cdot}\Phi000y{\cdot}0 \to 0{\cdot}\Phi0\Phi0y{\cdot}0 \to 0{\cdot}\Phi0\Phi\Phi y{\cdot}0,$$

where y is the unknown (and irrelevant) state of N. Since we know that both state $0{\cdot}1000b{\cdot}0$ and state $0{\cdot}0011f{\cdot}1$ are reachable in the GMW analysis of $\hat{N}$, we know by

Theorem 1 that the final value of z in Algorithm A is Φ. Thus the result of Algorithm A must be $0 \cdot \Phi\Phi\Phi\Phi y \cdot \Phi$.

If we next apply Algorithm B starting with state $0 \cdot \Phi\Phi\Phi\Phi y \cdot \Phi$, we obtain the result $0 \cdot 0\Phi\Phi\Phi y \cdot \Phi$. By Theorem 2 we know that there must be a state in the GMW outcome with $z = 0$. This contradicts the assumption that z remains 1, and proves that network N cannot exist.

It is an immediate consequence of Theorem 5 that the behaviors of many common components like JOIN, TOGGLE and latch cannot be implemented delay-insensitively in the input/output mode [7, 9]. Moreover, Theorem 5 can be generalized as follows. A behavior is *simple deterministic* [9] if it is a fundamental-mode behavior, it is deterministic, every transition from an unstable state goes directly to a stable state, and at most one input or output variable changes in any transition. Such a behavior is *nontrivial* if there exits at least one input state a and at least two stable whole states $a \cdot b \cdot c$ and $a \cdot b' \cdot c'$ such that $c \neq c'$ and one of the states is reachable from the other. Using Theorems 3 and 5, it is possible to prove the following [7]:

Theorem 6. *No nontrivial simple deterministic behavior with a binary input has a delay-insensitive implementation by a gate circuit operated in input/output mode.*

For the arbiter result of the previous section, we needed the assumption that any circuit realizing an arbiter behavior must be started in a stable state. This restriction is not needed [9, 39], and Theorem 4 also holds for the input/output mode.

8. Conclusions

We have shown that there are many ways of defining delay-insensitivity. One of the minimal requirements we can impose as the correctess of a network is that it makes a transition to the correct final state. Even with this simple concern, very few behaviors have delay-insensitive implementations by gate circuits. The proofs of several such results are possible if we use the result linking GMW analysis to ternary simulation.

Acknowledgements

The author wishes to thank David Dill, Jo Ebergen, Mark Josephs, Luciano Lavagno, Jan Tijmen Udding, Tom Verhoeff, Hao Zhang, and an anonymous referee for their comments on this paper.

References

[1] J.H. Anderson, M.G. Gouda, A new explanation of the glitch phenomenon, Acta Informatica 28 (1991) 297–309.

[2] D.L. Black, On the existence of delay-insensitive fair arbiters: trace theory and its limitations, Distributed Comput. 1 (4) (1986) 205–225.

[3] S.D. Brookes, A.W. Roscoe, An improved failures model for communicating sequential processes, Lect. Notes Comput. Sci. 197 (1984) 281–305.
[4] E. Brunvand, R.F. Sproull, Translating concurrent programs into delay-insensitive circuits, Proc. of Int. Conf. on Computer-Aided Design, IEEE Press, 1989, pp. 262–265.
[5] E. Brunvand, Translating concurrent communicating programs into asynchronous circuits, Ph.D. Thesis, Computer Science Department, Carnegie Mellon University, 1991. Also, Tech. Rept. CMU-CS-91-198.
[6] J.A. Brzozowski, J.C. Ebergen, Recent developments in the design of asynchronous circuits, in: J. Csirik, J. Demetrovics, F. Gécseg (Eds.), Proc. of Fundamentals of Computation Theory, Lecture Notes in Computer Science, No. 380, Springer, Berlin, 1989, pp. 78–94.
[7] J.A. Brzozowski, J.C. Ebergen, On the delay-sensitivity of gate networks, IEEE Trans. Comput. 41 (11) (1992) 1349–1360.
[8] J.A. Brzozowski, C-J. Seger, A characterization of ternary simulation of gate networks, IEEE Trans. Comput. C-36 (11) (1987) 1318–1327.
[9] J.A. Brzozowski, C-J. Seger, Asynchronous Circuits, Springer, New York, NY, 1995.
[10] J.A. Brzozowski, M. Yoeli, On a ternary model of gate networks, IEEE Trans. Comput. C-28 (3) (1979) 178–183.
[11] J.A. Brzozowski, H. Zhang, Delay-insensitivity and semi-modularity, to appear in Formal Methods in System Design.
[12] M.E. Bush, M.B. Josephs, Some limitations to speed-independence in asynchronous circuits, Proc. 2nd Inte. Symp. on Advanced Research in Asynchronous Circuits and Systems, March 18–21, 1996, Aizu-Wakamatsu, Japan, IEEE Computer Society Press, Los Alamitos, CA, 1996, pp. 104–111.
[13] T.-A. Chu, Synthesis of self-timed VLSI circuits from graph-theoretic specifications, Ph.D. Thesis, Dept. of Electrical Engineering and Computer Science, Massachusets Institute of Technology, June 1987.
[14] D.L. Dill, Trace theory for automatic hierarchical verification of speed-independent circuits, Ph.D. Thesis, Computer Science Department, Carnegie Mellon University, February 1988. Also, The MIT Press, Cambridge, Massachusets, 1989.
[15] J.C. Ebergen, Translating programs into delay-insensitive circuits, Ph.D. Thesis, Department of Mathematics and Computing Science, Eindhoven University of Technology, Eindhoven, The Netherlands, October 1987. Also, CWI Tract 56, Centre for Mathematics and Computer Science, Amsterdam, The Netherlands, 1989.
[16] J.C. Ebergen, A formal approach to designing delay-insensitive circuits, Distributed Comput. 5 (3) (1991) 107–119.
[17] E.B. Eichelberger, Hazard detection in combinational and sequential switching circuits, IBM J. Res. Develop. 9 (1965) 90–99.
[18] S. Hauck, Asynchronous design methodologies: an overview, Proc. IEEE 83 (1) (1995) 69–93.
[19] He Jifeng, M.B. Josephs, C.A.R. Hoare, A theory of synchrony and asynchrony, in: M. Broy, C.B. Jones (Eds.), Programming Concepts and Methods, North-Holland, Amsterdam, 1990, pp. 459–478.
[20] C.A.R. Hoare, Communicating Sequential Processes, Prentice-Hall, Englewood Cliffs, NJ, 1985.
[21] M.B. Josephs, Receptive process theory, Acta Inform. 29 (1992) 17–31.
[22] M.B. Josephs, J.T. Udding, An algebra for delay-insensitive circuits, in: E.M. Clarke, R.P. Kurshan (Eds.), Computer-Aided Verification, AMS-ACM, providence, RI, 1990, pp. 147–175.
[23] M.B. Josephs, J.T. Udding, An overview of DI-algebra, in: T.N. Mudge, V. Milutinovic, L. Hunter (Eds.), Proc. the 26th Annual Hawaii Int. Conf. on System Sciences, 1993, pp. 329–338.
[24] M. Kishinevsky, A. Kondratyev, A. Taubin, V. Varshavsky, Concurrent Hardware, Wiley, Chichester, England, 1994.
[25] L. Lavagno, A. Sangiovanni-Vincentelli, Algorithms of Synthesis and Testing of Asynchronous Circuits, Kluwer Academic Publishers, Boston, 1993.
[26] P.G. Lucassen, A denotational model and composition theorems for a calculus of delay-insensitive specifications, Ph.D. Thesis, Rijksuniversiteit Groningen, Groningen, The Netherlands, May 1994.
[27] A.J. Martin, A delay-insensitive fair arbiter, Technical Report 5193:TR:85, Computer Science Department, California Institute of Technology, Pasadena, CA, 1985.
[28] A.J. Martin, The Limitations to delay-insensitivity in asynchronous circuits, in: W.J. Dally (Ed.), Proc. 6th MIT Conf. on Advanced Research in VLSI, MIT Press, Cambridge, MA, 1990.
[29] C. Mead, L. Conway, Introduction to VLSI Systems, Addison-Wesley, Reading, MA, 1980.
[30] R.E. Miller, Switching Theory, Volume 2: Sequential Circuits and Machines, Wiley, New York, 1965.
[31] R. Milner, Communication and Concurrency, Prentice-Hall, Englewood Cliffs NJ, 1989.

[32] C.E. Molnar, T.P. Fang, F.U. Rosenberger, Synthesis of delay-insensitive modules, in: H. Fuchs (Ed.), Proc. 1985 Chapel Hill Conf. on VLSI, Computer Science Press, Rockville, Maryland, 1985, pp. 67–86.
[33] E.F. Moore, Gedanken experiments on sequential machines, in: C.E. Shannon, J. McCarthy (Eds.), Automata Studies, Annals of Mathematics Study 34, Princeton University Press, Princeton NJ, 1956, pp. 129–153.
[34] D.E. Muller, W.C. Bartky, A theory of asynchronous circuits, Proc. of an Int. Symp. on the Theory of Switching, Annals of Computing Laboratory of Harvard University, vol. 29, 1959, pp. 204–243.
[35] H. Schols, A formalisation of the foam rubber wrapper principle, Master's Thesis, Department of Mathematics and Computing Science, Eindhoven University of Technology, Eindhoven, The Netherlands, February 1985.
[36] H. Schols, Delay-insensitive communication, Ph.D. Thesis, Department of Mathematics and Computing Science, Eindhoven University of Technology, Eindhoven, The Netherlands, February 1985.
[37] C-J.H. Seger, Models and algorithms for race analysis in asynchronous circuits, Ph.D. Thesis, Department of Computer Science, University of Waterloo, Waterloo, Ontario, Canada, May 1988, pp. 186
[38] C-J.H. Seger, On the existence of speed-independent circuits, Theoret. Comput. Sci. 86 (2) (1991) 343–364.
[39] C-J. Seger, J.A. Brzozowski, Generalized ternary simulation of sequential circuits, Theoret. Inform. Appl. 28 (3/4) (1994) 159–186.
[40] N. Shintel, M. Yoeli, Synthesis of modular networks from Petri-net specifications, Technical Report #743, Computer Science Department, Technion, Haifa, Israel, June 1992.
[41] J.T. Udding, Classification and composition of delay-insensitive circuits, Ph.D. Thesis, Department of Mathematics and Computing Science, Eindhoven University of Technology, Eindhoven, The Netherlands, September 1984.
[42] J.T. Udding, On the non-existence of delay-insensitive fair arbiters, Technical Memorandum 306, Computer Systems Laboratory, Washington University, St. Louis, MO, July 1885.
[43] J.T. Udding, A formal model for defining and classifying delay-insensitive circuits and systems, Distributed Comput. 1 (4) (1986) 197–204.
[44] S.H. Unger, Hazards and delays in asynchronous sequential switching circuits, IRE Trans. Circuit Theory CT-6 (1959) 12–25.
[45] S.H. Unger, Asynchronous Sequential Switching Circuits, Wiley-Interscience, New York, 1969.
[46] K. van Berkel, Handshake Circuits, Cambridge University Press, Cambridge, England, 1993.
[47] J.L.A. van de Snepscheut, Trace theory and VLSI design, Ph.D. Thesis, Department of Computing Science, Eindhoven University of Technology, Eindhoven, The Netherlands, May 1983. Also, Lecture Notes in Computer Science, vol. 200, Springer, Berlin, 1985.
[48] T. Verhoeff, Characterizations of delay-insensitive communication protocols, Computing Science Notes No. 89/06, Department of Mathematics and Computing Science, Eindhoven University of Technology, May 1989.
[49] T. Verhoeff, A theory of delay-insensitive systems, Ph.D. Thesis, Department of Mathematics and Computing Science, Eindhoven University of Technology, Eindhoven, The Netherlands, May 1994.
[50] H. Zhang, Delay-insensitive networks, MMath Thesis, Department of Computer Science, University of Waterloo, Waterloo, ON, Canada, June 1997.