A Technique for Finding and Verifying Speed-Dependences in Gate Circuits [†]

Radu Negulescu

Department of Computer Science
University of Waterloo
Waterloo, Ontario, Canada N2L 3G1

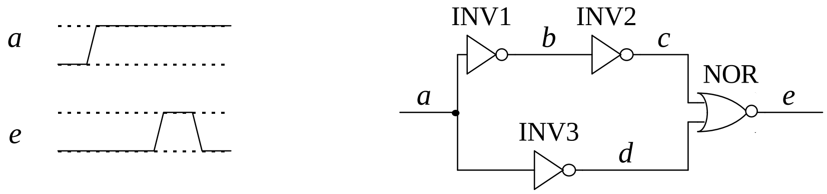
radu@maveric.uwaterloo.ca
ftp://cs-archive.uwaterloo.ca/cs-archive/CS-97-28/CS-97-28.ps.Z

August 11, 1997

Abstract

Before sizing the delays of the components in a circuit, it is often necessary to find delay constraints that would ensure the correctness of the circuit, and to verify the sufficiency of these constraints. We formalize constraints of a certain type as processes in a metric-free model. The circuit specification and components are also represented as processes, to be coupled and compared with the constraint processes. We use a BDD-based tool to verify a circuit together with a set of constraints, and to find hints as to which constraints are needed. We illustrate this technique on a circuit that uses extended isochronic forks.

1. Introduction



(a)

(b)

Figure 1: Is the waveform (a) realized by the circuit (b)?

Consider the problem of designing a circuit that responds to a rising input transition by issuing an output pulse, as in Figure 1 (a). A straightforward implementation would be that in Figure 1 (b): from an initial stable state where *a* is low, the rising transition on *a* causes falling transitions on *b* and *d*, then *e* rises, after which *c* rises and *e* falls.

[†] This research was supported by an Ontario Graduate Scholarship and by Grant No. OGP0000871 from the Natural Sciences and Engineering Research Council of Canada.

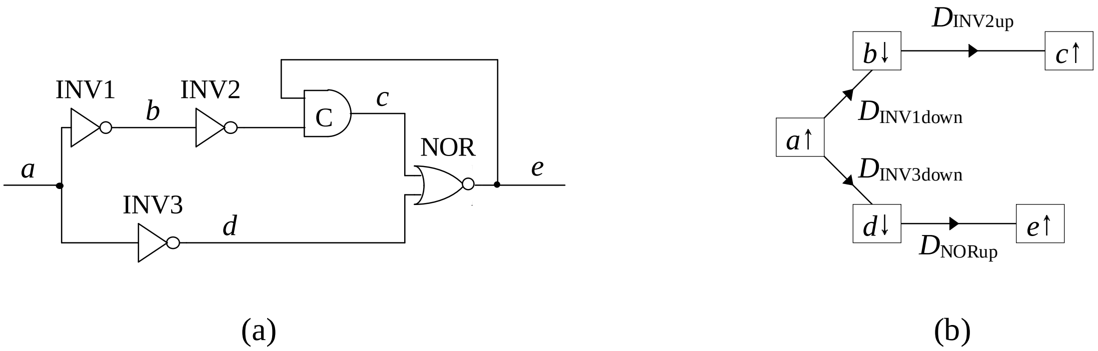


Figure 2: Ensuring correct behavior: (a) by extra circuitry; (b) by relative delays.

However, if INV1 and INV2 are quicker than INV3 and NOR, it is possible that the output pulse will never occur or will be corrupted, since *c* might rise and pull *e* down before *e* completes the upgoing transition. To prevent this problem, one may introduce extra circuitry, as in Figure 2 (a). (The newly introduced gate is a C-element, which copies at the output the value of the inputs if the inputs have the same value, and keeps the output constant if the inputs have different values.) However, the extra circuitry increases the area, delay, and power consumption of the circuit. Fortunately, a cheaper and more efficient solution is available. One can size the component delays in such a way that

$$D_{\text{INV1down}} + D_{\text{INV2up}} > D_{\text{INV3down}} + D_{\text{NORup}} \tag{1}$$

where *D*'s represent gate delays, and 'up' and 'down' refer to transitions of output signals. The two chains of delays involved are shown in Figure 2 (b).

Often, circuit designers face the problem of finding delay constraints *before* knowing the actual delays or delay bounds of the components. Then the question arises whether the detected constraints are sufficient to guarantee correctness of the circuit. The component delays are sized and optimized only after the delay constraints and the correctness of the circuit are established.

To our knowledge, other timing analysis methods have not focused on this specific problem. The methods in [BGM91] and [MD92] only compute bounds on timing separations of events and do not directly verify the functionality of a circuit. The verification method in [My95] requires the component delay bounds to be given, thus it does not address our problem. The timing-reliability problems in [KN94] are related to the problem we are studying, but [KN94] focuses on quantitative analysis for known delay constraints, rather than on finding and verifying the constraints. In [BY75] and [BY76], *almost-equal-delay* models were proposed to analyze a circuit under the assumption that the gate delays are roughly equal; however, [BY75] and [BY76] do not address our problem because their delay constraints are fixed (always a triangle inequality).

In this paper, we propose a semi-automatic technique for finding delay constraints and verifying their sufficiency before the component delays are sized. We use a metric-free process space from [Ne95] to model circuit specifications, components, and delay constraints. Although we have not done efficiency comparisons between our model and timed models, metric-free models should in principle be simpler than timed models and more tractable computationally. We perform the verification by a tool for manipulating such metric-free processes, and the tool also indicates potential flaws by producing executions where the flaws show up. Typically, such an execution contains two events out of order, or an undesirable event, suggesting a delay constraint that needs to be introduced. The tool is based on binary decision diagrams (BDDs). Process spaces are briefly described in Section 2 and the tool in Section 3.

We have applied our technique to circuits from Sun Microsystems Laboratories [MJCL97] and Philips Research Laboratories [vBHP95, Pe96]. In Section 6, we illustrate our technique by analyzing the delay constraints in a handshake-latch circuit from [vBHP95].

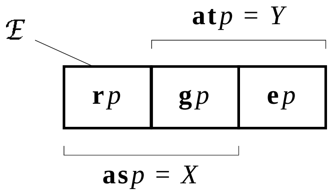
2. Process Space Basics

Process spaces are a simple theory for a large class of interacting systems, including distributed systems, digital circuits, and communication protocols as particular cases. Process spaces unify several correctness concerns by letting the users choose executions of the type most appropriate for their problems. In process spaces, the executions of a system can be sequences of events, functions of time, etc., but a priori we do not impose any structure on executions.

Formalism

The process space formalism is built over an arbitrary set $\mathcal{E}$, as follows. An *execution* is an element of $\mathcal{E}$. A *process* over $\mathcal{E}$ is a pair (X, Y) of subsets of $\mathcal{E}$ such that

(2) $X \cup Y = \mathcal{E}$.



Notation	Name	Set
as p	*accessible*	X
at p	*acceptable*	Y
r p	*reject*	$\overline{Y}$
g p	*goal*	$X \cap Y$
e p	*error*	$\overline{X}$

Figure 3: Execution sets of a process.

A process represents a contract between a device and its environment: the device guarantees that only executions from X may occur, whereas the environment guarantees that only executions from Y may occur. Executions from X are called *accessible* and those from Y are called *acceptable*−the device can 'access' and 'accept' them, respectively. A process splits $\mathcal{E}$ into three disjoint subsets, as in Figure 3. Executions from $\overline{Y}$ are called *rejects* and must be avoided by the environment; executions from $\overline{X}$ are called *errors* and must be avoided by the device; executions from $X \cap Y$ are called *goals* and are legal for both device and environment. The fact that $X \cup Y = \mathcal{E}$, written equivalently $\overline{X} \cap \overline{Y} = \emptyset$, formalizes a separation of responsibilities between the device and the environment: if an execution is in $\overline{Y}$, the environment guarantees to avoid it, thus the device does not need to avoid it too, and that execution is not in $\overline{X}$. The sets of accessible, acceptable, error, reject, and goal executions of process p are denoted by **as**p, **at**p, **e**p, **r**p, and **g**p, respectively.

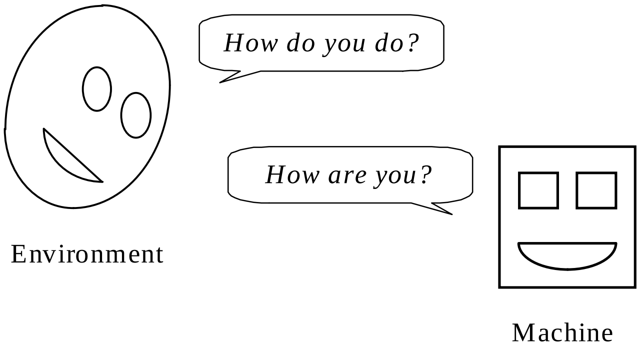


Figure 4: Etiquette machine.

For example, imagine a simple etiquette machine. Proper etiquette is defined as a conversation composed of "How do you do" and "How are you," and the conversation can be arbitrarily long. However, the available vocabulary includes another phrase, "XXX," that nobody wants to hear. If our machine hears "How do you do" or "How are you," it responds by "How do you do" or "How are you." If it hears

“XXX,” the machine might become out of order and may respond by anything, or give no response at all.

To model this machine in process spaces, we take the executions to represent conversations, consisting of strings of greetings between the machine and the environment. For instance, ⟨Environment: How are you⟩ ⟨Machine: XXX⟩ ⟨Machine: How are you⟩ is in $\mathcal{E}$, but ⟨Environment: How do you do XXX⟩ is not in $\mathcal{E}$. In this example we indicate in each greeting who delivered it, because it does matter who says an ⟨XXX⟩, for instance.

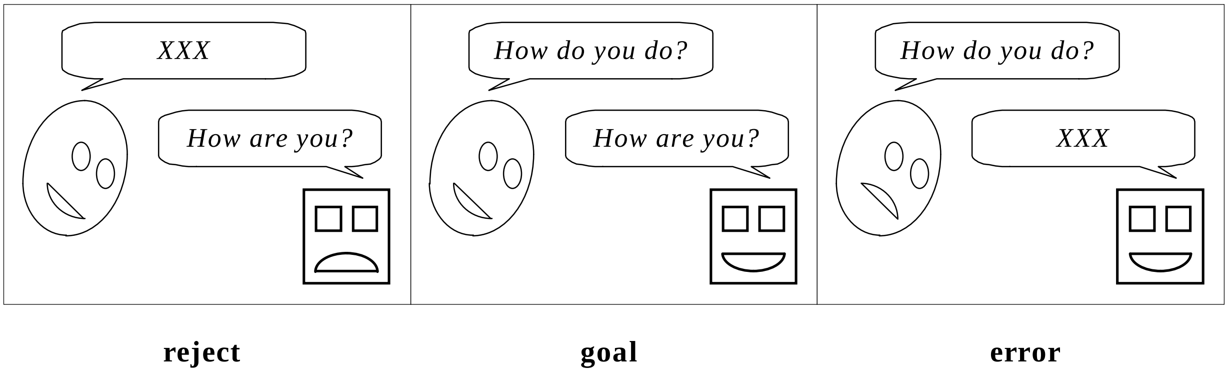
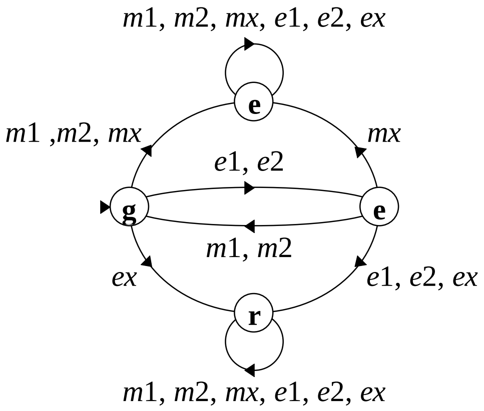


Figure 5: Watch your mouth!

Figure 5 gives examples of goals, rejects, and errors for the etiquette machine. In a goal execution, everybody behaves and everybody is content. In a reject execution, the environment behaves improperly and violates the requirements of the machine, whereas in an error execution, the machine behaves improperly and violates the requirements of the environment.

Greeting	Shorthand
Machine: How do you do	*m*1
Machine: How are you	*m*2
Machine: XXX	*mx*
Environment: How do you do	*e*1
Environment: How are you	*e*2
Environment: XXX	*ex*

(a)



(b)

Figure 6: Process of the etiquette machine: (a) actions; (b) process.

Figure 6 shows in full detail a process for the etiquette machine, as a state machine. The edges are labeled with greetings. To reduce clutter, we use shorthands for the greetings, and we sometimes put several greetings on a single edge. The shorthands are listed in Figure 6 (a). The state markings **r**, **g** and **e** in Figure 6 (b) stand for reject, goal, and error, respectively. The initial state is marked with an incoming arrowhead. A string of greetings is a reject, goal, or error for this process if that string is spelled by a path starting at the initial state and ending at a state marked **r**, **g**, or **e**, respectively. Since every greeting is possible in every state, and thus every string in $\mathcal{E}$ is on a path starting at the initial state, the rejects, goals, and errors cover the whole $\mathcal{E}$. A priori, any conversation is possible, because we don’t know what the environment or the machine would do. However, we forbid the environment or the machine from doing certain things by qualifying those things as rejects or errors.

In other words, the formal representation requires us to classify every string of greetings, including some which were more or less hand-waived in the informal descriptions above. For example, ⟨Environment: XXX⟩ ⟨Machine: XXX⟩ is a reject, because the environment faulted first; ⟨Machine:

How do you do⟩ ⟨Environment: XXX⟩ ⟨Machine: How are you⟩ is an error, because the machine is not allowed to speak out of turn; ⟨Environment: How do you do⟩ ⟨Environment: How do you do⟩ ⟨Machine: How are you⟩ is a reject, because the environment is not allowed to speak out of turn, either; the empty string is a goal, because neither the environment nor the machine have done anything wrong; and, the execution consisting of just ⟨Environment: How are you⟩ is an error, because it would be impolite for the machine not to respond to the greeting. Regarding the last execution, note that an execution represents an entire conversation in this example, not just the beginning of a conversation. Notice that the error state on the right must be exited by a polite greeting from the machine in order to reach a goal state.

Although the etiquette machine is non-deterministic, since it can choose between two available greetings, the state machine in Figure 6 (b) is, formally, a complete deterministic automaton: it has exactly one initial state, and, for each state and for each action, there is exactly one edge going out of that state and labeled with that action. Non-deterministic automata can also be used by our tool to represent and manipulate processes, but, for simplicity, we do not use them in this paper.

Operations and Conditions

Process spaces provide several operations on processes and conditions for correctness. The most important ones are summarized below.

A process is called *robust* if it has no rejects; any execution is acceptable for that process. Robustness models the property that process *p* ‘operates correctly all by itself,’ i.e., that *p* does not impose any requirements on the environment. In other words, a robust process does not rely on the environment to avoid certain executions. The set of robust processes over an execution set $\mathcal{E}$ is denoted by $\mathcal{R}_{\mathcal{E}}$. Formally,

$$(3) \qquad p \in \mathcal{R}_{\mathcal{E}} \iff \mathbf{r}p = \varnothing \iff \mathbf{at}p = \mathcal{E} \,.$$

For example, the process in Figure 6 is not robust, because the executions beginning with ⟨Environment: XXX⟩ are rejects for that process. Robustness is an *absolute* correctness condition, meaning that a process has a correctness property by itself, rather than with respect to a specification.

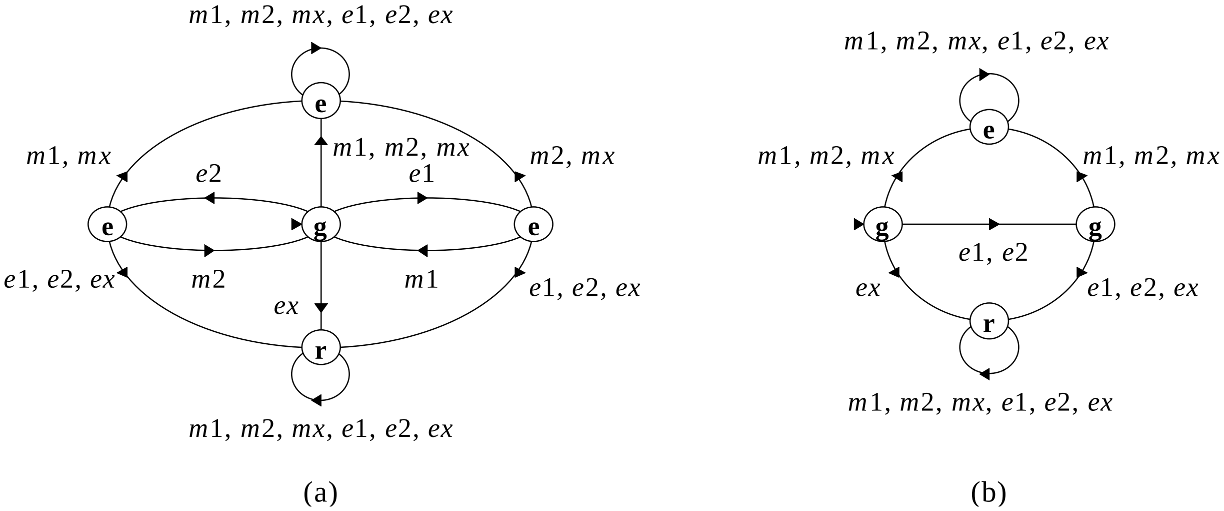


Figure 7: Examples for refinement: (a) repeat machine; (b) quiet machine.

Refinement is a binary relation on processes. Refinement, written $p \sqsubseteq q$, formalizes that process *q* is a ‘satisfactory substitute’ for process *p*. To be a satisfactory substitute, *q* should tolerate at least those environment situations that *p* tolerates, which means *q* has a larger acceptable set. Also, *q* should offer more guarantees than *p*, in terms of avoided executions, which means *q* has a smaller accessible set. Formally,

$$(4) \qquad p \sqsubseteq q \iff \mathbf{at}p \subseteq \mathbf{at}q \;\wedge\; \mathbf{as}p \supseteq \mathbf{as}q \,.$$

Refinement is a relative correctness condition: $p \sqsubseteq q$ can be interpreted as 'a specification p is met by an implementation q.'

Why is it undesirable to have many accessible executions? The smaller the set of accessible executions, the more determinate is the behavior specified by a process. For example, consider a 'repeat machine' that behaves similarly to the etiquette machine, except that, after a polite greeting by the environment, the repeat machine responds by the same greeting. The process in Figure 7 (a) for the repeat machine was obtained from the process in Figure 6 (b) by splitting the rightmost state, to remember the particular greeting used by the environment. Now, observe that the repeat machine is not refined by the etiquette machine, since execution $e1\ m2$ is accessible for the etiquette machine but not for the repeat machine. In other words, the etiquette machine is not a good substitute for the repeat machine, since the etiquette machine might commit a gaffe if the environment is stiff enough to require identical responses.

For another example, consider a 'quiet machine' that has the same requirements as the original etiquette machine, but would not respond to any greeting. The process for such a machine is shown in Figure 7 (b). Execution $e1$ is accessible because it is legal for the quiet machine to stop after an $e1$. However, execution $e1$ is not accessible for the etiquette machine. The quiet machine is not a good substitute for the etiquette machine, because of stopping after $e1$; the fault is detected as an extraneous accessible execution.

The examples above show how to detect safety faults and deadlock faults in this process space over finite words. On the other hand, the repeat machine does refine the etiquette machine, as is easily verified. This is because there is no deadlock and no illegal output event, although it might be considered 'unfair' to respond always by the same greeting. Such unfairness faults are ignored by the finite words process space we use in this paper, but they can be detected by a process space over infinite words, as shown in [Ne95]. One also verifies that neither the repeat machine nor the etiquette machine refine the quiet machine, and the quiet machine does not refine the repeat machine, either.

Reflection, denoted by $-$, is a unary operation on processes. Reflection swaps the accessible and acceptable sets of a process, that is, for process p,

(5) $$-p \;=\; (\mathbf{at}p,\ \mathbf{as}p)\ .$$

Considering that a process represents a contract between a device and its environment from the device point of view, reflection turns the table, i.e., the reflected process represents the environment viewpoint in the same contract. From the definition, it follows that the rejects of a process p are the errors of $-p$, and viceversa. Informally, notice that p requires its environment to avoid executions from $\mathbf{r}p$, while the reflected process $-p$, representing the environment viewpoint, guarantees to avoid those executions. Also, p guarantees to avoid executions from $\mathbf{e}p$, while the reflected process $-p$ requires its environment to avoid those executions.

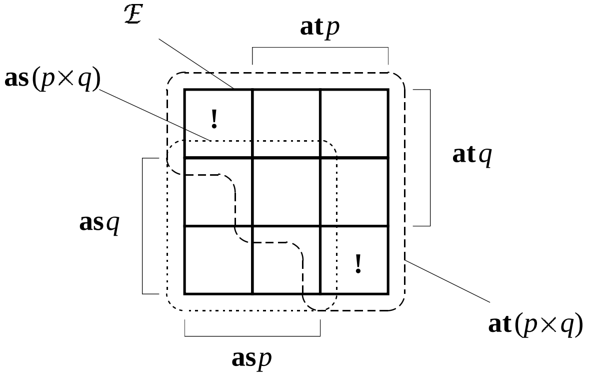


Figure 8: The product operation.

Product, written ×, is a binary operation on processes. The product yields a model for the system of two processes operating 'jointly.' Formally, the product of two processes p and q is defined by

(6) $$\mathbf{as}\,(p \times q) \;=\; \mathbf{as}p \cap \mathbf{as}q\ \text{, and}$$

(7) $$\mathbf{at}\,(p \times q) \;=\; \mathbf{at}p \cap \mathbf{at}q\ \cup\ \overline{\mathbf{as}p \cap \mathbf{as}q}\ .$$

To interpret the formal definition of product, notice that executions that are accessible to both participating devices are accessible for the system, and executions that are acceptable to both devices are acceptable for the system. However, these two intersections do not cover the whole execution set $\mathcal{E}$; in Figure 8, we can see that the executions marked '!' have not been accounted for so far. These executions are errors for the resulting system, because they should be avoided by one of the devices. Accordingly, these executions are acceptable, but not accessible for the resulting system.

The set of acceptable executions of the product can be rewritten as

$$(8) \qquad \mathbf{at}\,(p \times q) \;=\; \mathbf{at}p \cap \mathbf{at}q \;\cup\; \mathbf{r}p \cap \mathbf{e}q \;\cup\; \mathbf{e}p \cap \mathbf{r}q\ .$$

That is, the set of acceptable executions of the product is obtained by augmenting $\mathbf{at}p \cap \mathbf{at}q$ with the executions marked '!', which are errors for one of the devices and rejects for the other.

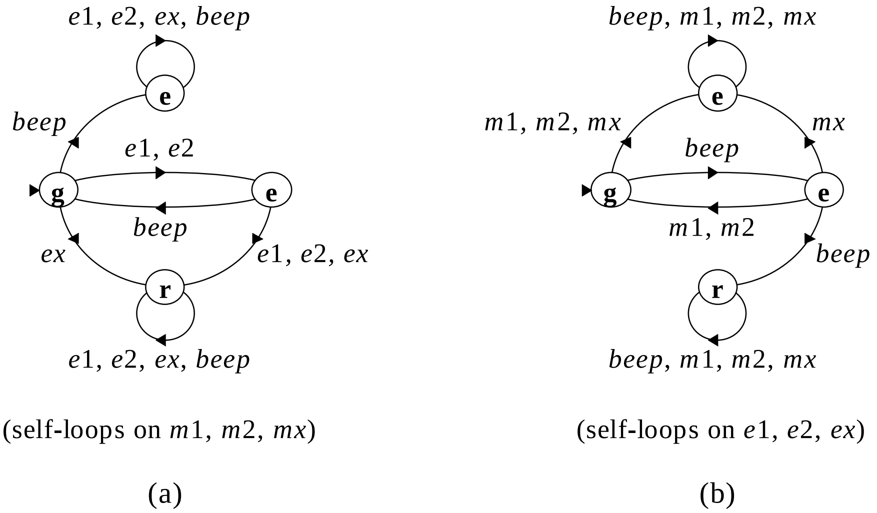


Figure 9: Example for product: (a) Listener; (b) Speaker.

To illustrate the product, let us consider an example of an etiquette machine which is realized as a system of two parts, call them Listener and Speaker. Listener emits a "beep" each time it hears a greeting from the environment, provided that the greeting is not out of turn and not an "XXX." Speaker delivers a polite greeting each time it hears a "beep."

The processes for Listener and Speaker are over the execution set

$$\mathcal{E} \;=\; \{e1,\, e2,\, ex,\, m1,\, m2,\, mx,\, beep\}^*\ ,$$

that is, the executions are strings of greetings and beeps, using the shorthands from Figure 6 (a). The processes for Listener and Speaker are shown in Figure 9. These processes are similar to the process in Figure 6, except that certain actions have been omitted from the state machines. We normally omit those actions that are ignored by a device, in the sense that they do not change the state of the device. More precisely, at every state there should be self-loops labeled with those actions.

Now, let us discuss some of the executions of the product of Listener and Speaker. Execution *e*2 *beep m*1 is accessible and acceptable for both Listener and Speaker, since it leads each of them to a goal state. By the definition of product and Figure 8, this execution is acceptable and accessible for the product process as well. Execution *e*1 *e*2 is accessible and not acceptable for Listener and it is accessible and acceptable for Speaker, since the environment has spoken too fast for Listener, but Speaker has stayed in its initial state, which is a goal state; this execution is accessible and not acceptable for the product process. An execution corresponding to a square marked '!' in Figure 8 is *e*1 *beep beep*; this execution is acceptable and not accessible for Listener and it is accessible and not acceptable for Speaker; thus this execution is acceptable and not accessible for the product process. If a part guarantees to avoid a certain execution, the whole system guarantees to avoid that execution.

One can verify that the original etiquette machine is refined by the product of Listener and Speaker. This product, however, is not refined by the etiquette machine; a counter-example execution is *beep*, revealing that the etiquette machine does not control action *beep*. (For the latter refinement relationship to hold, we would need to 'hide' *beep* from the device-environment interface. However, the hiding operation is not discussed here, as it is not needed in the analysis we perform.)

Algebraic Properties and Structured Verification

Several algebraic properties of process spaces can be used to facilitate formal verification. The refinement relationship is reflexive, transitive, and antisymmetric. Reflection is its own inverse. The process space product is commutative, associative, and idempotent. Furthermore, product satisfies the following monotonicity property with respect to refinement: for processes p, q, and r,

(9) $$p \sqsubseteq q \ \Rightarrow \ p \times r \ \sqsubseteq \ q \times r \, .$$

In words, if p is refined by q, then p coupled with r is refined by q coupled with the same r.

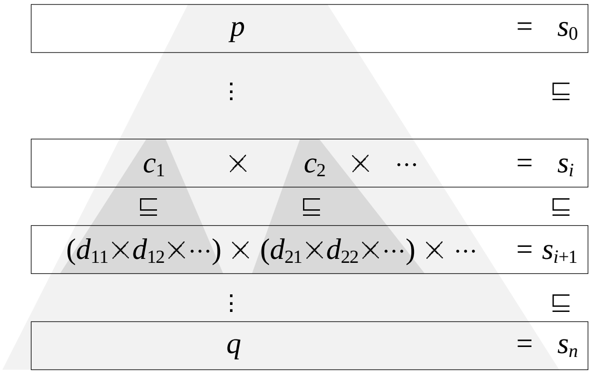

p
$= \ s_0$
$\vdots$
$\sqsubseteq$
$c_1 \quad \times \quad c_2 \ \times \ \cdots$
$= \ s_i$
$\sqsubseteq$
$\sqsubseteq$
$\sqsubseteq$
$(d_{11}{\times}d_{12}{\times}\cdots) \times (d_{21}{\times}d_{22}{\times}\cdots) \times \cdots$
$= s_{i+1}$
$\vdots$
$\sqsubseteq$
q
$= \ s_n$

Figure 10: Hierarchical and modular verification.

Transitivity of $\sqsubseteq$ and monotonicity of $\times$ allow hierarchical and modular verification, which can reduce computational costs by breaking a global verification task into smaller tasks. The problem is to determine whether $p \sqsubseteq q$, where p represents a specification and q an implementation. Typically, one uses several intermediate specifications s_0, s_1, …, s_n such that $s_0 = p$ and $s_n = q$, as shown in Figure 10. The intermediate specifications (including p and q) may be broken into components; for instance, $s_i = c_1 \times c_2 \times \cdots$ and $s_{i+1} = (d_{11} \times d_{12} \times \cdots) \times (d_{21} \times d_{22} \times \cdots) \times \cdots$. One verifies, for each j, that $c_j \sqsubseteq d_{j1} \times d_{j2} \times \cdots$. By monotonicity of $\times$ with respect to $\sqsubseteq$, one obtains $s_i \sqsubseteq s_{i+1}$. By the same procedure, one obtains $s_k \sqsubseteq s_{k+1}$ for each k in $\{0, \ldots, n-1\}$. By transitivity, $p \sqsubseteq q$ follows.

The following property permits to verify whether an implementation refines a specification by placing the implementation in the environment of the specification, and then checking robustness. For processes p and q,

(10) $$p \sqsubseteq q \ \Leftrightarrow \ -p \times q \in \mathcal{R}_{\mathcal{E}} \, .$$

Execution Types

Although in the general process space theory executions have no structure, for a particular application we need to choose particular executions. This choice is done by a tradeoff between precision and complexity of the analysis to be performed. For the problem at hand, we rule out executions that contain explicit timing information in the form of timestamps, timing intervals, etc. 'Timed' executions would permit to verify quantitative properties such as maximum or minimum delays, but they are harder to handle than 'untimed' executions. We limit our choice to finite and infinite sequences of events, with no explicit timing associated to the events. A process space over infinite sequences would capture faults such as unfairness and livelock, which are ignored by the finite sequence model. However, finite sequences are easier to handle and they capture at least safety faults and global deadlocks, which are our primary concerns. In this paper we settle for finite sequences of events, i.e. finite words over a given alphabet. As in the etiquette machine example, these words represent sequences of events during the *entire* operation of a circuit, not just the beginning of the operation.

Related Formalisms

The particular process space formalism that uses finite sequences of events as executions, as well as the conditions and operations in this formalism and their algebraic properties, are closely related to several previous theories of concurrency, notably to [BR85], [Di89], [Eb89], [Eb91], [Ho85], [Jo92], and [Ve94].

Methods for hierarchical and modular verification are common in concurrency theory. Properties similar to (10), linking relative and absolute notions of correctness, were given previously for instance in [Di89], [Eb91], and [Ve94]. Notice, however, that we have eliminated all alphabet restrictions from hierarchical and modular verification and from property (10): arbitrary processes can be connected by product and compared by refinement, regardless of the relationships between their inputs, outputs, and other kinds of actions, while the algebraic properties are shown to hold for arbitrary connections and comparisons, too.

3. FIREMAPS Basics

FIREMAPS is a tool for manipulation of processes whose executions are finite words and whose execution sets are given by finite state machines. The label FIREMAPS stands for finitary and regular manipulation of processes and systems, and also refers vaguely to mapping firings of a Petri net. FIREMAPS has over one hundred operators, including the process space operations and conditions described in Section 2.

For efficiency, FIREMAPS uses a package of BDD routines. BDDs (binary decision diagrams) are data structures for representing Boolean functions [Br86]. BDDs allow for greater average-case efficiency of the Boolean operations than the traditional representations of Boolean functions. The BDD package, presented in [BMB90], was obtained from the Model Checking Group at Carnegie Melon University. FIREMAPS was written by the author, at the University of Waterloo.

In this section, we present a minimum of FIREMAPS syntax and commands necessary to understand the examples of application of our analysis technique for speed-dependences. FIREMAPS is used for other purposes too, but they are not the subject of this paper.

Syntax and Some Data Types

FIREMAPS uses the Polish notation for expressions, where an expression consists of the operator, followed by the operands, while the operands may be constants, variables, or similar expressions. This is in reverse to the HP calculator convention, where the operands are typed in first, then the operator. This notation may not be very common, but it offers several advantages. Firstly, it does not need parentheses, and expressions have less clutter. For instance, 2 * (3 + 4) is written * 2 + 3 4 in Polish notation. Secondly, parsing is made trivial, and we can easily add new commands and functionality to the tool. Thirdly, it does not need operator precedences, which means that the users have less trivia to memorize.

FIREMAPS has several data types: process (`p`), action list (`a`), integer list (`l`), string (`t`), integer (`n`), bit (`b`), etc. Each data type is designated by a single letter, which is used to form some of the operator names as shown below. For each data type, there are operators for reading, writing, and variable assignment, plus some file operations. If `x` is the letter of a data type, then the operators for read, write, and variable assignment are `(xr)`, `(wx)`, and `(=x)`, respectively. The reading operator `(xr)` must be followed by an ASCII representation of the object being read, and it returns that object. The writing operator takes as operand an object and returns nothing. The variable assignment operator takes as operands the variable name and the object being assigned, and it returns nothing. In general, the operators that return nothing constitute commands and can be invoked from the FIREMAPS prompt.

For example, the following two commands read a string, assign it to a variable `my_string`, and print the value of `my_string`.

```
(=t) my_string (tr) "sample string"
(wt) my_string
# the reply was:  "sample string"
```

The # sign starts a comment which ends at the end of the line in which # appears. In response to the first command, the tool does not produce any output. In response to the second command, the tool prints `"sample string"`, as claimed in the comment. We often indicate the results of a FIREMAPS session by inserting such comments in the script.

For the variable assignment for processes, we use the shorthand = instead of `(=p)`.

FIREMAPS exits angrily upon the slightest syntax error. In compensation, however, there are facilities for syntax checking. The FIREMAPS script can be read from a file instead of being typed in each time it is used, and also there is a command that disables the more computationally-intensive process operations.

To input the commands from the file `myscript` instead of the console, just type

```
(in) myscript
```

at the FIREMAPS prompt. There are no quotes around the file name. This command also works with a path instead of just a file name, if the path is from the directory where FIREMAPS was invoked. For example, if the file `myscript` is located in the directory `mydirectory` which is located in the directory where FIREMAPS was invoked, then the command is

```
(in) mydirectory/myscript
```

The command `(pdis)` (for ‘process-operation disable’) disables/enables the costly operations. To do syntax checking on a FIREMAPS script, we invoke `(pdis)` and then we run FIREMAPS normally. The results will not be valid, because the expensive routines are replaced by stubs; on the other hand, we find out in a short time whether the script has proper syntax.

A quick way to exit FIREMAPS is to type in a syntax error. The proper way, however, is to type in the command `(q)`. A list of FIREMAPS operators and commands can be printed by `(h)` or `(help)`.

Some Data Formats

Action and integer lists have a simple format, containing the number of elements in the list, a colon, and then the list itself. For instance,

```
5: a b a c a
```

is a valid action list,

```
2: 15 22
```

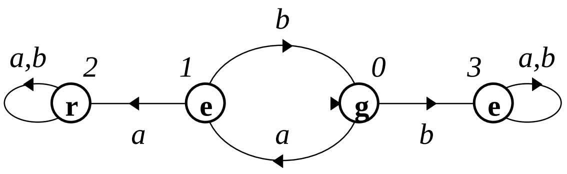
is a valid integer list, and

```
0:
```

is both a valid action list and a valid integer list. On the other hand,

```
2: a
2: a b a
```

are not valid action lists, because the indicated counts do not correspond to actual numbers of actions.



(a)

```
2 actions; 4 states; 8 edges;
actions: a, b;
states: 0 sti, 1 t, 2 s, 3 t;
edges:
from 0: a 1, b 3;
from 1: a 2, b 0;
from 2: a 2, b 2;
from 3: a 3, b 3.
```

(b)

Figure 11: Process format: (a) process; (b) FIREMAPS representation.

Processes have a more intricate, but self-explaining format. For example, the process in Figure 11 (a) is represented by the text in Figure 11 (b). For reference, we have annotated the states in Figure 11 (a) by state codes in italics. The FIREMAPS format for processes should be self-explaining, except for the short strings next to the state codes in Figure 11 (b). These short strings describe the type of a state: if `s` is present, the state is accessible; if `t` is present, the state is acceptable; if `i` is present, the state is initial. There can be more than one initial state. To comply with process space theory, any state that is reachable from an initial state must be either accessible or acceptable or both; otherwise, restriction (2) would be violated.

Process Conditions and Operations

The process space operations product and reflection are implemented by the FIREMAPS operators `*` and `-` respectively. The process space conditions refinement and robustness are implemented by the FIREMAPS operators `[=` and `(R)`, respectively.

The reflection operator takes as arguments two processes and returns a bit. To see that bit, we can print it using the print bit command `(wb)`, as in

```
(wb) [= ep eq
```

where `ep` and `eq` are expressions returning processes. Similarly, the robustness operator takes as argument a process and returns a bit. To see that bit, we can print it, as in

```
(wb) (R) ep
```

where `ep` is an expression returning a process.

If a process is not robust, we may want to know why, and FIREMAPS can produce an action list that represents a reject execution of that process. This is an important facility for diagnosis. The operator is `(aR)`; it takes a process as argument, and it returns an action list representing a reject execution, if one exists. The resulting action list can be printed, assigned to variables, etc. Typically, we print the resulting action list by the command `(wa)`, as in

```
(wa) (aR) ep
```

where `ep` is an expression returning a process. Note, however, that if the `(aR)` operator is invoked with a robust process as argument, then no reject execution exists, no execution is returned by the operator, an error is flagged and FIREMAPS exits. Check robustness before asking for a counter-example to robustness.

Counter-examples to refinement can be obtained as counter-examples to robustness, using the process space property (10) from Section 2. To obtain an execution that contradicts $p \sqsubseteq q$, we ask for an execution that contradicts the robustness of $-p \times q$, as in

```
(wa) (aR) * - ep eq
```

where `ep` and `eq` are expressions returning the processes p and q, respectively.

Integer	State type	Sti marking
7	goal, initial	sti
6	goal, not initial	st
5	reject, initial	si
4	reject, not initial	s
3	error, initial	ti
2	error, not initial	t

(a)

Integer	Trap type
6	goal trap
4	reject trap
2	error trap
0	no trap

(b)

Figure 12: Types for simulation commands: (a) state types; (b) trap types.

To interpret a word that has been found as a counter-example to refinement, we can check the effect of that word on the processes involved. There are several simulation commands in FIREMAPS, taking a process and an action list as arguments, and yielding an integer list as the result. The integers in the integer lists represent the states that have been reached following the given action list. The integers can be: the actual state codes, if the command used is `(apply)`; state types (accessible, acceptable, initial), if the command used is `(sti_apply)`; or trap types, if the command used is `(trap_apply)`. A state has a trap type iff the states reachable from that state are all goals, all errors, or all rejects. The state types and trap types are shown in the tables in Figure 12.

For example, suppose that the process in Figure 11 is assigned to the variable `p`, the action list `3: a b b` is assigned to the variable `t`, and we invoke three simulation commands as follows.

```
= p (pr) 2 actions; 4 states; 8 edges;
actions: a, b;
states: 0 sti, 1 t, 2 s, 3 t;
edges:
  from 0: a 1, b 3;
  from 1: a 2, b 0;
  from 2: a 2, b 2;
  from 3: a 3, b 3.

(=a) t (ar) 3: a b b

(wl) (apply) p t
# the reply was:  4: 0 1 0 3
(wl) (sti_apply) p t
# the reply was:  4: 7 2 7 2
(wl) (trap_apply) p t
# the reply was:  4: 0 0 0 2
```

The first command produces the codes of the states visited, as given in Figure 11: the first state visited is state 0, then state 1, then state 0 again, and then state 3. The second command produces the sti-types of the states visited, which can be found in the table in Figure 12 (a): the first state has type 7, the code for `sti`, i.e., it is a goal and an initial state; the second state has type 2, the code for `t`, i.e., it is an error state; etc. The third command produces the trap types: no trap for states 0 and 1, because goals, errors, and rejects are reachable from these states; and trap type 2, i.e. error trap, for state 3, because only errors, state 3 itself, are reachable from state 3.

To decide refinement, robustness, or to find a counter-example to robustness, FIREMAPS uses a BDD-based breadth-first search of the state-spaces. The search consists of several passes, where at each pass a new layer of states is visited. For monitoring the advance of the computation, FIREMAPS has the command `(trace)`. This command turns on and off the reporting of the number of layers of states that have been visited during the computation.

Sometimes we need to construct a process by changing the initial state of an original model. The operator `(quot)`, for quotient[1], changes the initial state of a process. This operator takes as arguments the process and an action list, and it returns another process which behaves like the original process after the given actions have occurred. In fact, this operator takes the quotients of the languages of the given process with respect to the given word. For example, if `ep` is an expression returning the process in Figure 11, the command

```
(wp) (quot) ep (ar) 3: a b a
```

prints the following process, which is similar to that in Figure 11 but has a different initial state:

```
2 actions; 4 states; 8 edges;
actions: a, b;
states: 0 st, 1 ti, 2 s, 3 t;
edges:
  from 0: a 1, b 3;
  from 1: a 2, b 0;
  from 2: a 2, b 2;
  from 3: a 3, b 3.
```

Systems

In typical applications, we use FIREMAPS to either verify or diagnose (provide a counter-example to) the robustness of the product of several processes. Applications that use refinement are, by property (10) from Section 2, particular cases of applications that use robustness.

Verifications and diagnosis of robustness of the product of several processes can be made much more efficient if the product is not computed explicitly, but an equivalent operation is done on the set of processes involved.

These optimizations are very substantial. For their sake, we have introduced in FIREMAPS a data structure called system, which is a list of processes, and has the code letter `P`. A system is built by adding processes to an empty system or by uniting two systems. The operator `(empty)` returns an empty system, the operator `(add) eP ep` takes a system `eP` and a process `ep` and returns a new system by inserting `ep` into `eP`. The operator `(addP) eP1 eP2` takes systems `eP1` and `eP2` and returns their concatenation. There is an assignment operator for systems, `(=P)`. The operator `(prodP) eP` returns the product of the processes in system `eP`.

The operator `(RP) eP` takes a system `eP` and returns a bit representing robustness of the product of the processes in `eP`, while the operator `(aRP) eP` takes a system `eP` and returns an action list representing a counter-example execution to the robustness of the product of the processes in `eP`, if any such execution exists.

For example, to obtain an execution that contradicts $p \sqsubseteq q_1 \times q_2 \times q_3$, we ask FIREMAPS to print a counter-example to robustness of product for a system *sys* containing $-p$, q_1, q_2, and q_3, as in:

```
(=P) sys (add) (add) (add) (add) (empty) - ep eq1 eq2 eq3
(wa) (aRP) sys
```

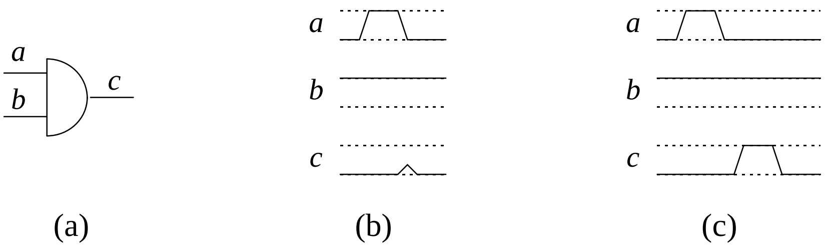
where `ep`, `eq1`, `eq2`, and `eq3` are expressions returning the processes p, q_1, q_2, and q_3, respectively.

[1] Similar operators are called "after" in [Ve94] and elsewhere. We try to use the terminology first introduced for formal languages.

4. CMOS Gate Models

In this section we discuss process space models of Boolean gates and some other circuit components, such as C-elements. Such digital components can be modeled by finite state machines; however, there are some subtleties, because the behavior of a gate is not fully standardized in the presence of hazards.

We are interested in components that have a single output and whose behavior is described by a Boolean function of the inputs and the output, which we call the *instability function*, in the following manner. The instability function specifies when the output value does not correspond to the input values, and thus the output signal is about to undergo a transition. We refer to all such components as *CMOS gates*. For example, consider an AND gate with three inputs a, b, and c, and with output d. The instability function is $(a \wedge b \wedge c) \oplus d$, where $\oplus$ is the exclusive-or sign. Noting that exclusive-or is the negation of equivalence, we see that the AND is unstable whenever d is different from the conjunction of a, b, and c. For another example, consider a C-element with inputs a and b and with output c. The instability function is $a \wedge b \wedge \overline{c} \vee \overline{a} \wedge \overline{b} \wedge c$. In words, the C-element is unstable whenever a and b are high and c is low, or a and b are low and c is high. In general, a CMOS gate that has output b and realizes the Boolean function $F(a_1, a_2, ...)$ has the instability function $b \oplus F(a_1, a_2, ...)$, and a CMOS gate that realizes Martin's production rules [Ma90] $U(a_1, a_2, ...) \rightarrow b\uparrow$ and $D(a_1, a_2, ...) \rightarrow b\downarrow$ has the instability function $U(a_1, a_2, ...) \wedge \overline{b} \vee D(a_1, a_2, ...) \wedge b$.



(a) (b) (c)

Figure 13: Responses to a short pulse: (a) gate and signals; (b) incomplete transition; (c) delayed pulse.

The behavior of a CMOS gate is to a large extent standardized, but still there are some situations where the behavior depends on the particular implementation. For example, consider the AND gate in Figure 13 (a). Suppose input b is high and input a is low. Suppose a rises and then falls again before the output gets a chance to rise. What happens next? Depending on the implementation, the output signal might not become high, as shown in Figure 13 (b), or it might have a delayed pulse, as in Figure 13 (c). Roughly speaking, CMOS gate models where an output transition is canceled upon retraction of the input excitation, as in Figure 13 (b), are called *inertial* models, whereas models where output transitions copy precisely the changes in the input excitation, as in Figure 13 (c), are called *ideal* models. Variations of these models and other CMOS gate models are also possible. For a detailed discussion of gate and delay models, see [BS95].

Situations where a CMOS gate becomes unstable and then stable again before undergoing an output transition, as in Figure 13, are called *hazards*. Since several behaviors of a gate may be expected in a hazard situation, we need to know whether hazards may occur and, if so, for which gates. One can simply forbid hazards at first by using gate specifications that declare hazards illegal. We call such CMOS gate specifications *hazard-intolerant* models. Then, any hazards will be detected as violations of the gate specifications. Should any hazards show up, one can use more detailed specifications to model the particular behaviors of those gates exposed to hazards.

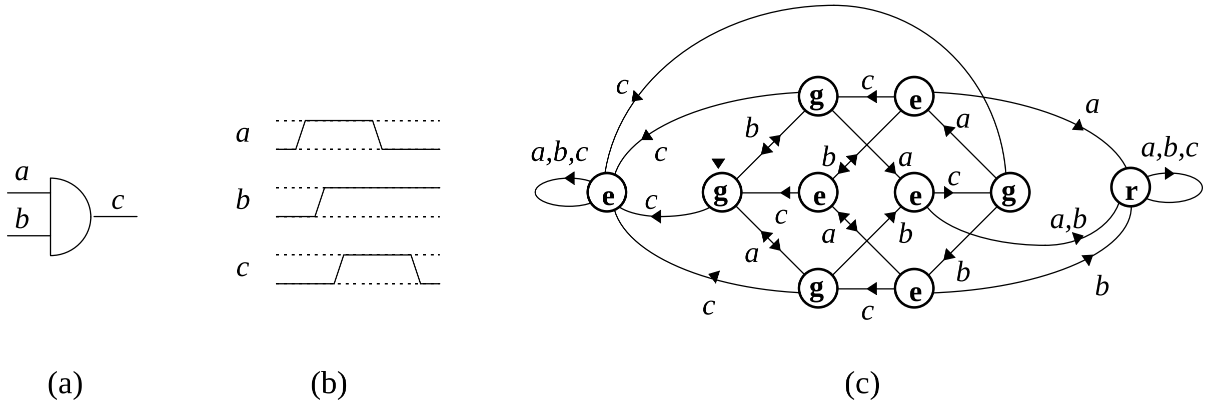


Figure 14: Hazard-intolerant model: (a) gate and signals; (b) execution *abcac*; (c) process.

For an example of a hazard-intolerant model in process spaces, consider the AND gate in Figure 14 (a). Executions represent waveforms as finite sequences of actions from {*a*, *b*, *c*}. Each action represents a rising or falling transition on the corresponding signal. For instance, the waveform in Figure 14 (b) is represented by *abcac*. The process for the hazard-intolerant model of the AND gate is shown as a state machine in Figure 14 (c). The state machine has ten states: eight states for the eight possible combinations of logic levels of the signals of the gate, one error trap state for executions containing an illegal output, and one reject trap state for executions containing an illegal input. Unless otherwise specified, we assume that in the initial state all signals are low. Then, in the neighboring state after an *a* transition, the signal values become *a* high, *b* low, *c* low; and so on. Let us discuss some executions in the hazard-intolerant model. Execution *abcabc* is a goal because it leads back to the initial state, which is a goal state. Execution *aca* contains an invalid output, and, accordingly, it leads to the error state on the left. Execution *ab* is an error because it leads to an unstable state, and the gate should respond by a *c*, not stop after *ab*. (In this paper, we are taking an execution to represent the entire operation of the gate, not just the beginning of the operation.) Finally, execution *abb* represents a hazard, modeled as a reject, because the gate becomes unstable after the first *b* and stable again after the second *b*, without performing an output transition.

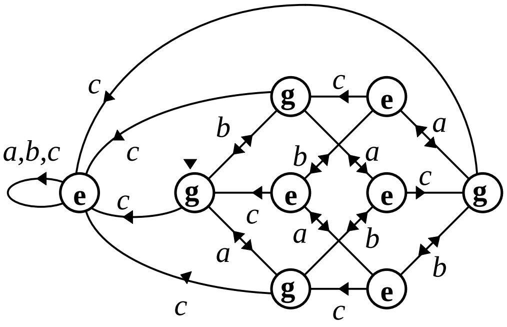


Figure 15: Process for the inertial model of the gate in Figure 14 (a).

For an example of an inertial model in process spaces, consider the process in Figure 15, representing the inertial model of the AND gate in Figure 14 (a). This process is similar to the hazard-intolerant model, except that input transitions from an unstable state to a stable state are permitted. For instance, execution *abb* represents a hazard as described above, but it leads to a goal state of the inertial model in Figure 15.

The hazard-intolerant models extend to arbitrary CMOS gates the asynchronous models used e.g. in [Di89] for Boolean gates, while the inertial models correspond to the widely known inertial delay models described e.g. in [BS95].

FIREMAPS provides operators for building CMOS gate models directly from the instability functions of the gates. These operators are `(hazard-intolerant)` for the hazard-intolerant model and `(inertial)` for the inertial model. Each of these operators takes as arguments a Boolean function representing the instability function of a CMOS gate and an action list containing just the output of the gate. This format was chosen for convenience. The operator `(Br)` reads and returns a Boolean function.

The format of a Boolean function read by `(Br)` is, again, in the Polish notation. The function uses the symbols for the input and output signals of the gate as variables. The Boolean operators used in the function are represented by: `&` for conjunction, `|` for disjunction, `!` for negation, `^` for exclusive-or, `->` for implication, and `<->` for equivalence.

The following expression returns the hazard-intolerant model for an AND gate with inputs *a*, *b*, and *c* and output *d* :

```
(hazard-intolerant)
   (Br) ^ d & & a b c
   (ar) 1: d
```

The `(hazard-intolerant)` operator returns an internal representation of the hazard-intolerant process for the AND gate, and takes as arguments a Boolean function, read and returned by `(Br)`, and an action list, read and returned by `(ar)`. The Boolean function is the instability function $d \oplus (a \wedge b \wedge c)$ of the AND gate, as we have discussed at the beginning of this section. The action list contains just one action, *d*, which is the output of the gate.

Notice that the gate inputs were not listed separately, but the output was. The operator `(hazard-intolerant)` reads all gate actions from the expression for the instability function. The gate output is given in a separate action list because there is no way to distinguish the output from the inputs by reading the instability function only.

The following expression returns the inertial model of a C-element with inputs *a* and *b* and output *c* :

```
(inertial)
   (Br)  |  & & a b ! c  & & ! a ! b c
   (ar) 1: c
```

The instability function of this C-element is thus $a \wedge b \wedge \overline{c} \vee \overline{a} \wedge \overline{b} \wedge c$ and the output is *c*.

5. Chain Constraints

To enforce a particular ordering of two events, one can either rely on causality or on relative delays. If causality between two events is used, one may introduce some extra circuitry that does not allow the second event to occur until the first event has occurred. The causality approach has the inconvenience that the delay of the extra circuitry is introduced between the two events. If relative delays are used instead, one chooses the delays of the circuit components so that the total delay on a path from a third event to the first event is shorter than the total delay on a path to the second event. This way, the delay between the first and the second events can be reduced arbitrarily, and area and power savings can be achieved.

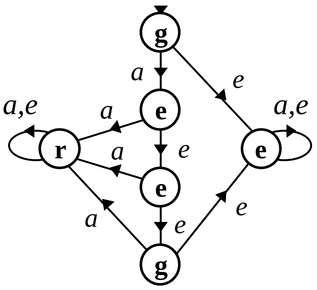


Figure 16: Specification process.

For example, consider again the circuit in Figure 1 (b). The pulse specification in Figure 1 (a) is represented by the finite-state process in Figure 16. From the initial state, *a* is supposed to switch, then *e*, and then *e* again. Illegal inputs lead to a reject state, and illegal outputs to an error state. Stopping before two *e* transitions is also illegal, and the corresponding states are error states. Stopping is legal after two *e* transitions and before the first *a* transition, since normally the environment does not need to ever initiate the operation of the circuit. Events other than *a* and *e* are ignored by this interface specification, meaning that they do not change the state of the process. The gate behaviors are represented by hazard-intolerant models. The problem is to determine whether the specification process is refined by the product of the gate processes. FIREMAPS reported that refinement does not hold and produced counter-example execution *abcd*, indicating a hazard at the NOR gate. This execution shows that the circuit might end up in a stable state where signals *a* and *c* are high and signals *b*, *d*, and *e* are low, while an *e* transition is expected. Thus, in absence of delay constraints, the circuit might not behave as specified.

One way to deal with the flaw above is to introduce extra circuitry, as in Figure 2 (a), for which the product of the hazard-intolerant models of the gates does refine the specification process above.

Another way to deal with the flaw, without introducing new gates, is to impose constraint (1). To verify the circuit with the constraint, we model constraint (1) as a process that forbids certain executions by declaring them to be errors. It is the responsibility of an imaginary constraint device to avoid executions such as *abcd*, *abdc*, *abd*. The constraint process is built automatically by FIREMAPS from the lists of actions involved, the values of the corresponding signals at the state where the 'race' starts, and the two competing delay chains. In this case, the actions involved in the constraint are *a*, *b*, *c*, *d*, and *e*; the competing delay chains start at the initial state; the chain with the longer delay is *abc*; and, the chain with the shorter delay is *ade*. FIREMAPS reported that refinement holds between the specification process in Figure 16 and the product of the new constraint process and the gate processes.

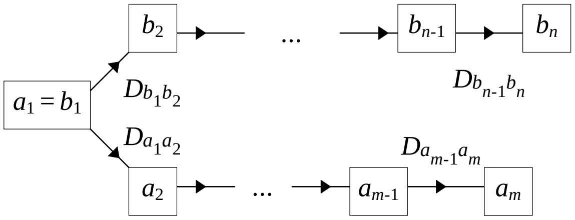


Figure 17: General form of competing delay chains.

In general, we use a constraint of the form

$$(11)\qquad D_{b_1b_2} + ... + D_{b_{n-1}b_n} > D_{a_1a_2} + ... + D_{a_{m-1}a_m}$$

to ensure that a_m will occur before b_n, where a_1, ..., a_m, b_1, ..., b_n are events such that a_1 is the same as b_1, and the *D*'s are delays between the events indicated in subscripts; see Figure 17. The interpretation of such a constraint is that the *b* chain of events will not be completed before the *a* chain, unless one of the *a* or *b* actions involved occurs out of order, disrupting the chains. Such a constraint is different from the simple delay inequality $D_{b_1b_n} > D_{a_1a_m}$; we shall discuss this point on an example later in this section. We call the constraints of the form (11) *chain constraints*.

We model a chain constraint as a process. The starting point of the delay chains is a state called the *base state* of a process called the *base process*. The base process 'monitors' the circuit, and, if the base state is reached and the first event of the chains occurs, then a 'race' between the two delay chains is started. Recall that the first event is the same for the two chains.

For example, consider the chain constraint in expression (1), referring to the circuit in Figure 1 (b). The 'race' starts when signals *b* and *d* are high, signals *a*, *c*, and *e* are low, and an upgoing transition on *a* occurs. The base process is a five-dimensional hypercube, to keep track of the logic values of these five signals. Each state of the hypercube corresponds to one of the 32 possible configurations of logic values of these five signals. The base state corresponds to the signal configuration mentioned above. The initial state of the chain constraint process corresponds to the initial state of the circuit. In this example the base state is the same as the initial state, but in general they can be different.

The chain constraint process is obtained by adding new states to the base process, to monitor the progress through the two delay chains after the base state. The first event of the chains is now taken to lead from the base state to one of the new states. If the chain that is supposed to take longer is completed before the other chain of events, then an error trap state is reached. If one of the chains is left before any chain is completed, we go back to the states of the original base process. Except for the error trap state, all states of the chain constraint process are goal states. The chain constraint process enforces that the chain that is supposed to take longer will not be completed before the other, but the chain constraint process does not intervene in any other way in the behavior of the circuit or the environment.

For example, consider again the circuit in Figure 1 (b). The full listing of the process for the chain constraint in expression (1) is given in Appendix 1. The initial state of this process happens to be its base state. Execution *abdc* leads through states 0, 33, 35, 36, and 32, to the error trap state (state 32) of the chain constraint process, because the chain *abc* is not supposed to be completed before the chain *ade*. Execution *abdadb* leads through states 0, 33, 35, 36, 10, 2, and 0 to a goal state (the initial state), because the chains have been left by the second *a* transition.

Thus, the meaning of a chain constraint process is to forbid every execution *u* that splits into three parts u_1, u_2, and u_3, such that u_1 puts the base process into the base state and u_2 violates a delay chain inequality. For example, execution *abdcdda* splits up into u_1 = the empty word, u_2 = *abdc*, and u_3 = *dda*. The violation occurs if u_2 is the 'interleaving' of two words *v* and *w* such that *v* is the first chain of actions (the chain that is supposed to take longer) and *w* is a prefix of the second chain of actions. In the example above, *v* = *abc* and *w* = *ad*. More precisely, u_2 is the interleaving of *v* and *w* if each of *v* and *w* can be obtained by deleting from u_2 the actions of the other chain. In the example, *v* can be obtained by deleting *d* and *e*, and *w* by deleting *b* and *c*.

Notice that, if the chains are disrupted by an extraneous event, the constraint does not apply. For instance, execution *abdac* is a goal for the process in Appendix 1, and thus it is not ruled out by the respective chain constraint. This is because the second *a* event is not expected in the chains.

FIREMAPS provides the operator `(cc)` for building a chain constraint process from the base process, the base state, and the two chains of events. A chain constraint process is yielded by an expression of the form

```
(cc) ep ea1 ea2 ea3
```

where `ep` is an expression returning the base process, and `ea1`, `ea2`, and `ea3` are expressions returning action lists for a word leading to the base state, for the slow delay chain, and for the quick delay chain, respectively. Hypercubes can also be built by a special operator, `(hypercube)`, which takes as argument an action list with the actions of the hypercube.

For example, a process listing like that in Appendix 1 can be obtained by the commands:

```
= pulse (cc)                                                # (A1.1)
  (hypercube) (ar) 5: a b c d e                             # (A1.1.1)
  (ar) 0:                                                   # (A1.1.2)
  (ar) 3: a b c                                             # (A1.1.3)
  (ar) 3: a d e                                             # (A1.1.4)

(wp) pulse                                                  # (A1.2)
```

Line (A1.1.1) above yields the base process, a hypercube with five actions: *a*, *b*, *c*, *d*, and *e*. Line (A1.1.2) reads an action list that leads from the initial state of the base process to the base state; in this case, the base state is the initial state itself, and the action list is empty. Lines (A1.1.3) and (A1.1.4) read action lists for the competing delay chains; the chain that is supposed to have a larger delay goes first, in line (A1.1.3). Command (A1.2) prints the process.

The action list for a base state does not have any restrictions, but the following restrictions apply on the two action lists for delay chains taken by the `(cc)` operator. Let *x* and *y* be two such action lists.

(i) All the actions in *x* and *y* must be actions of the base process as well;
(ii) *x* and *y* must have a common prefix of length at least one;
(iii) After the largest common prefix, *x* and *y* should have at least one action each;
(iv) Each action of *x* that comes after the largest common prefix of *x* and *y* should be different from each action of *y* that comes after the largest common prefix of *x* and *y*.

The question arises whether one can simply sum up the delays in a chain constraint and use a simpler constraint. For instance, would the chain constraint

(12) $D_{a\uparrow c\uparrow} > D_{a\uparrow e\uparrow}$

have the same effect on the circuit in Figure 1 (b) as the constraint (1)? The answer is no, because chain constraint (12) is inadvertently strong. We used the tool to find counter-examples to refinement between the circuit with chain constraint (12) and the circuit with chain constraint (1), as follows:

```
= pulse2 (cc)
  (hypercube) (ar) 3: a c e
  (ar) 0:
  (ar) 2: a c
  (ar) 2: a e

(=a) init (ar) 2: b d
= inv1 (quot) (inertial) (Br) ^ b ! a (ar) 1: b  init
= inv2 (quot) (inertial) (Br) ^ c ! b (ar) 1: c  init
= inv3 (quot) (inertial) (Br) ^ d ! a (ar) 1: d  init
= nor (quot) (inertial) (Br) ^ e ! | c d (ar) 1: e  init

(wa) (aR) *
  - * * * * inv1 inv2 inv3 nor pulse2
  * * * * inv1 inv2 inv3 nor pulse
# the reply was:  6: a b a a d c
```

where `pulse2` is the new chain constraint and `inv1`, `inv2`, `inv3`, `nor` are the gates in the circuit. The tool has produced execution *abaadc* which is an error for the new chain constraint process but a goal for the original chain constraint process. This execution is allowed by (1) but it is inadvertently ruled out by (12), because the process for (12) does not notice that the delay chain *abc* was started earlier than the delay chain *ade*. (This execution is not permitted by the specification in Figure 1 (a) either, but this is beside the point; in some other application, we might need to compare this circuit with another specification.)

In general, one must be careful when deciding to simplify a chain constraint, because a simpler delay inequality such as (12) may inadvertently guarantee to avoid some potentially dangerous executions, and thus it may mask bugs if it is used in verification instead of the more detailed constraint. One should check that the circuit with a simplified constraint is equivalent to the circuit with the original constraint (double refinement) before using the simplified constraint.

6. Case Study: A Handshake Latch

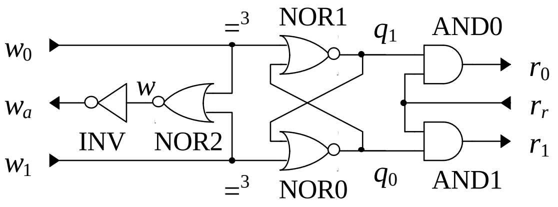


Figure 18: Handshake latch [vBHP95].

The handshake latch circuit in Figure 18 was given in [vBHP95] as an example of usage of extended isochronic forks. A handshake latch is intended to operate as a one-bit memory cell with handshake protocols for reading and writing. Initially, the circuit inputs and outputs are low. A write cycle is supposed to start by raising w_0, for writing a 0, or w_1, for writing a 1. The circuit should raise w_a ('write acknowledge') in response; then w_0 or w_1, whichever was raised, is lowered; and then the circuit should lower w_a, completing the write cycle. A read cycle is started by raising r_r ('read request'); the circuit should raise r_0 if a 0 was stored or r_1 if a 1 was stored; then r_r is lowered; and then the circuit should lower either r_0 or r_1, whichever was raised. The read and write cycles must be mutually exclusive.

The $=^3$ signs indicate extended isochronic forks of depth 2 [vBHP95]. Such a fork bounds the difference in the delays through two layers of gates by the delay in a third layer of gates. In [vBHP95], it is only mentioned that "the fork from w_0 extends to q_0 and w_a," not to r_0; this fork constrains the difference between the delay from w_0 to w_a and the delay from w_0 to q_0 to be less than the delays in those gates that have w_a or q_0 as input. The fork from w_1 is symmetrical to that from w_0.

To illustrate our technique, we model the circuit, its specification, and the extended isochronic fork assumptions as processes, and we use FIREMAPS to compare the circuit and the fork assumptions with the specification described above. The FIREMAPS session is shown in Appendix 2; in the following, we discuss the commands used.

The specification is constructed from several partial specifications by commands (A2.1) through (A2.4) in Appendix 2. Process `writeack` from (A2.1) models the protocol obeyed by signals w_0, w_1, and w_a, as described above, and process `readack` from (A2.2) models the protocol obeyed by r_0, r_1, and r_r, also described above. Process `mutex-consistency` from (A2.3) requires mutual exclusion between the read, write 0, and write 1 operations, and it ensures consistency between the value that is read and the value that was written last.[2] In (A2.3) we assumed that the value stored initially is 0; the analysis would be the same if the value stored initially were 1.

Command (A2.4) builds the overall specification process `spec` from the partial specification processes. We take the product of the reflections of the partial specifications, rather than the product of the partial specifications themselves. To put it briefly, this is because the product operation models the joining of devices in a system, whereas what we join in this case are 'pieces of the environment.'

The processes for the circuit components are built by commands (A2.5) through (A2.10). At first, we use hazard-intolerant models for CMOS gates; we shall replace some of these models later as necessary. Although hazards are not always bugs, it is better to know about them if they may occur, and to ignore them if they are not important. The initial states of these processes have to be changed from all signals at zero to the initial state of the circuit. Initially, signals w_0, w_1, w_a, r_0, r_1, r_r, and q_1 are low and signals w and q_0 are high. For NOR0, AND0, AND1, and NOR2, the initialization sequences of `(quot)` contain one transition for each signal that should be high. For NOR1 and INV, however, the initialization sequences cannot just have one transition for q_0 and w, respectively, because those transitions would be hazards (situations where an output transition is enabled and then disabled without being completed), putting these gates into their reject trap states. Thus, for these gates, we use some longer initialization sequences, still setting all signals to their desired initial values. For instance, for NOR1 we apply $q_1\ q_0\ q_1$, first setting signal q_1 high to stabilize the gate, then q_0 high and q_1 low again to reach the desired values.

Commands (A2.11) and (A2.12) check that, under hazard-intolerant models, the gates NOR2 and INV can be replaced by an OR gate. If the verification holds with the OR gate, then it will also hold with the NOR2 and INV gates instead, by the properties for hierarchical and modular verification.

At this point we check whether the specification is satisfied by the implementation by verifying refinement between their processes. As discussed in Section 3, this is equivalent to checking robustness of product for a system containing all gate processes and the reflection of the specification process. For efficiency, we prefer to work with systems. Command (A2.13) constructs the system to check by inserting processes to an empty system. Command (A2.14) enables tracing, to monitor the progress of the verification task. Command (A2.15) performs the check. The tool finds that refinement does not hold, and we ask for a counter-example execution in command (A2.16). Again, we work with a system instead of single processes. Command (A2.17) disables tracing.

Execution $r_r r_1$ has been found as a counter-example to refinement, indicating that, although initially a 0 is stored in the latch, a 1 is read! We have also checked that, if a 1 is written, a 0 is (inadvertently) read: we write a 1 and then we read the stored value, through simulation commands (A2.18) and (A2.19). In (A2.18) we try to read a 1 and reach an error state, while in (A2.19) reading a 0 leads to a goal state.

[2] In this paper we use a rather verbose format for processes because the state machines are relatively small and this format shows all their details. FIREMAPS also supports some more concise formats, in which the reject trap states, the error trap states, and the 'illegal' edges leading to such states are not shown in the script, but are filled in implicitly by the program.

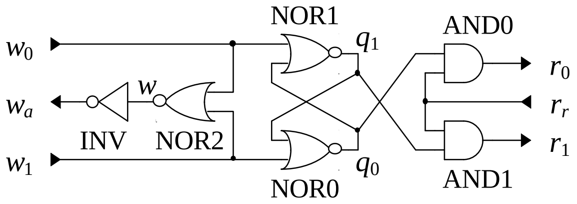


Figure 19: Relabeled handshake latch circuit.

The AND gates appear to have been connected incorrectly, and we fix them as in Figure 19. The correction is minor, amounting to a simple relabeling of the output terminals. Commands (A2.20), (A2.21), and (A2.22) rebuild the processes for the ANDs and the system for the whole implementation accordingly. Commands (A2.23) through (A2.26) redo the verification task.

Another counter-example to refinement, execution $w_1\ w_a\ w_1\ w_a$, is found for the corrected circuit. By the simulation commands (A2.27), (A2.28), and (A2.29), we see that this execution is a reject for the implementation and for the process of NOR0, although it is a goal for the specification. This execution is a hazard for gate NOR0, although the environment has obeyed the specification.

What is the meaning of this hazard? To see what could happen in this ‘hazardous’ situation, we replace the hazard-intolerant models of the NOR gates by inertial models, in commands (A2.30), (A2.31), and (A2.32), and we redo the verification in commands (A2.33) through (A2.36). The counter-example to refinement obtained this time is execution $w_1\ w_a\ w_1\ w_a\ r_r\ r_0$, showing that an invalid value can be read if the w_1 signal is reset before NOR0 performs an output transition. That is, the written value is retracted before the circuit can store it, and then a wrong value is read.

To prevent reading a wrong value as in $w_1\ w_a\ w_1\ w_a\ r_r\ r_0$, we should use the extended isochronic fork assumptions from [vBHP95]. In both the original circuit and the corrected circuit, the difference of the sums of delays $D_{\mathrm{NOR0}} + D_{\mathrm{NOR1}}$ and $D_{\mathrm{NOR2}} + D_{\mathrm{INV}}$ must be less than D_{AND0} because of one of the forks and less than D_{AND1} because of the other fork, where D’s denote gate delays.

Usually, an extended isochronic fork assumption breaks down into several chain constraints, of which only a few are of interest for the functionality of the circuit. Here, we need a constraint that would compel $q_1\!\uparrow$ to occur before the erroneous reading can take place. From the isochronic fork assumptions, we extract an upper bound on the time of occurrence of $q_1\!\uparrow$ by aiming to have D_{NOR1} on the ‘smaller’ side of a delay inequality:

$$D_{\mathrm{AND0}} > |(D_{\mathrm{NOR1}} + D_{\mathrm{NOR0}}) - (D_{\mathrm{NOR2}} + D_{\mathrm{INV}})| \qquad \text{(the fork from } w_0 - \text{see comments below)}$$

$$\Rightarrow \quad D_{\mathrm{AND0}} > (D_{\mathrm{NOR1}} + D_{\mathrm{NOR0}}) - (D_{\mathrm{NOR2}} + D_{\mathrm{INV}}) \qquad \text{(one of the two bounds above)}$$

$$\Rightarrow \quad D_{\mathrm{NOR2}} + D_{\mathrm{INV}} + D_{\mathrm{AND0}} > D_{\mathrm{NOR0}} + D_{\mathrm{NOR1}} \qquad \text{(re-arranging the terms)}$$

$$\Rightarrow \quad D_{w_1\uparrow\, w\downarrow} + D_{w\downarrow\, w_a\uparrow} + D_{r_r\uparrow\, r_0\uparrow} > D_{w_1\uparrow\, q_0\downarrow} + D_{q_0\downarrow\, q_1\uparrow} \qquad \text{(picking a particular case of signal levels)}$$

$$\Rightarrow \quad D_{w_1\uparrow\, w_a\uparrow} + D_{r_r\uparrow\, r_0\uparrow} > D_{w_1\uparrow\, q_0\downarrow} + D_{q_0\downarrow\, q_1\uparrow} \qquad \text{(since we hide } w \text{ in the circuit model)}$$

A chain constraint is obtained by adding positive delays $(D_{w_a\uparrow\, w_1\downarrow} + D_{w_1\downarrow\, w_a\downarrow} + D_{w_a\downarrow\, r_r\uparrow})$ to the left-hand side:

$$D_{w_1\uparrow\, w_a\uparrow} + D_{w_a\uparrow\, w_1\downarrow} + D_{w_1\downarrow\, w_a\downarrow} + D_{w_a\downarrow\, r_r\uparrow} + D_{r_r\uparrow\, r_0\uparrow} > D_{w_1\uparrow\, q_0\downarrow} + D_{q_0\downarrow\, q_1\uparrow} \tag{13}$$

This line of reasoning is probably not the most natural way to extract a chain constraint from an extended isochronic fork assumption, since we started from the fork assumption for w_0 and we ended with delay chains that are triggered by a w_1 event. The matter is somewhat confused by the swapping of the AND gates, which affects the delay assumptions as well. Also, the intermediate delays $D_{w_a\uparrow\, w_1\downarrow}$, $D_{w_1\downarrow\, w_a\downarrow}$, and $D_{w_a\downarrow\, r_r\uparrow}$ had to be added because the AND gates are not used immediately after latching a new value, but only after w_a is lowered and r_r is raised. On the other hand, one may have to overcome many such difficulties in practical applications, and this is what we try to illustrate in a case study.

Commands (A2.37) and (A2.38) construct processes for chain constraint (13) and a symmetrical chain constraint, by means of the special operators `(cc)` and `(hypercube)`, described in Section 5. In (A2.39), we build a system `cons` by inserting the two chain constraint processes into an empty system.

Command (A2.41) checks refinement between the specification and the product of the implementation and the constraint processes, by checking robustness of product of a system containing the gate processes, the constraint processes, and the reflection of the specification process.

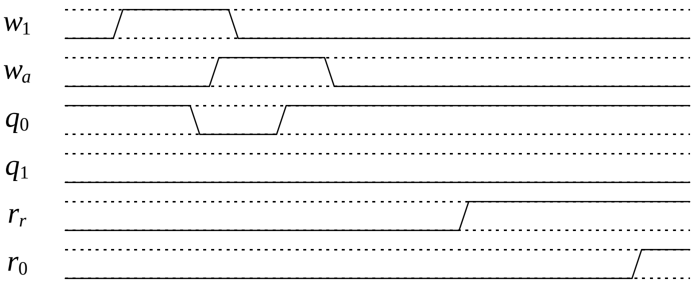


Figure 20: Objection to the delay assumptions for the handshake latch.

A new counter-example is found in (A2.42). To simplify the presentation, we permute a few events in the new counter-example execution, and we obtain execution $e = w_1\ q_0\ w_a\ w_1\ q_0\ w_a\ r_r\ r_0$, representing the waveform in Figure 20. Execution e is also a counter-example to refinement, as shown by the simulation command (A2.44). This execution is not ruled out by the chain constraint (13) because the second w_1 event disrupts the chains and the delay inequality does not apply. Also, it is easy to see that execution e is compatible with both entire isochronic fork assumptions, not just with the chain constraints we have extracted from these assumptions: since r_0 is the last event in e, the delays in the AND gates can be arbitrarily large, larger than the difference between $D_{\mathrm{NOR0}} + D_{\mathrm{NOR1}}$ and $D_{\mathrm{NOR2}} + D_{\mathrm{INV}}$.

What if the fork assumptions bounded the branch delay difference not only by the delay in the AND gates, but also by the delay of a hypothetical gate in the environment, where the extra gate has w_a as input? Wouldn't the extra gate avoid the danger in Figure 20 by retarding the $w_1{\downarrow}$ transition till after a transition on q_1? Not necessarily, since the extra gate delay may or may not intervene in the environment delay from $w_a{\uparrow}$ to $w_1{\downarrow}$. The environment delay from $w_a{\uparrow}$ to $w_1{\downarrow}$ can be made arbitrarily short, shorter than the extra gate delay, for instance, by using extended isochronic forks of the kind used in the circuit itself.

The waveform in Figure 20 contains an undesirable $q_0{\uparrow}$ event. To forbid this event, we needed some stronger delay assumptions. Although, strictly speaking, the fork assumptions from [vBHP95] do not guarantee to avoid execution e above, [vBHP95] seems to intend the environment delay from $w_a{\uparrow}$ to $w_1{\downarrow}$ or to $w_0{\downarrow}$ to be larger than the extra gate delay above, thus larger than the delay differences in the fork assumptions. Accordingly, we imposed that a $q_1{\uparrow}$ event occurs before $w_1{\downarrow}$, and we imposed a symmetrical constraint for $q_0{\uparrow}$ and $w_0{\downarrow}$:

$$(14) \qquad D_{w_1\uparrow\, w_a\uparrow} + D_{w_a\uparrow\, w_1\downarrow} \ > \ D_{w_1\uparrow\, q_0\downarrow} + D_{q_0\downarrow\, q_1\uparrow}$$

$$(15) \qquad D_{w_0\uparrow\, w_a\uparrow} + D_{w_a\uparrow\, w_0\downarrow} \ > \ D_{w_0\uparrow\, q_1\downarrow} + D_{q_1\downarrow\, q_0\uparrow}$$

Command (A2.47) builds a system containing the two new chain constraint processes. Command (A2.49) verifies refinement between the specification and the product of the implementation and the constraint processes, by checking robustness of product for a corresponding system. This time, refinement holds, which means that the new chain constraints suffice to guarantee that the circuit meets the given specification. Command (A2.51) ends the session.

7. Concluding Remarks

In Section 5 we formalized constraints of a certain form, which we called chain constraints, as processes in a metric-free process space from [Ne95]. In Section 6 we analyzed a handshake latch circuit from [vBHP95]. The circuit was found to be correct, except for a swapping of output labels and an objection to the original delay assumptions. We verified that the relabeled circuit meets its specification under slightly stronger delay assumptions. The verification and diagnosis were done by a BDD-based tool, taking less than half a minute on a typical workstation for the script in Appendix 2. The analysis technique is widely applicable and we have used it for several other small industrial designs as well.

Chain constraints and related delay assumptions appear to be used often in the design of low-level CMOS gate circuits. Chain constraints can save area and power, and can increase speed by replacing parts in the circuit. Also, chain constraints can be manipulated conveniently by a metric-free formalism and tool: they can be found in an almost systematic manner and they can be verified automatically. Moreover, safety margin evaluations such as those from [KN94] can be applied to chain constraints to assess the reliability of a circuit in the presence of delay fluctuations.

The circuits we have tried were designed for asynchronous operation, but we do not see any conceptual problems in applying our analysis technique to synchronous circuits, as long as we are dealing with low-level, gate implementations. The handshake latch itself might be operated in a clocked environment. One possible direction for further work is to apply chain constraints to analyze other types of systems, not just gate circuits. We are currently targeting switch-level circuits, i.e. MOS transistor circuits where the transistors are approximated by switches. Another direction is to automate the entire analysis procedure, by deriving new chain constraints from the counter-example executions automatically.

It would be interesting to relate our technique to the more detailed timing methods from [My95], although there may be a difficulty in doing that, since, as shown at the end of Section 5, the chain constraints we use may not boil down to inequalities involving single delays. Also, it may be interesting to adapt our analysis technique to other metric-free formalisms, especially to those in [BR85], [Di89], [Eb89], [Eb91], [Ho85], [Jo92], and [Ve94], which are closely related to the formalism used in this paper. Chain constraints should be straightforward to represent in the cited formalisms. However, there may be a difficulty in transferring our technique to some of these formalisms. Notice that the actions of a chain constraint process are outputs, because the process controls them. In process spaces, although there is no formal distinction among inputs, outputs, inouts (actions that may change input/output direction during the operation of the circuit), and other kinds of actions, an output is easy to recognize because illegal output events lead to an error trap state. We effectively connect outputs of the chain constraint processes to outputs of the gate processes, whereas [Di89], [Jo92], and [Ve94] in their present forms forbid connecting outputs by parallel composition.

In general, we hope that the high flexibility provided by the absence of connectivity restrictions and the abstraction of the notion of execution will make process spaces suitable for dealing with various ‘niche’ applications.

Finally, let us give a suggestion regarding isochronic forks and extended isochronic forks. We have noticed that it is rare that both upgoing and downgoing transitions of a signal need delay constraints; more often, only one direction needs to be constrained. We propose *up* or *down* isochronic forks, meaning that the delay assumptions apply only when the input signal of the fork undergoes a rising or falling transition, respectively. This way, the costs of implementing the forks may be reduced.

Acknowledgments The author is grateful to Charles E. Molnar for frequent consultations and the opportunity to apply and push forward this work during a summer internship at Sun Microsystems Laboratories, and to J. A. Brzozowski for his advice and many comments on the manuscript. Also, the author thanks Ad Peeters and Jo C. Ebergen for their feedback on earlier versions of the manuscript.

Appendix 1. Listing of a Chain Constraint Process

```
5 actions; 37 states; 185 edges;
actions: e, a, c, b, d;
states: 0 sti, 1 st, 2 st, 3 st, 4 st, 5 st, 6 st, 7 st, 8 st, 9 st,
  10 st, 11 st, 12 st, 13 st, 14 st, 15 st, 16 st, 17 st, 18 st, 19 st,
  20 st, 21 st, 22 st, 23 st, 24 st, 25 st, 26 st, 27 st, 28 st, 29 st,
  30 st, 31 st, 32 t, 33 st, 34 st, 35 st, 36 st;
edges:
  from 0: e 16, a 33, c 4, b 2, d 8;
  from 1: e 17, a 0, c 5, b 3, d 9;
  from 2: e 18, a 3, c 6, b 0, d 10;
  from 3: e 19, a 2, c 7, b 1, d 11;
  from 4: e 20, a 5, c 0, b 6, d 12;
  from 5: e 21, a 4, c 1, b 7, d 13;
  from 6: e 22, a 7, c 2, b 4, d 14;
  from 7: e 23, a 6, c 3, b 5, d 15;
  from 8: e 24, a 9, c 12, b 10, d 0;
  from 9: e 25, a 8, c 13, b 11, d 1;
  from 10: e 26, a 11, c 14, b 8, d 2;
  from 11: e 27, a 10, c 15, b 9, d 3;
  from 12: e 28, a 13, c 8, b 14, d 4;
  from 13: e 29, a 12, c 9, b 15, d 5;
  from 14: e 30, a 15, c 10, b 12, d 6;
  from 15: e 31, a 14, c 11, b 13, d 7;
  from 16: e 0, a 17, c 20, b 18, d 24;
  from 17: e 1, a 16, c 21, b 19, d 25;
  from 18: e 2, a 19, c 22, b 16, d 26;
  from 19: e 3, a 18, c 23, b 17, d 27;
  from 20: e 4, a 21, c 16, b 22, d 28;
  from 21: e 5, a 20, c 17, b 23, d 29;
  from 22: e 6, a 23, c 18, b 20, d 30;
  from 23: e 7, a 22, c 19, b 21, d 31;
  from 24: e 8, a 25, c 28, b 26, d 16;
  from 25: e 9, a 24, c 29, b 27, d 17;
  from 26: e 10, a 27, c 30, b 24, d 18;
  from 27: e 11, a 26, c 31, b 25, d 19;
  from 28: e 12, a 29, c 24, b 30, d 20;
  from 29: e 13, a 28, c 25, b 31, d 21;
  from 30: e 14, a 31, c 26, b 28, d 22;
  from 31: e 15, a 30, c 27, b 29, d 23;
  from 32: e 32, a 32, c 32, b 32, d 32;
  from 33: e 17, a 0, c 5, b 35, d 34;
  from 35: e 19, a 2, c 32, b 1, d 36;
  from 34: e 25, a 8, c 13, b 36, d 1;
  from 36: e 27, a 10, c 32, b 9, d 3.
```

Appendix 2. FIREMAPS Session for a Handshake Latch

```
#  SPECIFICATION PROCESSES

= writeack (pr)                                                   # (A2.1)
  3 actions; 8 states; 24 edges;
  actions: w0, w1, wa;
  states: 0 sti, 1 t, 2 st, 3 t, 4 st, 5 t, 6 t, 7 s;
  edges:
    from 0: w0 1, w1 5, wa 6;
    from 1: w0 7, w1 7, wa 2;
    from 2: w0 3, w1 7, wa 6;
    from 3: w0 7, w1 7, wa 0;
    from 4: w0 7, w1 3, wa 6;
    from 5: w0 7, w1 7, wa 4;
    from 6: w0 6, w1 6, wa 6;
    from 7: w0 7, w1 7, wa 7.

= readack (pr)                                                    # (A2.2)
  3 actions; 8 states; 24 edges;
  actions: r0, r1, rr;
  states: 0 sti, 1 t, 2 st, 3 t, 4 st, 5 t, 6 t, 7 s;
  edges:
    from 0: r0 6, r1 6, rr 3;
    from 1: r0 0, r1 6, rr 7;
    from 2: r0 6, r1 6, rr 1;
    from 3: r0 2, r1 4, rr 7;
    from 4: r0 6, r1 6, rr 5;
    from 5: r0 6, r1 0, rr 7;
    from 6: r0 6, r1 6, rr 6;
    from 7: r0 7, r1 7, rr 7.
```

```
= mutex-consistency (pr)                                          # (A2.3)
  6 actions; 16 states; 96 edges;
  actions: rr, w0, w1, wa, r0, r1;
  states: 0 sti, 1 t, 2 st, 3 t, 4 st, 5 t, 6 st, 7 t,
     9 t, 10 st, 11 t, 13 t, 14 st, 15 t,
     8 t, 12 s;
  edges:
    from  0: rr 13, w0  1, w1  5, wa  8, r0  8, r1  8;
    from  1: rr 12, w0 12, w1 12, wa  2, r0  8, r1  8;
    from  2: rr 12, w0  3, w1 12, wa  8, r0  8, r1  8;
    from  3: rr 12, w0 12, w1 12, wa  0, r0  8, r1  8;
    from  4: rr  9, w0  1, w1  5, wa  8, r0  8, r1  8;
    from  5: rr 12, w0 12, w1 12, wa  6, r0  8, r1  8;
    from  6: rr 12, w0 12, w1  7, wa  8, r0  8, r1  8;
    from  7: rr 12, w0 12, w1 12, wa  4, r0  8, r1  8;
    from  9: rr 12, w0 12, w1 12, wa  8, r0  8, r1 10;
    from 10: rr 11, w0 12, w1 12, wa  8, r0  8, r1  8;
    from 11: rr 12, w0 12, w1 12, wa  8, r0  8, r1  4;
    from 13: rr 12, w0 12, w1 12, wa  8, r0 14, r1  8;
    from 14: rr 15, w0 12, w1 12, wa  8, r0  8, r1  8;
    from 15: rr 12, w0 12, w1 12, wa  8, r0  0, r1  8;
    from  8: rr  8, w0  8, w1  8, wa  8, r0  8, r1  8;
    from 12: rr 12, w0 12, w1 12, wa 12, r0 12, r1 12.

= spec - * * - writeack - readack - mutex-consistency             # (A2.4)


#  VERIFICATION WITHOUT DELAY CONSTRAINTS--ORIGINAL CIRCUIT

= nor0 (quot) (hazard-intolerant) (Br) ^ q0 ! | w1 q1 (ar) 1: q0 (ar) 1: q0
                                                                  # (A2.5)
= nor1 (quot) (hazard-intolerant) (Br) ^ q1 ! | w0 q0 (ar) 1: q1
   (ar) 3: q1 q0 q1                                               # (A2.6)
= and0 (quot) (hazard-intolerant) (Br) ^ r0 & q1 rr (ar) 1: r0 (ar) 0:
                                                                  # (A2.7)
= and1 (quot) (hazard-intolerant) (Br) ^ r1 & q0 rr (ar) 1: r1 (ar) 1: q0
                                                                  # (A2.8)
= nor2 (quot) (hazard-intolerant) (Br) ^ w ! | w0 w1 (ar) 1: w (ar) 1: w
                                                                  # (A2.9)
= inv (quot) (hazard-intolerant) (Br) ^ wa ! w (ar) 1: wa (ar) 3: wa w wa
                                                                  # (A2.10)

= or (hazard-intolerant) (Br) ^ wa | w0 w1 (ar) 1: wa             # (A2.11)
(wb) [= or * nor2 inv                                             # (A2.12)
# the reply was:  TRUE

(=P) impl (add) (add) (add) (add) (add) (empty)
   nor0 nor1 or and0 and1                                         # (A2.13)

(trace)                                                           # (A2.14)
(wb) (RP) (add) impl - spec                                       # (A2.15)
# the reply was:  FALSE
(wa) (aRP) (add) impl - spec                                      # (A2.16)
# the reply was:  2: rr r1
(trace)                                                           # (A2.17)

(wl) (sti_apply) (prodP) impl (ar) 8: w1 q0 q1 wa w1 wa rr r1     # (A2.18)
# the reply was:  9: 7 2 2 2 6 2 6 2 2
(wl) (sti_apply) (prodP) impl (ar) 8: w1 q0 q1 wa w1 wa rr r0     # (A2.19)
# the reply was:  9: 7 2 2 2 6 2 6 2 6


#  VERIFICATION WITHOUT DELAY CONSTRAINTS--CORRECTED CIRCUIT

= and0 (quot) (hazard-intolerant) (Br) ^ r0 & q0 rr (ar) 1: r0 (ar) 1: q0
                                                                  # (A2.20)
= and1 (quot) (hazard-intolerant) (Br) ^ r1 & q1 rr (ar) 1: r1 (ar) 0:
```

```
                                                                  # (A2.21)
(=P) impl (add) (add) (add) (add) (add) (empty)
   nor0 nor1 or and0 and1                                         # (A2.22)

(trace)                                                           # (A2.23)
(wb) (RP) (add) impl - spec                                       # (A2.24)
# the reply was:  FALSE
(wa) (aRP) (add) impl - spec                                      # (A2.25)
# the reply was:  4: w1 wa w1 wa
(trace)                                                           # (A2.26)

(wl) (sti_apply) (prodP) impl (ar) 4: w1 wa w1 wa                 # (A2.27)
# the reply was:  5: 7 2 2 2 4
(wl) (sti_apply) nor0 (ar) 4: w1 wa w1 wa                         # (A2.28)
# the reply was:  5: 7 2 2 4 4
(wl) (sti_apply) spec (ar) 4: w1 wa w1 wa                         # (A2.29)
# the reply was:  5: 7 2 6 2 6


#  VERIFICATION WITHOUT DELAY CONSTRAINTS--INERTIAL MODEL

= nor0 (quot) (inertial) (Br) ^ q0 ! | w1 q1 (ar) 1: q0 (ar) 1: q0
                                                                  # (A2.30)
= nor1 (quot) (inertial) (Br) ^ q1 ! | w0 q0 (ar) 1: q1 (ar) 3: q1 q0 q1
                                                                  # (A2.31)
(=P) impl (add) (add) (add) (add) (add) (empty)
   nor0 nor1 or and0 and1                                         # (A2.32)

(trace)                                                           # (A2.33)
(wb) (RP) (add) impl - spec                                       # (A2.34)
# the reply was:  FALSE
(wa) (aRP) (add) impl - spec                                      # (A2.35)
# the reply was:  6: w1 wa w1 wa rr r0
(trace)                                                           # (A2.36)


#  VERIFICATION WITH DELAY CONSTRAINTS--PASS 1

= cc0  (cc)                                                       # (A2.37)
  (hypercube) (ar) 6: w1 q1 q0 r0 wa rr
  (ar) 0:
  (ar) 6: w1 wa w1 wa rr r0
  (ar) 3: w1 q0 q1

= cc1  (cc)                                                       # (A2.38)
  (hypercube) (ar) 6: w0 q0 q1 r1 wa rr
  (ar) 2: q0 q1
  (ar) 6: w0 wa w0 wa rr r1
  (ar) 3: w0 q1 q0

(=P) cons (add) (add) (empty) cc0 cc1                             # (A2.39)

(trace)                                                           # (A2.40)
(wb) (RP) (add) (addP) impl cons - spec                           # (A2.41)
# the reply was:  FALSE
(wa) (aRP) (add) (addP) impl cons - spec                          # (A2.42)
# the reply was: 8: w1 wa q0 w1 wa rr q0 r0
(trace)                                                           # (A2.43)

(wl) (sti_apply) (prodP) (add) (addP) impl cons - spec
    (ar) 8: w1 q0 wa w1 q0 wa rr r0                               # (A2.44)
# the reply was:  9: 7 2 2 2 2 2 6 2 4


#  VERIFICATION WITH DELAY CONSTRAINTS--PASS 2

= cc2 (cc)                                                        # (A2.45)
```

```
  (hypercube) (ar) 4: w1 q1 q0 wa
  (ar) 0:
  (ar) 3: w1 wa w1
  (ar) 3: w1 q0 q1

= cc3 (cc)                                                       # (A2.46)
  (hypercube) (ar) 4: w0 q1 q0 wa
  (ar) 2: q0 q1
  (ar) 3: w0 wa w0
  (ar) 3: w0 q1 q0

(=P) cons (add) (add) (empty) cc2 cc3                            # (A2.47)

(trace)                                                          # (A2.48)
(wb) (RP) (add) (addP) impl cons - spec                          # (A2.49)
# the reply was:  TRUE
(trace)                                                          # (A2.50)


# QUITTING

(q)                                                              # (A2.51)
```

References

[vBHP95] K. van Berkel, F. Huberts, A. Peeters. Stretching quasi delay insensitivity by means of extended isochronic forks. In *Proceedings of the Second Working Conference on Asynchronous Design Methodologies*, South Bank University, London, UK, IEEE Computer Society Press, 1995.

[BRB90] K. S. Brace, R. L. Rudell, and R. E. Bryant. Efficient implementation of a BDD package. In *Proceedings of the 27th ACM/IEEE Design Automation Conference*, pp. 40-45, June 1990.

[BR85] S. D. Brookes and A. W. Roscoe. An improved failures model for communicating sequential processes. In *Proceedings NSF-SRC Seminar on Concurrency*, pp. 281-305, 1985.

[Br86] R. E. Bryant. Graph based algorithms for Boolean function manipulation. *IEEE Transactions on Computers*, C-35(8):677-691, August 1986.

[BGM91] J. A. Brzozowski, T. Gahlinger, and F. Mavaddat. Consistency and satisfiability of timing specifications. *Networks*, pp. 91-107, January 1991.

[BS95] J. A. Brzozowski and C.-J. H. Seger. *Asynchronous Circuits.* Springer Verlag, 1995.

[BY75] J. A. Brzozowski and M. Yoeli. Models for analysis of races in sequential circuits. In LNCS 28, pp.26-31, Springer Verlag, June 1975.

[BY76] J. A. Brzozowski and M. Yoeli. *Digital Networks.* Prentice-Hall, 1976.

[BM88] S. M. Burns, A. J. Martin. Syntax-directed translation of concurrent programs into self-timed circuits. In J. Allen and F. Leighton, eds., *Proceedings of the Fifth MIT Conference on Advanced Research in VLSI*, pp. 35-50, MIT Press, 1988.

[Di89] D. L. Dill. *Trace Theory for Automatic Hierarchical Verification of Speed-Independent Circuits*. An ACM distinguished dissertation, MIT Press, 1989.

[Eb89] J. C. Ebergen. Translating programs into delay-insensitive circuits. CWI Tract 56, Centre for Mathematics and Computer Science, Amsterdam, The Netherlands, 1989.

[Eb91] J. C. Ebergen. A formal approach to designing delay-insensitive circuits. *Distributed Computing*, (5): 107-119, 1991.

[Ho85] C. A. R. Hoare. *Communicating Sequential Processes*. Prentice Hall, 1985.

[Jo92] M. B. Josephs. Receptive process theory. *Acta Informatica*, 29(1):17-31, 1992.

[KN94] M. Kuwako and T. Nanya. Timing-reliability of asynchronous circuits based on different delay models. In *International Symposium on Advanced Research in Asynchronous Circuits and Systems*, Salt Lake City, USA, pp. 22-31, IEEE Computer Society Press, 1994.

[Ma90] A. J. Martin. Programming in VLSI: from communicating processes to delay-insensitive circuits. In C.A.R. Hoare, ed., *Developments in Concurrency and Communication*, UT Year of Programming Series, pp. 1-64, Addison-Wesley, 1990.

[MD92] K. McMillan and D. L. Dill. Algorithms for interface timing verification. ACM International Workshop on Timing Issues in the Specification and Synthesis of Digital Systems, February 1992.

[MJCL97] C. E. Molnar, I. W. Jones, W. S. Coates, J. K. Lexau. A FIFO ring performance experiment. In *International Symposium on Advanced Research in Asynchronous Circuits and Systems*, Eindhoven, The Netherlands, pp. 279-289, IEEE Computer Society Press, 1997.

[My95] C. J. Myers. *Computer-Aided Synthesis and Verification of Gate-Level Timed Circuits*. Ph.D. Thesis, October 1995.

[Ne95] R. Negulescu. Process Spaces. Technical Report CS-95-48, Computer Science, University of Waterloo, December 1995.
`http://maveric.uwaterloo.ca/~radu/ps.html`

[Pe96] Ad M.G. Peeters. *Single-Rail Handshake Circuits*. Ph.D. Thesis, Eindhoven University of Technology, Eindhoven, The Netherlands, 1996.

[Ve94] T. Verhoeff. *A Theory of Delay-Insensitive Systems*. Ph.D. Thesis, Eindhoven University of Technology, Eindhoven, The Netherlands, 1994.