Stable Production Rule Sets are Deterministic

Karl Papadantonakis

Computer Science Department
California Institute of Technology

caltechCSTR/2003.003

Stable Production Rule Sets are Deterministic

Karl Papadantonakis
5/10/2003

Introduction

Production Rule Sets (PRS) [AM] are a digital event-based model for CMOS circuits. A priori, the behaviors of CMOS circuits are described by differential equations, so one might ask if any results from basic DE theory can be stated in the digital model. Basic DE theory gives us two results that might be useful for analysing CMOS circuits: the existence theorem, and the uniqueness theorem. While the existence theorem certainly has an analogy in the digital model, the uniqueness theorem has two problems. First, there is noise, so one must use the Fundamental Equality [JHH] directly, instead of the uniqueness theorem. Second, while the fundamental inequality can be used to understand certain phenomena (such as metastability, which we will discuss in the last section), it inherently can only characterize behavior for a short time in the future (its strength decays exponentially), which is not good enough for a circuit designer, who wants to control (some aspect of) a circuit's behavior for all time.

Fortunately, the following statements about production rule sets have been conjectured:

- "stable PRS is deterministic; therefore it cannot implement an arbiter"
- "stable PRS is deterministic; therefore the sequence of values sent on any channel is fixed"

These statements are of practical importance: the first tells us not to waste time trying to design an arbiter using stable PRS, and the second tells a circuit designer that if his PRS is stable, then for each input sequence, he need only consider a particular execution in order to argue its correctness.

To give meaning to these statements, one must define "deterministic". The term "deterministic" for PRS is introduced but not defined in [MM2]. I believe that a proper definition should make the above two statements true, while also giving exactly the property of an arbitrary execution of a given production rule set which is "determined".

Prior Work

Though my proofs are self-contained, the notion of "stability" for PRS (as defined by Alain Martin) has been described as "strong confluence" [MM2], and the relation between strong confluence and determinism (in the context of event-rule systems) has been suggested as early as 1980 [RMil]. Strong confluence has been proven to hold for stable CSP [SZ] and for stable PRS [MM2], but the definition of determinism in the former paper is rather complicated (it relies on being able to partition the system into testable components) and might apply only to CSP and not PRS, and I do not feel that the latter paper rigorously explores the consequences of strong confluence (it does not even give a definition of determinism). However, when discussing the "determinism lemma" part of my proof with Mani Chandy, he suggested that the notion of getting from one execution to another through local interchanges is standard. Indeed, this idea is used in [SZ] and elsewhere.

Regardless of whether my definitions are consequences of previous ones, I believe that: 1) my paper is the first to rigorously explore the consequences of stability for PRS, including the first to define determinism in this context, and 2) my paper is the first to prove, in a matter of two pages, without referring to difficult theorems from other texts, that the properties conjectured above to be determined are in fact determined.

Regarding the second comment, there exists previous work (under Alain Martin) which attempts to distinguish the "computation" from the "execution" (the computation is a class of executions). Some of the "determined properties" we will discuss may be determined in XER-systems, [TL] and it may be that all stable PRS can be modelled as XER-systems. In fact, the "XER-system" may itself be a "determined property", in the sense we will define. However, all of our results about determinism will be obtained more directly (without reformulating infinite executions into periodic ones, and without referring to a construct such as an XER-system. Also note that in our treatment, we do not have to make a special case for deterministic execution spaces that are not partial orders). Also we will explore the applicability of our results to arbitrary (not necessarily stable) PRS.

Background

Production Rule Sets

A *production rule set* (PRS) [AM] is a digital model of the dynamic behavior of a network of CMOS transistors. It is an ordered tuple $(\mathcal{N}, \mathcal{R}, s_0)$ where $\mathcal{N}$ is a finite set of *nodes*, $s_0 : \mathcal{N} \to \{0,1\}$, is an arbitrary *initial state*, and $\mathcal{R}$ is a set of *rules*. For each node $t \in \mathcal{N}$, there is exactly one pair of rules (with *target* t), written:

$$g_{t+}(n_1, \ldots, n_k) \to t\uparrow$$
$$g_{t-}(n_1, \ldots, n_k) \to t\downarrow$$

This pair is sometimes referred to as the *gate* g_t.

The *guards* g_{t+} and g_{t-} are boolean-valued functions of boolean node values. For example, we can talk about "the boolean value of $g_{t+}(n_1, \ldots, n_k)$" if we assign boolean values to $n_1, \ldots, n_k$. Also, given a global state s, i.e. any function $s : \mathcal{N} \to \{0,1\}$, we can evaluate $g_{t+}(s)$ and $g_{t-}(s)$. If the rules of a gate are complementary, i.e. if $g_{t+}(s) = \neg g_{t-}(s)$ for all s, then we say the gate is *static*.

Note: in some contexts, it is convenient to skip writing a production rule if the guard is never true, so that a target can have only one rule (e.g. there is a rule for $t\uparrow$ without a rule for $t\downarrow$). Also, it is sometimes convenient to write rules of the form $a \vee b \to x\downarrow$ as two separate rules. For clarity, however, we will not allow such notation in this paper; hence rules always occur in pairs: two for each target, and every node $t \in \mathcal{N}$ is the target of some pair of rules. This means that a rule is identified by its target node and direction. E.g. if t is a node, then $t\uparrow$ identifies a rule.

Enabled Rules

A state s *enables* a rule $g(n_1, \ldots, n_k) \to t\uparrow$ if $g(s) = 1$ and $s(t) = 0$. Similarly $g(n_1, \ldots, n_k) \to t\downarrow$ is enabled if $g(s) = 1$ and $s(t) = 1$.

Note: in some contexts, one considers a rule "enabled but vacuous" if the guard holds but the target already has the value to be assigned ($t\uparrow$ *assigns* 1; $t\downarrow$ *assigns* 0). However, in this paper we will not consider such rules to be enabled, because then the class of stable production rule sets would be trivial.

Rule and State Sequences

Given a production rule set $(\mathcal{N}, \mathcal{R}, s_0)$ and any (possibly infinite) sequence of rules $r_1, r_2, \ldots$, we will define the *sequence of states* $s_0, s_1, \ldots$. We define the *state* s_{i-1} *before rule* r_i, by induction: the state before rule r_1 is s_0. The state s_{i+1} (the state before rule r_{i+1}, also known as the state after rule r_i) is the same as s_i, except that t is changed to 0 if $r_i = t\downarrow$, or 1 if $r_i = t\uparrow$.

Executions

A rule sequence is an *execution* iff

1. Only enabled rules execute: For all $i \geq 1$, the state before r_i (i.e., s_{i-1}) enables r_i, and
2. If s_i enables a rule, then some state after s_i does not enable the rule (traditionally, such a rule must execute eventually, but our definition applies to both stable and unstable PRS).

There is an *existence theorem*: any (possibly empty) finite rule sequence satisfying (1) can be extended to an execution (note: we required $\mathcal{N}$ to be finite for this).

Noninterference

An execution is *noninterfering* if for every node t, none of the execution's states simultaneously satisfies the guards $t\uparrow$ and $t\downarrow$. A production rule set is noninterfering if all its executions are. Only noninterfering PRS is implementable in CMOS. However noninterference will not be necessary to prove the results of this paper.

Stability

An execution is *stable* if for each rule $r = g(\ldots) \to t\ldots$, and for each state s_i in which r is enabled, r remains enabled until it is executed. For example, if r assigns 0, then $s_i(t) \wedge g(s_i) = s_{i+1}(t) \wedge g(s_{i+1}) = \cdots = s_j(t) \wedge g(s_j) = 1$, where s_j is the first state after s_i with $r_{j+1} = r$. A production rule set is stable if all its executions are. Martin Synthesis [AM] constructs stable production rule sets (unless there are arbiters; see section on arbiters below).

Determinism

Given a stable production rule set and a fixed *reference execution* E_0, we will show that the following property is determined for any execution E: there exists a *path* from E_0 to E. Intuitively, we can get from E_0 to E through a (possibly infinite) sequence of small steps, without ever leaving the space of executions. In some sense, this is saying that the space of executions is *connected*.

Prefix Equality

If two (rule) sequences S and S' have the same first n terms (rules), then we write $S \stackrel{n}{=} S'$. Notice that for two rule sequences with $S \stackrel{n}{=} S'$, we have $s(S)_n = s(S')_n$. I.e., both sequences must arrive in the same state after performing the same n steps from the same initial state.

Path

For any PRS (not necessarily stable), and any executions E_0 and E, a (possibly infinite) sequence of executions $E_0, E_1, \ldots$ is a *path* from E_0 to E (denoted $E_0 \rightharpoonup E$) iff:

1. Convergence: For any $n \leq |E|$, there exists j such that $E_j \stackrel{n}{=} E_{j+1} \stackrel{n}{=} \cdots \stackrel{n}{=} E$. (Notice that a finite path must end at E)
2. Small steps: For all $j \geq 0$, E_j is the same as E_{j+1}, except for a single interchange of two adjacent terms (i.e. two rules executed in a different order).

The Determinism Lemma

Consider any execution E of a stable production rule set $R = (\mathcal{N}, \mathcal{R}, s_0)$ with reference execution E_0. We will construct a path $E_0 \rightharpoonup E$ by induction on n, for $0 \le n \le |E|$: given a finite path $E_0 \rightharpoonup E_k$ with $E_k \overset{n}{=} E$, we extend it to a finite path $E_0 \rightharpoonup E_{k'}$ with $E_{k'} \overset{n+1}{=} E$. (This results in a single sequence whose tail terms agree with E in the first n terms, for any n, hence satisfying property 1 of the path).

Base case: The singleton sequence $\{E_0\}$ is an "empty path" $E_0 \rightharpoonup E_0$ (property 1 holds because all terms equal E_0, and property 2 holds because there are no pairs to check). Also $E_0 \overset{0}{=} E$ holds, vacuously.

Inductive step: Consider a finite path $E_0 \rightharpoonup E_k$ with $E_k \overset{n}{=} E$. Since $s(E_k)_n = s(E)_n$ and E is an execution, rule $r(E)_{n+1}$ is enabled by $s(E_k)_n$. Furthermore, E_k is a stable execution (because all executions of R are stable) so states $s(E_k)_n \ldots s(E_k)_{n'}$ must enable $r(E)_{n+1}$, where n' is the smallest index with $r(E_k)_{n'+1} = r(E)_{n+1}$.

Notice that if $n' = n$, then we are done, because then $E_k \overset{n+1}{=} E$. This will serve as the base case for a secondary induction, in which we extend the path one step at a time, decreasing n' (bringing $r(E_k)_{n'+1}$ one step earlier) at each inductive step:

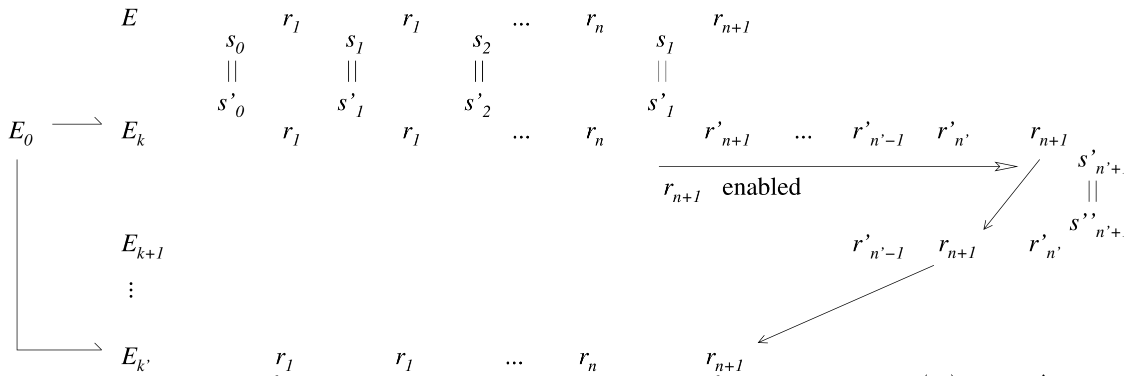

Figure 1. The path from E_k to $E_{k'}$. Note: r is shorthand for the sequence $r(E)$, and r' is shorthand for $r(E_k)$, etc.

Now suppose $n' > n$. Let E_{k+1} be the rule sequence which is identical to E_k, except that $r(E_k)_{n'+1}$ is interchanged with $r(E_k)_{n'}$. We claim that E_{k+1} is an execution.

First, notice that since $E_k \overset{n'-1}{=} E_{k+1}$, all rules before $r(E_{k+1})_{n'}$ are enabled before being executed. Second, we have already noted that states $s(E_k)_{n'}$ and $s(E_k)_{n'-1}$ enable $r(E_k)_{n'+1}$. Therefore $s(E_{k+1})_{n'-1} = s(E_k)_{n'-1}$ enables both $r(E_{k+1})_{n'+1} = r(E_k)_{n'}$ and $r(E_{k+1})_{n'} = r(E_k)_{n'+1}$. Executing $r(E_{k+1})_{n'}$ next cannot disable $r(E_{k+1})_{n'+1}$, for then we could construct an unstable execution (by the existence theorem). Therefore both rules are enabled before being executed. Notice that (by our definition of enabled) two rules that are simultaneously enabled cannot have the same target node. Therefore the result of executing the two rules does not depend on the order in which they are executed. Hence $s(E_{k+1})_{n'+1} = s(E_k)_{n'+1}$, $s(E_{k+1})_{n'+2} = s(E_k)_{n'+2}$, etc., so all rules after $r(E_{k+1})_{n'+1}$ are enabled before being executed.

Finally, we observe that execution property (2) holds: any enabled rule is eventually disabled, because both executions have the same tail. $\square$

The Permutation Lemma

We now claim that for any path $E_0 \rightharpoonup E$ constructed by the determinism lemma, all executions on this path are permutations of E. Each step of the path is itself a permutation, so it suffices to show that E_0 is a permutation of E.

Starting with $n = 0$, the construction finds the first occurence of $r(E)_n$ somewhere in E_0 and moves it to the front, building a prefix of E. Similarly, all other action occurences in E appear somewhere in E_0. This index map is therefore injective. It is also surjective: by swapping E_0 and E in the determinism lemma, we find that each rule has the same number of occurences in both executions. Therefore occurence k of any rule in E must be the image of occurence k of that rule in E_0. $\square$

Consequences of the Determinism Lemma

A property of sequences is said to be *continuous* if, when holding for any sequence of sequences that converge (see *path* definition, above) to a sequence S, it must also hold for S itself. Unfortunately, many useful determined properties will not be continuous, because they will not be invariant under *indefinite* postponement of actions. So we use a weaker definition.

A property of sequences is said to be *semi-continuous* if, when holding for any sequence of sequences converging to S *in which all terms of the sequence are permutations of* S, it must also hold for S itself. For example, the property counting the number of 'y's in a sequence is semi-continuous, but not continuous, since it does not hold for the sequence of sequences $\{y\}, \{x, y\}, \{x, x, y\}, \ldots$, which converges to $x, x, x, \ldots$.

All Semi-Continuous, Interchange-Invariant Properties are Determined

Determinism Theorem. *For any stable production rule set R, any semi-continuous property P of executions of R which is preserved by single interchanges will be determined. I.e., any such P has a unique value for all executions of R.*

Proof: Choose E_0 canonically using the existence theorem. We claim that $P(E) = P(E_0)$ for any execution E. By the determinism lemma, there is a path $E_0, E_1, \ldots$ converging to E. Since each step in this path is a single interchange, $P(E_i) = P(E_0)$ holds for all i, by induction. By the permutation lemma, all E_i are permutations of E. Since P is semi-continuous, $P(E) = P(E_0)$ must also hold. □

Transition Counts of All Nodes are Determined

As a simple example, let P_t denote the number (possibly ∞) of times that a node switches (i.e. the number of times that a rule with target t is executed). Clearly P_t is interchange-invariant. P_t is also semi-continuous: in fact, if $P_t(S) = n$ holds for any S then it will hold for all permutations of S.

As an even simpler example, the property $P_{|\cdot|}$, measuring the length of an execution, is determined for stable production rule sets. $P_{|\cdot|}$ is in fact continuous, and clearly interchange-invariant. The length of an execution is a first-order approximation to the energy consumed by a circuit implementation; thus the energy consumed on any execution is the same, to first order.

Handshake Sequences are Determined

Since "Deterministic CHP" [KP2] can be compiled [AM] into stable PRS, one would expect the handshake sequences corresponding to the determined value sequences in such systems to be themselves determined. In fact, all we need to assume is stability of the PRS.

Given any production rule set $(\mathcal{N}, \mathcal{R}, s_0)$, a *handshake on* (L, R) is any pair of node sets (L, R) such that $(L, R, \mathcal{N}-L-R)$ is a partition of $\mathcal{N}$, and the projection onto $L \cup R$ of *every* execution E can be written as a sequence of *handshake phases*

$$L_1\uparrow, R_1\uparrow, L_1\downarrow, R_1\downarrow, L_2\uparrow, R_2\uparrow, L_2\downarrow, R_2\downarrow, L_3\uparrow, \ldots \tag{2}$$

where the $L_i \updownarrow$ and $R_i \updownarrow$ are *nonempty* sequences of rules (which may depend on E) such that all rules in $L_i\uparrow$ assign some node in L to **true**, all rules in $L_i\downarrow$ assign some node in L to **false**, and similarly for $R_i \updownarrow$.

For a particular execution E, the *handshake sequence on* (L, R), denoted $P_{(L,R)}$, is the sequence $L_1\uparrow, R_1\uparrow, L_1\downarrow, R_1\downarrow, \ldots$ viewed as a sequence of sets. I.e. permutations within any individual handshake phase are ignored.

Suppose the production rule set is stable. We now claim that $P \stackrel{\text{def}}{=} P_{(L,R)}$ is determined. First, P is semi-continuous: consider any E and any sequence $E_0, E_1, \ldots$ of permutations of E converging to E, and such that $P(E_0) = P(E_i)$ for all i. We will show that $P(E) = P(E_0)$ by showing $P(E) \stackrel{j}{=} P(E_0)$ for all j. Consider any nonnegative $j \leq |P(E)|$. After n steps (for some n), E has performed j handshake steps, i.e., $|P(E[1..n])| = j$. For some k, $E_k \stackrel{n}{=} E$. Hence $P(E) \stackrel{j}{=} P(E_k) = P(E_0)$. Technically (since we only considered $j \leq |P(E)|$) we must also show that $P(E_0)$ is no longer than $P(E)$; this follows from the fact that E_0 is a permutation of E.

Finally, P is interchange-invariant. Consider any execution E, and any single interchange resulting in execution E'. If the interchange involves any node in $\mathcal{N} - L - R$, then E and E' have the same projection onto $L \cup R$, so clearly $P(E) = P(E')$. If both nodes are part of the same phase, then $P(E) = P(E')$ because the sets of actions in this phase are the same. So the interchanged nodes must be members of adjacent phases, and interchanging them produces an illegal phase sequence – violating the assumptions that E' is an execution and (L, R) is a handshake – so this never happens. Thus $P(E) = P(E')$ always.

Interchange-Invariant Properties are Not Necessarily Determined

Finally, here is an example demonstrating why we require the property to be semi-continuous. Let $\overline{P_\infty}$ be the property which is true for all sequences that can be permuted to $a\!\uparrow, a\!\downarrow, b\!\uparrow, b\!\downarrow, a\!\uparrow, a\!\downarrow \ldots$ using only a finite number of interchanges. Clearly $\overline{P_\infty}$ is interchange-invariant, and $\overline{P_\infty}$ is indeed satisfied by some executions of the system consisting of two independent self-occluding inverters with respective nodes a and b. But it is not satisfied by all executions of that system, for example by this one: $b\!\uparrow, b\!\downarrow, a\!\uparrow, a\!\downarrow, b\!\uparrow, b\!\downarrow \ldots$.

Arbiters

Now we will try to understand (using both analog and digital arguments) why stable PRS cannot implement an arbiter specification. An *arbiter* is any PRS fragment implementing the following HSE specification [AM]:

$$
\begin{array}{l}
*[[a \longrightarrow ap\!\uparrow; [\neg a]; ap\!\downarrow \\
\;\;\mid b \longrightarrow bp\!\uparrow; [\neg b]; bp\!\downarrow \\
\;]]
\end{array}
$$

For the benefit of readers who are not familiar with HSE, here is the above specification in plain language: From the initial state, (which has all interface nodes a,b,ap,bp low) the *environment* can raise a and/or b at any time (possibly "at the same time"); the arbiter must eventually choose one input which went high, and raise the corresponding output, and complete a handshake on that output, before starting again.

Proof 1, using the Metastability Theorem

This proof relies on the conjecture that a CMOS circuit implementing a stable PRS executes each rule in constant time (or less). This is only a conjecture simply because the theory of modelling analog circuits using PRS is not fully developed. But the argument is simple: if the guard of a rule is stable, then while the target is switching, the pulling network remains on, always suppling at least current i, thereby charging the output capacitance C in at most time C/i.

Observe that the arbiter must always take a finite number of steps in order to make a decision, for otherwise it would repeat a state, and we could construct an execution that never makes a decision (violating the arbiter specification). Combining this with the delay conjecture, we obtain an analog circuit implementing an arbiter with constant response time τ. Notice that if we have a rise at time τ, then we have a circuit which always produces a stable bp output by time 3τ, depending on the time at which b rises, even if b is only permitted to rise between times 0 and 2τ. This contradicts the *metastability theorem* (see appendix, below). ▫

This argument comes from the generic theory of analog circuits. However, given that we have a digital model of our circuits (namely PRS), it should be possible to prove the same fact directly about the digital model, without ever referring to analog circuit implementations. Fortunately, with the determinism theorem, this is possible:

Proof 2, using the Determinism Theorem

Suppose there is a stable PRS fragment implementing the arbiter. Consider a simple environment in which we add two unconditional rules, $a\!\uparrow$ and $b\!\uparrow$. By any reasonable definition of stability for PRS fragments, this environment/fragment combination is a stable PRS. Construct an execution starting with $a\!\uparrow$ but never doing $b\!\uparrow$; it must eventually do $ap\!\uparrow$, because this execution must be valid without the $b\!\uparrow$ rule (intuitively, the arbiter does not know about the $b\!\uparrow$ rule until it executes). This execution never does $bp\!\uparrow$, because a is never withdrawn. Similarly, there is another execution which does $bp\!\uparrow$ but not $ap\!\uparrow$. This contradicts the determinism theorem, which implies that the transition counts of all nodes are determined. ▫

The Metastability Theorem

Consider a "time-digitizing" circuit with one input x which may have a single rise (with a fixed, given, nonzero rise time and fixed, continuous shape) at any time between t_0 and t_1. The circuit also has one output y, which, *for any valid input*, after time t_y, never enters a given "illegal signal" range I (a fixed, given, open voltage interval that must exclude and separate the "high" and "low" levels). Furthermore, if the input arrives at time t_0, the output is "high" at time t_y, while if the input arrives at time t_1, the output is "low" at time t_y:

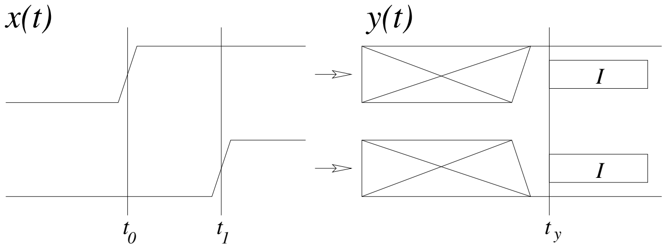

Figure 3. Behavior of the time-digitizing circuit.

Using the fundamental inequality, we will now show that such a circuit does not exist.

Let x_0 be the input signal (of the given rise shape) arriving at the time which is the least upper bound of all arrivial times that make the final output high. Notice that this time is arbitrarily close to arrival times that make the final output different: for any ϵ, there is some x_ϵ whose rise time is ϵ different from x_0, which makes the output different. Also, since the input shape is continuous, for any ε, there is some legal x_ε such that $|x_0(t) - x_\varepsilon(t)| < \varepsilon$ for all t, while x_0 and x_ε cause outputs that differ by $|I|$ at time t_y. By the fundamental inequality [JHH], there exists a *Lipschitz constant* K (essentially an upper bound on all transconductance constants in the circuit) such that:

$$|I| \leq |y_0(t_y) - y_\varepsilon(t_y)| \leq \left(\frac{\varepsilon}{K}\right)\left(e^{Kt_y} - 1\right) \tag{4}$$

but clearly this cannot be satisfied for all ε. □

Notice that, because of the exponential term, a circuit might be able to (and can, in careful designs) distinguish extremely small differences in input arrival times if t_y is only slightly increased. However, for any t_y, there will always be at least some small nonempty region of arrival times that cannot be handled. This is why synchronous circuits that have to respond to asynchronous events always have a nonzero probability of failure.

Deterministic Classes of Executions of Unstable PRS

Often, a PRS is designed from both stable and unstable components. The system can be analysed as a stable system, if each *deterministic class* of executions of the unstable components is individually considered. We will now define the deterministic classes of executions of arbitrary PRS, prove their determinism, and show that for fixed behavior of the unstable components, all executions fall in a single class.

A *strong path* $E \rightharpoonup F$ is any path such that all executions on the path are stable *and* E is a permutation of F. For arbitrary PRS, we define the relation "$\rightleftharpoons$" on its executions, as follows: $E \rightleftharpoons F$ iff there exist strong paths $E \rightharpoonup F$ and $F \rightharpoonup E$.

Lemma. *$\rightleftharpoons$ is an equivalence relation.*

Proof: $\rightleftharpoons$ is obviously symmetric, and $\rightleftharpoons$ is reflexive, as there is an empty path starting from any execution, which is a permutation of itself.

$\rightleftharpoons$ is transitive: Consider two paths $E_0 \rightharpoonup F_0$ and $F_0 \rightharpoonup G$. We construct a path $E_0 \rightharpoonup G$ as follows: For each term g_i of G, the path $F_0 \rightharpoonup G$ modifies, for some n, only the first n terms of F_0, leaving all subsequent rules and states in F_0 unchanged. Thus, to extend $E_0 \rightharpoonup G$ so as to fix g_i, we (1) include the part of F_0 that we need "on demand" (by including enough of $E_0 \rightharpoonup F_0$ so that $E_0 \rightharpoonup F_0$ will never change the first n terms again), and (2) include one step of $F_0 \rightharpoonup G$. Notice that at any point in this construction we can undo all steps from phase (2) and obtain some term (which, by assumption, is a stable execution) in $E_0 \rightharpoonup F_0$. Therefore all terms on our path are stable executions. The reverse path is similar. And clearly E_0 is a permutation of G, since both are permutations of F_0. □

Generalized Determinism Theorem. *For any production rule set, with execution space $\mathcal{E}$, any semi-continuous, interchange-invariant property P is well-defined on $\mathcal{E}/\rightleftharpoons$, i.e. its value depends only on the deterministic class of an execution.*

Proof: Consider any E and E_0 in this class. By assumption, there exists a strong path $E_0, E_1, \ldots$, converging to E, such that all E_i are stable permutations of E. Since each step is a single interchange, $P(E_i) = P(E_0)$ holds for all i, by induction. Since P is semi-continuous, $P(E) = P(E_0)$ must also hold. □

Finally, consider a mixed system in which some components are stable and others are unstable. One way to "fix" the behavior of the unstable components by only considering stable executions in which all nodes of each unstable component switch in a determined sequence. Now consider any two stable executions in which the nodes of all unstable components switch in the same sequences. Attempt to construct a path from one execution to the other. This path construction will not reorder the node sequence of any unstable component, because this sequence is the same in both endpoint executions. Therefore each execution on the path will have no instabilities in the unstable components – hence each such execution will be stable. Therefore the path is a strong path, and both executions are members of the same deterministic class.

A Practical Application: The Trace Calculator

So far, we have presented several operations on executions (namely, various constructions of paths from one execution to another) that can be useful in checking the correctness of designs. Current production rule simulators (such as `cosmos`, `prsim`, `csim`, and `ksim`) all have only one data type: the *state* (conceptually, that is: there are event queues, but these are for performance only: they do not add information or enhance functionality). If this data type were replaced by the *trace* (execution), then the following new and useful operations would become possible:

- **stepping backwards**. One can easily examine any prefix of a trace. Also, since our execution model does not allow vacuous transitions, the state anywhere in the trace can be quickly determined by stepping forwards *or backwards* from any cached "keyframe states".
- **path construction**. Given any two executions, the program can begin constructing a path between them, according to the determinism lemma. If they are part of the same component, the program succeeds and returns the path. If they are not part of the same component, the program always returns a third execution with an instability. This would be useful for finding bugs that cause problems long after they occur. For example, suppose we find trace A, which behaves as the designer expects, but trace B deadlocks. Clearly (by the fact that transition counts are determined) there is an instability, but it has not been found – in fact the deadlock may be very indirectly related to the instability. With the "path construction" feature however, the program attempts to get to trace A from trace B. Notice that it will construct *as much of A as is possible* given that the trace must deadlock. The first term in the resulting trace that differs from A will be absolutely essential in order for the deadlock to occur.
- **targeted trace modification**. Current simulators that can randomly explore the execution space must construct each new execution from scratch. If the instability being sought only occurs because of an interleaving that occurs late in the execution, a new execution must still be created to test each interleaving. Instead, we can use the fact that there is always a path from the stable execution to an unstable one, to attempt to find an unstable execution by constructing a path from the base execution (and the path can be targeted to a particular time in the trace. E.g. the user could specify each interchange manually or algorithmically).
- **class membership**. The program can attempt the path construction, but return only a **true/false** answer, indicating whether the two traces are members of the same deterministic class. Of course, if the traces belong to different classes, then the program should also verify that the resulting instability is specially marked by the designer as part of an unstable component.
- **fast class membership**. To quickly guess whether two traces belong to the same class, a number of stock "determined properties" (e.g. transition counts and handshake sequences, if known) can be used as heuristics: if they have the same values for both traces, then the two traces *might* be members of the same deterministic class, but if any property has different values for the two traces, then they definitely belong to different classes.
- **intra-class testing**. Current simulators cannot be told (without manually crippling the input rules) to explore only a single determinism class; however, if one begins by choosing a class (e.g. a particular sequence of outputs from an arbiter) by specifying a reference member, one can then do "targeted trace modification" with the restriction that searching does not "cross over" the boundaries imposed by unstable traces. This would allow other properties (performance properties, and "unexpected" instabilities – i.e. unstable rules not marked by the designer as such) to be tested within individual determinism classes.
- **finding instabilities "from endpoints"**. This operation would search for traces in different classes. Before attempting a path construction for each new trace under consideration, it could discard traces unless the "fast class membership" heuristics indicate that the two traces are in different classes. Then a path construction would be guaranteed to return a trace with an instability.

Finally, notice that a trace is not an intractably large piece of data: `csim` already effectively stores it when it constructs "critical paths", and, although traces are usually not bounded, the number of terms in traces that are typically examined is usually smaller than the total number of rules in the PRS.

Bibliography

[SZ] Smith, S.F. and Zwarico, A.E. Correct Compilation of Specifications to Deterministic Asynchronous Circuits. *Formal Methods in System Design 7*, 1995, pp. 155-226.

[RMil] Robin Milner. *A Calculus of Communicating Systems*, Lecture Notes in Computer Science 92, Springer-Verlag Berlin Heidelberg, 1980.

[AL] Andrew Lines. *Pipelined Asynchronous Circuits*, Masters thesis, Caltech, revised June 1998.

[AM] Alain J. Martin. *Synthesis of Asynchronous VLSI Circuits*, Caltech Computer Science Tech Report, 1990.

[JHH] J. H. Hubbard, *Differential Equations: A Dynamical Systems Approach*, Springer-Verlag New York, 1991.

[KP] Karl Papadantonakis, *MiniMIPS Decomposition (based on that of Andrew Lines).*

[KP2] Karl Papadantonakis, *What is "Deterministic CHP", and is Slack Elasticity That Useful?*, Master's thesis, Caltech, 2002.

[TL] Tak Kwan Lee, *A General Approach to Performance Analysis and Optimization of Asynchronous Circuits*, PhD thesis, Caltech, 1995.

[MM] Rajit Manohar and Alain J. Martin. Slack Elasticity in Concurrent Computing. *Proceedings of the Fourth International Conference on the Mathematics of Program Construction*, Lecture Notes in Computer Science 1422, pp. 272-285, Springer-Verlag 1998

[MM2] Rajit Manohar and Alain J. Martin. QDI circuits are Turing-Complete.