1

Topics in Asynchronous Circuit Theory*

Janusz Brzozowski

School of Computer Science, University of Waterloo,
Waterloo, ON, Canada N2L 3G1
brzozo@uwaterloo.ca http://maveric.uwaterloo.ca

Summary. This is an introduction to the theory of asynchronous circuits — a survey of some old and new results in this area. The following topics are included: benefits of asynchronous design, gate circuits, Boolean analysis methods, the theory of hazards, ternary and other multi-valued simulations, behaviors of asynchronous circuits, delay-insensitivity, and asynchronous modules used in modern design.

1.1 Motivation

This paper is a survey of some aspects of asynchronous circuit theory. No background in digital circuits is assumed. The mathematics used includes sets, relations, partial orders, Boolean algebra, and basic graph theory; other concepts are introduced as needed. Only a general knowledge of theoretical computer science is required, in particular, automata and formal languages.

Most computers have a fixed-frequency signal, called a *clock,* that controls their operations; such circuits are called *synchronous*. Because of this central control, the design of synchronous circuits is relatively easy. Circuits operating without a clock are *asynchronous*. Here, the components communicate with each other using some sort of "hand shaking" protocol. This makes the design of asynchronous circuits more challenging.

Several reasons for studying asynchronous circuits are given below; for a more complete discussion of these reasons, as well as an informal general introduction to computing without a clock, we refer the reader to a recent article by Sutherland and Ebergen [41].

- **Higher speed**
  In a synchronous circuit, the clock frequency is determined by the slowest components. In an asynchronous one, each component works at its own pace; this may lead to a higher average speed, especially in applications in which slow actions are rare.

* This research was supported by the Natural Sciences and Engineering Research Council of Canada under Grant No. OGP0000871.

- **Lower energy consumption**
  Up to 30% of the total energy may be used by the clock and its distribution system, and energy is wasted if nothing useful is being done. In asynchronous circuits, idle components consume negligible energy — an important feature in battery-operated systems.
- **Less radio interference**
  The fixed clock frequency in a synchronous circuit results in strong radio signals at that frequency and its harmonics; this interferes with cellular phones, aircraft navigation systems, *etc.* In asynchronous circuits, the radiated energy tends to be distributed over the radio spectrum.
- **Asynchronous interfaces**
  Computers with different clock frequencies are often linked in networks. Communication between such computers is necessarily asynchronous.
- **Better handling of metastability**
  An element that arbitrates between two requests goes to one state, if the first request is granted, and to another state, if the second request wins. If the two requests arrive simultaneously, the arbiter enters an intermediate *metastable* state of unstable equilibrium, and can remain there for an unbounded amount of time [14]. In a synchronous circuit, a long-lasting metastable state is likely to cause an error. An asynchronous circuit waits until the metastable state is resolved, thus avoiding errors.
- **Scaling**
  As integrated-circuit technology improves, components become smaller. However, delays in long wires may not scale down proportionally [25]. Under such circumstances, a synchronous design may no longer be correct, while an asynchronous circuit *can* tolerate such changes in relative delays.
- **Separation of Concerns**
  In asynchronous designs, it is possible to separate physical correctness concerns (*e.g.*, restrictions on delay sizes) in the design of basic elements from the mathematical concerns about the correctness of the behavior of networks of basic elements. This cannot be done in synchronous designs.

Another reason for studying asynchronous circuits is their generality. Synchronous circuits can be viewed as asynchronous circuits satisfying certain timing restrictions. Finally, asynchronous circuits present challenging theoretical problems. Building blocks more complex than gates must be used, *e.g.*, *rendezvous elements,* which detect when two actions have both been completed, and *arbiters,* which decide which of two competing requests should be served. Efficient methods are needed for detecting *hazards,* unwanted pulses that can cause errors. The presence of many concurrent actions leads to an exponential number of possible event sequences; this "state-space explosion" must be overcome. Methods are needed for the design of *delay-insensitive* networks whose behavior should be correct even in the presence of arbitrary component and wire delays.

1.2 Gates

Logic circuits have been analyzed and designed using Boolean algebra since Shannon's pioneering paper in 1938 [38]. A *Boolean algebra* [10] is a structure $\mathbf{B} = (\mathcal{A}, +, *, ^-, 0, 1)$, where $\mathcal{A}$ is a set, $+$ and $*$ are binary operations on $\mathcal{A}$, $^-$ is a unary operation on $\mathcal{A}$, $0, 1 \in \mathcal{A}$ are distinct, and the laws of Table 1.1 hold. The laws are listed in *dual* pairs: to find the dual of a law interchange $+$ and $*$, and 0 and 1. Multiplication has precedence over addition.

Table 1.1. Laws of Boolean algebra

B1	$a + a = a$	B1′	$a * a = a$
B2	$a + b = b + a$	B2′	$a * b = b * a$
B3	$a + (b + c) = (a + b) + c$	B3′	$a * (b * c) = (a * b) * c$
B4	$a + (a * b) = a$	B4′	$a * (a + b) = a$
B5	$a + 0 = a$	B5′	$a * 1 = a$
B6	$a + 1 = 1$	B6′	$a * 0 = 0$
B7	$a + \overline{a} = 1$	B7′	$a * \overline{a} = 0$
B8	$\overline{(\overline{a})} = a$		
B9	$a + (b * c) = (a + b) * (a + c)$	B9′	$a * (b + c) = (a * b) + (a * c)$
B10	$\overline{(a + b)} = \overline{a} * \overline{b}$	B10′	$\overline{(a * b)} = \overline{a} + \overline{b}$

The smallest Boolean algebra is $\mathbf{B}_0 = (\{0, 1\}, +, *, ^-, 0, 1)$, where $+$, $*$, and $^-$ are the OR, AND, and NOT operations of Table 1.2.

Table 1.2. The operations $+$, $*$, and $^-$ in $\mathbf{B}_0$

a_1	a_2	$a_1 + a_2$
0	0	0
0	1	1
1	0	1
1	1	1

a_1	a_2	$a_1 * a_2$
0	0	0
0	1	0
1	0	0
1	1	1

a	$\overline{a}$
0	1
1	0

A Boolean function $f(x_1, \ldots, x_n)$ is a mapping $f : \{0, 1\}^n \to \{0, 1\}$, for $n \geq 0$. Given $f : \{0, 1\}^n \to \{0, 1\}$ and $g : \{0, 1\}^n \to \{0, 1\}$, define, for each $a = (a_1, \ldots, a_n) \in \{0, 1\}^n$, $(f + g)(a) = f(a) + g(a)$, $(f * g)(a) = f(a) * g(a)$, and $\overline{f}(a) = \overline{f(a)}$, where $+$, $*$, and $^-$ on the right are the operations of $\mathbf{B}_0$.

Let $\mathbf{B}_n$ be the set of all 2^{2^n} Boolean functions of n variables. Then $(\mathbf{B}_n, +, *, ^-, \mathbf{0}, \mathbf{1})$, is a Boolean algebra where the operations $+$, $*$, and $^-$ are the operations on Boolean functions defined above, and $\mathbf{0}$ and $\mathbf{1}$ are the functions that are identically 0 and 1, respectively.

Let $0, 1, x_1, \ldots, x_n$ be distinct symbols. A *Boolean expression* over $x_1, \ldots, x_n$ is defined inductively:

1. 0, 1, $x_1, \ldots, x_n$ are Boolean expressions.
2. If E and F are Boolean expressions, then so are $(E + F)$, $(E * F)$, and $\overline{E}$.
3. Any Boolean expression is obtained by a finite number of applications of Rules 1 and 2.

Boolean expressions are used to denote Boolean functions. Expressions 0, 1, and x_i denote $\mathbf{0}$, $\mathbf{1}$, and the function that is identically equal to x_i, respectively. The meaning of other expressions is obtained by structural induction.

Signals in digital circuits take two values, "low voltage" and "high voltage," represented by 0 and 1, respectively. A *gate* is a circuit component performing a Boolean function. It has one or more *inputs*, one *output*, and a *direction* from the inputs to the output. Symbols of some commonly used gates are shown in Fig. 1.1. In the first row are: a *buffer* or *delay* with the identity function, $y = x$; the AND gate, with multiplication in $\mathbf{B}_0$, $y = x_1 * x_2$; the OR gate, with addition in $\mathbf{B}_0$, $y = x_1 + x_2$; the XOR gate with the exclusive-or function $y = x_1 \not\equiv x_2$, which has value 1 if the inputs differ, and 0 if they agree. The gates in the second row perform the complementary functions: the *inverter* or NOT gate; the NAND gate; the NOR gate; and the equivalence gate, EQUIV.

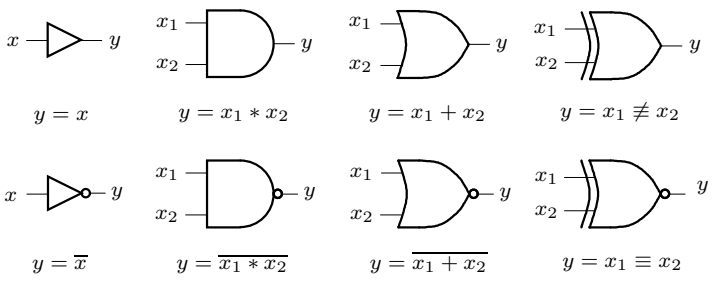

Fig. 1.1. Gate symbols

For some applications, the Boolean-function model of a gate is sufficient. To study general circuit behaviors, however, it is necessary to take gate delays into account. Thus, if the gate inputs change to produce a new output value, the output does not change at the same time as the inputs, but only after some delay. Therefore, a more accurate model of a gate consists of an ideal gate performing a Boolean function followed by a delay.

Figure 1.2(a) shows our conceptual view of the new model of an inverter. Signal Y is a fictitious signal that would be produced by an inverter with zero delay, the rectangle represents a delay, and y is the inverter output. Figure 1.2(b) shows some inverter waveforms, assuming a fixed delay. In our binary model, signals change directly from 0 to 1 and from 1 to 0, since we cannot represent intermediate values. The first waveform shows the input x as a function of time. The ideal inverter with output Y inverts this waveform with zero delay. If the delay were ideal, its output would have the shape $y'(t)$.

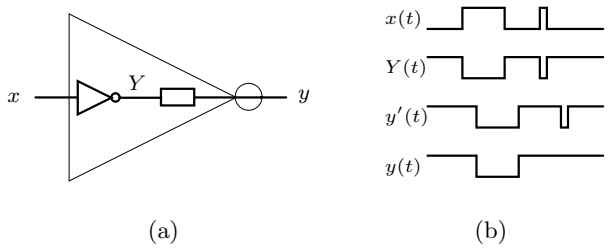

Fig. 1.2. Inverter: (a) delay model, (b) waveforms

However, if an input pulse is shorter than the delay, the pulse does not appear at the output, as in $y(t)$. In other words, the delays in our model are *inertial*. Also, we assume that the delay can change with time in an arbitrary fashion. This is a pessimistic model, but it has been used often, and it is one of the simplest. For a detailed discussion of delay models see [10].

Unless stated otherwise, we assume that each gate symbol represents a Boolean function followed by a delay. Thus, the buffer gate (with the identity function) is equivalent to a delay; hence we use a triangle to represent a delay.

1.3 Gate Circuits

A gate circuit consists of *external input terminals* (or simply *inputs*), gates, *external output terminals* (or simply *outputs*), *forks*, and *wires*. The inputs, gates, and outputs are labeled by variables $x_1, \ldots, x_m$, $y_1, \ldots, y_n$ and $z_1, \ldots, z_p$, respectively. Let $\mathcal{X} = \{x_1, \ldots, x_m\}$, $\mathcal{Y} = \{y_1, \ldots, y_n\}$, and $\mathcal{Z} = \{z_1, \ldots, z_p\}$. Forks are junctions with one input and two or more outputs. The environment supplies binary values to the input terminals. An external input terminal, a gate output and a fork output must be connected by a wire to a gate input, a fork input, or an external output terminal, and *vice versa*, except that we do not permit an input x to be connected to an output z, because x and z would be disjoint from the rest of the circuit. Note that a wire connects two points only; multiple connections are done through forks.

Figure 1.3(a) shows a circuit called the NOR *latch.* Forks are shown by small black dots (circled in this figure). Input x_1 is connected to an input of NOR gate y_1 by wire w_1, *etc.*

In analyzing a circuit, it is usually not necessary to assume that *all* components have delays. Several models with different delay assumptions have been used [10]. We use the *gate-delay* model. In this model, in the latch of Fig. 1.3(a), gate variables y_1 and y_2 are assumed to have nonzero delays, and wires have zero delays. The choice of delays determines the *(internal) state variables* of the circuit. In our example, we have decided that the state of the two gates suffices to give us a proper description of the behavior of the latch.

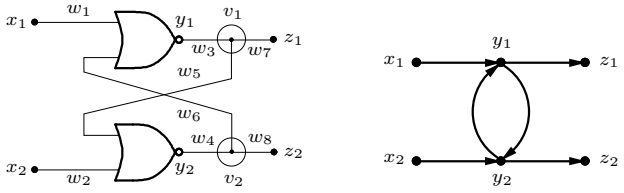

Fig. 1.3. The NOR latch: (a) circuit, (b) network

Having selected gates as state variables, we model the circuit by a *network graph,* which is a directed graph [1] (digraph) (V, E), where $V = \mathcal{X} \cup \mathcal{Y} \cup \mathcal{Z}$ is the set of vertices, and $E \subseteq V \times V$ is the set of directed edges defined as follows. There is an edge (v, v') if vertex v is connected by a wire, or by several wires via forks, to vertex v' in the circuit, and vertex v' depends functionally on vertex v. The network graph of the latch is shown in Fig. 1.3(b).

Suppose there are incoming edges from external inputs $x_{j_1}, \ldots, x_{j_{m_i}}$ and gates $y_{k_1}, \ldots, y_{k_{n_i}}$ to a gate y_i, and the Boolean function of the gate is

$$f_i : \{0,1\}^{m_i+n_i} \to \{0,1\}. \tag{1.1}$$

The *excitation function* (or *excitation*) of gate y_i is

$$Y_i = f_i(x_{j_1}, \ldots, x_{j_{m_i}}, y_{k_1}, \ldots, y_{k_{n_i}}). \tag{1.2}$$

If the incoming edge of output variable z_i comes from vertex v, then the *output equation* is $z_i = v$.

In the example of the latch, we have the excitations and output equations:

$$Y_1 = \overline{x_1 + y_2}, \quad Y_2 = \overline{x_2 + y_1}; \quad z_1 = y_1, \quad z_2 = y_2.$$

A *network* $N = (x, y, z, Y, E_z)$ of a circuit consists of an input m-tuple $x = (x_1, \ldots, x_m)$, a state n-tuple $y = (y_1, \ldots, y_n)$, an output p-tuple $z = (z_1, \ldots, z_p)$, an n-tuple $Y = (Y_1, \ldots, Y_n)$ of excitation functions, and a p-tuple $E_z = (z_1 = y_{h_1}, \ldots, z_p = y_{h_p})$ of output equations. In examples, we use circuit diagrams, and leave the construction of the networks to the reader.

A network is *feedback-free* if its network graph is acyclic; an example is given in Fig. 1.4. In a feedback-free network, one can define the concept of *level*. External input vertices are of level 0. A gate vertex is of level 1, if its inputs are of level 0. Inductively, a gate is of level k, if its inputs come from vertices of level $\leq k-1$ and at least one of them comes from a vertex of level $k-1$. An output $z_i = y_j$ has the level of y_j.

In the circuit of Fig. 1.4, y_1 and y_2 are of level 1, y_3 is of level 2, y_4 is of level 3, and y_5 and z_1 are of level 4. The excitations and output equations are:

$$Y_1 = \overline{x_1}, \quad Y_2 = x_1 * x_2, \quad Y_3 = y_1 * x_3, \quad Y_4 = y_2 + y_3, \quad Y_5 = x_1 * y_4; \quad z_1 = y_5.$$

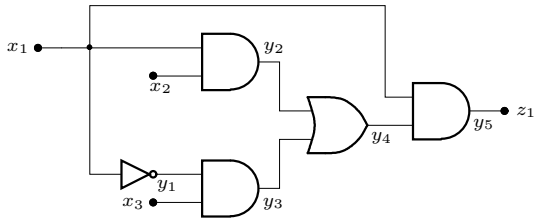

Fig. 1.4. A feedback-free circuit

1.4 Binary Analysis

Our analysis of asynchronous circuits by Boolean methods follows the early work of Muller [28, 31, 32], but we use the terminology of [7, 10, 12, 19].

A *total state* $c = a \cdot b$ of a network $N = (x, y, z, Y, E_z)$ is an $(m+n)$-tuple of values from $\{0,1\}$, the first m values (m-tuple $a = (a_1, \ldots, a_m)$) being the inputs, and the remaining n (n-tuple $b = (b_1, \ldots, b_n)$), the state variables $y_1, \ldots, y_n$. The $\cdot$ is used for readability. The excitation function f_i of (1.1) is extended to a function $f_i' : \{0,1\}^{m+n} \to \{0,1\}$ as follows:

$$f_i'(a \cdot b) = f_i(a_{j_1}, \ldots, a_{j_{m_i}}, b_{k_1}, \ldots, b_{k_{n_i}}). \tag{1.3}$$

Usually we write f_i' simply as f_i; the meaning is clear from the context. A state variable y_i is *stable* in state $a \cdot b$ if $b_i = f_i(a \cdot b)$; otherwise it is *unstable*. The set of unstable state variables is $U(a \cdot b)$. A state $a \cdot b$ is *stable* if $U(a \cdot b) = \emptyset$.

We are interested in studying the behavior of a network started in a given state $a \cdot b$ with the input kept constant at $a \in \{0,1\}^m$. A state in which two or more variables are unstable is said to contain a *race*. We use the "general multiple-winner" (GMW) model, where any number of variables can "win" the race by changing simultaneously. With this in mind, we define a binary relation R_a on the set $\{0,1\}^n$ of states of N: For any $b \in \{0,1\}^n$,

bR_ab, if $U(a \cdot b) = \emptyset$, *i.e.*, if total state $a \cdot b$ is stable, and

bR_ab^K, if $U(a \cdot b) \neq \emptyset$, and K is any nonempty subset of $U(a \cdot b)$,

where b^K is b with all the variables in K complemented. Nothing else is in R_a. As usual, we associate a digraph with the relation R_a, and denote it G_a. The set of all states reachable from b in relation R_a is

$$\mathit{reach}(R_a(b)) = \{c \mid bR_a^*c\}, \tag{1.4}$$

where R_a^* is the reflexive-and-transitive closure of R_a. We denote by $G_a(b)$ the subgraph of G_a corresponding to $\mathit{reach}(R_a(b))$.

In examples, we write binary tuples as binary words, for simplicity. The circuit of Fig. 1.5 has binary input x, and state $y = (y_1, y_2, y_3)$. The excitations are $Y_1 = x + y_1$, $Y_2 = \overline{y_1}$, and $Y_3 = \overline{x * y_2 * y_3}$. Graph $G_1(011)$ for this circuit is shown in Fig. 1.6(a), where unstable entries are underlined. There are three

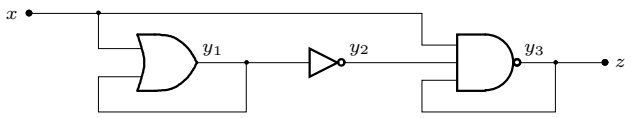

Fig. 1.5. A circuit with a transient oscillation

cycles in $G_1(011)$: $(011, 010)$, $(111, 110)$, and (101). In general, a cycle of length 1 is a stable state, while a cycle of length > 1 is an *oscillation*. A cycle in which a variable y_i has the same value in all of the states of the cycle, and is unstable in the entire cycle is called *transient*. Under the assumption that the delay of each gate is finite, a circuit cannot remain indefinitely in a transient cycle. In the example above, both cycles $(011, 010)$ and $(111, 110)$ are transient.

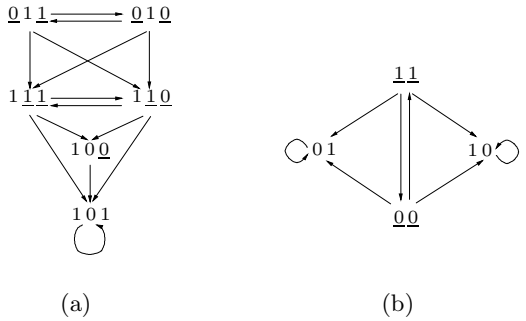

Fig. 1.6. Graphs: (a) $G_1(011)$ for Fig. 1.5, (b) $G_{00}(11)$ for Fig. 1.3

One of the important questions in the analysis of asynchronous circuits is: What happens after the input has been held constant for a long time? (This concept will be made precise.) In the example of Fig. 1.6(a), we conclude that the circuit can only be in stable state 101 after a long time, because unstable gates must eventually change, since they have finite delays.

A different kind of cycle is present in graph $G_{00}(11)$ of Fig. 1.6(b) for the NOR latch. In state 11 there is a two-way race. If y_1 wins the race, the state becomes 01, and the instability that was present in y_2 in state 11 has been removed by the change in y_1. This shows that our mathematical model captures the inertial nature of the delays: the instability in y_2 did not last long enough, and was ignored. Similarly, if y_2 wins the race, stable state 10 is reached. Here the race is *critical,* since its outcome depends on the relative sizes of the delays. A third possibility exists, namely, the cycle $(11, 00)$. The circuit can remain in this cycle only if both delays are perfectly matched; any imbalance will cause the circuit to leave the cycle and enter one of the two stable states. For this reason, such cycles are called *match-dependent*. In a

crude way they represent the phenomenon of metastability [10, 14]. Let the set of *cyclic states* reachable from b in the graph of R_a be

$$cycl(R_a(b)) = \{s \in \{0,1\}^n \mid bR_a^* s \text{ and } sR_a^+ s\}, \tag{1.5}$$

R_a^+ being the transitive closure of R_a. Let the set of *nontransient cyclic states* be $cycl\_nontrans(R_a(b))$. The *outcome* of state b under input a is the set of all the states reachable from some state that appears in a nontransient cycle of $G_a(b)$. Mathematically, we have

$$out(R_a(b)) = \{s \mid bR_a^* c \text{ and } cR_a^* s, \text{ where } c \in cycl\_nontrans(R_a(b))\}. \tag{1.6}$$

This definition captures the intuitive notion that a circuit can be in a state d after a sufficiently long time only if d is in the outcome. For a more detailed discussion of these issues, see [10], where the following result is proved:

Theorem 1. *Let N be a network with n state variables with delay sizes bounded by D. If N is in state b at time 0 and the input is held constant in the interval 0 to t, $t \geq (2^n - 2)D$, then the state at time t is in $out(R_a(b))$.*

In the examples of Fig. 1.6 (a) and (b), the outcomes are $\{101\}$ and $\{00, 01, 10, 11\}$ respectively. We leave it to the reader to verify that the outcome of graph $G_{111}(10110)$ of the circuit of Fig. 1.4 consists of the stable state 01011.

The size of graph $G_a(b)$ is exponential in the number of state variables, and this makes the computation of outcome inefficient. Better methods for determining properties of the outcome will be given later.

1.5 Hazards

In calculating the outcome, we find the final states in which the network may end up after an input change; this is a "steady-state" property. We now turn our attention to "transient properties", *i.e.*, the sequences of values that variables go through before reaching their final values in the outcome.

Hazards are unwanted pulses occurring in digital circuits because of the presence of "stray" delays. It is important to detect and eliminate them, because they can result in errors. Early work on hazards was done by Huffman [24], McCluskey [26], and Unger [45], among others. For more details and references, see [6], where the use of multi-valued algebras for hazard detection is discussed. Recent results of Brzozowski and Esik [5] unify the theory of hazard algebras. We begin by a brief outline of this theory, which has its roots in Eichelberger's work on ternary simulation [10, 18]; see also [7].

Consider the feedback-free circuit of Fig. 1.4 redrawn in Fig. 1.7. Suppose the input $x = (x_1, x_2, x_3)$ has the constant value 011; one verifies that the state $y = (y_1, \ldots, y_5)$ becomes 10110, as shown by the first value of each

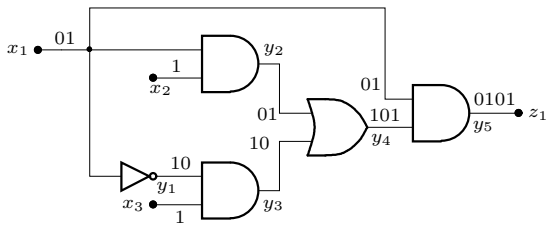

Fig. 1.7. Illustrating hazards

variable in the figure. If the input becomes 111, the following changes take place: $y_1 : 1 \to 0$, $y_2 : 0 \to 1$, and $y_3 : 1 \to 0$, as shown by the second values. Two possibilities exist for y_4. If y_2 changes first, or at the same time as y_3, then y_4 remains 1. If y_3 changes first, however, there is a period during which both y_2 and y_3 are 0, and y_4 may have a 0-pulse. Such a pulse is called a *static hazard*—static, because y_4 has the same value before and after the input change, and hence is not supposed to change. For y_5, if the change in x_1 is followed by a double change in y_4 as shown, then the output has a *dynamic hazard:* y_5 should change only once from 0 to 1, but changes three times. The 1-pulse can lead to an error. In summary, hazards are unexpected signal changes: In the case of a static hazard, there is a nonzero even number of changes where there should be none; in case of a dynamic hazard, there is a nonzero even number of changes in addition to the one expected change.

Hazards can be detected using binary analysis by examining all paths in graph $G_a(b)$. For example, one verifies that the following path exists in graph $G_{111}(10110)$ of the circuit in Fig. 1.7: $\pi = \underline{1}\,0\,1\,1\,\underline{0} \to 0\,\underline{0}\,1\,1\,1 \to 0\,\underline{0}\,0\,\underline{1}\,1 \to 0\,\underline{0}\,0\,0\,\underline{1} \to 0\,\underline{0}\,0\,0\,0 \to 0\,1\,0\,\underline{0}\,0 \to 0\,1\,0\,1\,\underline{0} \to 0\,1\,0\,1\,1$. Along this path, we have the changes $y_4 : 1 \to 0 \to 1$, and $y_5 : 0 \to 1 \to 0 \to 1$.

In general, graph $G_a(b)$ may have as many as 2^n nodes, and the binary method is not practical for hazard detection; hence alternate methods have been sought [6]. We describe one such method, which uses "transients."

A *transient* is a nonempty binary word in which no two consecutive letters are the same. The set of all transients is $\mathbf{T} = \{0, 1, 01, 10, 010, 101, \ldots\}$. Transients represent signal waveforms in the obvious way. For example, waveform $x(t)$ in Fig. 1.2(b) corresponds to the transient 01010.

By *contraction* of a binary word s, we mean the operation of removing all letters, say a, repeated directly after the first occurrence of a, thus obtaining a word $\hat{s}$ of alternating 0s and 1s. For example, the contraction of the sequence 01110001 of values of y_5 along path π above is 0101. For variable y_i, such a contraction is the *history* of that variable along π and is denoted $\boldsymbol{\sigma}_i^{\pi}$.

We use boldface symbols to denote transients. If $\mathbf{t}$ is a transient, $\alpha(\mathbf{t})$ and $\omega(\mathbf{t})$ are its first and last letters, respectively. Also, $z(\mathbf{t})$ and $u(\mathbf{t})$ denote the number of 0s and 1s in $\mathbf{t}$, respectively. We write $\mathbf{t} \leq \mathbf{t}'$ if $\mathbf{t}$ is a prefix of $\mathbf{t}'$; $\leq$ is a partial order on $\mathbf{T}$. Let $[n]$ denote the set $\{1, \ldots, n\}$. We extend the partial

order $\leq$ to n-tuples of transients: for $\mathbf{t} = (\mathbf{t}_1, \ldots, \mathbf{t}_n)$ and $\mathbf{t}' = (\mathbf{t}'_1, \ldots, \mathbf{t}'_n)$, $\mathbf{t} \leq \mathbf{t}'$, if $\mathbf{t}_i \leq \mathbf{t}'_i$, for all $i \in [n]$. We denote by $\mathbf{t} \circ \mathbf{t}'$ concatenation followed by contraction [7], *i.e.*, $\mathbf{t} \circ \mathbf{t}' = \widehat{\mathbf{t}\mathbf{t}'}$. The suffix relation $\preceq$ is also a partial order.

In the algebra we are about to define, gates process transients instead of 0s and 1s. Thus it is necessary to extend Boolean functions to the domain of transients. Let $f : \{0,1\}^n \to \{0,1\}$ be any Boolean function. Its *extension* to $\mathbf{T}$ is the function $\mathbf{f} : \mathbf{T}^n \to \mathbf{T}$ defined as follows: For any tuple $(\mathbf{t}_1, \ldots, \mathbf{t}_n)$ of transients, $\mathbf{f}(\mathbf{t}_1, \ldots, \mathbf{t}_n)$ is the longest transient produced when $\mathbf{t}_1, \ldots, \mathbf{t}_n$ are applied to the inputs of a gate performing function f. A method for computing the extension of any function is given in [5], as well as formulas for the extensions of some common functions. Here we give only the extensions of complementation, and the 2-input AND and OR functions. If $\mathbf{t} = a_1 \ldots a_n$ is any transient, where the a_i are in $\{0,1\}$, and f is complementation, then $\mathbf{f}(\mathbf{t}) = \overline{\mathbf{t}} = \overline{a_1} \ldots \overline{a_n}$. If f is the 2-input OR function, then $\mathbf{s} = \mathbf{f}(\mathbf{t}_1, \mathbf{t}_2)$ is the word in $\mathbf{T}$ determined by the conditions[2]

$$\begin{aligned}
\alpha(\mathbf{s}) &= \alpha(\mathbf{t}_1) + \alpha(\mathbf{t}_2) \\
\omega(\mathbf{s}) &= \omega(\mathbf{t}_1) + \omega(\mathbf{t}_2) \\
z(\mathbf{s}) &= \begin{cases} 0 & \text{if } \mathbf{t}_1 = 1 \text{ or } \mathbf{t}_2 = 1 \\ z(\mathbf{t}_1) + z(\mathbf{t}_2) - 1 & \text{otherwise.} \end{cases}
\end{aligned}$$

Dually, if f is the 2-input AND function, then $\mathbf{s} = \mathbf{f}(\mathbf{t}_1, \mathbf{t}_2)$ is given by

$$\begin{aligned}
\alpha(\mathbf{s}) &= \alpha(\mathbf{t}_1) * \alpha(\mathbf{t}_2) \\
\omega(\mathbf{s}) &= \omega(\mathbf{t}_1) * \omega(\mathbf{t}_2) \\
u(\mathbf{s}) &= \begin{cases} 0 & \text{if } \mathbf{t}_1 = 0 \text{ or } \mathbf{t}_2 = 0 \\ u(\mathbf{t}_1) + u(\mathbf{t}_2) - 1 & \text{otherwise.} \end{cases}
\end{aligned}$$

We denote by $\otimes$ and $\oplus$ the extensions of the 2-argument Boolean AND ($*$) and OR ($+$) functions, respectively. Algebra $C = (\mathbf{T}, \oplus, \otimes, ^-, 0, 1)$, is called the *change-counting algebra*, and is a commutative de Morgan bisemigroup [5], *i.e.*, it satisfies laws B2, B3, B5, B6, B8, B10 of Boolean algebra from Table 1.1 and their duals.

We refer to C as the *algebra of transients*, and use it to define efficient simulation algorithms for detecting hazards. There are two algorithms, A and B, and two versions, A and $\tilde{\text{A}}$, of the first algorithm. Algorithm A is general, whereas $\tilde{\text{A}}$ applies only if the initial state is stable. It is proved in [7] that A and $\tilde{\text{A}}$ produce the same result if the initial state is stable, under the condition that the network contains *input delays*, *i.e.*, delays in the wires leaving the input terminals; these delays can be viewed as gates performing identity functions.

For brevity we describe only Algorithm $\tilde{\text{A}}$. In this section, we ignore the output terminals and consider only inputs and state variables.

Let $N = (x, y, z, Y, E_z)$ be a network. The *extension* of N to the domain $\mathbf{T}$ of transients is $\mathbf{N} = (\mathbf{x}, \mathbf{y}, \mathbf{z}, \mathbf{Y}, \mathbf{E_z})$, where $\mathbf{x}$, $\mathbf{y}$, and $\mathbf{z}$, are variables taking

[2] In $\alpha(\mathbf{s})$ and $\omega(\mathbf{s})$, $+$ represents Boolean OR, whereas in $z(\mathbf{s})$ and $u(\mathbf{s})$ it is addition.

values from $\mathbf{T}$, and the Boolean excitation functions are replaced by their extensions to $\mathbf{T}$.

Let $a \cdot b$ be a (binary) total state of $\mathbf{N}$. Algorithm $\tilde{\mathrm{A}}$ is defined as follows:

Algorithm $\tilde{\mathbf{A}}$
$\mathbf{a} = \tilde{a} \circ a$;
$\mathbf{s}^0 := b$;
$h := 1$;
$\mathbf{s}^h := \mathbf{S}(\mathbf{a} \cdot \mathbf{s}^0)$;
while $(\mathbf{s}^h <> \mathbf{s}^{h-1})$ **do**
    $h := h + 1$;
    $\mathbf{s}^h := \mathbf{S}(\mathbf{a} \cdot \mathbf{s}^{h-1})$;

We illustrate the algorithm in Table 1.3 using the circuit of Fig. 1.7. The initial state $011 \cdot 10110$ is stable. We change the input to $(1, 1, 1)$ and concatenate $(0, 1, 1)$ and $(1, 1, 1)$ component-wise with contraction, obtaining $(01, 1, 1)$ as the new input. In the first row with this input, y_1, y_2, and y_5 are unstable in the algebra of transients; they are all changed *simultaneously* to get the second row. Now y_3 is unstable, and changes to 10 in the third row. In the third row, y_4 is unstable; according to the definition of the extended OR, the output becomes 101. In the fifth row, y_5 becomes 0101. At this point, the state $(10, 01, 10, 101, 0101)$ is stable, and the algorithm stops. Note that the last row shows that y_1, y_2, and y_3 have single, hazard-free changes, that y_4 has the static hazard 101, and y_5, the dynamic hazard 0101. The final state, 01011, is shown by the last bits of the transients.

Table 1.3. Simulation in Algorithm $\tilde{\mathrm{A}}$

	$\mathbf{x}_1$	$\mathbf{x}_2$	$\mathbf{x}_3$	$\mathbf{y}_1$	$\mathbf{y}_2$	$\mathbf{y}_3$	$\mathbf{y}_4$	$\mathbf{y}_5$
initial state	0	1	1	1	0	1	1	0
	01	1	1	1	0	1	1	0
	01	1	1	10	01	1	1	01
	01	1	1	10	01	10	1	01
	01	1	1	10	01	10	101	01
result $\tilde{\mathrm{A}}$	01	1	1	10	01	10	101	0101

Algorithm $\tilde{\mathrm{A}}$ results in a state sequence that is nondecreasing with respect to the prefix order [5]:

$$\mathbf{s}^0 \leq \mathbf{s}^1 \leq \ldots \leq \mathbf{s}^h \leq \ldots$$

Algorithm $\tilde{\mathrm{A}}$ may not terminate, for example, for the NOR latch with initial state $11 \cdot 00$ and inputs changing to 00. If Algorithm $\tilde{\mathrm{A}}$ does terminate, let its result be $\mathbf{s}^{\tilde{A}}$. Note that $\mathbf{s}^{\tilde{A}} = \mathbf{S}(\mathbf{a}, \mathbf{s}^{\tilde{A}})$, *i.e.*, the last state is stable.

The next theorem shows that it is possible to reduce the set of state variables and still obtain the same result using simulation with the remaining

variables. A set $\mathcal{F}$ of vertices of a digraph G is a *feedback-vertex set* if every cycle in G contains at least one vertex from $\mathcal{F}$.

To illustrate variable removal, consider the NOR latch of Fig.1.3. If we choose y_1 as the only feedback variable, we eliminate y_2 by substituting $\overline{y_1 + x_2}$ for it, obtaining $Y_1 = \overline{x_1 + y_2} = \overline{x_1 + \overline{y_1 + x_2}} = \overline{x_1} * (y_1 + x_2)$. In this way we obtain a *reduced* network[3] $\dot{N}$ of N. We perform similar reductions in the extended network $\mathbf{N}$ to obtain $\dot{\mathbf{N}}$. The proof of the following claim is similar to an analogous theorem proved in [7] for Algorithm A:

Theorem 2. *Let $\mathcal{F}$ be a feedback-vertex set of a network $\mathbf{N}$, and let $\dot{\mathbf{N}}$ be the reduced version of $\mathbf{N}$ with vertex set $\mathcal{X} \cup \mathcal{F}$. If Algorithm $\tilde{A}$ terminates on $\mathbf{N}$, the final state of $\mathbf{N}$ agrees with that of $\dot{\mathbf{N}}$ for the state variables in $\mathcal{F}$.*

We now compare the simulation algorithm to binary analysis. The next theorem [7] shows that the result of simulation "covers" the result of binary analysis in the following sense. Suppose π is a path of length h starting at b in $G_a(b)$, and the $n-$tuple of histories of the state variables along π is $\boldsymbol{\sigma}^\pi$. Then $\boldsymbol{\sigma}^\pi$ is a prefix of the simulation result after h steps. More formally, we have

Theorem 3. *For all paths $\pi = s^0, \ldots, s^h$ in $G_a(b)$, with $s^0 = b$, we have $\boldsymbol{\sigma}^\pi \leq \mathbf{s}^h$, where $\mathbf{s}^h$ is the $(h+1)$st state in the sequence resulting from Algorithm $\tilde{A}$.*

Corollary 1. *If Algorithm $\tilde{A}$ terminates with state $\mathbf{s}^H$, then for any path π in $G_a(b)$, $\boldsymbol{\sigma}^\pi \leq \mathbf{s}^H$.*

Only partial results are known about the converse of Corollary 1. It has been shown by Gheorghiu [19, 20] that, for feedback-free circuits of 1- and 2-input gates, if appropriate wire delays are taken into account, there exists a path π in $G_a(b)$ such that $\boldsymbol{\sigma}^\pi = \mathbf{s}^H$.

Alternatives to simulation with an infinite algebra are described next.

1.6 Ternary Simulation

Algebra C permits us to count an arbitrary number of signal changes; an alternative is to count precisely only up to some threshold $k-1$, $k > 1$, and consider all transients of length $\geq k$ as equivalent [5]. For $k > 1$, define relation $\sim_k$ as follows; For $\mathbf{s}, \mathbf{t} \in \mathbf{T}$, $\mathbf{s} \sim_k \mathbf{t}$ if either $\mathbf{s} = \mathbf{t}$ or $\mathbf{s}$ and $\mathbf{t}$ are both of length $\geq k$. Thus the equivalence class $[\mathbf{t}]$ of $\mathbf{t}$ is the singleton $\{\mathbf{t}\}$ if the length of $\mathbf{t}$ is $< k$; all the remaining elements of $\mathbf{T}$ are in one class that we denote Φ_k, or simply Φ, if k is understood.

Relation $\sim_k$ is a congruence relation on C, meaning that for all $\mathbf{s}, \mathbf{t}, \mathbf{w} \in \mathbf{T}$, $\mathbf{s} \sim_k \mathbf{t}$ implies $(\mathbf{w} \oplus \mathbf{s}) \sim_k (\mathbf{w} \oplus \mathbf{t})$, and $\overline{\mathbf{s}} \sim_k \overline{\mathbf{t}}$. Thus, there is a unique algebra $C_k = C/\sim_k = (\mathbf{T}_k, \oplus, \otimes, ^-, 0, 1)$, where $\mathbf{T}_k = \mathbf{T}/\sim_k$ is the quotient set of

[3] After the substitution we must compute the outputs differently. We still have $z_1 = y_1$, but $z_2 = y_2 = \overline{x_2 + y_1}$ here.

equivalence classes of $\mathbf{T}$ with respect to $\sim_k$, such that the function from $\mathbf{T}$ to $\mathbf{T}_k$ taking a transient to its equivalence class is a homomorphism, *i.e.*, preserves the operations and constants. The operations in $\mathbf{T}_k$ are as follows: $\overline{[\mathbf{t}]} = [\,\overline{\mathbf{t}}\,]$, $[\mathbf{t}] \oplus [\mathbf{t}'] = [\mathbf{t} \oplus \mathbf{t}']$, and $[\mathbf{t}] \otimes [\mathbf{t}'] = [\mathbf{t} \otimes \mathbf{t}']$. The quotient algebra C_k is a commutative de Morgan bisemigroup with $2k-1$ elements [5].

For Algebras $\mathbf{T}_k$, Algorithm Ã always terminates; we denote its result by $\mathbf{s}^{\tilde{A}}$. Following Eichelberger [10, 18], we define a second simulation algorithm, Algorithm B:

Algorithm B
$\mathbf{t}^0 := \mathbf{s}^{\tilde{A}}$;
$h := 1$;
$\mathbf{t}^h := \mathbf{S}(a, \mathbf{t}^{h-1})$;
while $\mathbf{t}^h <> \mathbf{t}^{h-1}$ **do**
    $h := h + 1$;
    $\mathbf{t}^h := \mathbf{S}(a, \mathbf{t}^{h-1})$;

Now the network is started in the state which results from Algorithm Ã. The input is set to its final binary value a, and Algebra $\mathbf{T}_k$ is used for the computation of the next state. It is easy to verify that Algorithm B results in a sequence of states that is nonincreasing in the suffix order $\succeq$:

$$\mathbf{s}^{\tilde{A}} = \mathbf{t}^0 \succeq \mathbf{t}^1 \succeq \ldots \succeq \mathbf{t}^B.$$

The network reaches a stable state, *i.e.*, $\mathbf{t}^B = \mathbf{S}(a, \mathbf{t}^B)$.

The smallest quotient algebra $\mathbf{T}_2$ is isomorphic to the well-known ternary algebra [10]. Its operations are shown in Table 1.4, and satisfy all the laws of Boolean algebra from Table 1.1, except B7 and its dual, and also the laws of Table 1.5. The third element, Φ_2 (denoted simply as Φ) can be interpreted as the *uncertain* value, whereas 0 and 1 are *certain* values. The following partial order, the *uncertainty partial order* [10, 30], reflects this interpretation: $\mathbf{t} \sqsubseteq \mathbf{t}$ for all $\mathbf{t} \in \{0, \Phi, 1)$, $0 \sqsubseteq \Phi$, and $1 \sqsubseteq \Phi$.

Table 1.4. The operations $\oplus$, $\otimes$, and $^-$ in $\mathbf{T}_2$

$\mathbf{t}_1 \oplus \mathbf{t}_2$	$\mathbf{t}_2$ 0	Φ	1
0	0	Φ	1
$\mathbf{t}_1$ Φ	Φ	Φ	1
1	1	1	1

$\mathbf{t}_1 \otimes \mathbf{t}_2$	$\mathbf{t}_2$ 0	Φ	1
0	0	0	0
$\mathbf{t}_1$ Φ	0	Φ	Φ
1	0	Φ	1

$\mathbf{t}$	$\overline{\mathbf{t}}$
0	1
Φ	Φ
1	0

Algorithm Ã in $\mathbf{T}_2$ for the circuit of Fig. 1.7 is shown in the first part of Table 1.6. Here, the inputs that change become uncertain, and this uncertainty "spreads" to some of the gates. Then, in Algorithm B, the inputs are set to their final values, and some (or all) of the uncertainty is removed.

Table 1.5. Ternary laws

T1 $\overline{\Phi} = \Phi$
T2 $(\mathbf{t} \oplus \overline{\mathbf{t}}) \oplus \Phi = \mathbf{t} \oplus \overline{\mathbf{t}}$ T2′ $(\mathbf{t} \otimes \overline{\mathbf{t}}) \otimes \Phi = \mathbf{t} \otimes \overline{\mathbf{t}}$

Table 1.6. Ternary simulation

	$\mathbf{x}_1$	$\mathbf{x}_2$	$\mathbf{x}_3$	$\mathbf{y}_1$	$\mathbf{y}_2$	$\mathbf{y}_3$	$\mathbf{y}_4$	$\mathbf{y}_5$
initial state	0	1	1	1	0	1	1	0
	Φ	1	1	1	0	1	1	0
	Φ	1	1	Φ	Φ	1	1	Φ
	Φ	1	1	Φ	Φ	Φ	1	Φ
result $\tilde{\text{A}}$	Φ	1	1	Φ	Φ	Φ	Φ	Φ
	1	1	1	Φ	Φ	Φ	Φ	Φ
	1	1	1	0	1	Φ	Φ	Φ
	1	1	1	0	1	0	1	Φ
result B	1	1	1	0	1	0	1	1

Eichelberger [18] did not formally relate the result of ternary simulation to binary analysis. Partial results in this direction were obtained by Brzozowski and Yoeli [12], where it was shown that the result of Algorithm B "covers" the outcome of binary analysis, and it was conjectured that the converse also holds if wire delays are taken into account. The conjecture was proved by Brzozowski and Seger [9]. We briefly outline these results.

By a *complete* network we mean one in which each wire has a delay. The following characterizes the result of Algorithm $\tilde{\text{A}}$, where *lub* denotes the least upper bound with respect to the uncertainty partial order $\sqsubseteq$:

Theorem 4. *Let $N = (x, y, z, Y, E_z)$ be a complete binary network, and let $\mathbf{N} = (\mathbf{x}, \mathbf{y}, \mathbf{z}, \mathbf{Y}, \mathbf{E_z})$ be its ternary counterpart. If N is started in total state $\tilde{a} \cdot b$ and the input changes to a, then the result $\mathbf{s}^{\tilde{A}}$ of Algorithm $\tilde{A}$ for $\mathbf{N}$ is equal to the least upper bound of the set of states reachable from the initial state in the GMW analysis of N, i.e.,*

$$\mathbf{s}^{\tilde{A}} = lub\, reach(R_a(b)).$$

The characterization of the result of Algorithm B is given by

Theorem 5. *Let $N = (x, y, z, Y, E_z)$ be a complete binary network, and let $\mathbf{N} = (\mathbf{x}, \mathbf{y}, \mathbf{z}, \mathbf{Y}, \mathbf{E_z})$ be its ternary counterpart. If N is started in total state $\tilde{a} \cdot b$ and the input changes to a, then the result $\mathbf{t}^B$ of Algorithm B is equal to the least upper bound of the outcome of the GMW analysis, i.e.,*

$$\mathbf{t}^B = lub\, out(R_a(b)).$$

Ternary simulation can also detect static hazards. For a detailed discussion of these issues see [10].

Theorem 6. *A complete network N started in total state $\tilde{a} \cdot b$ with the input changing to a has a static hazard on variable s_i if and only if its ternary extension $\mathbf{N}$ has the following property: The result $\mathbf{s}_i^{\tilde{\mathrm{A}}}$ of Algorithm $\tilde{A}$ is Φ, while the result $\mathbf{t}_i^B$ of Algorithm B is equal to the initial value b_i.*

1.7 Simulation in Other Finite Algebras

Ternary simulation detects static hazards, but does not explicitly identify them. As k is increased, simulation in Algebra $\mathbf{T}_k$ provides more and more information. Tables 1.7 and 1.8 show simulations of the circuit of Fig. 1.7 in $\mathbf{T}_3$ and $\mathbf{T}_4$, which have five and seven elements, respectively. Quinary simulation (in $\mathbf{T}_3$) reveals in Algorithm $\tilde{\mathrm{A}}$ the signals which have no hazards, and detects both the static and dynamic hazards by the presence of a Φ; whereas septenary simulation (in $\mathbf{T}_4$) explicitly identifies the static hazard in y_4. For $k = 5$, the nonary Algorithm $\tilde{\mathrm{A}}$ (in $\mathbf{T}_5$) is identical in this case to simulation in Algebra C in Table 1.3, where the dynamic hazard is explicitly identified as 0101; consequently, Algorithm B is no longer needed.

Table 1.7. Quinary simulation

	$\mathbf{x}_1$	$\mathbf{x}_2$	$\mathbf{x}_3$	$\mathbf{y}_1$	$\mathbf{y}_2$	$\mathbf{y}_3$	$\mathbf{y}_4$	$\mathbf{y}_5$
initial state	0	1	1	1	0	1	1	0
	01	1	1	1	0	1	1	0
	01	1	1	10	01	1	1	01
	01	1	1	10	01	10	1	01
	01	1	1	10	01	10	Φ	01
result $\tilde{\mathrm{A}}$	01	1	1	10	01	10	Φ	Φ
	1	1	1	10	01	10	Φ	Φ
	1	1	1	0	1	10	Φ	Φ
	1	1	1	0	1	0	1	Φ
result B	1	1	1	0	1	0	1	1

Several other multi-values algebras have been proposed for hazard detection over the years [6]. It was shown in [5] that all the successful algebras can be derived from Algebra C.

1.8 Fundamental-Mode Behaviors

Until now, we have considered only what happens in a network started in a particular total state: we have computed the states of the outcome and checked for the presence of hazards. In most applications, a network started in a stable total state should move to another stable total state when its

Table 1.8. Septenary simulation

	$\mathbf{x}_1$	$\mathbf{x}_2$	$\mathbf{x}_3$	$\mathbf{y}_1$	$\mathbf{y}_2$	$\mathbf{y}_3$	$\mathbf{y}_4$	$\mathbf{y}_5$
initial state	0	1	1	1	0	1	1	0
	01	1	1	1	0	1	1	0
	01	1	1	10	01	1	1	01
	01	1	1	10	01	10	1	01
	01	1	1	10	01	10	101	01
result $\tilde{\text{A}}$	01	1	1	10	01	10	101	Φ
	1	1	1	10	01	10	101	Φ
	1	1	1	0	1	10	101	101
	1	1	1	0	1	0	1	101
result B	1	1	1	0	1	0	1	1

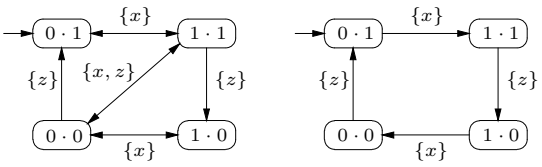

Fig. 1.8. Inverter behavior: (a) unrestricted, (b) fundamental-mode

inputs change, and its outputs should have no hazards. We now examine the response of a network to a *sequence* of input changes. We use three examples, an inverter, a latch and a flip-flop, to illustrate how sequential behaviors are modeled by automata, and how the operation of a circuit can be restricted to avoid unreliable responses. The flip-flop is also an example of a synchronous circuit analyzed by asynchronous methods to show the details of its operation.

Since we are modeling physical processes, we assume that only a finite number of input changes can take place in any finite time interval. If we do not put any other restrictions on the environment, the behaviors that result can be unreliable, and therefore not useful, as is shown below.

We represent network behaviors by digraphs with states as vertices. A directed edge from one vertex to another is a *transition*, and the set of input and output variables changing in each transition is indicated as a label.

Consider an inverter operating in an unrestricted environment. Suppose it has input x, internal state y, output $z = y$, and starts in stable state $x \cdot y = x \cdot z = 0 \cdot 1$. If the input changes to 1, the inverter changes to state $1 \cdot 1$, which is unstable; see Fig. 1.8(a). From state $1 \cdot 1$, the inverter can move to state $1 \cdot 0$. However, in state $1 \cdot 1$, the environment may change x back to 0 before the inverter has a chance to respond; then state $0 \cdot 1$ can again be reached. In state $1 \cdot 1$, it is also possible that y changes to 0 as x is changing,

resulting in a transition to $0 \cdot 0$. Thus, if the input is withdrawn, we do not know which state will result. The rest of the behavior is similar.

It is clear that some restrictions must be imposed on the environment to avoid unreliable behavior. A restriction that has been used for many years is operation in *fundamental mode* [10]. In this mode, a circuit starts in a stable state, and an input can only be changed if the circuit is stable. For each input change from a stable total state we use binary analysis to find the outcome. It is tacitly assumed that the circuit always reaches a unique stable state, and that the input is held constant until this happens. The fundamental-mode behavior of the inverter is shown in Fig. 1.8(b).

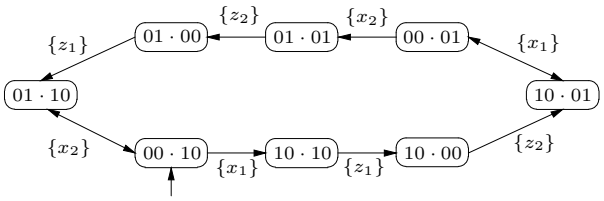

Fig. 1.9. Part of fundamental-mode behavior of latch

As a second example, consider the latch of Fig. 1.3 started in total state $00 \cdot 10$; see Fig. 1.9. Note that $z_1 = y_1$ and $z_2 = y_2$. State $00 \cdot 10$ is stable. If the input changes to 01, the latch remains stable. When the input is 10, the excitations are $Y_1 = 0$, $Y_2 = \overline{y_1}$. If the input changes to 10 from state $00 \cdot 10$, the new state $10 \cdot 10$ is unstable, and $y_1 y_2$ changes as follows: $\underline{1}\,0 \rightarrow 0\,\underline{0} \rightarrow 0\,1$. Thus the outcome is a unique stable state; there are no races, oscillations or hazards. The rest of the behavior is symmetric.

The latch behavior above is not only fundamental-mode, but is also restricted by the condition that both inputs should never be 1 at the same time. The latch is used as a memory device, storing a value, traditionally called Q, in variable $y_1 = z_1$, and its complement $\overline{Q}$ in variable $y_2 = z_2$. Input x_1 then corresponds to the *reset* input R, and x_2 is the *set* input S. When $RS = 00$, the latch has two stable states $Q\overline{Q} = 01$ and $Q\overline{Q} = 10$; it *remembers* the previously stored value. If $RS = 10$, the latch is *reset* to, or stays in, state $Q\overline{Q} = 01$, and, if $RS = 01$, the latch is *set* to, or stays in, state $Q\overline{Q} = 10$. If the input 11 is applied and held, both Q and $\overline{Q}$ become 0, violating the condition that $\overline{Q}$ should be the complement of Q. Furthermore, if both inputs then become 0 at the same time, the latch enters a match-dependent oscillation, as shown in Fig. 1.6(b). Since the latch's behavior is unpredictable after it leaves the oscillation, the input combination 11 is not used in practice.

The circuit of Fig. 1.10 is a *master/slave toggle flip-flop*. It has two inputs: the *toggle* input t, and the *clock* input c. It is normally considered a synchronous circuit, but we analyze it as an asynchronous circuit operating in fundamental mode and under the restriction that t should not change

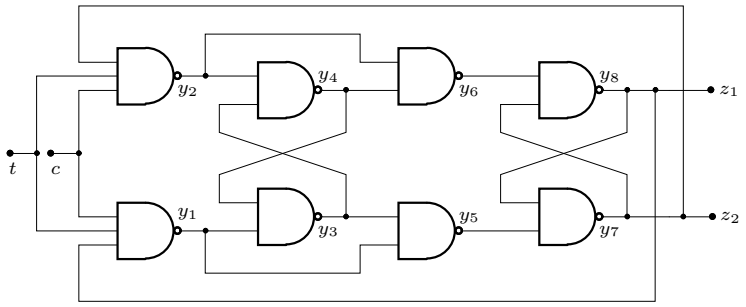

Fig. 1.10. Toggle flip-flop

when $c = 1$. Variables y_3, y_4 form the *master* latch, and y_7, y_8 are the *slave* latch; these variable values will be shown in boldface. Suppose $ct = 00$, and $y = y_1 \dots y_8 = 1\,1\,\mathbf{0}\,\mathbf{1}\,1\,0\,\mathbf{0}\,\mathbf{1}$; this state is stable, both latches hold the values 01, and the *slave follows the master*. If t becomes 1, there is no change. If c then becomes 1, and the input is constant at $ct = 11$, the excitations are: $Y_1 = \overline{y_8}$, $Y_2 = \overline{y_7}$, $Y_3 = \overline{y_1 * y_4}$, $Y_4 = \overline{y_2 * y_3}$, $Y_5 = \overline{y_1 * y_3}$, $Y_6 = \overline{y_2 * y_4}$, $Y_7 = \overline{y_5 * y_8}$, $Y_8 = \overline{y_6 * y_7}$. The internal state changes as follows:

$$\underline{1}\,1\,\mathbf{0}\,\mathbf{1}\,1\,0\,\mathbf{0}\,\mathbf{1} \to 0\,1\,\underline{\mathbf{0}}\,\mathbf{1}\,1\,0\,\mathbf{0}\,\mathbf{1} \to 0\,1\,\mathbf{1}\,\underline{\mathbf{1}}\,1\,0\,\mathbf{0}\,\mathbf{1} \to 0\,1\,\mathbf{1}\,\mathbf{0}\,1\,\underline{0}\,\mathbf{0}\,\mathbf{1} \to 0\,1\,\mathbf{1}\,\mathbf{0}\,1\,1\,\mathbf{0}\,\mathbf{1}.$$

The new values 10 are now stored in the master, while the slave remembers the old values 01. When the clock again becomes 0, we have

$$\underline{0}\,1\,\mathbf{1}\,\mathbf{0}\,1\,1\,\mathbf{0}\,\mathbf{1} \to 1\,1\,\mathbf{1}\,\mathbf{0}\,\underline{1}\,1\,\mathbf{0}\,\mathbf{1} \to 1\,1\,\mathbf{1}\,\mathbf{0}\,0\,1\,\underline{\mathbf{0}}\,\mathbf{1} \to 1\,1\,\mathbf{1}\,\mathbf{0}\,0\,1\,\mathbf{1}\,\underline{\mathbf{1}} \to 1\,1\,\mathbf{1}\,\mathbf{0}\,0\,1\,\mathbf{1}\,\underline{\mathbf{0}}.$$

The slave again follows the master, and the operation can be repeated when c next becomes 1. We can summarize the flip-flop behavior as follows: If $t = 0$ when the c becomes 1, the outputs of the flip-flop do not change. If $t = 1$ when c becomes 1, the flip-flop *toggles*, *i.e.*, its outputs are complemented. For more examples of this approach to analysis see [11].

In general, a *behavior* of a network $N = (x, y, z, Y, E_z)$ with m inputs $x_1, \dots, x_m$, n state variables $y_1, \dots, y_n$, and p outputs $z_1, \dots, z_p$ is a tuple

$$B = (x, y, z, Q, q_\epsilon, T, E_z),$$

where $Q = \{0,1\}^{m+n}$ is the set of *total states,* $q_\epsilon \in Q$ is the *initial state,* $T \subseteq Q \times Q - \{(q,q) \mid q \in Q\}$ is the set of *transitions,* to be defined.

Several types of behaviors have been studied [10]; here we consider only certain special ones. The *unrestricted transition relation* is defined as follows. In any total state $a \cdot b$, $a \in \{0,1\}^m$, $b \in \{0,1\}^n$, if bR_ab', $b \neq b'$ or $a \neq a'$, we have $(a \cdot b, a' \cdot b') \in T$. The behavior of the inverter in Fig. 1.8(a) is an example of unrestricted behavior.

The *fundamental-mode transition relation* is defined as follows: $(a \cdot b, a' \cdot b') \in T$ if and only if bR_ab' and either $b = b'$ or $a = a'$. Thus, an input can

change only if $a \cdot b$ is stable; otherwise, the state follows the R_a relation. The latch behavior in Fig. 1.9 is a part of its fundamental-mode behavior.

A behavior is *direct* if it is fundamental-mode and every transition from an unstable state leads directly to a stable state. An example of a direct behavior is shown in Fig. 1.11 for the latch. Here the details of the two 2-step transitions of Fig. 1.9 are suppressed and only the final outcome is shown.

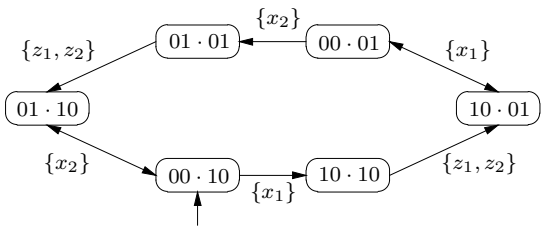

Fig. 1.11. Part of direct behavior of latch

With each behavior $B = (x, y, z, Q, q_\epsilon, T, E_z)$, we associate a finite nondeterministic *behavior automaton* $A = (\Sigma, Q, q_\epsilon, F, T)$, where $\Sigma = 2^{\mathcal{X} \cup \mathcal{Z}} \setminus \{\emptyset\}$ is the input alphabet of the automaton consisting of nonempty subsets of the set of input and output variables. Elements of Σ are used as labels on the transitions; the label is the empty word ϵ if there are no input or output changes accompanying a transition. Also, $F = Q$, *i.e.*, every state is final (accepting). The language $L = L(A)$ of all words accepted by A is always *prefix-closed, i.e.*, the prefix of every word in the language is also in the language.

1.9 Delay-Insensitivity

Behavior automata can be used as specifications of behaviors to be realized by networks. Ideally, a network realizing a given behavior should be *delay-insensitive*, in the sense that it should continue to realize the behavior in the presence of arbitrary gate and wire delays. As we shall see, very few behaviors can be realized delay-insensitively. This will be made more precise.

Fundamental-mode operation has the major drawback that the environment needs to know how long to wait for a stable state to be reached, and this is an unrealistic requirement. An alternative is the "input/output mode" of operation [3, 4, 10, 15, 29]. Here the environment is allowed to change an input after a response from the component has been received, or no response is expected. To illustrate this mode of operation, we present the circuit of Fig. 1.12(a) consisting of two delays and an OR gate.

Suppose the circuit is started in total state $0 \cdot 000$, which is stable. In fundamental mode, if the input changes to 1, the internal state changes as follows: $\underline{0}\,0\,0 \rightarrow 1\,\underline{0}\,0 \rightarrow 1\,1\,\underline{0} \rightarrow 1\,1\,1$. If the input then changes to 0 again,

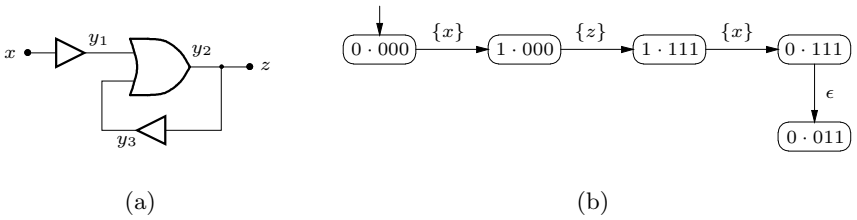

Fig. 1.12. Illustrating operating modes: (a) circuit, (b) direct behavior

we have $\underline{1}\,1\,1 \rightarrow 0\,1\,1$. The direct behavior corresponding to these two input changes is shown in Fig. 1.12(b).

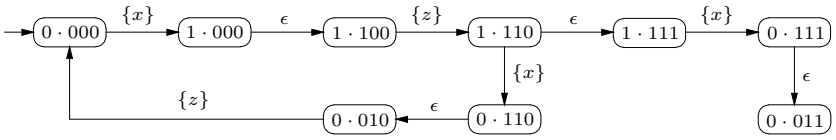

Fig. 1.13. Part of input/output-mode behavior

On the other hand, suppose the network operates in *input/output mode.* Then the environment is permitted to change the input after the output has changed, without waiting for the entire state to stabilize. A part of the input/output-mode behavior is shown in Fig. 1.13. While the correct input/output sequence xzx is still present, it is also possible for the circuit to have the sequence $xzxz$, which is *not* permitted in fundamental-mode.

Even in fundamental mode, very few behaviors are realizable by delay-insensitive networks. Suppose a behavior B has transitions $a{\cdot}b \rightarrow a'{\cdot}b \rightarrow a'{\cdot}b'$, $a' \cdot b' \rightarrow a \cdot b' \rightarrow a \cdot b''$, and $a \cdot b'' \rightarrow a' \cdot b'' \rightarrow a' \cdot b'''$, where $a \cdot b$, $a' \cdot b'$, $a \cdot b''$, and $a' \cdot b'''$ are stable. A result of Unger's [44, 45] states that, if $b' \neq b'''$, any network realizing B has an *essential hazard*, meaning that, if wire delays are taken into account, the network state reached from $a{\cdot}b$ when the input changes to a' may be either b' or b''', implying that the network is not delay-insensitive. Unger's result has been rephrased by Seger [10, 36, 37]:

Theorem 7. *Let N be any complete network, let $a{\cdot}b$ be a stable state of N, and assume that $out(R_{a'}(b)) = \{b'\}$, $out(R_a(b')) = \{b''\}$, and $out(R_{a'}(b'')) = \{b'''\}$. Then $b''' = b'$.*

This theorem shows that behavior B either has no delay-insensitive realization (b' and b''' can be confused by any network purporting to realize it), or is very restricted (cannot have $b''' \neq b'$). Seger's proof uses the results of Section 1.6 about the equivalence of ternary simulation and binary analysis.

Behaviors realizable by delay-insensitive networks operating in the input/output mode are even more restricted, as was shown in the example of Fig. 1.12(a). It has been shown by Brzozowski and Ebergen [4, 10] that a simple behavior similar to that of Fig. 1.12(b), where a first input change produces an output change, but the second input change produces no response, cannot be realized by any gate circuit operating in input/output mode. Consequently, most components used in asynchronous circuit design cannot be delay-insensitive. It is, therefore, necessary to design such components using some timing restrictions. However, it *is* possible to design *networks* of such components to operate independently of the component and wire delays.

1.10 Delay-Insensitivity in Networks

We introduce a very general definition of an asynchronous module, which includes not only gates but also more complex components used in asynchronous design. We suppress the details of the implementation of the module, and use only an abstract model [13].

Definition 1. *A* module *is a nondeterministic sequential machine* $M = (\mathcal{S}, \mathcal{X}, y, \mathcal{Z}, \delta, \lambda)$ *of the Moore type, where*

- $\mathcal{S}$ *is a finite set of* internal states*;*
- $\mathcal{X} = \{x_1, \ldots, x_m\}$ *is the set of* binary input variables*, where* $m \geq 0$*;*
- y *is the* internal state variable *taking values from* $\mathcal{S}$*;*
- $\mathcal{Z} = \{z_1, \ldots, z_p\}$ *is the set of* binary output variables*, where* $p \geq 0$*;*
- δ *is the* excitation function*,* $\delta : \{0,1\}^m \times \mathcal{S} \to 2^{\mathcal{S}} \setminus \{\emptyset\}$*, satisfying the restriction that for any* $a \in \{0,1\}^m$ *and* $b \in \mathcal{S}$*, either* $\delta(a,b) = \{b\}$ *(in which case (a,b) is stable) or* $b \notin \delta(a,b)$ *((a,b) is unstable);*
- $\lambda = (\lambda_1, \ldots, \lambda_p)$ *is the* output function*,* $\lambda : \mathcal{S} \to \{0,1\}^p$*.*

If $m = 0$, the module is a *source*, and if $p = 0$, then it is a *sink*. If the cardinality of $\delta(a,b)$ is 1 for all a, b, then the module is *deterministic*. For deterministic modules we write $\delta(a,b) = c$, instead of $\delta(a,b) = \{c\}$. If the module is deterministic, δ is independent of $\mathcal{S}$, $\mathcal{S} = \{0,1\}$, $\mathcal{Z} = \{z\}$, and $z = y$, then the module is equivalent to a gate.

Figure 1.14 shows six deterministic 2-state modules. In each of them, the state set is $\{0,1\}$, but we could also use any abstract 2-element set like $\{a,b\}$. Notation $y \cdot z$ means that the state is y and the associated output is z. The excitations and outputs are: (a) $\delta = x$, $z = y$, (b) $\delta = \overline{x}$, $z = y$, (c) $\delta = x$, $z_1 = z_2 = y$, (d) $\delta = x_1 \not\equiv x_2$, $z = y$, (e) $\delta = x_1 * x_2 + (x_1 + x_2)y$, $z = y$, and (f) $\delta = x_1 + \overline{x_2} * y$, $z_1 = y$, $z_2 = \overline{y}$. Modules (a), (b), and (d) are gates.

The module in Fig. 1.15 is nondeterministic; it is a primitive sort of *arbiter*. An input $x_i = 0$ indicates no request, and $x_i = 1$ is a request, for $i = 1, 2$. If the inputs are 00, there are no requests, the state is 0, and the outputs are 00. If $x = 10$, x_1 is requesting service; the arbiter grants this service by

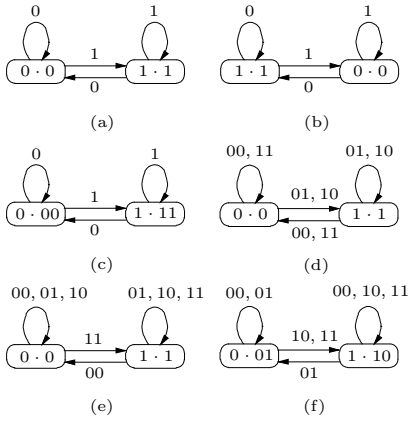

Fig. 1.14. 2-state modules: (a) delay, (b) inverter, (c) (isochronic) fork, (d) XOR, (e) C-element, (f) set-dominant latch

changing state to 1, and setting $z_1 = 1$. The arbiter remains in this state until x_1 becomes 0; the second input is ignored, since the request was granted to x_1. If the input becomes 01 in state 1, the arbiter first returns to state 0, to terminate the service to x_1, and then makes a grant to x_2. The situation is symmetric if $x = 01$ in state 00. If $x = 11$, two transitions possible: the arbiter nondeterministically decides to serve either x_1 or x_2.

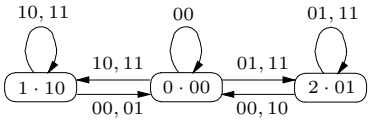

Fig. 1.15. Arbiter

Module outputs cannot change before the internal state changes, since the outputs are uniquely determined by the state. There are no delays in the output wires, since wire delays *are* often negligible, especially if the module takes up a small area of a chip. Also, designers sometimes use the *isochronic fork* assumption, that signals at two ends of a fork arrive at the same time, and we can model this. If needed, delay modules can be added to the wires.

We consider only *closed* or *autonomous* networks of modules; such networks are without external inputs or outputs. An open network can be made closed by adding a module representing the environment. A *network N of modules* consists of a set $\{M^1, \ldots, M^n\}$ of modules and a set of wires, each wire connecting an output z_k^h of module M^h to an input x_j^i of module M^i.

Each module output must be connected to a module input, and *vice versa*. As in the case of gates, a wire can connect only two points.

The set of network state variables is $\mathcal{Y} = \{y^1, \ldots, y^n\}$, and the excitation of module i is δ_i. A state of N is an n-tuple in $\mathcal{S} = \mathcal{S}^1 \times \ldots \times \mathcal{S}^n$. The excitation of each module is a function of $y^1, \ldots, y^n$.

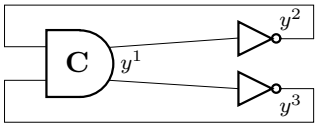

Fig. 1.16. Network

A network consisting of a 2-output C-element and two inverters is shown in Fig.1.16. Here the C-element has excitation $\delta_1 = y^2 * y^3 + (y^2 + \underline{y^3}) * y^1$, and two outputs, $z_1^1 = z_2^1 = y^1$. The inverter excitations are $\delta_2 = \delta_3 = \overline{y^1}$. Suppose the state of the network is 011. Since each inverter is in state 1, its output is 1, and the inputs of the C-element are both 1. The C-element is therefore unstable. At the same time, the outputs of the C-element are both 0, and both inverters are stable. This analysis leads to the behavior of Fig. 1.17.

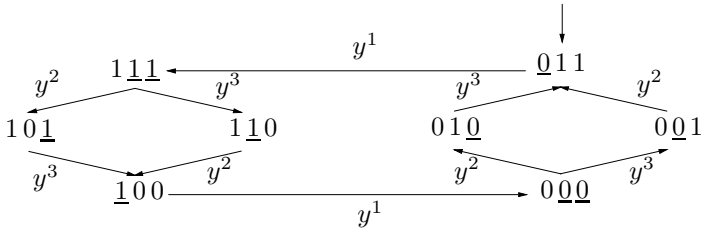

Fig. 1.17. Network behavior

In the analysis of networks of modules we use the *general single-winner* (GSW) model. It turns out that this simplification is justifiable, as we shall see later. Let $\mathcal{R}'$ be the GSW relation; then states s and t are related by $\mathcal{R}'$ if they differ in exactly one component i (that is $s_i \neq t_i$), y^i is unstable in state s, and $t_i \in \delta_i(s)$. In general, the GSW behavior of a network N is an initialized directed graph $B = (q, Q, \mathcal{R})$, where $q \in \mathcal{S}$ is the initial state, Q is the set of states reachable from q by the GSW relation, and $\mathcal{R}$ is the GSW relation $\mathcal{R}'$ restricted to Q. Moreover, we label each edge of this graph by the variable that changes during the corresponding transition.

A behavior $B = (q, Q, \mathcal{R})$ can be viewed as a nondeterministic finite automaton $\mathcal{B} = (\mathcal{Y}, Q, q, \mathcal{R}, F)$, where $\mathcal{Y}$ is the input alphabet, Q is the set of

states, q is the initial state, $\mathcal{R}$ is the labeled transition relation, and $F = Q$ is the set of final states. The language $L(\mathcal{B})$ is the set of all possible sequences of module variable changes.

Let N be a network started in state q. A *delay extension* of N is a network $\hat{N}$ started in $\hat{q}$ and obtained from N by adding any number of delays in the wires; here $\hat{q}$ is an extension of q obtained by adding values for the new delays in such a way that they are initially stable. Behavior $\hat{B} = (\hat{q}, \hat{Q}, \hat{\mathcal{R}})$ of $\hat{N}$ is *livelock-free* if every sequence of changes of only the inserted delays is finite. Every delay-extension is livelock-free [13].

A network is *strongly delay-insensitive* with respect to a state q, if, for any delay extension $\hat{N}$, the behavior $\hat{B} = (\hat{q}, \hat{Q}, \hat{\mathcal{R}})$ of $\hat{N}$ is observationally equivalent to the behavior $B = (q, Q, \mathcal{R})$ of N. Roughly speaking, two behaviors are *observationally equivalent*, or *weakly bisimilar*, if they can simulate each other in a step-by-step fashion. Of course, we ignore the added delays in $\hat{N}$, and compare only that part of the state that corresponds to the original modules. Bisimilar states must agree in the observable variables, for any transition from one state there must be a corresponding transition from the other state, and these transitions must lead to bisimilar states.

We say that $\hat{B}$ is *safe* with respect to B, if, whenever $\hat{s} \in \hat{Q}$ and $s \in Q$ are bisimilar, then, any sequence

$$\hat{s} \xrightarrow{*} \hat{u} \xrightarrow{y^i} \hat{v} \xrightarrow{*} \hat{t}$$

from state $\hat{s}$ to $\hat{t}$ (that involves zero or more unobservable changes, followed by a change in y^i, followed by zero or more unobservable changes), implies the existence of a corresponding sequence from s to t, where $\hat{t}$ and t are bisimilar:

$$\hat{s} \xrightarrow{*} \hat{u} \xrightarrow{y^i} \hat{v} \xrightarrow{*} \hat{t} \quad \Rightarrow \quad s \xrightarrow{y^i} t.$$

Behavior $\hat{B}$ is *complete* with respect to B if the converse implication holds. One verifies that $\hat{B}$ is always complete with respect to B [13]. Therefore to verify strong delay-insensitivity, one needs to check only safety. This definition involves an infinite test, because there are infinitely many delay-extensions of a network. However, there is a finite test for the equivalent property of quasi semi-modularity.

Semi-modularity is a concept introduced by Muller [31, 32] for deterministic networks of gates. This notion has been generalized in [13] as follows. *Quasi semi-modularity* of a behavior B requires that, once a state variable y^i becomes unstable, and b is a possible next state for y^i, then b should remain a possible next state until y^i changes. In other words, no change in some other variable y^j can remove b from the excitation of y^i.

To illustrate these notions, we present a part of a network in Fig. 1.18. In the top left, the two delays have outputs 1, and excitations $\{0\}$ and $\{1\}$. The arbiter is in state 0, and has excitation $\{1, 2\}$. After the top delay changes to 0, the excitation of the arbiter becomes $\{2\}$. This is a violation of quasi

semi-modularity, because a change in the delay has caused the arbiter state 1 to be removed from the excitation.

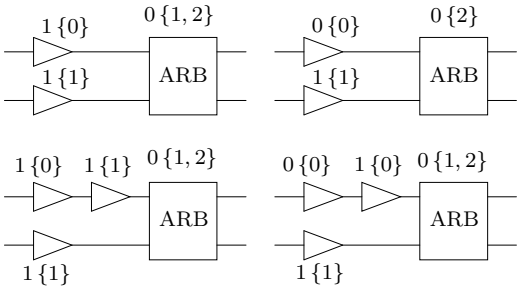

Fig. 1.18. Safety violation

The bottom parts illustrate what happens if an initially stable delay is added. After the left delay is changed, the arbiter retains its original excitation, because of the added delay. Thus, in the extension of the network, the state of the arbiter can become 1, while this is not possible in the original network. This is a violation of safety, and this network is not strongly delay-insensitive.

The following results have been proved by Brzozowski and Zhang [13, 49]:

Theorem 8. *If a network is strongly delay-insensitive, then its behavior is quasi semi-modular.*

The converse result holds under the condition that each wire has at least one delay. A network satisfying this condition is called *delay-dense*.

Theorem 9. *If the behavior of a delay-dense network is quasi semi-modular, then the network is strongly delay-insensitive.*

In our analysis we have used the GSW model, but strong delay-insensitivity and quasi semi-modularity can be generalized to the GMW model. The following results of Silver's [39, 40] justify the use of the GSW model.

Theorem 10. *A network is strongly delay-insensitive in the GSW model if and only if it is strongly delay-insensitive in the GMW model.*

Theorem 11. *The GSW behavior of a network is single-change quasi semi-modular if and only if its GMW behavior is multiple-change quasi semi-modular.*

1.11 Trace Structures

This section is an introduction to formal specifications of asynchronous circuits, and is based on the work of Ebergen [16, 17]. The design of modules,

as we have seen, necessarily involves making some assumptions about relative sizes of delays. Assuming that the modules have been designed, Ebergen describes their behavior in a program notation inspired by Hoare's Communicating Sequential Processes [23]. Using this notation, one can then design networks of modules, and reason about their correctness. These correctness concerns are now separate from the timing concerns.

We begin by defining the behavior of several modules using a notation similar to regular expressions. Our view is now more abstract, and hides the implementation details. Thus we no longer need to know whether a signal is high or low, but interpret a signal change as a *communication* at a circuit terminal. Because this communication can be an input or an output, depending on whether we are talking about a module or its environment, we use the following notation: $a?$ denotes an input event, whereas $a!$ is an output. Note that a is not a variable, but a label of a terminal.

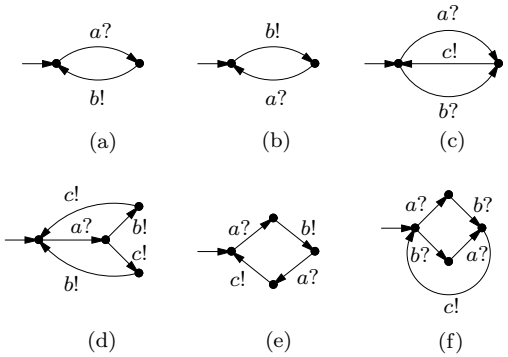

Fig. 1.19. Module state graphs: (a) WIRE, (b) IWIRE, (c) MERGE, (d) FORK, (e) TOGGLE, (f) JOIN

Figure 1.19 shows the state graphs of several modules. The specification of each module is a contract between the module and the environment. The WIRE receives an input $a?$ and responds by producing an output $b!$ Thus, the correct event sequences for the WIRE are: $\epsilon, a?, a?b!, a?b!a?, a?b!a?b!, \ldots$ A convenient way to describe this language is to denote all input/output alternations by the regular expression $(a?b!)^*$ and then close this set under the prefix operation. Thus the language of a WIRE is $\mathbf{pref}((a?b!)^*)$. It is not correct for the environment to send two consecutive inputs to the WIRE without receiving an output after the first signal. It is also incorrect for the WIRE to produce an output without receiving an input, or to produce two consecutive outputs, without an input between them. These rules are similar to the input/output mode discussed earlier.

The *initialized wire* or IWIRE first produces an output, and then acts like a WIRE; its language is $\mathbf{pref}((b!a?)^*)$. A MERGE module has two inputs $a?$ and

$b?$ and one output $c!$ It receives a signal on either one of its input terminals, and then produces an output; its language is $\mathbf{pref}((a? + b?)c!)^*)$. A FORK has one input $a?$ and two outputs $b!$ and $c!$, and its language is $\mathbf{pref}((a?b!c! + a?c!b!)^*)$. A TOGGLE has one input $a?$ and two outputs $b!$ and $c!$, and its language is $\mathbf{pref}((a?b!a?c!)^*)$. The JOIN is a component that detects whether both of its input events $a?$ and $b?$ have happened before responding with an output $c!$, and its language is $\mathbf{pref}(((a?b? + b?a?)c!)^*)$. Thus, this module provides synchronization. A similar specification of a 3-input JOIN is long and clumsy, as the reader can verify. To overcome this, a new operation $\|$, a type of parallel composition called *weave*, is introduced. Using $\|$, we can write $\mathbf{pref}(((a?\|b?)c!)^*)$ instead of $\mathbf{pref}(((a?b? + b?a?)c!)^*)$. In trace theory, parallel events are denoted by interleaving, as they are in the GSW model.

Expressions, like those above, describing module languages are called "commands." Formally, a *command* over an alphabet Σ is defined inductively as follows:

1. ABORT, SKIP, and $a?$ and $a!$, for all $a \in \Sigma$, are commands.
2. If E and F are commands, then so are: (EF), $(E + F)$, E^*, $\mathbf{pref}E$, $E{\downarrow_\Gamma}$, for $\Gamma \subseteq \Sigma$, and $(E\|F)$.
3. Any command is obtained by a finite number of applications of 1 and 2.

The following order of operator precedence is used to simplify notation: star, **pref**, concatenation, addition, and weave at the lowest level.

Next, we define the semantics of commands. A *trace structure* is a triple (I, O, T), where I is the *input alphabet*, O is the *output alphabet*, $I \cap O = \emptyset$, and $T \subseteq (I \cup O)^*$ is the *trace set*, a *trace* being another term for a word over the *alphabet* $\Sigma = I \cup O$ of the trace structure. The *meaning* of a command is a trace structure defined below, where we equate a command with the trace structure that it represents. The notation $t{\downarrow_\Gamma}$ for $\Gamma \subseteq \Sigma$ denotes the projection of the word t to the alphabet Γ obtained from t by replacing all the letters of t that are not in Γ by the empty word.

$$\text{ABORT} = (\emptyset, \emptyset, \emptyset) \tag{1.7}$$
$$\text{SKIP} = (\emptyset, \emptyset, \{\epsilon\}) \tag{1.8}$$
$$a? = (\{a\}, \emptyset, \{a\}) \tag{1.9}$$
$$a! = (\emptyset, \{a\}, \{a\}) \tag{1.10}$$

Next, if $E = (I, O, T)$ and $E' = (I', O', T')$, then

$$(EE') = (I \cup I', O \cup O', TT') \tag{1.11}$$
$$(E + E') = (I \cup I', O \cup O', T \cup T') \tag{1.12}$$
$$E^* = (I, O, T^*) \tag{1.13}$$
$$\mathbf{pref}E = (I, O, \{u \mid uv \in T\}) \tag{1.14}$$
$$E{\downarrow_\Gamma} = (I \cap \Gamma, O \cap \Gamma, \{t{\downarrow_\Gamma} \mid t \in T\}) \tag{1.15}$$
$$(E\|E') = (I \cup I', O \cup O', \{t \in (\Sigma \cup \Sigma')^* \mid t{\downarrow_\Sigma} \in T, t{\downarrow_{\Sigma'}} \in T'\}) \tag{1.16}$$

The operation $E\downarrow_\Gamma$ is that of *hiding* symbols. When an output $a!$ of one module becomes an input $a?$ of another, both these symbols can become internal, and may be removed from the description of the combined system. To illustrate the weave operation, let $E = (\{a\}, \{c\}, \{\epsilon, a, ac\})$ and $E' = (\{b\}, \{c\}, \{\epsilon, b, bc\})$. Then $(E \| E') = (\{a, b\}, \{c\}, \{\epsilon, a, b, ab, ba, abc, bac\})$.[4] As another example, reconsider the JOIN. We can also write its language as $\mathbf{pref}(a?c! \| b?c!)^*$; the concatenation operations enforce that an input precedes the output, and the weave provides synchronization on the common letter c, making sure that an output occurs only after both inputs have arrived. The C-element is similar to the JOIN, except that its inputs may be "withdrawn" after they are supplied. Another name for the C-element is the *rendezvous*.

As we have already stated, a trace structure specifies all possible sequences of communications that can occur between a component and its environment. An input or output produced when it is not allowed by the specification represents a violation of *safety.* If each specified trace can indeed occur, then a *progress* condition is satisfied. Note that the occurrence of all traces is not guaranteed, and the progress condition is not sufficiently strong to exclude *deadlock* (a path in the behavior where the desired events do not occur, though they may occur in another path) and *livelock*. In spite of these limitations, the theory has been used successfully.

We close this section with an example of the RGDA arbiter, where the letters stand for "request, grant, done, acknowledge." The arbiter communicates with two processes, each with four terminals called r_1, g_1, d_1, a_1 and r_2, g_2, d_2, a_2. The communications with each process taken separately must follow the RGDA pattern; this is captured by the commands $\mathbf{pref}(r_1?g_1!d_1?a_1!)^*$ and $\mathbf{pref}(r_2?g_2!d_2?a_2!)^*$. Thus the arbiter receives a "request," issues a "grant," receives a "done" signal and produces an "acknowledge" signal. Of course, this specification is not complete, because we must ensure that the arbiter serves only one process at a time. To specify that the process must be served exclusively, we insist that a "grant" can be made only after a "done" has been received. This is covered by the command $\mathbf{pref}(g_1!d_1? + g_2d_2!)^*$. Since all three conditions are required, the complete specification becomes

$$\mathbf{pref}(r_1?g_1!d_1?a_1!)^* \parallel \mathbf{pref}(r_2?g_2!d_2?a_2!)^* \parallel \mathbf{pref}(g_1!d_1? + g_2d_2!)^*.$$

1.12 Further Reading

The literature on asynchronous circuits is quite extensive; see [2] for a survey on delay-insensitivity and additional references. The term *delay-insensitivity* was introduced in connection with the Macromodules project [29]; Molnar used the term "foam-rubber wrapper" to describe a component with delays in its input and output wires [29]. Seitz developed a design methodology called

[4] The weave is not the same as the *shuffle* operation in formal language theory; the shuffle of E and E' includes such words as ac and acb.

self-timed systems (Chapter 7 in [27]). Van de Snepscheut introduced trace theory for modeling asynchronous circuits [47]. Udding [42, 43] was the first to formalize the concept of delay-insensitivity and of the foam-rubber wrapper postulate. A different formalization of the latter was done by Schols [34]. A modified trace theory was used by Dill [15]. Communicating sequential processes were studied by Hoare [23], and a similar model was proposed by He, Josephs, and Hoare [21]. Algebraic methods for asynchronous circuits were introduced by Josephs and Udding [22]. Van Berkel studied "handshake processes" [46]. Verhoeff extended Udding's rules for delay-insensitivity to include progress concerns [48]. Negulescu developed the algebra of "process spaces" for asynchronous circuits [33].

For a survey of asynchronous design methodologies see [8]. The annual proceedings of the International Symposium on Advanced Research in Asynchronous Circuits and Systems published by the IEEE Computer Society are an excellent source on information on the progress in asynchronous design. An extensive bibliography on asynchronous circuits is available at `http://www.win.tue.nl/async-bib/`.

Acknowledgment I am very grateful to Mihaela Gheorghiu and Jo Ebergen for their constructive comments.

References

1. J.A. Bondy and U.S.R. Murty, *Graph Theory with Applications,* American Elsevier, 1976
2. J.A. Brzozowski, "Delay-Insensitivity and Ternary Simulation," *Theoretical Computer Science,* vol. 245, pp. 3-25, 2000
3. J.A. Brzozowski and J.C. Ebergen, "Recent Developments in the Design of Asynchronous Circuits," *Proc. Fundamentals of Computation Theory,* J. Csirik and J. Demetrovics and F. Gécseg, eds., pp. 78–94, LNCS, Springer, 1989
4. J.A. Brzozowski and J.C. Ebergen, "On the Delay-Sensitivity of Gate Networks," *IEEE Trans. on Computers,* vol. 41, no. 11, pp. 1349–1360, November 1992
5. J.A. Brzozowski and Z. Ešik, "Hazard Algebras," *Formal Methods in System Design,* vol. 23, issue 3, pp. 223–256, November 2003
6. 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., Physica-Verlag, Heidelberg, pp. 3–24, 2003
7. J.A. Brzozowski and M. Gheorghiu, "Gate Circuits in the Algebra of Transients," *Theoretical Informatics and Applications,* to appear
8. J.A. Brzozowski, S. Hauck, and C-J. Seger, "Design of Asynchronous Circuits," Chapter 15 in [10]
9. J.A. Brzozowski and C-J. Seger, "A Characterization of Ternary Simulation of Gate Networks," *IEEE Trans. Computers*, vol. C-36, no. 11, pp. 1318–1327, November 1987
10. J.A. Brzozowski and C-J. Seger, *Asynchronous Circuits,* Springer, Berlin, 1995
11. J.A. Brzozowski and M. Yoeli, "Digital Networks," Prentice-Hall, 1976

12. J.A. Brzozowski and M. Yoeli, "On a Ternary Model of Gate Networks," *IEEE Trans. on Computers,* vol. C-28, no. 3, pp. 178–183, March 1979
13. J.A. Brzozowski and H. Zhang, "Delay-Insensitivity and Semi-Modularity," *Formal Methods in System Design,* vol. 16, no. 2, pp. 187–214, March 2000
14. T.J. Chaney and C.E. Molnar, "Anomalous Behavior of Synchronizer and Arbiter Circuits," *IEEE Trans. on Computers,* vol. C-22, no. 4, pp. 421–422, 1973
15. D.L. Dill, *Trace Theory for Automatic Hierarchical Verification of Speed-Independent Circuits,* PhD Thesis, Computer Science Department, Carnegie Mellon University, February 1988. Also, The MIT Press, Cambridge, MA, 1989
16. J.C. Ebergen, *Translating Programs into Delay-Insensitive Circuits,* PhD Thesis, Department of Mathematics and Computing Science, Eindhoven University of Technology, Eindhoven, The Netherlands, October 1987. Also, CWI Tract 56, Centre for Math. and Computer Science, Amsterdam, The Netherlands, 1989
17. J.C. Ebergen, "A Formal Approach to Designing Delay-Insensitive Circuits," *Distributed Computing,* vol. 5, no. 3, pp. 107–119, 1991
18. E.B. Eichelberger, "Hazard Detection in Combinational and Sequential Switching Circuits," *IBM J. Res. and Dev.,* vol. 9, pp. 90–99, 1965
19. M. Gheorghiu, *Circuit Simulation Using a Hazard Algebra*, MMath Thesis, Dept. of Computer Science, University of Waterloo, Waterloo, ON, Canada, 2001. `http://maveric.uwaterloo.ca/publication.html`
20. M. Gheorghiu and J.A. Brzozowski, "Simulation of Feedback-Free Circuits in the Algebra of Transients," *Int. J. Foundations of Computer Science*, vol. 14, no. 6, pp. 1033-1054, December 2003
21. J. He, M.B. Josephs, and C.A.R. Hoare, "A Theory of Synchrony and Asynchrony," pp. 459–478 in M. Broy and C.B. Jones, eds., *Programming Concepts and Methods,* North-Holland, Amsterdam, 1990
22. M.B. Josephs and J.T. Udding, "An Algebra for Delay-Insensitive Circuits," pp. 147–175 in E.M. Clarke and R.P. Kurshan, eds., *Computer-Aided Verification,* AMS-ACM, Providence, RI, 1990
23. C.A.R. Hoare," *Communicating Sequential Processes,* Prentice-Hall, Inc., Englewood Cliffs, NJ, 1985
24. D.A. Huffman, "The Design and Use of Hazard-Free Switching Circuits," *J. ACM*, vol. 4, pp. 47–62, 1957
25. A.J. Martin, "Compiling Communicating Processes into Delay-Insensitive VLSI Circuits," *Distributed Computing,* vol. 1, pp. 226–234, 1986
26. E.J. McCluskey, "Transient Behavior of Combinational Logic Circuits," in *Redundancy Techniques for Computing Systems,* R. H. Wilcox and W. C. Mann, eds., Spartan Books, Washington, DC, pp. 9–46, 1962
27. C. Mead and L. Conway, *Introduction to VLSI Systems,* Addison-Wesley, Reading, MA, 1980
28. R.E. Miller, *Switching Theory, Volume II: Sequential Circuits and Machines,* Wiley, New York, 1965
29. C.E. Molnar, T.P. Fang, and F.U. Rosenberger, "Synthesis of Delay-Insensitive Modules," *Proc. 1985 Chapel Hill Conference on VLSI,* H. Fuchs, ed., Computer Science Press, Rockville, Maryland, pp. 67–86, 1985
30. M. Mukaidono, "Regular Ternary Logic Functions—Ternary Logic Functions Suitable for Treating Ambiguity," *Proc. 13th Ann. Symp. on Multiple-Valued Logic,* pp. 286–291, 1983
31. D.E. Muller, *A Theory of Asynchronous Circuits,* Tech. Report 66, Digital Computer Laboratory, University of Illinois, Urbana-Champaign, Illinois, USA, 1955

32. D.E. Muller and W.S. Bartky, A Theory of Asynchronous Circuits, in *Proc. Int. Symp. on the Theory of Switching,* Annals of the Computation Laboratory of Harvard University, Harvard University Press, pp. 204–243, 1959
33. R. Negulescu, *Process Spaces and Formal Verification of Asynchronous Circuits,* PhD Thesis, Dept. of Computer Science, University of Waterloo, Waterloo, ON, Canada, 1998
34. 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
35. C-J.H. Seger, *Ternary Simulation of Asynchronous Gate Networks,* MMath Thesis, Dept. of Comp. Science, University of Waterloo, Waterloo, ON, Canada, 1986
36. C-J.H. Seger, *Models and Algorithms for Race Analysis in Asynchronous Circuits,* PhD Thesis, Dept. of Comp. Science, University of Waterloo, Waterloo, ON, Canada, 1988
37. C-J.H. Seger, "On the Existence of Speed-Independent Circuits," *Theoretical Computer Science,* vol. 86, no. 2, pp. 343–364, 1991
38. C.E. Shannon, "A Symbolic Analysis of Relay and Switching Circuits," *Trans. AIEE,* vol. 57, pp. 713–723, 1938
39. S. Silver, *True Concurrency in Models of Asynchronous Circuit Behaviors,* MMath Thesis, Dept. of Computer Science, University of Waterloo, Waterloo, ON, Canada, 1998
40. S. Silver and J.A. Brzozowski, "True Concurrency in Models of Asynchronous Circuit Behavior," *Formal Methods in System Design,* vol. 22, issue 3, pp. 183—203, May 2003.
41. I.E. Sutherland and J. Ebergen, "Computers without Clocks," *Scientific American,* pp. 62–69, August 2002
42. J.T. Udding, *Classification and Composition of Delay-Insensitive Circuits,* PhD Thesis, Department of Mathematics and Computing Science, Eindhoven University of Technology, Eindhoven, The Netherlands, September 1984
43. J.T. Udding, "A Formal Model for Defining and Classifying Delay-Insensitive Circuits and Systems," *Distributed Computing,* vol. 1, no. 4, pp. 197–204, 1986
44. S.H. Unger, "Hazards and Delays in Asynchronous Sequential Switching Circuits," *IRE Trans. on Circuit Theory,* vol. CT–6, pp. 12–25, 1959
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,* PhD Thesis, Department of Computing Science, Eindhoven University of Technology, Eindhoven, The Netherlands, May 1983. Also, *Lecture Notes in Computer Science,* vol. 200, Springer-Verlag, Berlin, 1985
48. T. Verhoeff, *A Theory of Delay-Insensitive Systems,* PhD Thesis, Department of Mathematics and Computing Science, Eindhoven University of Technology, Eindhoven, The Netherlands, May 1994
49. H. Zhang, *Delay-Insensitive Networks,* MMath Thesis, Dept. of Computer Science, University of Waterloo, Waterloo, ON, Canada, 1997