Production Rule Verification for Quasi-Delay-Insensitive Circuits

James N. Cook

Department of Computer Science
California Institute of Technology
Pasadena, California

In Partial Fulfillment of the Requirements
for the Degree of Master of Science

June 11, 1993

Dedicated to Jerry, Ellen,
and especially Emily.

1 Introduction

Circuit designs have become extremely complex. The need to manage this complexity has led to the development of new automated circuit design methods. Currently these methods lean toward *silicon compilation*, the automated transformation of a high level circuit description into a transistor-level, implementable design. A particularly interesting class of circuits for automatic generation are *delay-insensitive* circuits. Delay-insensitive circuits are designed to operate correctly with any arbitrary but finite delay in wires or in operators [14]. Delay-insensitive circuits must be asynchronous, since the use of a clock would bound the range of delays possible for correct operation. The property of delay-insensitivity has several desirable consequences for automated design [3, 10]:

- **Facilitated layout.** A delay-insensitive design will function correctly after arbitrary changes to the lengths of its wires.
- **Elimination of global clock signals.** This eliminates difficulties in distributing the clock signal simultaneously to all parts of the circuit.
- **Inherently modular designs.** Any component of a delay-insensitive design can be replaced by another logically equivalent component, even if the new component has different delays.
- **Speed optimization.** Transistors can be arbitrarily resized without concern for the correctness of the circuit.
- **Increased robustness.** Delay-insensitive designs are less sensitive than other designs with respect to manufacturing process variations, operating temperatures, source voltages, etc.

Of course, the above advantages come at some cost. Elimination of the global clock signal requires that subcircuits must generate completion signals to indicate that they are done with their computation. This requires additional wiring and increases the complexity of the subcircuits. In addition, the concurrent nature of computation in such circuits can be more difficult to analyze than computation in clocked circuits.

The concurrent nature of delay-insensitive circuit design can make them extremely difficult to debug. Although many errors show up as shorts or hazards during simulation, there is no guarantee that they will appear. Simulations are performed by assigning delays to various components of the circuit; it is possible that these assigned delays will mask such an error. Errors undetected in simulation will cause a circuit design to lose its delay-insensitivity—the circuit's correctness becomes dependent on the actual delays being similar to those assumed in the simulation.

Fortunately these types of errors can be detected in high-level descriptions of a circuit design. Circuits can be conveniently expressed as lists of *production rules*, a notation developed by Alain Martin at the California Institute of Technology [8, 10]. Lists of production rules can be guaranteed to be free of shorts and hazards by examining them for two properties called *stability* and *noninterference*. Delay-insensitive circuits must have these properties for all possible sets of component delays. While it is possible to manually verify sets of production rules for these properties, such manual verification is error prone and becomes unwieldy for all but the smallest circuits.

In this document, we present an automated method for the verification of delay-insensitive circuits expressed as production rules. We begin with a description of Martin's design method and specification of the production rule notation. We precisely define stability and noninterference and relate them to delay-insensitivity. We give sequential and parallel algorithms for performing verification. We provide several examples of the verification method and describe our implementation of the algorithms. We conclude with a summary of this work.

2 Production Rules

Alain Martin's research group at the California Institute of Technology (Caltech) has developed a synthesis method and a set of design tools for *quasi-delay-insensitive* circuits. The class of completely delay-insensitive circuits has been proven to be quite limited [9]; quasi-delay-insensitive circuits are delay-insensitive under the assumption of *isochronic forks* and allow the design of a larger class of circuits. Under the isochronic fork assumption, some of the gate outputs that are connected to multiple gate inputs are labeled isochronic—these outputs are assumed to arrive at all the connected inputs simultaneously (see [8, 10] for details). Quasi-delay-insensitive circuits are a superset of speed-independent circuits; speed-independent circuits can be considered as quasi-delay-insensitive circuits where all wires are labeled isochronic [2]. In this document we exclusively consider quasi-delay-insensitive circuits, despite the use of the phrase delay-insensitive to refer to them.

The synthesis method works as follows. The designer begins by writing a program describing the circuit's behavior; this program has the form of a collection of concurrently executing sequential processes that communicate over one-way data channels. The notation used for these programs is called CSP (Communicating Sequential Processes) and is based on C.A.R. Hoare's original notation [4]. The CSP program is next transformed into a *handshaking expansion* by reducing it to an equivalent set of processes where all communication actions have been replaced with manipulations of shared variables. This handshaking expansion is then transformed into a set of *production rules*, in which all explicit sequencing has been removed. Production rules are the lowest level program description in the design method; they can be easily simulated and can be automatically implemented in CMOS. They are also sufficiently general to be useful in specifying VLSI circuits outside the context of this design method.

In the production rule notation a circuit is described in terms of its variables and the conditions under which transitions on these variables occur. Each production rule has the form $G \to S$, where G is a boolean expression on the circuit's variables and S is a *simple assignment* (e.g., $z\uparrow$ and $z\downarrow$ correspond to $z := true$ and $z := false$). G is called the *guard* of the production rule. Multiple production rules with identical guards are often written with a single guard and several simple assignments. $G \to x\uparrow, y\downarrow, z\uparrow$ is an abbreviated form of $G \to x\uparrow$, $G \to y\downarrow$, $G \to z\uparrow$. A production rule *fires* (executes its assignment) some time after its guard evaluates to true. If the firing of a production rule does not change any circuit variable's value, then the firing is called *vacuous*. If a firing does change some variable, the firing is called *effective*.

Production rules describe both combinatorial and state-holding gates. Figure 1 shows the production rules and transistor implementations of a NAND gate and an inverting Muller C-element. Note that the staticizer used to hold the C-element's state is not explicitly described in the production rules. This is due to the semantics of production rule firing: each firing is equivalent to an assignment to a variable. Thus the variable should hold its value until some other production rule fires and a different assignment takes place.

$$
\begin{array}{lcl@{\qquad\qquad}lcl}
\neg a \vee \neg b & \rightarrow & c\uparrow & d \wedge e & \rightarrow & f\downarrow \\
a \wedge b & \rightarrow & c\downarrow & \neg d \wedge \neg e & \rightarrow & f\uparrow
\end{array}
$$

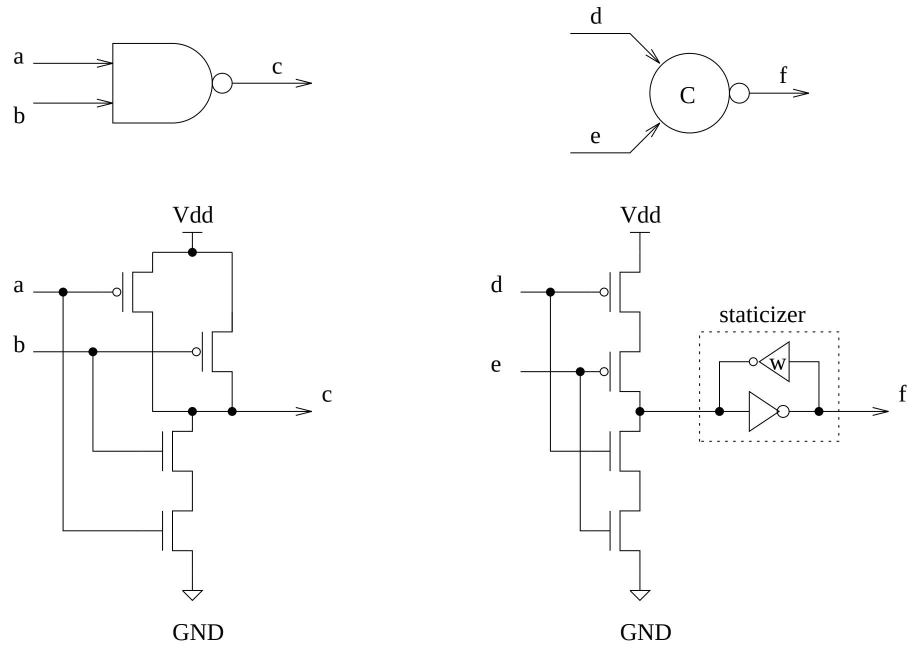


Figure 1: Implementation of NAND gate and inverting C-element.

There are two properties that production rule sets must satisfy; these properties are termed stability and noninterference. Stability is defined as follows:

> A production rule $G \rightarrow S$ is *stable* if every time G becomes true it remains true until the assignment S is completed.

Noninterference only relates to *complementary* production rules, that is, production rules of the form $G1 \rightarrow z\uparrow$ and $G2 \rightarrow z\downarrow$ for some z. Noninterference is defined as follows:

Two complementary production rules $G1 \rightarrow z\uparrow$ and $G2 \rightarrow z\downarrow$ are *noninterfering* if and only if $\neg(G1 \wedge G2)$ holds invariantly.

We call a set of production rules stable and noninterfering if each individual production rule is stable and all complementary production rules are noninterfering [10].

Informally, the stability and noninterference requirements for production rules follow directly from their implementation in CMOS. Figure 2 gives the implementation of two complementary production rules that will illustrate these requirements.

$$\begin{array}{lcl} \neg a & \rightarrow & d\downarrow \\ b \wedge c & \rightarrow & d\uparrow \end{array}$$

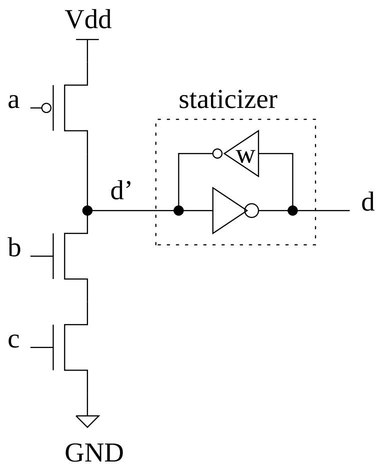


Figure 2: Production rule implementation.

The stability and noninterference requirements now become clear. If production rule $\neg a \rightarrow d\downarrow$ is unstable then a may become false momentarily and quickly return to true. Thus the conducting path from Vdd to d' may not remain conducting sufficiently long for node d to be set high, which could assign an indeterminate value to d. Furthermore, if these two production rules are interfering then $\neg a$ and $b \wedge c$ might become simultaneously true, resulting in a short from Vdd to GND that could also assign an indeterminate value to d.

Although simulation can be used to check for instability and interference, it is not guaranteed to find all such problems. Take, for example, a production rule set containing the following production rules:

$$
\begin{array}{llll}
(1) & a & \rightarrow & b\uparrow \\
(2) & b & \rightarrow & a\downarrow \\
(3) & a \wedge b & \rightarrow & c\uparrow
\end{array}
$$

Assume that the simulator approximates physical delays by assigning to each production rule a delay between the time it will be enabled and when it will fire. (The delay could be based, for example, on the sizes and fanouts of the gates that would implement the circuit.) Let the delays for (1), (2), and (3) be 10, 30, and 20 time units respectively. Then whenever the simulator finds a true the following sequence will occur:

00	a becomes true
10	(1) fires and b becomes true
30	(3) fires and c becomes true
40	(2) fires and a becomes false

Thus the simulator will find that whenever $a \wedge b$ becomes true, it remains true until $c\uparrow$ completes. Unfortunately this is not true in general—if the delay associated with (2) was changed from 30 time units to 10 then a would become false before (3) would be able to fire. Thus production rule (3) would be unstable and the circuit could fail. This circuit is therefore not delay-insensitive; changing component delays can change its behavior. To detect this sort of error more exhaustive verification methods than simple simulation are required.

3 Verification Method

We begin with the single assumption we require for verification. We require only that the production rule set be *closed* before we can check for stability and noninterference.

> A set of production rules is *closed* if and only if for every variable other than Reset used in a production rule's guard there exists a production rule describing an assignment to that variable.

The assumption of closure is necessary because we intend to verify our production rule set with method similar to simulation. We need to have production rules for each variable that will change, including the circuit's inputs from the environment. Note that exact specification of the environment with production rules is not always possible; for example, synchronization and arbitration cannot be expressed in terms of production rules. See Appendix A for information about how environment specification is done in practice.

3.1 Sequential Algorithm

Assume a closed production rule set P with variables x_1, x_2, ..., x_n.

A circuit state is a vector with one element per circuit variable. Each vector element can have the value 0 or 1.[1] For example, $S[k]$ is the value of x_k in state S.

A production rule is a two-tuple consisting of a boolean expression and an assignment. We define several functions on states and production rules.

For production rule p, $\mathbf{trans}(p)$ is the simple assignment (transition) that will be performed by that production rule. For example, $\mathbf{trans}(\neg x_1 \wedge x_2 \vee x_3 \rightarrow x_4\!\uparrow) = x_4\!\uparrow$.

For production rule p and state S, $\mathbf{enb}(p, S)$ is true if and only if the boolean expression of p evaluates to true (i.e., p is enabled) when its variables are assigned the values in S.

For production rule p and state S, $\mathbf{eff}(p, S)$ is true if and only if $\mathbf{enb}(p, S) \wedge ((\mathbf{trans}(p) = x_k\!\uparrow \wedge S[k] = 0) \vee (\mathbf{trans}(p) = x_k\!\downarrow \wedge S[k] = 1))$. Thus, $\mathbf{eff}(p, S)$ is true if and only if if production rule p can cause the circuit to change state when in state S. We call such production rules *effectively enabled* in S.

[1] If the circuit's reset logic is also being tested, vector elements can also have the value U for undefined.

For production rule p and state S, $\mathbf{result}(p, S)$ is defined as follows:

- $\mathbf{trans}(p) = x_k\uparrow \wedge \mathbf{enb}(p, S)$ implies $\mathbf{result}(p, S) = \langle ..., S[k-1], 1, S[k+1], ...\rangle$.
- $\mathbf{trans}(p) = x_k\downarrow \wedge \mathbf{enb}(p, S)$ implies $\mathbf{result}(p, S) = \langle ..., S[k-1], 0, S[k+1], ...\rangle$.
- $\neg\mathbf{enb}(p, S)$ implies $\mathbf{result}(p, S) = S$.

For example, $\mathbf{result}(x_1 \rightarrow x_2\uparrow, \langle 1, 0\rangle) = \langle 1, 1\rangle$, but $\mathbf{result}(x_1 \rightarrow x_2\uparrow, \langle 0, 0\rangle) = \langle 0, 0\rangle$.

---

let $R = \{S_{init}\}$ and let $M = \{\}$
while $R \neq \{\}$ **do**
    remove state S from R
    let $E = \{p \in P \mid \mathbf{enb}(p, S)\}$
    let $E' = \{p \in E \mid \mathbf{eff}(p, S)\}$
    if $\exists p, q \in E, 1 \leq k \leq n$ such that $\mathbf{trans}(p) = x_k\uparrow$ and $\mathbf{trans}(q) = x_k\downarrow$ **then**
        report interference between p and q in state S
    for each $p \in E'$ **do**
        let $S' = \mathbf{result}(p, S)$
        if $\exists q \in E'$ such that $\neg\mathbf{enb}(q, S')$ **then**
            report q unstable
        if $S' \notin M$ **then**
            let $R = R \cup \{S'\}$
    end for
    let $M = M \cup \{S\}$ (mark S)
end while

Figure 3: Sequential verification algorithm.

Figure 3 gives the sequential verification algorithm in terms of the above functions and set operations. The algorithm searches all states that can be reached by production rule firings beginning in initial state S_{init}. During this search, if the algorithm finds a state in which two complementary production rules are both enabled, it reports an interference problem. If it finds a transition between states that disables an enabled production rule before it can fire, it reports an instability.

R and M are sets of circuit states. All states in R remain to be examined. M is a set of "marked" states which the algorithm has already examined. S and S' are states. E and E' are sets of production rules, with all production rules in E enabled in S and all production rules in E' effectively enabled in S. S_{init} is the circuit's initial state. P is a set containing all the circuit's production rules.

Claim: This algorithm finds all instabilities and interferences in production rule set P over variables $x_1, \ldots, x_n$.

Proof: Define a directed graph G as follows. Let each vertex of the graph be one of the 2^n possible circuit states. Let there be an edge from vertex x to vertex y if there exists a production rule p such that $\textbf{result}(p, x) = y$. Note that G may contain cycles. Let all vertices be initially unmarked. We claim that all vertices reachable from the vertex corresponding to state S_{init} will eventually be marked.

First we prove that the algorithm terminates. Since there are a finite number of production rules the inner **for** loop must terminate. Consider the set $R \cup M$. States are never removed from this set—if a state is removed from R it is eventually added to M. Initially $S_{init} \notin M$. Furthermore, each S' added to R is not in M (i.e. $R \cap M = \emptyset$), due to the test in the **if** statement. Thus each iteration of the **while** loop moves one state from R to M. Since the number of possible reachable states on n variables is finite, $|R \cup M|$ is finite. Therefore R will eventually become empty and the **while** loop will eventually terminate. Thus the algorithm terminates.

Suppose there exists some state reachable from S_{init} that is unmarked upon termination. Then there must exist some nonempty subset U of unmarked states reachable from S_{init}. The first statement of the algorithm places S_{init} into R, so the **while** loop will execute at least once and S_{init} will be marked. Since S_{init} is initially marked and all states in U are reachable from S_{init}, there must exist at least one marked vertex in G connected to an unmarked vertex. In the algorithm, before a state is marked, all states immediately reachable from it are added to R. Additionally, each S removed from R is eventually marked. Since the algorithm only terminates when R is empty, there can be no such marked vertex connected to an unmarked vertex. Thus all reachable vertices are eventually marked, so all reachable states are eventually visited.

Assume that two production rules p_1 and p_2 are interfering. By the definition of interference, there exists some reachable circuit state T such that $\textbf{enb}(p_1, T) \wedge \textbf{enb}(p_2, T)$. Since all reachable states are marked, state T must be marked by the algorithm. Before T is marked, however, set E will contain both p_1 and p_2, and the interference between p_1 and p_2 will be reported.

Assume that some production rule q is unstable. Then, by the definition of stability there exists production rule p and reachable state T such that $\textbf{eff}(p, T) \wedge \textbf{eff}(q, T) \wedge \neg\textbf{enb}(q, \textbf{result}(p, T))$. Since all reachable states are marked, T will be marked by this algorithm. Before T is marked, however, set E' will contain both p and q. Thus S' will eventually become $\textbf{result}(p, T)$ and the instability will be reported. $\square$

3.2 NP-Hardness

The need to check all circuit states reachable from S_{init} makes the time-complexity of the above algorithm exponential in the number of circuit variables. It is unlikely that this time-complexity can be substantially improved, as both instability and interference checking are NP-hard. (We will later discuss how to efficiently implement this algorithm.)

Problem: Given a set P of production rules, an initial state S_{init}, and two complementary production rules p_1 and p_2 in P, decide if p_1 and p_2 are interfering (simultaneously enabled in a state reachable from the initial state under some set of delays for production rule firings).

Theorem: Interference checking is NP-hard.

Proof: To prove that interference checking is NP-hard, we reduce the satisfiability problem (SAT) to it. Let E be an instance of SAT—a boolean expression in conjunctive normal form—over variables $x_1, \ldots, x_k$. Construct production rule set P over variables $x_1, \ldots, x_k$ and e as:

$$\begin{array}{lll} true & \rightarrow & x_1\uparrow, ..., x_k\uparrow \\ true & \rightarrow & e\downarrow \\ E & \rightarrow & e\uparrow \end{array}$$

Let the initial state S_{init} be the state in which all variables in P are false. We claim that E is satisfiable if and only if there exists interference between $true \rightarrow e\downarrow$ and $E \rightarrow e\uparrow$. If E is satisfiable, there exists some assignment to $x_1, ..., x_k$ such that E evaluates to true. Let $x_1, ..., x_j$ be the true variables in this assignment. There exists a set of delays in which $true \rightarrow x_1\uparrow$ through $true \rightarrow x_j\uparrow$ all have shorter delays than $true \rightarrow x_{j+1}\uparrow$ through $true \rightarrow x_k\uparrow$. Thus the circuit can reach a state in which $x_1, ..., x_j$ are true and $x_{j+1}, ..., x_k$ are false. In this state both $true \rightarrow e\downarrow$ and $E \rightarrow e\uparrow$ will be enabled, so there will be interference between these two production rules.

Conversely, if there exists interference between $true \rightarrow e\downarrow$ and $E \rightarrow e\uparrow$, then there exists some state in which $E \rightarrow e\uparrow$ is enabled, which implies that E can evaluate to true. Construction of the above production rule set can be done in polynomial time. Thus satisfiability reduces to interference detection and the reduction can be done in polynomial time. Therefore interference detection is NP-hard. □

Problem: Given a set P of production rules, an initial state S_{init}, and a production rule p, decide if p is unstable. (p is unstable if and only if there exist states S_1 and S_2 reachable from S_{init} such that p is effectively enabled in S_1, disabled in S_2, and the firing of some production rule moves the circuit from S_1 to S_2.

Theorem: Instability checking is NP-hard.

Proof: To prove that instability checking is NP-hard, we reduce the satisfiability problem to it. Let E be an instance of SAT over variables $x_1, \ldots, x_k$. Construct production rule set P over $x_1, \ldots, x_k$ and e, f as:

$$\begin{array}{lll} true & \rightarrow & x_1\uparrow, ..., x_k\uparrow \\ E & \rightarrow & e\uparrow \\ \neg e & \rightarrow & f\uparrow \end{array}$$

Let initial state S_{init} be the state in which all variables in P are false. We claim that E is satisfiable if and only if production rule $\neg e \rightarrow f\uparrow$ is unstable. If E is satisfiable, there exists some assignment to $x_1, ..., x_k$ such that E evaluates to true. Let $x_1, ..., x_j$ be the true variables in this assignment. There exists a set of delays in which the delay for $E \rightarrow e\uparrow$ is less than some constant D, and the delays for $true \rightarrow x_1\uparrow$ through $true \rightarrow x_j\uparrow$ are at least D time units shorter than the delays for $true \rightarrow x_{j+1}\uparrow$ through $true \rightarrow x_k\uparrow$ and $\neg e \rightarrow f\uparrow$. Thus the circuit can reach a state in which $x_1, ..., x_j$ are true, $x_{j+1}, ..., x_k$ are false, and $\neg e \rightarrow f\uparrow$ is effectively enabled but has not yet fired. In this state both $E \rightarrow e\uparrow$ and $\neg e \rightarrow f\uparrow$ will be enabled; this state corresponds to S_1. Production rule $E \rightarrow e\uparrow$ will then fire, disabling $\neg e \rightarrow f\uparrow$. Thus $\neg e \rightarrow f\uparrow$ is unstable.

Conversely, if $\neg e \rightarrow f\uparrow$ is unstable, then there exists some state in which e is true. Since $E \rightarrow e\uparrow$ is the only production rule encoding a transition on e and the initial state has all variables false, E must have evaluated to true and therefore be satisfiable. Construction of the above production rule set can be done in polynomial time. Thus satisfiability reduces to instability detection and the reduction can be done in polynomial time. Therefore instability detection is NP-hard. $\square$

3.3 Parallel Algorithm

The problem of circuit state space searching appears suitable for parallel solution, since there are potentially many states to search and each state can be examined independently. This problem is not merely a tree search, however, because the state graph may contain cycles. When new states to check are discovered, they must be tested for membership in the set of states already examined to prevent checking states more than once. This membership check complicates the parallel algorithm. We now describe a parallel adaptation of the above algorithm for the message-passing model of computation.

The basic idea for the parallel algorithm is to distribute the sets R and M over several processes. Each process will have local sets R_i and M_i such that the union of all R_i forms R and the union of all M_i forms M. The way in which these sets are distributed will greatly affect the efficiency of this algorithm.

Distributing the set R is trivial. Whenever a new state is generated it can be sent to an arbitrary process for checking—checking a state for stability and interference requires only the state itself and a description of the circuit's production rules. We require that each process has access to a copy of the production rule set being checked.[2]

Distributing the set M is more difficult. We cannot arbitrarily assign states to processes, as each process needs to check states for membership in the global set M. Thus we are forced to take a somewhat atypical approach to parallelization: rather than directly mapping work onto processes, we instead map the state space. The goal of this mapping is to have each process be "responsible" for some subset of the state space. Each process i will maintain in M_i a record of previously checked states for subset i. It will use R_i to store a set of states remaining to be checked; all states in R_i will also belong to subset i. We therefore need to develop a mapping from states to processes that can be used to generate these subsets.

Our state space to process mapping needs to be reasonably *uniform* for the parallel computation to be efficient. By uniform we mean that the number of states stored in each M_i should be roughly equal. If any one process is responsible for a large number of states then the speed of the computation will be limited by the speed with which that process can decide membership and generate successor states. Furthermore, the space required to store all reachable states may be much larger than the storage space available for a single process; if the state space is unevenly divided then some processes may run out of storage.

One important consideration in choosing such a mapping is that we desire the circuit's *reachable* states to map uniformly onto the processes. This is more difficult than mapping all possible states onto processes, since we have little *a priori* knowledge of which states will be reachable.

Fortunately, there is a way to perform this mapping: we use techniques developed for *hashing*. Since a state is a list of boolean values, we represent it with a sequence of bits, which we can consider as either a string of 8 bit characters or as a large integer. We can then use existing hash functions on these strings or integers to perform the mapping. Thus, for any state S, $\mathbf{hash}(S)$ is the number of the process responsible for that state.

We can utilize our hash function to distribute both R and M. Whenever we generate a new state S' to be examined, we send it to process $\mathbf{hash}(S')$. That

[2] This is not an unreasonable requirement, since both the number of production rules and the size of their representations tend to be small. For example, the control circuitry for an asynchronous microprocessor required less than two hundred production rules to describe.

process can then check if S' has been previously examined by examining its local M_i set and, if not, adding it to its local R_i set for future examination. This method is particularly efficient because it does not require processes to query each other about states belonging to M. Each state is sent directly to the process responsible for it and no response message is required.

Each process i runs the following program:
let $R_i = \{\}$ and let $M_i = \{\}$
repeat
    while message pending **do**
        receive state T
        if $T \notin M_i$ **then**
            let $R = R \cup \{T\}$
    end while
    remove state S from R_i
    let $E = \{p \in P \mid \mathbf{enb}(p, S)\}$
    let $E' = \{p \in E \mid \mathbf{eff}(p, S)\}$
    if $\exists p, q \in E, 1 \leq k \leq n$ such that $\mathbf{trans}(p) = x_k\uparrow$ and $\mathbf{trans}(q) = x_k\downarrow$ **then**
        report interference between p and q in state S
    for each $p \in E'$ **do**
        let $S' = \mathbf{result}(p, S)$
        if $\exists q \in E'$ such that $\neg\mathbf{enb}(q, S')$ **then**
            report q unstable
        let $d = \mathbf{hash}(S')$
        if $d = i$ and $S' \notin M_i$ **then**
            let $R_i = R_i \cup \{S'\}$
        else if $d \neq i$ **then**
            send S' to process d
    end while
    let $M_i = M_i \cup S$ (mark S)
end repeat

The algorithm is initiated by sending S_{init} to process $\mathbf{hash}(S_{init})$.

Figure 4: Parallel verification algorithm.

Figure 4 gives the parallel verification algorithm. This algorithm is almost identical to the sequential one; the only difference is that whenever a new state S' is generated it is not necessarily added to the local R_i. Instead it may be sent to some other process as dictated by the hash function's mapping. As mentioned above, this also takes care of the distributed M membership check.

An important consideration in message-passing parallel algorithms is the locality of communication. If messages are sent arbitrarily between processes then the algorithm will be less efficient than if we can somehow guarantee that processes will only send to processes on "nearby" computing nodes. This increased efficiency stems from two sources: first, shorter physical communication distances decrease the time required to transport the message, and second, the message passing network may be able to carry more messages simultaneously. Of course, this optimization will be dependent on the architecture of the multicomputer used to run the algorithm: the physical wires between processors determine which are neighbors and which are not. For multicomputers that use a hypercube architecture, such as C. L. Seitz's Cosmic Cube [13], we can select a hash function that will cause most communications to be from a process to one of its neighbors.

The ability to localize communication in this manner stems from a crucial realization about production rule simulation: *When we fire an effectively enabled production rule, exactly one bit of the state changes.* This follows directly from the definition of effective. All states S' that are generated by the above algorithm will differ from S in exactly one bit. Furthermore, *neighboring processors in a hypercube have IDs that differ in exactly one bit.* Our goal is therefore to find a hash function which, in addition to uniformly distributing our reachable circuit states, maps states which differ in exactly one bit to processors with IDs that differ in as few bits as possible. In practice, uniformity can be traded for locality of communication. We give several hash functions with different locality/uniformity ratios.

One deficiency of this method is that it only produces local communication on hypercube architectures. However, the problem of mapping hypercubes onto other architectures has been studied by other researchers [1, 6, 7], and will not be discussed here.

3.3.1 String Hashing

Substantial amounts of research has been spent looking for simple hash functions that generate uniform distributions from string inputs. One such hash function, described by Pearson [12], has the benefits of being string-based and quick to compute even on small microprocessors. The function is computed as follows:

```
h := 0;
for i in 1..length(state) loop
  h := Table[ h XOR state[i] ];
end loop;
return h MOD process_count;
```

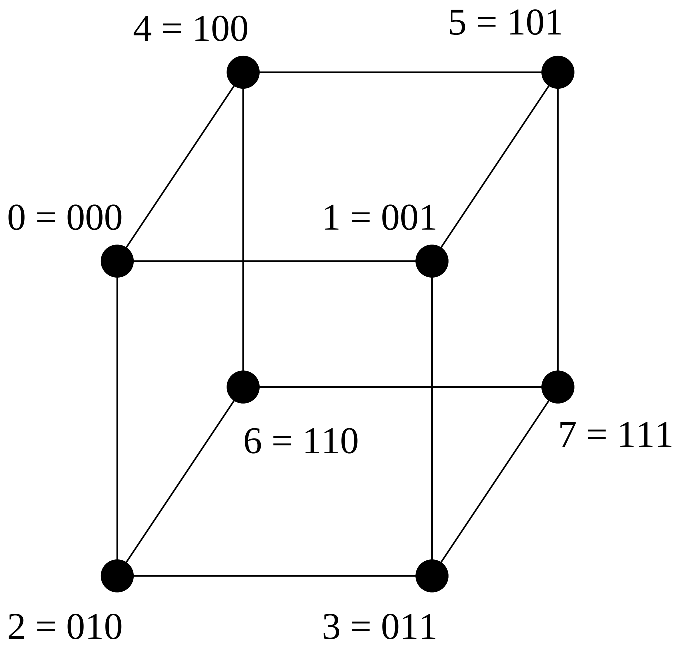


Figure 5: Processor IDs in a hypercube architecture.

(As given, this hash function only works for up to 256 processes. In [12] Pearson describes ways of extending it to larger values.) `Table` contains the numbers 0–255 in random order. We use each character of `String` to store 8 bits of state information. This hash function was found to produce the most uniform distribution of states to processes of any hash function tried. Unfortunately, this hash function does not guarantee any sort of locality, and in our tests produced uniform, random communication distances.

3.3.2 XOR Folding

Another approach we tried was to design a function specifically to generate local communications. One way of doing so is the following: Given 2^n processes, break the state bits into w separate n-bit words. Exclusive-or these words together, and return this value as the value of the hash function. This can be done as follows:

```
h := 0;
mask := (1 LEFT_SHIFT n) - 1;
for i in 1..w loop
  h := h XOR (state AND mask);
  state := state RIGHT_SHIFT n;
end loop
return h;
```

The exclusive-or operator has the property that for a XOR $b = c$, if a' differs from a in exactly one bit, then a' XOR $b = c'$, with c' differing from c in exactly one bit. Thus this hash function has the property that states differing in exactly one bit will generate hash values that differ in exactly one bit. All communications in the hypercube are therefore guaranteed to be with neighbors. Unfortunately, this hash function does not distribute states as uniformly as the Pearson's method.

3.3.3 Prime Hashing

There is a third choice for a hash function. We can treat our state as a large integer and hash it with the method suggested in Knuth [5].

```
return (state MOD prime) MOD process_count;
```

Since the state may contain a large number of bits the modulo operations must be implemented as multiprecision calculations, but this can be easily done by

	Pearson		XOR		Prime	
	SD	avg dist	SD	avg dist	SD	avg dist
Test Case 1 (avg 12 states/process)	3.04	2.11	6.85	1	4.04	1.35
Test Case 2 (avg 148 states/process)	10.37	2.01	25.24	1	48.85	1.44
Test Case 3 (avg 2400 states/process)	39.28	2.04	149.24	1	92.25	1.71
Test Case 4 (avg 450 states/process)	24.55	2.07	32.91	1	14.68	1.93

Table 1: Uniformity and communication distance in a 16-node hypercube (average internode distance = 2.13). For each example the standard deviation of the size of M_i is given, as is the average message distance. Test cases are cells from a parallel to serial converter and a cache controller.

utilizing the property $(a \cdot 2^w + b) \bmod p = ((((2^w \bmod p) \cdot (a \bmod p)) \bmod p) + (b \bmod p)) \bmod p$. Thus the modulo computation can be performed with w-bit words for any positive w. This hash function distributes states more uniformly than XOR folding and often produces shorter message distances than Pearson's method. This is due to the fact that changes in the low order bits in `state` are often, although not always, reflected in changes in the low order bits of the hash function result. Of course, this hash method depends on the ordering of the bits in the state, but on average it performs well.

Table 1 gives some data on the relative efficiency of the above schemes. In general, Pearson's hash function appears to be most useful for systems with limited memory (where uniform distribution of states is extremely important) or communication architectures where message locality is relatively unimportant. XOR folding seems most useful for hypercubes, where the relative inefficiency of the distribution is offset by the communication locality. Prime hashing seems to cover most systems in between.

4 Examples

We begin with a simple example illustrating the verification of a correct production rule set. The following set of production rules describes a simple oscillating circuit.

$$
\begin{array}{lll}
(1) & a & \rightarrow & b\uparrow \\
(2) & a & \rightarrow & c\uparrow \\
(3) & b \wedge c & \rightarrow & a\downarrow \\
(4) & \neg a & \rightarrow & b\downarrow \\
(5) & \neg a & \rightarrow & c\downarrow \\
(6) & \neg b \wedge \neg c & \rightarrow & a\uparrow
\end{array}
$$

The variables in this circuit will transition as follows: first $a\uparrow$, followed by both $b\uparrow$ and $c\uparrow$ in any order, followed by $a\downarrow$, followed by $b\downarrow$ and $c\downarrow$ in any order. Let us step through the sequential verification algorithm for this circuit.

For any circuit that does not have explicit reset production rules (i.e., rules involving a variable named "Reset") we assume that its initial state has all variables low. We write states as vectors; for example, $\langle 0, 1, 0 \rangle$ corresponds to the state with a low, b high, and c low. Our initial state is therefore $\langle 0, 0, 0 \rangle$. We place $\langle 0, 0, 0 \rangle$ into set R and begin the first **while** loop.

We choose and remove the state $\langle 0, 0, 0 \rangle$ from R. Production rules (4), (5) and (6) have true guards in this state and are therefore enabled. We let $E = \{(4), (5), (6)\}$. In this state, b is already low and c is already low, so production rules (4) and (5) are vacuous. Thus we let $E' = \{(6)\}$. There is no pair of production rules in E that calls for up and down transitions on the same variable, so there is no interference in this state. We enter the **for** loop and choose (6) in E'. Firing (6) in state $\langle 0, 0, 0 \rangle$ will lead to $\langle 1, 0, 0 \rangle$, so we set $S' = \langle 1, 0, 0 \rangle$. All production rules in E' are still enabled in S', thus there is no instability in the transition from $\langle 0, 0, 0 \rangle$ to $\langle 1, 0, 0 \rangle$. S' is not in R or M, so it is added to R. We then exit the **for** loop, add S to M and continue.

The diagram in Figure 6 shows the reachable state graph that will be explored by this algorithm. In no state is there a pair of production rules in E that call for up and down transitions on the same variable (e.g., (1) and (4) are never both in E). This implies that in no reachable state are two such production rules enabled, and the production rule set is therefore noninterfering.

Also note that for every state, each production rule in E' is contained in E for the neighboring state. This implies that if a production rule becomes enabled and is effective, then it will not be disabled until after firing. Thus the production rule set is stable as well. □

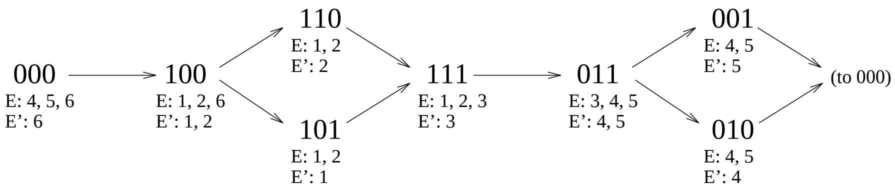


Figure 6: State graph explored by verification algorithm.

Let us now examine an incorrect production rule set. The following production rule set contains an unstable production rule.

$$
\begin{array}{lllll}
(1) & a & \rightarrow & b\uparrow \\
(2) & b & \rightarrow & a\downarrow \\
(3) & \neg a & \rightarrow & b\downarrow \\
(4) & \neg b & \rightarrow & a\uparrow \\
(5) & a \wedge b & \rightarrow & c\uparrow
\end{array}
$$

We begin, as above, with the initial state $\langle 0,0,0\rangle$ in set R. In this state production rules (3) and (4) are enabled and are placed into E. These two are obviously noninterfering, and we let E' be the effective production rules in E. Thus $E' = \{(4)\}$. The result of (4) in $\langle 0,0,0\rangle$ is $\langle 1,0,0\rangle$ so we add this state to R and $\langle 0,0,0\rangle$ to M.

In state $\langle 1,0,0\rangle$ production rules (1) and (4) are enabled and noninterfering. Only (1) is effective, and the result of (1) in $\langle 1,0,0\rangle$ is $\langle 1,1,0\rangle$. We add this state to R, add $\langle 1,0,0\rangle$ to M and continue.

In state $\langle 1,1,0\rangle$ production rules (1), (2), and (4) are enabled and noninterfering. Of these (2) and (4) are effective, so they are placed into E'. The result of (2) in $\langle 1,1,0\rangle$ is $\langle 0,1,0\rangle$. However, in $\langle 0,1,0\rangle$ production rule (4) is disabled. Thus it is possible for (4) to become effectively enabled and then disabled without firing. Since (4) is in E', the algorithm reports (4) as unstable, and therefore detects this problem.

5 Implementation

5.1 Efficiency Issues

As shown above, both stability and noninterference checking are NP-hard problems. Since the verification algorithm searches all reachable states for a set of production rules, it can be very slow (run time exponential in the number of circuit variables) in the worst case. In practice, however, circuits reach only a tiny fraction of their state space. Furthermore, there are several implementation "tricks" that can speed the run time of this algorithm.

The first and most important implementation issue is the choice of data structures for R and M. As the algorithm is written we will need to perform insertion, deletion, retrieval, and membership (for union) operations on R and insertion and membership on M. The membership operation must be particularly efficient, as membership in M is tested for every S', and membership in R is tested in the union (let $R = R \cup \{S'\}$) for those S' not in M. Fortunately, we can avoid many of these membership tests by implementing the sets R and $R \cup M$ instead of R and M. Since each S we remove from R is eventually added to M, we need not remove states from $R \cup M$. Furthermore, the final if statement requires only one membership check—if the state S' is not in $R \cup M$ it is added to both R and $R \cup M$. Thus we never need to check for membership in R and we can implement it efficiently with a FIFO queue. We will frequently check for membership in $R \cup M$, however, so we implement it with a hash table. All set operations can therefore be performed in $O(1)$ time on average.

The choice of data structure for the production rule set is also important. One good way to do this (due to Steve Burns) is to build an expression tree for each production rule's guard, include the transition at the root of the tree, and include only one copy of each variable in the forest of production rules. Thus each variable has a list of uplinks leading to expressions involving it and a list of downlinks leading to all transitions on it. Sets of enabled production rules can be stored as lists of pointers into this data structure. Expressions can be evaluated by associating a value with each node. When a production rule is fired, the expressions affected by the firing can be updated by traversing the uplinks of the variable to which the production rule assigned.

It is also not necessary to implement both E and E'. If only E is implemented, the function **result** can return a flag if an ineffective production rule is passed to it. The check for instability can be performed while updating the production rule data structure—if a transition was effectively enabled and the update routine disables it then instability can be reported.

$$a \wedge b \quad \rightarrow \quad c\uparrow$$

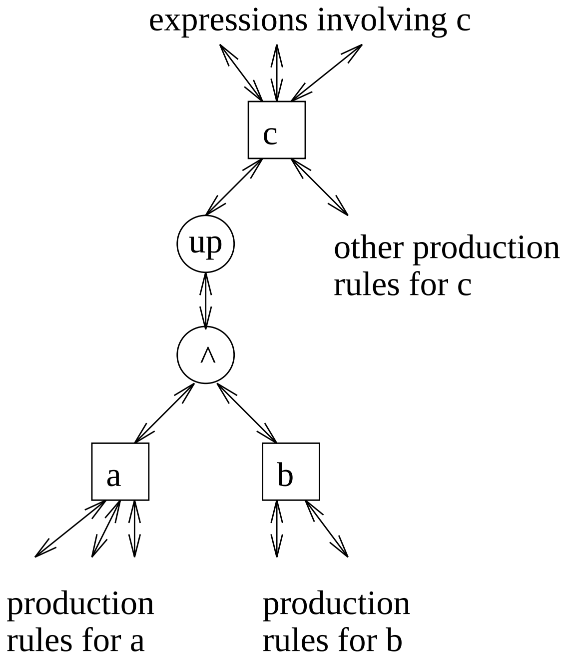


Figure 7: Proposed production rule structure.

Test Case	Variables	States	Time (sec)
P	26	190	< 1
S	23	2363	1.90
REG	18	9881	8.25
HALFCACHE	49	7176	10.02
SERIAL2	40	37590	41.40

Table 2: Run times for `prlint` on several test cases. P and S are highly sequential subcircuits in a parallel-serial converter. REG is a collection of register processes. HALFCACHE is a collection of cache controller subcircuits. SERIAL2 is a full serial-parallel converter. All times are for a SPARCstation IPX.

5.2 `prlint` Implementation

The CAST (Caltech Asynchronous Synthesis Tools) design tool package developed at the California Institute of Technology contains several programs that deal with production rules. One of these, `prsim`, is a fast and memory efficient production rule simulator. It can be used for high level simulation of large circuits described with production rules. Another suite of programs, `bubble`, `cellgen`, and `Vgladys`, transform a production rule description of a circuit into a CMOS implementation. Thus production rules can be used for description and testing of existing circuit designs as well as synthesis of new circuits. In either case, however, the production rules must be stable and noninterfering if they are to describe a correct circuit.

We have implemented the above sequential algorithm in a program called `prlint` which will be incorporated into the CAST design tool package. This program performs simple consistency checks on input production rule sets, then applies the algorithm to verify stability and interference.

The inner loop of `prlint` is given in Figure 8. We use a FIFO queue to store the states remaining to be checked and a hash table for those checked previously. The "prs" data structure contains a description of the production rule set being verified. Since applying a state to the production rule set and recomputing all the guards requires a substantial amount of computation, we use "fire_pr" and "undo_fire_pr" to make local changes to the data structure. With this method `prlint` checks approximately 1000 circuit states per second on a SPARCstation IPX.

Figure 9 shows sample output when `prlint` is run on the second example from the previous section. The initial warnings are generated by a simple syntax checker built into `prlint`; all following output is generated by the verification algorithm.

```
void simple_check(struct Prs* prs, StateVector initial)
{
   int i;
   int enabledcount;
   int result;
   StateVector state, stateprime;
   PRPtr enabled[MAX_ENABLED];
   struct Fifo* remaining;
   struct Hashtable* checked;

   remaining = create_Fifo();
   checked = createHashtable();

   put_Fifo(remaining, initial);
   fastAddHashtable(checked, initial);

   while (NULL ≠ (state = get_Fifo(remaining))) {

      apply_state(prs, state);
      enabledcount = build_enabled_list(prs, enabled);

      (void) check_interference(prs, enabled, enabledcount);
      for (i = 0; i < enabledcount; i++) {
         result = fire_pr(prs, enabled[i]);           /* fire PR and check stability */

         if (result ≠ VACUOUS && result ≠ EXCLUDED) {
            stateprime = new_state_vector(prs);
            retrieve_state(prs, stateprime);
            undo_fire_pr(prs, enabled[i]);
            if (NULL == associateHashtable(checked, stateprime)) {
               fastAddHashtable(checked, stateprime);
               put_Fifo(remaining, stateprime);
            }  /*if*/
         }  /*if*/

      }  /*for*/
   }  /*while*/

}  /*simple_check*/
```

Figure 8: Inner loop of sequential algorithm implementation.

```
C:\>prlint -printprs -printstates sample.prs

[Production Rules:]
a -> b+
~a -> b-
b & a -> c+
b -> a-
~b -> a+

Warning: Variable "c" has only one type of transition
Warning: Variable "c" is set, but not used
Warning: "Reset" variable not found.
Warning: (assuming all variables initialize to false)

Check Vars: b c a

[Run circuit...]
Checking: 000
Checking: 001
Checking: 101
Error: Unstable production rule:
        b & a -> c+
Checking: 111
Checking: 100
Checking: 110
Checking: 010
Checking: 011

[Statistics:]
Production rules: 5
Variables: 3
States Visited: 8
Possible States: 8
```

Figure 9: Sample `prlint` output.

`prlint` supports several command line options that provide more information about the circuit as it is simulated. These are documented in the on-line manual page.

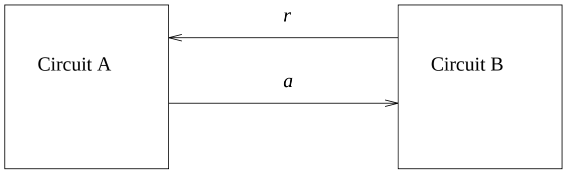
6 Conclusions

We have presented a method for the verification of quasi-delay-insensitive circuit designs. This verification requires that circuits be expressed in terms of *production rules*, which we have presented as a means to describe delay-insensitive circuits. Given this production rule description, we carry out a search of the circuit's reachable states and check for stability and interference errors, which correspond to possible shorts and hazards. We have shown how this search can be performed both sequentially and concurrently and have implemented both algorithms. We have given examples of the search method and have described our implementation of an automated verification tool.

We hope that this verification method and its implementation will be used not only to check circuits generated by Martin's synthesis method, but also as a way to verify the delay-insensitivity of other circuit designs.

A Channel Declarations

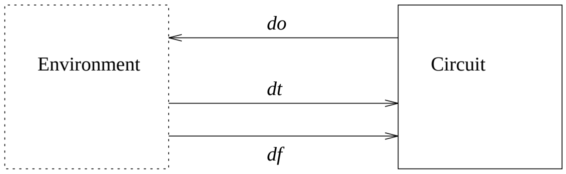
In Martin's synthesis method, parts of a circuit exchange information via *communication channels*. These channels are implemented as sets of wires that exchange data in a four-phase protocol. For example, consider two circuits that need to synchronize their operation. This can be done by performing a communication between the circuits that exchanges no data. Such communications can be implemented as follows:



Communication on this channel takes place as follows: When circuit A becomes ready to synchronize it raises wire r. When circuit B becomes ready to synchronize it waits for r to become high, then raises a in acknowledgment. A then lowers r and B lowers a. The part of the communication performed by circuit A is called the *active* part and the part performed by circuit B is called the *passive* part. Assume, then, that we only have production rules describing circuit A. In order to close the production rule set, we must specify the behavior of this channel. This can be done with the following production rules:

$$\begin{array}{lcl} r & \to & a\uparrow \\ \neg r & \to & a\downarrow \end{array}$$

In general, however, all behaviors of channels leading to a circuit's environment cannot be described solely with production rules. For example, consider a circuit with a one-bit input channel connected to the environment. Such a circuit would be implemented as follows:



Communication on this channel could be either *passive* or *active*. In a passive communication, the circuit would wait for either dt or df to be set by the environment.

Once it received this information it would raise d_o in acknowledgment. Then the environment would lower both dt and df, after which the circuit would lower d_o. In an active communication, however, the circuit would initiate the communication by raising d_o as a request. The environment would respond by raising either dt or df, which would be acknowledged by the circuit lowering d_o. The environment would complete the communication by lowering dt and df.

The problem with describing this circuit's behavior in production rules is that there is no mechanism for raising only one of two data wires. Assume we are describing a passive communication. If we use the following production rules for the environment, then both data wires will become high.

$$
\begin{array}{lcl}
\neg d_o & \rightarrow & dt\uparrow, df\uparrow \\
d_o & \rightarrow & dt\downarrow, df\downarrow
\end{array}
$$

It is also not possible to disable one of the first two production rules after the other has fired. Attempting this solution leads to the following production rules:

$$
\begin{array}{llcl}
(1) & \neg d_o \wedge \neg df & \rightarrow & dt\uparrow \\
(2) & \neg d_o \wedge \neg dt & \rightarrow & df\uparrow \\
(3) & d_o & \rightarrow & dt\downarrow, df\downarrow
\end{array}
$$

The problem with this production rule set is that it contains an instability. When all variables are false, production rules (1) and (2) are enabled. However, the firing of (1) disables (2) and vice versa, leading to an instability.

We have devised a notation called *port declarations* that compactly describes both synchronization and data channels. This notation is convenient for closing production rule sets, and is accepted as input in `prlint`. The syntax for these declarations is based on that of channel declarations in the CAST program `prif` [11]; these declarations can be copied into files that will be used with `prlint`.

The syntax for a port declaration is as follows:

⟨port type⟩ `port ( `⟨input list⟩` ; `⟨output list⟩` )`

A port type is one of `active` or `passive`. An input list is a comma separated list of variables representing the wires in the channel that are inputs to the circuit we are describing. An output list is a comma separated list of variables representing the output wires in this channel. During verification, at most one of the variables in the input list will be true at once. Further, it is an error for the circuit to set more than one output variable true simultaneously. Thus, a port declaration for the above passive communication would be:

```
passive port ( dt, df; do )
```

Port declarations can be thought of as macros that expand to lists of production rules (possibly involving an error flag) and lists of variables that need to be kept mutually exclusive. The above example would expand to:

$$
\begin{array}{lll}
d_o & \rightarrow & dt\downarrow, df\downarrow \\
\neg d_o & \rightarrow & dt\uparrow, df\uparrow \\
\text{excl(dt, df)} & &
\end{array}
$$

Thus the verifier will check states reached by firing $dt\uparrow$ and also by firing $df\uparrow$, but will be prevented from checking states reached by firing both $dt\uparrow$ and $df\uparrow$ by the "excl" statement. In `prlint` these mutual exclusion statements are handled by the "fire_pr" function called from the main checking loop. As can be seen in Figure 8 this function returns "EXCLUDED" if a production rule is prevented from firing by such a statement.

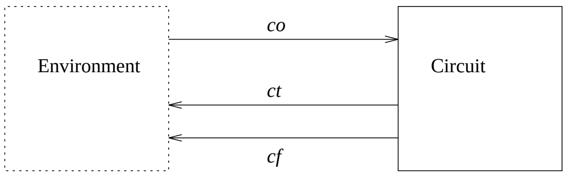
If instead we were describing an active communication we would use the following port declaration:

```
active port ( dt, df; do )
```

which would correspond to the following production rules

$$
\begin{array}{lll}
d_o & \rightarrow & dt\uparrow, df\uparrow \\
\neg d_o & \rightarrow & dt\downarrow, df\downarrow \\
\text{excl(dt, df)} & &
\end{array}
$$

Note that this example has essentially the same functionality as the one above—the circuit receives a single bit input from the environment. The change from passive to active requires the circuit to raise d_o before the environment responds; this is reflected in the reversal of the direction of the dt and df transitions.



For a final example, let us consider an active communication where a circuit sends a data bit to the environment. Again, the port declaration for this channel is simple:

```
active port ( ci; ct, cf )
```

This port declaration corresponds to the following production rules:

$$
\begin{array}{lcl}
\neg ct \wedge \neg cf & \rightarrow & c_i\downarrow \\
ct \vee cf & \rightarrow & c_i\uparrow \\
ct \wedge cf & \rightarrow & error\uparrow
\end{array}
$$

In this case the environment waits for one of ct, cf to become true before acknowledging with $c_i\uparrow$. If possible we would like to ensure that the input circuit does not raise both ct and cf; this would be a violation of the dual rail communication protocol. This is accomplished with the $ct \wedge cf \rightarrow error\uparrow$ pseudo-production rule. This production rule is treated by `prlint` like any other, but if the variable *error* becomes true then a special error is generated.

As shown above, port declarations give a compact description of communication between a circuit and its environment. Although most useful for specifying data channels, they may also be used to describe the synchronization channels previously mentioned. Furthermore, port declarations generalize easily to an arbitrary number of inputs and outputs. We have found the notation convenient for closing sets of production rules for use with `prlint` and believe that they present a useful extension to the production rule notation.

References

[1] Shahid H. Bokhari. On the mapping problem. *IEEE Tran. Computers*, 30(3):207–213, March 1981.

[2] Steven M. Burns. *Performance Analysis and Optimization of Asynchronous Circuits*. PhD thesis, California Institute of Technology, 1991.

[3] David L. Dill and Edmund M. Clarke. Automatic verification of asynchronous circuits using temporal logic. In Henry Fuchs, editor, *1985 Chapel Hill Conference on VLSI*, pages 127–143. Computer Science Press, 1985.

[4] C.A.R. Hoare. Communicating sequential processes. *Communications of the ACM*, 21(8):666–677, August 1978.

[5] Donald E. Knuth. *The Art of Computer Programming: Searching and Sorting*, volume 3. Addison-Wesley, 1981. Section 6.4.

[6] Ten-Hwang Lai and Alan P. Sprague. Placement of the processors of a hypercube. *IEEE Tran. Computers*, 40(6):714–722, June 1991.

[7] F. Thomson Leighton. *Introduction to Parallel Algorithms and Architectures: Arrays, Trees, Hypercubes*. Morgan Kaufmann Publishers, 1992. Sections 3.2 and 3.3.

[8] Alain J. Martin. Compiling communicating processes into delay-insensitive VLSI circuits. *Distributed Computing*, 1(4):226–234, 1986.

[9] Alain J. Martin. The limitations to delay-insensitivity in asynchronous circuits. In William J. Dally, editor, *Sixth MIT Conference on Advanced Research in VLSI*, pages 263–278. MIT Press, 1990.

[10] Alain J. Martin. Programming in VLSI: From communicating processes to delay-insensitive circuits. In C. A. R. Hoare, editor, *Developments in Concurrency and Communication*. Addison-Wesley, 1990. UT Year of Programming Institute on Concurrent Programming.

[11] Alain J. Martin et al. CAD tools for VLSI design. Report CS-TR-93-09, California Institute of Technology, 1993.

[12] Peter K. Pearson. Fast hashing of variable-length text strings. *Communications of the ACM*, 33(6):677–680, June 1990.

[13] Charles L. Seitz. The Cosmic Cube. *Communications of the ACM*, 28(1):22–33, January 1985.

[14] Jan L. A. van de Snepscheut. *Trace Theory and VLSI Design*, volume 200 of *Lecture Notes in Computer Science*. Springer-Verlag, 1985.