Rewriting semantics of production rule sets

Michael Katelman a,*, Sean Keller b, José Meseguer a

a Department of Computer Science, University of Illinois at Urbana-Champaign, Urbana, IL 61801, USA
b Computer Science Department, California Institute of Technology, Pasadena, CA 91125, USA

ARTICLE INFO

Article history:
Available online 7 August 2012

Keywords:
Asynchronous digital circuits
Formal semantics
Rewriting logic semantics
Hazard freedom
Deadlock freedom
Formal verification
Quasi-delay-insensitive circuits

ABSTRACT

This paper is about the semantics of production rule sets, a language used to model asynchronous digital circuits. Two formal semantics are developed and proved equivalent: a set-theoretic semantics that improves upon an earlier effort of ours, and an executable semantics in rewriting logic. The set-theoretic semantics is especially suited to meta-level proofs about production rule sets, whereas the executable semantics can be used with existing tools to establish, automatically, desirable properties of individual circuits. Experiments involving several small circuits are detailed wherein the executable semantics and the rewriting logic tool Maude are used to automatically check two important properties: hazard and deadlock freedom. In doing so, we derive several useful optimizations that make automatic checking of these properties more tractable.

© 2012 Elsevier Inc. All rights reserved.

1. Introduction

Asynchronous digital circuits have been employed to design low-power, high-performance microprocessors, e.g., [20], as well as in emerging applications such as systems-on-chip (SOCs), e.g., [18], soft-error-tolerant systems, e.g., [9], and nanoelectronics, e.g., [19]. The critical property that makes asynchronous circuits advantageous in these applications is their enormous immunity to both intrinsic and extrinsic timing variation. The most common failure mechanism in synchronous circuits, set-up and hold time variation of latches, is simply not present in many asynchronous circuit families. At present, the major difficulty in designing asynchronous circuits is that very few commercially supported asynchronous electronic design automation (EDA) tools or standard cell libraries exist, making design and implementation of asynchronous circuits more challenging than for synchronous ones.

The present work concerns the language of production rule sets, which was introduced as part of a correct-by-construction synthesis method for asynchronous digital circuits [15]. According to this methodology, designs are first given in a high-level hardware description language called Communicating Hardware Processes (CHP). The CHP description is synthesized into a semantically equivalent hierarchical network of gates and digital switches called a production rule set. From a given production rule set, one can then straightforwardly generate an equivalent representation in a variety of circuit technologies, including CMOS (complementary metal–oxide–semiconductor).

This paper addresses two issues concerning production rule sets. The first is the fundamental question of what does a production rule set mean? To that end, we treat production rule sets as a formal language and assign to that language a semantics. One of the most important aspects addressed by our semantics, and one that will consume a substantial portion of the development, are notions of circuit failure called hazards.

* Corresponding author.
E-mail addresses: katelman@uiuc.edu (M. Katelman), sean@async.caltech.edu (S. Keller), meseguer@uiuc.edu (J. Meseguer).

1 It should be noted for readers familiar with asynchronous circuit design that the semantics developed captures a particular timing assumption, specifically delay-insensitivity. Section 7 briefly addresses two more sophisticated timing assumptions, namely speed-independence and quasi-delay-insensitivity.
In addition to the theoretical importance of providing a solid mathematical foundation to understand production rule sets and the circuits derived from them, there are numerous practical benefits to having defined a precise formal semantics. In particular, a formal semantics helps facilitate a common understanding of what circuits designed using production rule sets are, and it affords newcomers to the field of asynchronous design an unambiguous framework in which to understand existing work. Additionally, a formal semantics provides a set of mathematical tools for proving properties about asynchronous circuits, and lays the foundation for the development of automated tools to reason about those circuits.

The second issue that this paper addresses is just this problem: that of automatically proving properties about production rule sets, much like one might prove properties about a software program. Specifically, we consider a notion of deadlock freedom that is appropriate for production rule sets, as well as a property called hazard freedom. Both of these properties are necessary conditions for a circuit to be considered correct. As we will demonstrate, another benefit of having a precise formal semantics is that, when that semantics is given in an executable way, some analyses, including deadlock and hazard freedom, can be made completely automatic. Executability is obtained in this paper through a semantic formalization in rewriting logic [21,22] which, through the rewriting logic engine Maude [5], offers various automated and semi-automated analysis possibilities.

Contributions. This paper, which primarily extends [10] but also builds on our earlier work from [12], makes several new contributions:

• A new set-theoretic semantics for production rule sets is developed, relative to a timing assumption called delay-insensitivity [16]. The semantics is an improvement on our earlier work from [12], providing a more familiar operational style and, we feel, improved clarity. This semantics is referred to here as \( S_{PRS} \).

• A new rewriting logic semantics is developed that improves on our earlier workshop paper [10], from which this current paper is derived. It has the desirable property that it matches almost exactly the set-theoretic semantics, and is referred to here as \( R_{PRS} \).

• The close relationship between \( S_{PRS} \) and \( R_{PRS} \) is proved formally through the development of a strong bisimulation between their induced transition systems. This establishes their correctness relative to each other, a common technique from programming language theory.

• The automated analysis results from [10], which demonstrated the feasibility of proving hazard freedom and deadlock freedom of asynchronous circuits described as production rule sets, are extended to apply to the new rewriting logic semantics.

• Our previous work on involving the more complex timing assumptions of speed-independence and quasi-delay-insensitivity is put into context with the new work presented here, so as to provide a complete picture of the current state of our work on the semantics of production rule sets and suggest where to proceed going forward in future work.

Organization. The remainder of the paper is organized as follows. Section 2 present some preliminaries on rewriting logic and Maude. Section 3 defines \( S_{PRS} \), a formal set-theoretic semantics of production rule sets for the delay-insensitive case. Section 4 defines \( R_{PRS} \), a formal executable semantics of production rule sets in rewriting logic. Section 5 defines and proves a strong bisimulation relation between transition systems induced by \( S_{PRS} \) and \( R_{PRS} \), thus establishing the relative correctness of \( S_{PRS} \) and \( R_{PRS} \). Section 6 concerns the use of the Maude tool to check deadlock and hazard freedom automatically. Section 7 briefly reviews our previous work on two additional timing assumptions, speed-independence and quasi-delay-insensitivity, rounding out the current state of our work on the semantics of production rule sets. Section 8 considers some related work, and Section 9 finishes with concluding remarks.

2. Preliminaries on rewriting logic and Maude

Rewriting logic [21] is a semantic framework for concurrency that has been shown to express in a simple and very natural way a wide range of concurrent systems. In particular, this applies not only to models of concurrent computation and to distributed algorithms, but also to programming languages, where it has stimulated the so-called rewriting logic semantics project [22], where semantic definitions of concurrent programming languages and hardware description languages (HDLs) are given by means of rewrite theories. As emphasized in [22] and exploited in detail in the current work, the rewriting logic semantics of a language is both mathematical and operational. That is, the mathematical semantics is given by the initial model of the rewrite theory; and the operational semantics is given by rewriting logic deduction and makes the language definitions executable and analyzable by model checking in a language like Maude [5]. This paper applies this framework to the case of the highly concurrent HDL of production rule sets, which specify asynchronous circuits. The mathematical semantics of the rewrite theory characterizing the behavior of production rule sets is used to prove the semantic equivalence between the rewriting logic semantics and a set-theoretic semantics; and the operational semantics is systematically used to show how asynchronous circuits specified this way can be formally analyzed by model checking.

A rewrite theory is a triple \((\Sigma, E, R)\), where \((\Sigma, E)\) is an equational theory (in this paper we will assume that \((\Sigma, E)\) is an order-sorted equational theory, with types, subtypes, and overloading of function symbols); and where \(R\) is a collection of possibly conditional rewrite rules of the form \(t \rightarrow t' \text{ if } \text{cond} \). What \((\Sigma, E)\) specifies are the states of the concurrent system being modeled, so that a state is an element of the initial algebra \(T_{\Sigma/E}\) specified by \((\Sigma, E)\). The operations in \(\Sigma\) should be
understood as distributed state constructors, so that a state, say, \( f(t_1, \ldots, t_n) \) is the parallel composition of \( f \) of the smaller states \( t_1, \ldots, t_n \). The equations \( E \) characterize the structural properties of such states (and also define any auxiliary function on states). For example, \( f \) may be associative and commutative, so that a distributed state built with \( f \) can be thought of as a “soup” of local states. What the rules in \( R \) specify are the local concurrent transitions possible in the given distributed system; that is, a rule \( t \rightarrow t' \) if \( \text{cond} \) specifies that whenever a fragment of the distributed state is an instance of the pattern \( t \) and satisfies the condition \( \text{cond} \), then that fragment can make an atomic local transition to the corresponding instance of the pattern \( t' \). Rewriting logic is intrinsically concurrent because it allows many such local concurrent transitions to take place simultaneously. In fact, what the deduction rules of rewriting logic [21] do is to specify and characterize all the possible concurrent transitions of the system so specified, so that concurrent computation and rewriting logic deduction become identical.

Maude [5] is an executable formal specification language and system directly based on rewriting logic. In Maude there are two main types of modules: (i) functional modules, which are equational specifications of the form \( \text{fmod} (\Sigma, E) \text{endfm} \) with an initial algebra semantics; and (ii) system modules, which are rewriting logic specifications of the form \( \text{mod} (\Sigma, E, R) \text{endfm} \) with an initial model semantics. Both functional and system modules are required to satisfy natural executability conditions such as confluence and termination of the equations \( E \), and coherence of the rules \( R \) with respect to the equations (see [5]), making them suitable for execution by Maude. Maude has user-definable syntax, plus keywords for specifying types (called sorts), subtypes, equations, and rewrite rules. This makes Maude specifications essentially self-explanatory: anybody familiar with the formalism of equations and rewrite rules can easily read the corresponding Maude versions as essentially typewriter versions of the mathematical textbook-like descriptions. Besides supporting simulation of specifications with its \text{reduce} command for functional modules and its \text{rewrite} command for system modules, Maude also has good support for model checking analysis. Specifically, failure of invariants can be uncovered with its breadth first search command; and LTL model checking of properties is provided by its \text{MODEL-CHECKER} module [5].

3. Set-theoretic semantics: \( \text{SPRS} \)

This section revisits our work in [12], providing a revised set-theoretic semantics for production rule sets for the delay-insensitive case. We refer generally to the formalization given in this section as \( \text{SPRS} \). Compared to [12], \( \text{SPRS} \) applies only to the delay-insensitive case, but gains a more familiar operational formalization and, as a result of the more limited scope, a treatment which is substantially clearer and more concise. The term “set-theoretic” is used to distinguish the semantics presented in this section, which uses just standard notions from mathematics, such as sets, functions, and relations, from the executable rewriting logic semantics which follows in Section 4.

The set-theoretic semantics is useful in various ways. It is suitable as a basis for formal proofs about production rule sets, an extensive example of which is developed in [12]. A clear formal semantics is also crucial to facilitating communication between, and a common understanding among, practitioners, as well as for helping newcomers to the field understand essential concepts.

Section 3.1 deals with the syntax of production rule sets. Section 3.2 introduces the semantics informally through a small example. For simplicity, hazards are omitted from the discussion in that section. Section 3.3 formalizes the semantics in detail, including hazards, and Section 3.4 returns to the example circuit and works through an execution that generates a hazard. Section 3.5 contains a discussion about production rule sets in the context of two somewhat similar formalisms for concurrency, guarded commands and communicating sequential processes.

3.1. Syntax

The “syntax” of production rule sets consists of a mathematical construct defining a single production rule, and then a mechanism for gathering together finite sets of these constructions; hence the name production rule sets. There is also a stylized way of writing production rule sets that we review below. The choice of which notation to use is just a matter of convenience.

**Definition 1** (Syntax of Production Rule Sets). Let \( Y \) denote a denumerable set of variables used to specify node names. A production rule is a triple \( (g, x, d) \), with \( g \), the guard, being a boolean expression involving variables from \( Y \), \( x \in Y \) is the transition variable, and \( d \in \{↑, ↓\} \) is the transition direction. A production rule set is a finite set of production rules.

A production rule \( (g, x, d) \) is often written in the following, stylized manner

\[
g \mapsto x \cdot d
\]

and a set of production rules \( \{(g_1, x_1, d_1), \ldots, (g_m, x_m, d_m)\} \) is often written as a newline-separated list of the individual rules

\[
g_1 \mapsto x_1 d_1 \\
\vdots \\
g_m \mapsto x_m d_m
\]
3.2. Example

The purpose of this section is to give, by way of example, an informal introduction to the dynamic behavior of production rule sets; that is, their semantics. The example circuit we consider is shown in Fig. 1. It is known as a 3-inverter ring oscillator.

Digital ring oscillators are typically among the first circuits designed and tested in new process technologies, and they can be used as timing elements and clock generators. The simplest ring-oscillator consists of an odd number of inverters connected sequentially to form a loop. Since the number of inverters is odd, the output of each inverter will change value in sequence perpetually; as such, the ring of inverters is said to oscillate. For electrical reasons, a single inverter ring does not oscillate, so the simplest ring oscillator contains three inverters.

Fig. 1 depicts a 3-inverter ring oscillator consisting of two simple inverters and one modified inverter. In order to simplify the presentation of certain undesired circuit behaviors, we have made it so that the transistors governing the inverter with output \( x_2 \) may switch independently.\(^2\) The production rule set corresponding to the 3-inverter ring oscillator depicted in Fig. 1 is

\[
\begin{align*}
&\neg x_3 \iff x_1 \uparrow & x_1 \iff x_{1,1} \uparrow & x_1 \iff x_{1,2} \uparrow & \neg x_{1,1} \iff x_2 \uparrow & \neg x_2 \iff x_3 \uparrow \\
&x_3 \iff x_1 \downarrow & \neg x_1 \iff x_{1,1} \downarrow & \neg x_1 \iff x_{1,2} \downarrow & x_{1,2} \iff x_2 \downarrow & x_2 \iff x_3 \downarrow
\end{align*}
\]

Let us assume that the oscillator begins in a state where the nodes take values according to a function \( \sigma : \{x_1, x_{1,1}, x_{1,2}, x_2, x_3\} \mapsto \{0, 1\} \) defined by

\[
\begin{align*}
x_1, x_{1,1}, x_{1,2}, x_3 &\mapsto 0 \\
x_2 &\mapsto 1
\end{align*}
\]

For the moment we will think of the semantics of production rule sets as essentially specifying all possible \( \sigma' \)'s reachable from \( \sigma \) in a single computation step.

Informally, the \( \sigma' \)'s reachable from \( \sigma \) are obtained by considering all rules with a true guard, choosing any subset of them, and then executing the right-hand sides of the rules in this set. In our example, all of the following rules have guards that are true

\[
\begin{align*}
&\neg x_3 \iff x_1 \uparrow & \neg x_1 \iff x_{1,1} \downarrow & \neg x_1 \iff x_{1,2} \downarrow & \neg x_{1,1} \iff x_2 \uparrow & x_2 \iff x_3 \downarrow
\end{align*}
\]

However, note that while all of these production rules have guards that evaluate to true in the current state, only the rule \( \neg x_3 \iff x_1 \uparrow \) can effect an observable change in the state of the circuit nodes (\( x_1 \) rises from 0 to 1); this is a notion we call enablement; the rule \( \neg x_3 \iff x_1 \uparrow \) is said to be enabled, whereas, for example, the rule \( x_2 \iff x_3 \downarrow \) is not enabled.

As there is only a single enabled rule, and because the semantics allows for selecting no rules during a step, there are only two possible \( \sigma' \)'s reachable from \( \sigma \); namely \( \sigma' = \sigma \) and the \( \sigma' \) defined by

\[
\begin{align*}
x_1, x_2 &\mapsto 1 \\
x_{1,1}, x_{1,2}, x_3 &\mapsto 0
\end{align*}
\]

From the above \( \sigma' \), where \( x_1 \) has switched, there are four \( \sigma'' \)'s subsequently reachable, one for each subset of \( \{x_{1,1}, x_{1,2}\} \). Both of these nodes are enabled to switch to 1, and in a single step either node may, independently, “choose” to switch or not switch.

\(^2\) In modern CMOS technologies (65 nm and smaller), transistor parameters vary significantly from their nominal values due to process-induced variation and random dopant fluctuation. Considering a large circuit with say billions of transistors, there will exist a few gates, e.g., inverters, with extreme parameter variation where, for example, the PFET is several orders of magnitude slower than the NFET. It then becomes reasonable to model such a gate with independently-controlled transistors.
The semantics as just described omits one major issue that will be handled in the formal semantics: hazards. The concept of a hazard corresponds to a circuit failure and will manifest itself by a node in the circuit taking a value \( x \) which is distinct from the usual 0 or 1. We will return to the 3-inverter ring oscillator in Section 3.4 to expand the example execution steps so that hazards and their semantics are accounted for.

### 3.3. Set-theoretic semantics

At a high level, our goal is to define a binary relation between program states, denoted \( \rightarrow_p \), that corresponds to one step of concurrent execution, relative to a production rule set \( P \). \( s \rightarrow_p s' \) means that it is possible to reach state \( s' \) from state \( s \) in one computation step. The space of executions is then given by the infinite \( \rightarrow_p \)-chains

\[
s_1 \rightarrow_p s_2 \rightarrow_p s_3 \rightarrow_p \cdots
\]

subject to a form of fairness described later.

Fix a production rule set \( P \). \( \text{Variable}_P \subseteq Y \) denotes the set of all variables occurring in \( P \). A state (with respect to \( P \)) is a pair

\[
(\sigma : \text{Variable}_P \rightarrow \text{Level}, H \subseteq \text{Variable}_P)
\]

where \( \text{Level} \equiv \{0, x, 1\} \). The set of all states (with respect to \( P \)) is denoted \( \text{State}_P \).

The \( \sigma \) component of a state \( (\sigma, H) \) serves the familiar purpose of specifying values for all nodes in the circuit, with the \( x \) value meaning that a hazard has been expressed at that node. As indicated in the introduction, the treatment of certain kinds of circuit failures called hazards is a central feature of our semantics, and a substantial portion of the development will be dedicated to defining the semantics of hazards. The second component of a state, \( H \) above, also pertains hazards.

Hazards come in two varieties, interference and instability hazards. An interference hazard occurs when a node is simultaneously being pulled both up and down in the current state, roughly corresponding to a short circuit. For a given valuation \( \sigma : \text{Variable}_P \rightarrow \text{Level} \), and we define a set

\[
\text{Interference}_{P, \sigma} \subseteq \text{Variable}_P
\]

such that \( y \in \text{Interference}_{P, \sigma} \) iff there exists \( g_1 \mapsto y \uparrow, g_2 \mapsto y \downarrow \in P \) such that \( \sigma(g_1) = \sigma(g_2) = 1 \). Note that the notation just used assumes a natural extension, used throughout this paper, of the domain of \( \sigma \) from variables to boolean expressions, such as \( g_1 \) and \( g_2 \). The one complication of this is that we are operating within a three-valued universe, instead of the usual two-valued one of booleans. Therefore, we will assume that the meaning of the usual boolean operators (\( \neg, \land, \lor \)) on \( \{0, 1\} \) are extended to \( \{0, x, 1\} \) according to the following equivalences.

\[
\neg x = x \quad x \land 0 = 0 \quad x \land 1 = x \quad x \lor 1 = 1 \quad x \lor 0 = x \quad x \land x = x \quad x \lor x = x.
\]

Instability hazards occur when a gate starts pulling toward a new output level, but before reaching a stable voltage level, the gate stops pulling. This is a property of a computation step, \( (\sigma, H) \rightarrow_p (\sigma', H') \), and is captured by a set

\[
\text{Instability}_{P, \sigma, \sigma'} \subseteq \text{Variable}_P
\]

To define this set, we first need an auxiliary notion, introduced informally in the example given in the previous section, called enablement. Given a valuation \( \sigma, \text{Enabled}_{P, \sigma} \subseteq \text{Variable}_P \) is defined so that \( y \in \text{Enabled}_{P, \sigma} \) if and only if

- \( \sigma(y) \not= 0 \) and there exists a \( g \mapsto y \downarrow \in P \) such that \( \sigma(g) = 1 \), or
- \( \sigma(y) \not= 1 \) and there exists a \( g \mapsto y \uparrow \in P \) such that \( \sigma(g) = 1 \).

Given enablement, \( y \in \text{Instability}_{P, \sigma, \sigma'} \) iff \( y \not\in \text{Enabled}_{P, \sigma} \), \( y \not\in \text{Enabled}_{P, \sigma'} \), and \( \sigma(y) = \sigma'(y) \).

For convenience, we define a third predicate which captures both of the above hazards, as well as the propagation of hazards that have been expressed previously.

\[
\text{Hazard}_{P, \sigma, \sigma'} \subseteq \text{Variable}_P
\]

is defined such that \( y \in \text{Hazard}_{P, \sigma, \sigma'} \) iff any of the following conditions are met:

- \( y \in \text{Interference}_{P, \sigma, \sigma'} \);
- \( y \in \text{Instability}_{P, \sigma, \sigma'} \);
- there exists a \( g \mapsto yd \in P \) such that \( \sigma'(g) = x \).

A set of actions, namely variable assignments and skip (with respect to \( P \)), is defined as

\[
\text{Action}_P \equiv \{\text{skip}\} \cup \{x := v \mid x \in \text{Variable}_P, v \in \text{Level}\}
\]
Given a set of actions $A \subseteq \text{Action}_P$ and a variable $x \in \text{Variable}_P$, we denote the subset of $x$-actions of $A$ as

$$A|_x \overset{\text{def}}{=} \{ y : v \in A \mid y = x \}$$

**Definition 2** (Set-theoretic Semantics of Production Rule Sets). Let

$$P = \{ r_1, \ldots, r_m \}$$

be a production rule set. The evaluation relation

$$\longrightarrow \subseteq (P \times \text{State}_P) \times \text{Action}_P$$

is defined inductively according to the following five inference rules, the first four governing the evaluation of the action of individual rules:

1. $$(g \mapsto x \uparrow, (\sigma, H)) \longrightarrow x := 1 \quad \sigma(g) = 1$$
2. $$(g \mapsto x \downarrow, (\sigma, H)) \longrightarrow x := 0 \quad \sigma(g) = 1$$
3. $$(g \mapsto x \downarrow, (\sigma, H)) \longrightarrow x := \chi \quad x \in H$$
4. $$(g \mapsto x \downarrow, (\sigma, H)) \longrightarrow \text{skip}$$

Then, the relation $\longrightarrow_P$ is defined by the following fifth rule, which combines the evaluation results of all of the rules $r_1, \ldots, r_m$, and, additionally, specifies the updated $H$ set.

$$(\sigma, H) \longrightarrow_P (\sigma[a_1, \ldots, a_m], H[\sigma, a_1, \ldots, a_m])$$

where, letting $A = \{ a_1, \ldots, a_m \}$, the node valuation function $\sigma$ is updated to

$$\sigma[a_1, \ldots, a_m](x) = \begin{cases} 1 & \text{if } A|_x = \{ x := 1 \} \\ 0 & \text{if } A|_x = \{ x := 0 \} \\ x & \text{if } A|_x = \{ x := x \} \text{ or } |A|_x| > 1 \\ \sigma(x) & \text{if } A|_x = \emptyset \end{cases}$$

and the set $H$ of possible hazards is updated to the set $H[\sigma, a_1, \ldots, a_m]$ such that for all $y \in \text{Variable}_P$, $y \in H[\sigma, a_1, \ldots, a_m]$ if

- $y \in \text{Hazard}_{P,\sigma,a_1,\ldots,a_m}$, or
- $y \in H$ and $\sigma(y) = \sigma[a_1, \ldots, a_m](y)$.

Note the fact that a pair $(g \mapsto x \downarrow, (\sigma, H))$ can evaluate to skip. skip has no effect on the state, which means that the semantics supports true concurrency, wherein a subset of the set $P$ of production rules actually contributes to a state transition $(\sigma, H) \longrightarrow_P (\sigma', H')$.

An execution is a mapping $\xi : \mathbb{N} \rightarrow \text{State}_P$ such that for all $j \in \mathbb{N}, \xi(j) \longrightarrow_P \xi(j + 1)$ and such that for all $y \in \text{Variable}_P$, it is not the case that there exists a $j \in \mathbb{N}$ where for all $i > j$, $y \in \text{Enabled}_{P,\xi_i}$ or $y \in H_i$, but $y$ never switches; that is, $\sigma_i(y) = \sigma_j(y)$. This condition is the aforementioned fairness constraint.

### 3.4. Example with hazards

Let us return to the example of Section 3.2 and work through a simple set of execution steps that result in a hazard. The hazard that will be manifested is an interference hazard at the inverter whose output is $x_2$. We will make crucial use of the independent control of that gate’s component transistors.

We begin again at the same place we did in Section 3.2, with the obvious exception that we now account for hazards. So, our initial state $s_0 = (\sigma_0, H_0)$ has $\sigma_0$ defined by

$$x_1, x_{1,1}, x_{1,2}, x_3 \mapsto 0$$
$$x_2 \mapsto 1$$

and $H_0 = \emptyset$. 

Going to \( s_1 = (\sigma_1, H_1) \), we let \( x_1 \) switch. Therefore, \( \sigma_1 \) is given by
\[
\begin{align*}
    x_1, x_2 & \rightarrow 1 \\
    x_1, x_1, x_2, x_3 & \rightarrow 0
\end{align*}
\]
and \( H_1 = \emptyset \).

The inference rules governing the action of individual rules always allow for a skip action to be generated and thus \( s \rightarrow^p s \) is always a legal computation step for any production rule set \( P \) and any state \( s \). Informally, this corresponds to choosing the empty set of rules with true guards. Along such lines, let us say that \( s_2 = s_1 \).

In going from \( s_2 = s_1 \) to \( s_3 = (\sigma_3, H_3) \), we will create the basic condition for the hazard to become expressed. Let \( \sigma_3 \) be such that \( x_1, x_2 \) switches, but \( x_1, x_1 \) does not; these are the only two currently enabled rules. That is, \( \sigma_3 \) is
\[
\begin{align*}
    x_1, x_2, x_3 & \rightarrow 1 \\
    x_1, x_1, x_3 & \rightarrow 0
\end{align*}
\]

The interesting aspect of this state change is that \( H_3 \) becomes non-empty. It is straightforward to check that \( x_2 \in \text{Hazard}_{P, \sigma_2, \sigma_3} \), and therefore that \( x_2 \in H_3 \). Since there are production rules \( \neg x_1, x_2 \uparrow, x_1, x_2 \downarrow \in P \), both with guards that are true in \( \sigma_3 \), then \( x_2 \) is witnessing an interference hazard in \( s_2 \); and one can show that \( H_3 = \{ x_2 \} \).

Finally, an \( X \) can become expressed in going to \( s_4 = (\sigma_4, H_4) \) with
\[
\begin{align*}
    x_1, x_1, x_2 & \rightarrow 1 \\
    x_1, x_3 & \rightarrow 0 \\
    x_2 & \rightarrow X
\end{align*}
\]
and \( H_4 = \{ x_3 \} \).

3.5. Concurrency in production rule sets

Although the language of production rule sets shares certain features with both guarded commands [7] and communicating sequential processes (CSP) [8], it is nevertheless quite different from both of the above formalisms. In particular, it is tempting to view production rule sets via similar constructs from guarded commands or CSP, but this is incorrect. As a simple, somewhat contrived example, consider the following production rule set, which describes how, depending on the current value of nodes \( x \) and \( y \) in the circuit, nodes \( z \) and \( w \) could be concurrently pulled up toward logical 1 (↑) or pulled down toward logical 0 (↓)
\[
\begin{align*}
    x & \rightarrow z \downarrow \\
    y & \rightarrow w \downarrow \\
    \neg y & \rightarrow w \uparrow,
\end{align*}
\]
A reasonable candidate translation into the language of guarded commands would be the statement
\[
\begin{align*}
    & \text{do} \quad x \rightarrow z := 0 \\
    & \quad \text{[ }] \quad y \rightarrow w := 0 \\
    & \quad \text{[ }] \quad \neg y \rightarrow w := 1, \\
    & \text{od}
\end{align*}
\]
Similarly, one might reasonably attempt to view the above production rule set as the following CSP parallel command
\[
\begin{align*}
    & * \text{[ }] \quad x \rightarrow z := 0 \quad || \\
    & * \text{[ }] \quad y \rightarrow w := 0 \quad || \\
    & * \text{[ }] \quad \neg y \rightarrow w := 1 
\end{align*}
\]
All three formalizations are, in fact, semantically different; most importantly, the production rule set exhibits both the possibility of only some of the production rules being fired, as well as a form of “true concurrency” which is different from the standard one-at-a-time semantics of the guarded command statement, or the interleaving semantics of CSP’s parallel command operator.

If \( x = y = z = w = 1 \), then, for the above production rule set, the following are all possible next states of \( z, w \) according to the production rule set semantics:
\[
\begin{align*}
    z &= 1, \quad w = 1; \\
    z &= 1, \quad w = 0; \\
    z &= 0, \quad w = 1; \\
    z &= 0, \quad w = 0.
\end{align*}
\]
In a single step of computation, neither the guarded command statement nor the parallel command can change both $z$ and $w$ to 0 since, according to the usual operational semantics of these systems, each change in the value of a variable requires a separate semantic step. The guarded command statement and the CSP parallel command are even different from each other, since performing an action requires peeling off different sets of syntactic constructs by the operational rules.

4. Rewriting logic semantics: $\mathcal{R}_{PRS}$

The purpose of this section is to translate the set-theoretic semantics of Section 3 into an executable one using rewriting logic [21], a formalism which has been shown [22] to be well suited for exactly this task. The particular notation used is that of the rewriting logic tool Maude [5]. As we will see, the rewriting logic semantics mimics closely the set-theoretic semantics. The rewriting logic theory described in this section will be referred to as $\mathcal{R}_{PRS}$.

Executability, in a variety of useful forms, is obtained as a direct result of employing the Maude tool, which allows us to simulate circuits as well as exhaustively check that a circuit satisfies desirable correctness properties, such as hazard freedom and deadlock freedom, for example. Exploiting the execution and formal analysis capabilities gained from the Maude specification is the subject of Section 6 (the entire Maude specification is available at [11]).

The syntax of production rule sets is defined first. Recall that a production rule is a triple $g \mapsto xd$ with $g$ a Boolean expression, $x$ a variable, and $d$ the transition direction. What is needed in rewriting logic are new sorts corresponding to these concepts and populated with appropriate terms. Maude's $\textit{QID}$ module [5, Section 9.2] provides our variables: strings of characters preceded by a single quote.

```plaintext
fmod AUX-SYNTAX is pr QID * (sort Qid to Variable) .
  sorts Guard Direction ProductionRule .
  subsort Variable < Guard .
  op not_ : Guard -> Guard .
  op _and_ : Guard Guard -> Guard .
  op _or_ : Guard Guard -> Guard .
  ops + - : -> Direction .
  op [-> _ _] : Guard Variable Direction -> ProductionRule .
endfm
```

Compared to the syntax from Section 3, the corresponding terms using Maude notation are very similar. The production rule $\neg y \mapsto w \uparrow$ becomes the term $[\text{not} 'y \mapsto 'w +]$ of sort ProductionRule in Maude, for example.

Obtaining an appropriate rewriting logic definition of sets of production rules is most easily accomplished by instantiating Maude's parameterized $\textit{SET}$ module ([5, Section 9.12.2]) with a view expressing the fact that elements of the set will be terms of sort ProductionRule. The module given next does exactly that; additionally, it renames the default sort and union operator to a more convenient syntax. The details of parameterized programming in Maude (theories, views, etc.) can be found in [5, Section 8.3].

```plaintext
view ProductionRule from TRIV to AUX-SYNTAX is
  sort Elt to ProductionRule .
endv

fmod SYNTAX is pr SET(ProductionRule) *
  ( sort Set(ProductionRule) to ProductionRuleSet
    , op _,-, to _ _
  ) .
endfm
```

Therefore, in the notation of the $\textit{SYNTAX}$ module, the production rule set

\[
\begin{align*}
x & \mapsto z \downarrow \\
y & \mapsto w \downarrow \\
\neg y & \mapsto w \uparrow,
\end{align*}
\]
becomes a term of sort \texttt{ProductionRuleSet}, written in Maude as
\begin{verbatim}
[ 'x -> 'z -]
[ 'y -> 'w -]
[not 'y -> 'w +]
\end{verbatim}

Continuing from the start of Section 3.3, we define an operator which takes a production rule set \( P \) as an argument and returns the set of \texttt{Variable} terms corresponding to the set \texttt{VariableP} defined in Section 3.3. Recall that \texttt{VariableP} was defined to be the set containing all of the variables occurring in \( P \). Variables can be embedded into the guard \( g \) of a rule \( g \mapsto xd \), and also include all transition variables \((x \in g \mapsto xd)\).

\begin{verbatim}
view Variable from TRIV to SYNTAX is
  sort Elt to Variable .
endv

fmod AUX-SEMANTICS-1 is
  pr SET{Variable} * (sort Set{Variable} to 2^Variable) .
  --- meta-variable declarations omitted
  op Variable-[_] : ProductionRuleSet -> 2^Variable .
  eq Variable-[ empty] = empty .
  eq Variable-([G -> Y D] P) =
    varsOf(G), Y, Variable-[P] .
  op varsOf : Guard -> 2^Variable .
  eq varsOf(Y) = Y .
  eq varsOf(not G) = varsOf(G) .
  eq varsOf(G1 and G2) = varsOf(G1), varsOf(G2) .
  eq varsOf(G1 or G2) = varsOf(G1), varsOf(G2) .
endfm
\end{verbatim}

Notice that we have omitted the meta-variable declarations used in the equations of the above module, something we will continue to do in subsequent modules. Each used variable is given a sort equal to the one declared for the operator argument in which it is positioned (see [11] for details).

Unlike the set \texttt{VariableP}, which was specified according an equationally defined function, the sets \texttt{Level} and \texttt{StateP} will be given entirely new sorts. Recall that \texttt{Level = \{0, X, 1\}} and that for a production rule set \( P \) a state is a pair \((\sigma, H)\) with \( \sigma : \texttt{VariableP} \to \texttt{Level} \) and \( H \subseteq \texttt{VariableP} \). The \( \sigma \) component is defined using Maude's \texttt{MAP} module [5, Section 9.13.1].

\begin{verbatim}
fmod AUX-SEMANTICS-2 is pr AUX-SEMANTICS-1 .
  sorts Level .
  ops 0 1 X : -> Level .
endfm

view Level from TRIV to AUX-SEMANTICS-2 is
  sort Elt to Level .
endv

fmod AUX-SEMANTICS-3 is
  pr MAP{Variable,Level} * (op _[_] to _(_)) .
  sort State .
  op (_,_,:) : Map{Variable,Level} 2^Variable -> State .
endfm
\end{verbatim}

Note that the parameter \( P \) of \texttt{StateP} is effectively ignored in our rewriting logic specification. The implication of this is that, in principle, one could introduce a state which has or lacks a valuation for any variable, regardless of whether or not that variable is in a production rule set \( P \) under consideration. This could be fixed through the use of memberships [5, Section 4], but the specification would continue to be unsatisfactory for efficiency and other reasons. Furthermore, if we begin with a correct state, the rules in the rewriting semantics will never lead to an inconsistent state; therefore, ignoring the parameter \( P \) in \texttt{StateP} is harmless.
As an example of what\textsc{AUX-SEMANTICS-3} provides, suppose that we have a state \((\sigma, H)\) for the above production rule set where 
\[
\begin{align*}
\sigma(x) &= 0 \\
\sigma(y) &= 0 \\
\sigma(z) &= 1 \\
\sigma(w) &= x
\end{align*}
\]
and \(H = \{y, z\}\); the representation of \((\sigma, H)\) as a term of sort \text{State} is written in the Maude notation as 
\[
(('x' \mapsto 0, 'y' \mapsto 0, 'z' \mapsto 1, 'w' \mapsto X), ('y', 'z'))
\]

Moving on to the definition of the various hazard-related concepts, we will require the ability to evaluate guards according to a three-valued valuation.

\begin{verbatim}
fm\text{od}\ AUX-SEMANTICS-4\ is\ pr\ AUX-SEMANTICS-3 .
... --- meta-variable declarations omitted

op _(_): Map{Variable, Level} Guard -> Level .
eq Sigma(not G1) = not3 Sigma(G1) .
eq Sigma(G1 and G2) = Sigma(G1) and3 Sigma(G2) .
eq Sigma(G1 or G2) = Sigma(G1) or3 Sigma(G2) .

op not3_ : Level -> Level [prec 24] .
eq not3 0 = 1 .
eq not3 1 = 0 .
eq not3 X = X .

op _and3_ : Level Level -> Level [assoc comm id: 1] .
eq X and3 0 = 0 .
eq 0 and3 0 = 0 .
eq X and3 X = X .

op _or3_ : Level Level -> Level [assoc comm id: 0] .
eq X or3 1 = 1 .
eq 1 or3 1 = 1 .
eq X or3 X = X .
endfm
\end{verbatim}

We are now in a position to handle the primary definitions having to do with hazards: \text{Interference}_{P, \sigma}, \text{Instability}_{P, \sigma, \sigma'}, and \text{Hazard}_{P, \sigma, \sigma'}; all of which are predicates on \text{Variable}_{P}. Consider \text{Interference}_{P, \sigma}. We declare an equationally defined function that takes two arguments, one a term of sort \text{ProductionRuleSet} corresponding to \(P\), and the second a term of sort \text{Map{Variable, Level}} corresponding to \(\sigma\), and returns a term of sort \text{2^{Variable}}.

Recall how \(\text{Interference}_{P, \sigma} \subseteq \text{Variable}_{P}\) was defined: for all \(y \in \text{Variable}_{P}, y \in \text{Interference}_{P, \sigma}\) if and only if there exists \(g_1 \mapsto y \uparrow, g_2 \mapsto y \downarrow \in P\) such that \(\sigma(g_1) = \sigma(g_2) = 1\). We accomplish this in Maude with two auxiliary functions. \text{InterfPred} determines if a variable satisfies the interference property and \text{InterfFilter} filters the set \text{Variable}_{P} by applying \text{InterfPred} to every variable in \(P\) one-by-one.

\begin{verbatim}
fm\text{od}\ AUX-SEMANTICS-5\ is\ pr\ AUX-SEMANTICS-4 .
... --- meta-variable declarations omitted

op InterfPred : Variable ProductionRuleSet Map{Variable, Level} -> Bool .

cq InterfPred(Y, P, Sigma) = true
if [G+ -> Y +] [G- -> Y -] P' := P
\ \ Sigma(G+) == 1 and Sigma(G-) == 1 .
eq InterfPred(Y, P, Sigma) = false [owise] .
endfm
\end{verbatim}
op Interference-{_,_} :
  ProductionRuleSet Map{Variable,Level} -> 2^Variable .
eq Interference-{P,Sigma} =
  InterfFilter(Variable-(P), P, Sigma) .

op InterfFilter :
  2^Variable ProductionRuleSet
  Map{Variable,Level} -> 2^Variable .
eq InterfFilter(empty , P, Sigma) = empty .
eq InterfFilter((Y,YS), P, Sigma) =
  if InterfPred(Y, P, Sigma) then Y else empty fi ,
  InterfFilter(YS, P, Sigma) .
endfm

Instability_{P,\sigma,\sigma'} \subseteq Variable_P was defined in Section 3 so that for all \( y \in \text{Variable}_P \), \( y \in \text{Instability}_{P,\sigma,\sigma'} \) if and only if \( y \in \text{Enabled}_{P,\sigma}, y \notin \text{Enabled}_{P,\sigma'}, \) and \( \sigma(y) = \sigma(y') \). Recall that \( \text{Enabled}_{P,\sigma} \subseteq \text{Variable}_P \) was defined so that for all \( y \in \text{Variable}_P, y \in \text{Enabled}_{P,\sigma} \) if and only if:

- \( \sigma(y) \neq 0 \) and there exists a \( g \mapsto y \downarrow \in P \) such that \( \sigma(g) = 1 \), or
- \( \sigma(y) \neq 1 \) and there exists a \( g \mapsto y \uparrow \in P \) such that \( \sigma(g) = 1 \).

The corresponding definitions in Maude are very similar, and use again the Pred and Filter pair idea from the definition of Interference. InstFilter is omitted because it is not substantively different from InterfPred (see [11]).

fmod AUX-SEMANTICS-6 is pr AUX-SEMANTICS-5 .
...
... --- meta-variable declarations omitted

op EnabledPred :
  Variable ProductionRuleSet Map{Variable,Level} -> Bool .
ceq EnabledPred(Y, P, Sigma) = true
  if Sigma(Y) /= 1
  \( \land \) [G+ \rightarrow Y +] P' := P
  \( \land \) Sigma(G+) == 1 .
ceq EnabledPred(Y, P, Sigma) = true
  if Sigma(Y) /= 0
  \( \land \) [G- \rightarrow Y -] P' := P
  \( \land \) Sigma(G-) == 1 .
eq EnabledPred(Y, P, Sigma) = false [owise] .

op InstPred :
  Variable ProductionRuleSet Map{Variable,Level}
  Map{Variable,Level} -> Bool .
ceq InstPred(Y, P, Sigma, Sigma') = true
  if EnabledPred(Y, P, Sigma )
  \( \land \) not EnabledPred(Y, P, Sigma')
  \( \land \) Sigma(Y) == Sigma'(Y) .
eq InstPred(Y, P, Sigma, Sigma') = false [owise] .

op Interference-{_,_,_} :
  ProductionRuleSet Map{Variable,Level}
  Map{Variable,Level} -> 2^Variable .
eq InterfFilter{P, Sigma, Sigma'} =
  InstFilter(Variable-(P), P, Sigma, Sigma') .

... InstFilter omitted
endfm
Hazard_{P, \sigma, \sigma'} is transcribed directly. Recall that Hazard_{P, \sigma, \sigma'} is just the union of Interference_{P, \sigma'} and Instability_{P, \sigma, \sigma'}, plus the propagation of any \mathcal{X} values.

```
fm mod AUX-SEMANTICS-7 is pr AUX-SEMANTICS-6 .
... --- meta-variable declarations omitted

  op HazardPred :
    Variable ProductionRuleSet Map{Variable,Level}
    Map{Variable,Level} -> Bool .
  ceq HazardPred(Y, P, Sigma, Sigma') = true
    if InterfPred(Y, P, Sigma') .
  ceq HazardPred(Y, P, Sigma, Sigma') = true
    if InstPred(Y, P, Sigma, Sigma') .
  ceq HazardPred(Y, P, Sigma, Sigma') = true
    if [G -> Y D] P' := P
         \ Sigma(G) == X .
  eq HazardPred(Y, P, Sigma, Sigma') = false [owise] .

  op Hazard-{_,_,_} :
    ProductionRuleSet Map{Variable,Level}
    Map{Variable,Level} -> 2^Variable .
  eq Hazard-{P, Sigma, Sigma'} =
    HazardFilter(Variable-{P}, P, Sigma, Sigma') .
... HazardFilter omitted
endfm
```

Subsequent to Hazard_{P, \sigma, \sigma'} we defined Action_{P} and A|_{y}, where A \subseteq Action_{P} and y \in Variable_{P}. Recall that actions are either pairs containing a variable and a level, or the special action \textit{skip}. The restriction operator on a set of actions picks those non-skip actions with a particular variable given as the first component.

```
fm mod AUX-SEMANTICS-8 is pr AUX-SEMANTICS-7 .
  sort Action .
  op _:=_ : Variable Level -> Action .
  op skip : -> Action .
endfm

view Action from TRIV to AUX-SEMANTICS-8 is
  sort Elt to Action .
endv

fm mod AUX-SEMANTICS-9 is pr AUX-SEMANTICS-8 .
  pr SET{Action} * (sort Set{Action} to 2^Action) .
... --- meta-variable declarations omitted

  op __|__ : 2^Action Variable -> 2^Action .
  eq (Y := V , A) | Y = Y := V , (A | Y) .
  eq A | Y = empty [owise] .
endfm
```
The following rewrite rules are used to mimic the effect of the four inference rules

\[
\begin{align*}
(g \mapsto x \uparrow, (\sigma, H)) & \to x := 1 & \sigma(g) = 1 \\
(g \mapsto x \downarrow, (\sigma, H)) & \to x := 0 & \sigma(g) = 1 \\
(g \mapsto x \Rightarrow, (\sigma, H)) & \to x \in H \\
(g \mapsto x \Rightarrow, (\sigma, H)) & \to \text{skip}
\end{align*}
\]

mod AUX-SEMANTICS-10 is pr AUX-SEMANTICS-9.

Notice that the \(<_,>_\) constructor yields a term of kind \([\text{Action}]\) (see [5, Section 3.5]), but without a proper sort. This will be crucial when we define the rewrite rule corresponding to \(\rightarrow_P\) to ensure that all of the \(\langle r_j, (\sigma, H) \rangle\) get rewritten according to the above rules into actions \(a_j\); that is, terms of sort \text{Action}.

The top-level rewrite rule that ultimately gives us \(\rightarrow_P\) relies on rewriting logic equivalents for \(\sigma[a_1, \ldots, a_m]\) and \(H[\sigma, a_1, \ldots, a_m]\). We start with \(\sigma[a_1, \ldots, a_m]\), which was defined in Section 3 according to

\[
\sigma[a_1, \ldots, a_m](x) = \begin{cases} 
1 & \text{if } A | x = \{ x := 1 \} \\
0 & \text{if } A | x = \{ x := 0 \} \\
x & \text{if } A | x = \{ x := x \} \text{ or } |A| > 1 \\
\sigma(x) & \text{if } A | x = \emptyset
\end{cases}
\]

\(H[\sigma, a_1, \ldots, a_m]\) is similarly straightforward. Following the definition from Section 3, \(H[\sigma, a_1, \ldots, a_m] \subseteq \text{Variable}_P\) such that for all \(y \in \text{Variable}_P, y \in H[\sigma, a_1, \ldots, a_m]\) iff

\[
\begin{itemize}
  \item y \in \text{Hazard}_{P,\sigma,a_1,\ldots,a_m}, \text{ or }
  \item y \in H \text{ and } \sigma(y) = \sigma[a_1, \ldots, a_m](y).
\end{itemize}
\]
Finally, we give a conditional rewrite rule that captures the earlier top-level inference rule

\[
\frac{\langle r_1, (\sigma, H) \rangle \rightarrow a_1 \quad \ldots \quad \langle r_m, (\sigma, H) \rangle \rightarrow a_m}{(\sigma, H) \rightarrow_p (\sigma[a_1, \ldots, a_m], H[\sigma, a_1, \ldots, a_m])}
\]

There are a couple of ways in which the rewriting logic definition appears different from the corresponding inference rule. First, note that since we are using the logical symbol \( \rightarrow \) (from rewriting logic) to define \( \rightarrow_p \) (from our static semantics), the production rule set parameter must be encoded in the terms being rewritten. This is the purpose of the \( \_\{\_, \_\}\{\}_\) constructor.

The second difference is that the single condition of the rewrite rule above

\[
mkActs(P, (\Sigma, H)) \Rightarrow A
\]

serves the purpose of the multiple premises of the inference rule

\[
\frac{\langle r_1, (\sigma, H) \rangle \rightarrow a_1 \quad \ldots \quad \langle r_m, (\sigma, H) \rangle \rightarrow a_m}{(\sigma, H) \rightarrow_p (\sigma[a_1, \ldots, a_m], H[\sigma, a_1, \ldots, a_m])}
\]
The reason for this difference is that the number of premises, \( m \), is not fixed, but rather scales dynamically with the size of the production rule set; in rewriting logic, however, the number of conditions in a rewrite rule is fixed.

Finally, it is important to note that while \( \text{mkActs} \) is only kinded, the variable \( \lambda \) has sort \( \text{Act} \). This ensures that all of the individual production rules are rewritten to actions in the condition, before a computation step is taken.

5. Strong bisimulation between \( S_{PRS} \) and \( R_{PRS} \)

This section establishes a strong bisimulation between two transition systems: one induced by the set-theoretic semantics of production rule sets, \( S_{PRS} \), and the other induced by the executable semantics given in rewriting logic, \( R_{PRS} \). In so doing, confidence is raised in the correctness of the two semantic formalizations, as well as in the use of the executable semantics as an analysis tool. This increased confidence applies in particular to the model checking results presented later in Section 6.

The strong bisimulation result is obtained as follows. First, we define a function, \( \text{cast}_P \), that maps states in the set-theoretic semantics to corresponding states in the rewriting logic semantics. We then make explicit the transition systems associated to both the semantics; and finally we show that the two are strongly bisimilar via \( \text{cast}_P \). This result yields as corollaries that hazard freedom and deadlock freedom, which are also defined formally later in this section, are preserved by the mapping between the two semantics.

We introduce a number of mathematical conventions that are used throughout this section. First, we assume that \( R_{PRS} \) is comprised as

\[
R_{PRS} = (\Sigma_{PRS}, E_{PRS}, \Sigma_{PRS}).
\]

Use of the sort name \( \text{Configuration} \) is overloaded to also denote the set

\[
T_{\Sigma_{PRS}/E_{PRS}, \text{Configuration}}
\]

of elements of \( T_{\Sigma_{PRS}/E_{PRS}} \) of sort \( \text{Configuration} \); the distinction will be clear from context.

5.1. \( \text{cast}_P \)

Fix a production rule set \( P \). Our bisimulation relation is defined by a function \( \text{cast}_P \) taking each state \( (\sigma, H) \in \text{State}_P \) to a corresponding term in the rewriting logic specification \( R_{PRS} \); specifically, \( \text{cast}_P((\sigma, H)) \) will be a term of sort \( \text{Configuration} \). That is, applying the overloading of \( \text{Configuration} \) specified above, \( \text{cast}_P \) is a function

\[
\text{cast}_P : \text{State}_P \rightarrow \text{Configuration}.
\]

\( \text{cast}_P \) is defined by means of two auxiliary \( \text{cast} \) functions, one that recurses over the structure of an element of \( \text{State}_P \), yielding a term of sort \( \text{State} \), and a second recursing over the structure of a production rule set and yielding a term of sort \( \text{ProductionRuleSet} \); specifically,

\[
\text{cast}_P(s) = \{ \text{cast}(s) \} \{ \text{cast}(P) \}.
\]

To simplify the presentation note that we have used “\( \text{cast} \)” in an ad-hoc polymorphic way to denote both the function that converts the state part, as well as the function that converts the production rule set. \( \text{cast} \) will also name all similar functions converting other types of objects in the set-theoretic semantics into terms in the rewriting logic semantics.

The definition of \( \text{cast} \) functions is largely routine. For most constructs in the set-theoretic formalization, there is a corresponding operator in the rewriting logic semantics with the same arguments and we simply generate that operator and then recurse. For example, individual production rules are cast as

\[
\text{cast}(g \mapsto xd) = [\text{cast}(g) \rightarrow \text{cast}(x) \text{cast}(d)]
\]

At the bottom are the atomic elements of the syntax, such as variables and the transition directions, which are cast as

\[
\text{cast}(\uparrow) = + \quad \text{cast}(\downarrow) = -
\]

Casting finite sets made out of simpler elements highlights an interesting point. The following definition is unambiguous and correct in rewriting logic, as well as in Maude, by asserting that the operator \( \_ \_ \_ \_ \_ \_ \_ \_ \_ \) juxtaposition, is associative, commutative, and idempotent. Specifically in Maude, its predefined \( \text{SET} \) module [5, Section 9.12.2] employs \textit{equational attributes} [5, Section 4.4.1] and associate-commutative rewriting for associativity and commutativity, and for idempotency an explicit equation is used.

\[
\text{cast}([r_1, \ldots, r_m]) = \text{cast}(r_1) \cdots \text{cast}(r_m)
\]

Similarly, valuation functions of the form \( \sigma : \text{Variable}_P \rightarrow \text{Level} \) can be viewed as sets of pairs, and are cast accordingly into
By way of summarizing, consider the production rule set corresponding to a single nand-gate
\[ P = \{ x_1 \land x_2 \mapsto y \downarrow, \neg x_1 \lor \neg x_2 \mapsto y \uparrow \} \]
and a state \( s = (\sigma, \emptyset) \) where \( \sigma(x_1) = \sigma(x_2) = \sigma(y) = 1. \) Then \( \text{cast}_P(s) \) yields the following term
\[
\begin{array}{l}
\langle 'x_1 |-> 1 , 'x_2 |-> 1, 'y |-> 1, \text{empty} \rangle \\
\{ [ 'x_1 \text{ and } 'x_2 |-> 'y - ] \\
[ (\text{not } 'x_1) \text{ or } (\text{not } 'x_2) |-> 'y + ] \}
\end{array}
\]

The following lemma will be useful to simplify some of the proofs given later.

**Lemma 1.** Let \( P \) be a production rule set.
\[
\text{cast}_P : \text{State}_P \rightarrow \text{Configuration}
\]
is injective.

**Proof.** Straightforward by induction on \( \text{State}_P. \) \( \square \)

### 5.2. Strong bisimulation

Having defined \( \text{cast}_P \) we are now in a position to state our main result establishing the strong bisimulation between \( \text{PRS} \) and \( \text{RPR} \). For notational convenience and symmetry, for a given production rule set \( P \), clear from context, we use \( \rightarrow_M \) to denote the relation \( \rightarrow_p \) defined according to the set-theoretic semantics \( \text{PRS} \). Similarly, we use \( \rightarrow_R \) to denote the one step rewriting relation induced by \( \text{RPR} \) on terms of sort \( \text{Configuration} \).

**Theorem 1.** Let \( P \) be a production rule set. Consider the transition systems
\[
T_M \overset{\text{def}}{=} (\text{State}_P, \rightarrow_M) \\
T_R \overset{\text{def}}{=} (\text{Configuration}, \rightarrow_R)
\]
then \( \text{cast}_P \), when seen as a relation, is a strong bisimulation between \( T_M \) and \( T_R \).

The following lemma will be useful in the proof of the above theorem.

**Lemma 2.** Let \( P \) be a production rule set. For all \( g \mapsto xd \in P, (\sigma, H) \in \text{State}_P, \) and \( a \in \text{Action}_P, \) we have
\[
\langle g \mapsto xd, (\sigma, H) \rangle \rightarrow a
\]
according to \( \text{PRS} \), if and only if
\[
\text{RPR} \vdash \text{cast}(\langle g \mapsto xd, (\sigma, H) \rangle) \rightarrow \text{cast}(a)
\]

**Proof.**
See Appendix A. \( \square \)

Having this lemma, we can proceed with a proof of Theorem 1.

**Proof of Theorem 1.**
See Appendix A. \( \square \)

The two correctness properties that we are concerned with, hazard freedom and deadlock freedom, can both be phrased in terms of simple reachability queries. For a transition system \( A = (A, \rightarrow \subseteq A \times A) \) and an element \( a \in A \), we let \( \text{Reach}_A(a) \subseteq A \) denote the set of reachable states from \( a \); i.e.,
\[
\{ a' \in A \mid a \rightarrow^* a' \}.
\]
The relative correctness of \( \text{PRS} \) and \( \text{RPR} \) with respect to these correctness properties will fall out through instances of the following corollary to Theorem 1.
Corollary 1. Let $P$ be a production rule set and $s_0 \in \text{State}_P$. For any pair of predicates $Q_M \subseteq \text{State}_P$ and $Q_R \subseteq \text{Configuration}$ such that $s \in Q_M$ if and only if $\text{cast}_P(s) \in Q_R$, then

$$\text{Reach}_{\text{TA}_M}(s_0) \subseteq Q_M \text{ if and only if } \text{Reach}_{\text{TA}_R}(\text{cast}_P(s_0)) \subseteq Q_R.$$ 

Proof of Corollary 1.
See Appendix A. □

5.3. Hazard freedom

Hazard freedom essentially asserts the impossibility of reaching a state where any node takes the value $X$. Let $P$ be a production rule set. We define a predicate $\text{Hazard!}_P \subseteq \text{State}_P$ such that for all $(\sigma, H) \in \text{State}_P$, $(\sigma, H) \in \text{Hazard!}_P$ if and only if there exists a $y \in \text{Variable}_P$ with $\sigma(y) = X$.

Definition 3. Let $P$ be production rule set and $\sigma : \text{Variable}_P \rightarrow \text{Level}$. We say that $\mathcal{R}_{\text{PRS}}$ exhibits hazard freedom with respect to $P$ and $\sigma$ if and only if for all $(\sigma', H') \in \text{Reach}_{\text{TA}_M}(\sigma, H)$

$$(\sigma', H') \notin \text{Hazard!}_P.$$ 

In rewriting logic, we can give an equationally-defined function, $\text{Hazard!}$, which is essentially the characteristic function of $\text{Hazard!}_P$.

```plaintext
op Hazard! : Configuration -> Bool [frozen].

eq Hazard!(((Y \rightarrow X, SIGMA), H)(P)) = true.

eq Hazard!((SIGMA, H)(P)) = false [owise].
```

Definition 4. Let $P$ be production rule set and $\sigma : \text{Variable}_P \rightarrow \text{Level}$. We say that $\mathcal{R}_{\text{PRS}}$ exhibits hazard freedom with respect to $P$ and $\sigma$ if and only if for all $c' \in \text{Reach}_{\text{TA}_R}(\text{cast}_P((\sigma, H)))$

such that we have $\text{Hazard!}(c') = \text{false}$. 

Proposition 1. Let $P$ be production rule set and $\sigma : \text{Variable}_P \rightarrow \text{Level}$. $\mathcal{R}_{\text{PRS}}$ exhibits hazard freedom with respect to $P$ and $\sigma$ if and only if $\mathcal{R}_{\text{PRS}}$ exhibits hazard freedom with respect to $P$ and $\sigma$.

Proof. According to Corollary 1, it is sufficient to show that for all $s \notin \text{State}_P$, $s \in \text{Hazard!}_P$ if and only if $\text{Hazard!}(\text{cast}_P(s)) = \text{false}$. This is straightforward by induction on states. □

5.4. Deadlock freedom

As with hazard freedom, deadlock freedom will be characterized with respect to a production rule set and an initial valuation. It is essentially an assertion of the impossibility of reaching a state where no rules are enabled. One small difference from the definition of enablement is needed to account for $X$ values, however. Equivalently, it asserts the impossibility of reaching a state where the only transitions that are possible are idle transitions.

Let $P$ be a production rule set and $\sigma : \text{Variable}_P \rightarrow \text{Level}$ and recall from Section 3.3 the definition of $\text{Enabled}_{P, \sigma} \subseteq \text{Variable}_P$. For all $y \in \text{Variable}_P$, $y \in \text{Enabled}_{P, \sigma}$ if and only if

- $\sigma(y) \neq 0$ and there exists a $g \mapsto y \downarrow \in P$ such that $\sigma(g) = 1$, or
- $\sigma(y) \neq 1$ and there exists a $g \mapsto y \uparrow \in P$ such that $\sigma(g) = 1$.

We define a new predicate, $\text{Switchable}_{P, \sigma, H} \subseteq \text{Variable}_P$, similar to $\text{Enabled}_{P, \sigma}$, but also with a parameter $H \subseteq \text{Variable}_P$. For a variable $y \in \text{Variable}_P$ and a state $(\sigma, H) \in \text{State}_P$, $y \in \text{Switchable}_{P, \sigma, H}$ if and only if one of the following is true:

- $y \neq X$ and $y \in H$.

The definition of deadlock freedom is then analogous to the definition of hazard freedom, with $\text{Switchable}_{P, \sigma, H} \neq \emptyset$ serving the purpose of $\sigma \notin \text{Hazard!}_P$. 

Definition 5. Let \( P \) be production rule set and \( \sigma : \text{Variable}_P \rightarrow \text{Level} \). We say that \( S_{\text{PRS}} \) exhibits deadlock freedom with respect to \( P \) and \( \sigma \) if and only if for all
\[
(\sigma', H') \in \text{Reach}_{T_M}((\sigma, \emptyset))
\]
\( \text{Switchable}_{P,\sigma',H'} \neq \emptyset \).

Above in Section 4 we defined the rewriting logic equivalent to the conditions that define whether a particular variable is in the set of things that are enabled. This was denoted \( \text{EnabledPred} \). We define a similar notion for \( \text{Switchable}_{P,\sigma,H} \), called \( \text{SwitchPred} \).

\[
\begin{align*}
\text{op SwitchPred} &: \text{Variable} \times \text{ProductionRuleSet} \times \text{Map}\{\text{Variable},\text{Level}\} \rightarrow \text{Bool} \\
\text{ceq \quad SwitchPred}(Y,P,\Sigma,H) &= \text{true} \\
&\quad \text{if } \Sigma(Y) \neq 1 \\
&\quad \text{if } \Sigma(G+) == 1 \\
&\quad \text{if } \Sigma(G-) == 1 \\
&\quad \text{if } Y \in H \\
&\quad \text{eq \quad SwitchPred}(Y,P,\Sigma,H) = \text{false} \quad \text{[otherwise]}. \\
\end{align*}
\]

To get \( \text{Switchable} \) from \( \text{SwitchPred} \), we simply need a function that gets all of the variables from a production rule set and then filters the result by \( \text{SwitchPred} \). This is entirely routine, and can be had in exactly the same way as we defined for, for example, \( \text{InterfFilter} \).

Definition 6. Let \( P \) be production rule set and \( \sigma : \text{Variable}_P \rightarrow \text{Level} \). We say that \( P \) exhibits deadlock freedom with respect to \( P \) and \( \sigma \) if and only if for all
\[
(SIGMA', H') \in \text{Reach}_{T_R}(\text{cast}_P((\sigma, \emptyset)))
\]
\( \text{Switchable}_{\{-P,SIGMA',H'\}} \neq \emptyset \).

Proposition 2. Let \( P \) be production rule set and \( \sigma : \text{Variable}_P \rightarrow \text{Level} \). \( S_{\text{PRS}} \) exhibits deadlock freedom with respect to \( P \) and \( \sigma \) if and only if \( R_{\text{PRS}} \) exhibits deadlock freedom with respect to \( P \) and \( \sigma \).

Proof. Similar to that for preservation of hazard freedom. \( \square \)

6. Hazard and deadlock freedom analysis with \( R_{\text{PRS}} \) and Maude

This section investigates the feasibility of using our executable semantics and the formal tools provided by Maude to prove hazard freedom and deadlock freedom, two properties that every asynchronous circuit must typically satisfy in order to be considered correct. All of the Maude source code and example circuits used for experimentation are open source and available at [11].

Section 6.1 briefly describes each of the asynchronous circuits we are subjecting to analysis. Section 6.2 describes the commands necessary to check hazard and deadlock freedom using Maude and the semantics \( R_{PR} \) presented in Section 4. These results demonstrate the need for some optimizations, which are described in Section 6.3, followed in Section 6.4 by modified analysis results with the optimizations enabled. Due to the highly asynchronous nature of production rule sets, there is an enormous state space explosion even for simple circuits; therefore, the optimizations we present are essential to model check circuits in practice.

6.1. Circuits analyzed

Our experiments cover six circuits of size varying from 12 production rules up to 130 production rules. The complete production rule set for each circuit can be found in [11].

- 3InverterRing (12 production rules): A ring oscillator is typically the first circuit used to demonstrate the viability of a new process technology.
Fig. 2. Hazard freedom and deadlock freedom model checking results, without optimizations. System configuration: Maude 2.5, Intel Xeon X5570 (2.93GHz, 8MiB L3), 24GiB RAM, 64-bit Linux, kernel version 2.6.18. “TIME” means the experiment timed out (30 minutes), and “MEM” means it reached a preset memory limit (4GiB).

- **ClosedBuffer** (26 production rules): Simple logical buffer stages are generally used to balance parallel paths in an asynchronous pipeline. Closing the buffer requires a source to produce tokens to send into the buffer and a sink to empty the buffer.
- **Toggle** (28 production rules): A circuit that alternates between sourcing a one or a zero token are essential components of most test harnesses.
- **PCHBAndFixed** (66 production rules): The PCHB (pre-charged half buffer) is a customized quasi delay-insensitive gate that is electrically similar to a stage of domino logic. A PCHB can be used in a data-path to perform computation, it can be used for control, or it can act as a combination thereof. In this instance the input to the PCHB is a fixed value.
- **1BitFullAdderFixed** (118 production rules): The bit-slice ripple-carry adder is ubiquitous in digital VLSI design. This variant is implemented as a quasi delay-insensitive PCHB with the input of the adder tied to a fixed value source.
- **PCHBAndToggle** (130 production rules): This instance of a PCHB AND-gate makes use of the toggle element to alternate the input pattern thus generating each possible input permutation.

### 6.2. Model checking experiments

The hazard and deadlock freedom analyses are performed using Maude’s **search** command [5, Section 12]. search does a breadth-first search enumerating all terms reachable from a given initial term through rewriting. If this set of reachable terms is finite and one is interested in checking computable invariants, as is the case for both hazard and deadlock freedom, then **search** acts as a decision procedure for the satisfaction of the given invariant.

As described above in Section 5.3, the invariant we want to check for hazard freedom is the negation of $\text{Hazard!}$, or alternatively, that no reachable state satisfies $\text{Hazard!}$. We use this second formulation, which is accomplished in Maude by executing the following command,

```
search [1] initialC =>* C:Configuration such that Hazard!(C).
```

The term `initialC` equals \( \text{cast}_P((\sigma, \emptyset)) \) where \( \sigma \) denotes the valuation of nodes of the device at reset. If no solution is returned, then the device is considered hazard-free with respect to that reset state. If not, then the device has a potential hazard.

The situation for deadlock freedom is analogous, except that the invariant for deadlock freedom is that `Switchable-\{\_\,\_\,\_\}` should never be empty; again, we use the dual formalization, however. The appropriate incantation is

```latex
\text{search [1] initialC =>* (SIGMA',H')(P) such that Switchable-(P,SIGMA',H') == empty.}
```

Applying these checks to each of the circuits described above in Section 6.1 we find that none of the checks are able to finish due to exhausting the system’s available memory resources, which are substantial for a contemporary system (24GiB). For consistency with the presentation of subsequent results, this initial experiment is reported in Fig. 2. Clearly, some form of simplification/optimization is needed to reduce memory consumption and gain tractability even for the very small circuits we are considering.

### 6.3. Performance optimizations

Two optimizations to the rewriting logic semantics, \( R_{PRS} \), are now developed. These are specifically aimed at addressing excessive memory consumption and result in tractable analysis of all but our largest circuit, PCHBAndToggle. In the case of the largest circuit, the analysis is still improved in the sense that, with respect to the system configuration limits that we set, it goes from being memory bound to being computation bound (Of course, it is possible that relaxing the time-limit constraint, say to an hour, might once again cause us to encounter a memory limitation; however, having observed system behavior as the experiment was run, this appears to be unlikely: the memory footprint was not growing very slowly when the time limit was reached.)

---

3 The reason why the search command cannot use the \( \Rightarrow! \) modality is because empty sets of actions can always produce idle transitions.
Out-of-control memory usage is primarily due to the condition of the top-level rewrite rule in \( R_{PRS} \), which, recalling from Section 4, is

\[
mkActs(P, (\Sigma, I)) \Rightarrow A.
\]

Suppose that we have the production rule set

\[
[ 'x' -> 'z' ] \\
[ 'y' -> 'w' ] \\
[ not 'y' -> 'w +' ]
\]

Applying \( mkActs \) to this set yields the following term of kind \( 2^{\text{Action}} \)

\[
< [ 'x' -> 'z' ], (\Sigma, I) > , \\
< [ 'y' -> 'w' ], (\Sigma, I) > , \\
< [ not 'y' -> 'w +' ], (\Sigma, I) >
\]

and each of the elements of this set are rewritten one-by-one until a term of sort \( 2^{\text{Action}} \) is obtained, e.g., say

\[
'z := 0 , \\
skip , \\
'w := X
\]

In deriving this term, Maude is necessarily inefficient, because it cannot know that the rewriting of each element of the set is independent from the others; that is, outside of the rule chosen to rewrite each element, the order in which these rewrites are applied is inconsequential, so rewriting

\[
< [ 'x' -> 'z' ], (\Sigma, I) > , \\
skip , \\
< [ not 'y' -> 'w +' ], (\Sigma, I) >
\]

and then

\[
'z := 0 , \\
skip , \\
< [ not 'y' -> 'w +' ], (\Sigma, I) >
\]

does not need to be considered separately from first rewriting

\[
'z := 0 , \\
< [ 'y' -> 'w' ], (\Sigma, I) > , \\
< [ not 'y' -> 'w +' ], (\Sigma, I) >
\]

followed by

\[
'z := 0 , \\
skip , \\
< [ not 'y' -> 'w +' ], (\Sigma, I) >
\]

Maude must, however, attempt all \( 2^k \) possible orderings, where \( k \) is the number of production rules, for what is really just a single possible next state.

The independent nature of the rewriting steps can be communicated to Maude by, instead of producing a set of terms to rewrite, having \( mkActs \) produce a list with some arbitrary order and then using matching to force the rewriting to iterate over this list.

The second optimization reduces the possible sets of actions that, at the condition of the top-most rewrite rule in \( R_{PRS} \), become bound to the variable \( A \). Accomplishing this reduction is done through a small modification to the inference rules for \( \rightarrow \) in Definition 2, so that, for example,

\[
(g \mapsto x \uparrow; (\sigma, H)) \rightarrow x \mapsto 1 \quad \sigma(g) = 1
\]

is modified so that the side condition includes also that \( \sigma(x) \neq 1 \); that is \( \sigma(g) = 1 \) becomes \( \sigma(g) = 1 \) and \( \sigma(x) \neq 1 \). Of course, this change must get reflected at the rewriting logic level as well. The correctness of this optimization, while not proved in detail, follows from the invariance of the updates to \( \sigma \) and \( H \) during a transition when, for example, \( \sigma(x) = 1 \) and one of the updating actions is \( x := 1 \).

### 6.4. Model checking experiments redux

The result of applying each optimization in isolation, as well as the aggregate effect of applying both in tandem, is shown in Fig. 3 for hazard freedom, and in Fig. 4 for deadlock freedom. With these optimizations, some of our example circuits can be checked quite quickly. Scalability clearly remains an issue, however, even after applying the above optimizations, though they accomplish much over the original semantics, which was, above all, designed for conceptual clarity.
Fig. 3. Hazard freedom model checking results, with optimizations. System configuration: Maude 2.5, Intel Xeon X5570 (2.93GHz, 8MiB L3), 24GiB RAM, 64-bit Linux, kernel version 2.6.18. “TIME” means the experiment timed out (30 minutes), and “MEM” means it reached a preset memory limit (4GiB).

<table>
<thead>
<tr>
<th>Circuit Name</th>
<th>Size</th>
<th>list opt</th>
<th>switch opt</th>
<th>all opts</th>
</tr>
</thead>
<tbody>
<tr>
<td>3InverterRing</td>
<td>12rl</td>
<td>TIME</td>
<td>MEM</td>
<td>TIME</td>
</tr>
<tr>
<td>ClosedBuffer</td>
<td>26rl</td>
<td>TIME</td>
<td>MEM</td>
<td>TIME</td>
</tr>
<tr>
<td>Toggle</td>
<td>28rl</td>
<td>TIME</td>
<td>MEM</td>
<td>TIME</td>
</tr>
<tr>
<td>PCHBAAndFixed</td>
<td>66rl</td>
<td>TIME</td>
<td>MEM</td>
<td>TIME</td>
</tr>
<tr>
<td>1BitFullAdderFixed</td>
<td>118rl</td>
<td>TIME</td>
<td>MEM</td>
<td>TIME</td>
</tr>
<tr>
<td>PCHBAAndToggle</td>
<td>130rl</td>
<td>TIME</td>
<td>MEM</td>
<td>TIME</td>
</tr>
</tbody>
</table>

Fig. 4. Deadlock freedom model checking results, with optimizations. System configuration: Maude 2.5, Intel Xeon X5570 (2.93GHz, 8MiB L3), 24GiB RAM, 64-bit Linux, kernel version 2.6.18. “TIME” means the experiment timed out (30 minutes), and “MEM” means it reached a preset memory limit (4GiB).

<table>
<thead>
<tr>
<th>Circuit Name</th>
<th>Size</th>
<th>list opt</th>
<th>switch opt</th>
<th>all opts</th>
</tr>
</thead>
<tbody>
<tr>
<td>3InverterRing</td>
<td>12rl</td>
<td>TIME</td>
<td>MEM</td>
<td>TIME</td>
</tr>
<tr>
<td>ClosedBuffer</td>
<td>26rl</td>
<td>TIME</td>
<td>MEM</td>
<td>TIME</td>
</tr>
<tr>
<td>Toggle</td>
<td>28rl</td>
<td>TIME</td>
<td>MEM</td>
<td>TIME</td>
</tr>
<tr>
<td>PCHBAAndFixed</td>
<td>66rl</td>
<td>TIME</td>
<td>MEM</td>
<td>TIME</td>
</tr>
<tr>
<td>1BitFullAdderFixed</td>
<td>118rl</td>
<td>TIME</td>
<td>MEM</td>
<td>TIME</td>
</tr>
<tr>
<td>PCHBAAndToggle</td>
<td>130rl</td>
<td>TIME</td>
<td>MEM</td>
<td>TIME</td>
</tr>
</tbody>
</table>

Therefore, more optimizations along the lines of those above, as well as clever new ideas will be needed in the future to make automatic checks for hazards and deadlock reliably tractable for modern circuits. Some additional tractability can be gained by looking at more practical timing assumptions, described in the next section, which further reduce the amount of concurrency.

7. Speed-independent and quasi-delay-insensitive circuits

The primary objective of this paper is to improve upon our work in [10, 12] for the unrestricted, or delay-insensitive, case, developing a clearer set-theoretic semantics, an executable rewriting logic semantics that is almost identical, and showing that the updated executable semantics is still capable of being used for automated formal analysis. However, two other timing assumptions, speed-independence [23, 25] and quasi-delay-insensitivity [15, 18], that we have worked on previously bear mention as well, so as to complete the picture of where our current efforts regarding the formal semantics and analysis of production rule sets and to understand how we intend go proceed in future work.

Toward that end, this section briefly presents analysis results on the same set of circuits used above, but under the speed-independence and quasi-delay-insensitive timing assumptions. The results given are with respect to an implementation of these timing assumptions in Maude [10]. That implementation was developed in accordance with [12], which provides a detailed account of our earlier effort to formalize these two timing assumptions in a non-executable way. In future work, as we mention in Section 9, it will be important to give similar treatment to these more complex timing assumptions as we have done in the previous sections of this paper for the delay-insensitive case, iterating on the non-executable semantics in order to make them clearer, and formally connecting the non-executable semantics with a revised executable semantics in rewriting logic. However, this is a very substantial undertaking, likely requiring the length of the current work for each timing assumption individually to specify both a non-executable and executable semantics and prove their equivalence through a suitable bisimulation; this is outside the scope of the current paper.
At a high level, both speed-independence and quasi-delay-insensitivity represent restrictions on relative delay of signals on forks, which occur when the output of a gate fans out to the input of two or more subsequent gates. Speed-independence imposes the restriction that if one branch of a fork switches to a new level, then all branches must switch simultaneously. On the other hand, quasi-delay-insensitivity allows for some branches of a fork to have stabilized before others do, but only until a sequence of “acknowledgments” from the stabilized branch courses through the circuit to the input of the gate connected to the non-stabilized branch of the original fork. Formal details of these timing assumptions, given as a non-executable semantics, can be found in [12] and spend great time defining precisely, for example, a notion of acknowledgment. Additional details are too extensive to go into additional detail here, but can be found in the above reference (see also [19]).

The behaviors admitted by delay-insensitivity, quasi-delay-insensitivity, and speed-independence are related as follows: delay-insensitivity admits strictly more behaviors than quasi-delay-insensitivity, which in turn admits strictly more behaviors than speed-independence. Both of the more restrictive timing assumptions reduce the set of possible device behaviors, thereby making formal analysis easier. The trade-off is that one must analyze the circuit separately to ensure that the assumptions made about timing are actually valid given the physics of the device.

In addition, the more restrictive timing assumptions have the added, although somewhat counter intuitive, advantage of being theoretically more capable, in the sense that the delay-insensitive timing assumption is so permissive that the set of useful production rule sets becomes limited because more of them imply hazardous circuit behaviors. A proof of this fact is developed in [14]. It is also worth noting that, for hazard freedom, we have shown previously that speed-independence and quasi-delay-insensitivity are equivalent, yielding a simpler check for the property relative to the quasi-delay-insensitivity assumption. The proof of this fact is developed in [12].

Figs. 5 and 6 present the results of analyzing the circuits from Section 6.1 under the more restrictive timing assumptions. Despite the fact that the number of behaviors is reduced, we found ourselves still unable to exhaustively prove hazard freedom and deadlock freedom for our largest circuit, PCHBAndToggle.

Finally, we experimented with an optimization specifically tailored for the speed-independence case, where we simply removed production rules corresponding to wires. With this optimization we were able to check hazard freedom for all of the example circuits listed above, as shown in Fig. 7. Due to the result from [12] cited above, this implies hazard freedom in the quasi-delay-insensitive case as well.
Fig. 6. Deadlock freedom model checking results for all three timing assumptions. System configuration: Maude 2.5, Intel Xeon X5570 (2.93GHz, 8MiB L3), 24GiB RAM, 64-bit Linux, kernel version 2.6.18. "–" indicates that the experiment timed out or exhausted available memory resources.

<table>
<thead>
<tr>
<th>Circuit Name</th>
<th>Size</th>
<th>delay-insensitive</th>
<th>speed-independence</th>
</tr>
</thead>
<tbody>
<tr>
<td>3InverterRing</td>
<td>12rl</td>
<td>no</td>
<td>yes</td>
</tr>
<tr>
<td>ClosedBuffer</td>
<td>26rl</td>
<td>–</td>
<td>yes</td>
</tr>
<tr>
<td>Toggle</td>
<td>28rl</td>
<td>–</td>
<td>yes</td>
</tr>
<tr>
<td>PCHBAEndFixed</td>
<td>66rl</td>
<td>–</td>
<td>no</td>
</tr>
<tr>
<td>1BitFullAdderFixed</td>
<td>118rl</td>
<td>–</td>
<td>–</td>
</tr>
<tr>
<td>PCHBAEndToggle</td>
<td>130rl</td>
<td>–</td>
<td>–</td>
</tr>
</tbody>
</table>

Fig. 7. Hazard freedom model checking results, wires removed. System configuration: Maude 2.5, Intel Xeon X5570 (2.93GHz, 8MiB L3), 24GiB RAM, 64-bit Linux, kernel version 2.6.18. "–" indicates that the experiment timed out or exhausted available memory resources.

<table>
<thead>
<tr>
<th>Circuit Name</th>
<th>Size</th>
<th>quasi-delay-insensitive</th>
</tr>
</thead>
<tbody>
<tr>
<td>3InverterRing</td>
<td>12rl</td>
<td>yes</td>
</tr>
<tr>
<td>ClosedBuffer</td>
<td>26rl</td>
<td>yes</td>
</tr>
<tr>
<td>Toggle</td>
<td>28rl</td>
<td>yes</td>
</tr>
<tr>
<td>PCHBAEndFixed</td>
<td>66rl</td>
<td>no</td>
</tr>
<tr>
<td>1BitFullAdderFixed</td>
<td>118rl</td>
<td>–</td>
</tr>
<tr>
<td>PCHBAEndToggle</td>
<td>130rl</td>
<td>–</td>
</tr>
</tbody>
</table>
8. Related work

The work presented in this paper covers two somewhat separate topics and therefore the related work falls into two distinct categories: the semantics of production rule sets, as a topic of interest in its own right, and the formal verification of asynchronous digital circuits, specifically hazard freedom.

Regarding the first topic, namely, the semantics of production rule sets, the current work improves upon our own earlier efforts in [10, 12]. The current efforts, including both the set-theoretic and the rewriting logic styles, provide a cleaner and simpler presentation of the delay-insensitive case. To the best of our knowledge, no other works have presented semantic issues as an end in and of themselves, but rather simply in support of some larger goal, such as Martin’s synthesis method [15,17]. The semantics has also been addressed in an auxiliary way to prove that the scope of possible circuits under delay-insensitivity is limited [16] and that quasi delay-insensitive circuits are Turing-complete [14]. The semantics from [15,17] was also examined in [26] in order to clarify the relationship between production rule sets and corresponding physical circuit implementations.

The second topic addressed is formal verification of asynchronous circuits, in particular verifying hazard freedom and deadlock freedom. Our work seems to be the first that attempts to use the formal executable semantics approach (modulo our work in [10], on which the current work is based). It is also the only work that we know of that provides an extensive formal verification platform for asynchronous circuits designed using production rule sets, which we get via Maude’s built in tools, including a full LTL model checker.

The methods developed in [3] use two versions of the circuit; one high-level and one low-level. Both designs are given as specialized automata, and while a full enumeration of the reachable state space in the high-level design is necessary, a careful analysis shows how to avoid doing the same for the low-level design. This yields a more efficient analysis of hazard free operation, since the high-level design has a smaller state space than the more detailed, low-level design. [28] uses the modern program analysis technique of abstract interpretation to reason about hazards in asynchronous circuits. [29] uses a standard symbolic model checker to verify hazard free operation of speed-independent circuits, and an older tool called prlint [6] purports to exhaustively check hazard free operation of a production rule set. prlint is no longer easily available, and we were unable to acquire a version capable of running on a modern Linux workstation.

A class of Petri nets, called signal transition graphs (STGs), can be used to model certain aspects of asynchronous circuits [13], and a number of works, e.g., [4,24,27,30,31] propose methods of model checking these Petri net specifications. Certain high-level properties such as liveness and fairness can be verified in this way, but the STG specification does not expose low-level circuit properties like the timing of forks.

9. Conclusion

This paper improves upon our earlier work in [10,12], providing a cleaner formal semantics of production rule sets for the delay-insensitive case, including both a set-theoretic semantics and an executable semantics in rewriting logic; which is, to the best of our knowledge, the first of its kind. The utility of our work here is, first and foremost, toward promoting a common understanding of what production rule sets mean, especially to those entering the field of asynchronous circuit design; and, secondly for the purpose of formal analysis of such circuits.

Regarding formal analysis, the set-theoretic semantics is perhaps best suited as the foundation for proving meta-theorems about production rule sets, as we did in [12]. The executable rewriting logic semantics is instead better suited to establishing the functional correctness of individual circuits, as certain obligations may be discharged automatically, as we showed above and in [10].

A number of challenges remain, some rather daunting. Firstly, the speed-independence and quasi delay-insensitivity cases from [12] should be further developed along the lines of what we did here for the delay-insensitive case. Second, there is the issue of scalability: we have been able to automatically check hazard freedom and deadlock freedom for circuits up to about one hundred production rules, but modern circuits can easily be four orders of magnitude larger.

One possibility is to investigate probabilistic methods in more detail, which are highly parallelizable and scale extremely well. Existing work on probabilistic rewrite systems and statistical model checking [1,2] allows for a rewriting-based approach to continue to be used, and perhaps even build directly on our work here.

Acknowledgments

The authors are indebted to the anonymous reviewers of this article for their thoughtful and detailed comments on earlier drafts. We are also grateful for the contributions of Alain J. Martin to our earlier efforts at formalizing the semantics of production rule sets under various timing assumptions, as well as for having developed the framework of production rule sets to begin with. Michael Katelman and José Meseguer were supported in part by NSF Grant CCF 09-05584.
Appendix A. Proof of Lemma 2, Theorem 1, and Corollary 1

Proof of Lemma 2.

By cases on \(a\):

\(\bullet\) (skip): It is enough to show that both \(\langle g \mapsto x \cdot d, (\sigma, H) \rangle \longrightarrow \text{skip}\), with respect to \(S_{PRS}\), and \(\text{cast}(\langle g \mapsto x \cdot d, (\sigma, H) \rangle) \longrightarrow \text{cast}(\text{skip})\), with respect to \(R_{PRS}\), hold unconditionally. That \(\langle g \mapsto x \cdot d, (\sigma, H) \rangle \longrightarrow \text{skip}\) holds unconditionally with respect to \(S_{PRS}\) is established according to the rule (where the variables used in the rule are not the same as those above; they are implicitly quantified)

\[
\langle g \mapsto x \cdot d, (\sigma, H) \rangle \longrightarrow \text{skip}
\]

Similarly, with respect to \(R_{PRS}\),

\[
R_{PRS} \vdash \text{cast}(\langle g \mapsto x \cdot d, (\sigma, H) \rangle) =
\langle \text{cast}(g) \to \text{cast}(x) \cdot \text{cast}(d), (\text{cast}(\sigma), \text{cast}(H)) \rangle
\longrightarrow \text{skip} = \text{cast}(\text{skip})
\]

due to the rewriting rule

\[
rl < [G \to Y D], (\Sigma, H) > \Rightarrow \text{skip}.
\]

\(\bullet\) (y := 1): (⇒) Suppose \(\langle g \mapsto x \cdot d, (\sigma, H) \rangle \longrightarrow y := 1\) holds with respect to \(S_{PRS}\); we show that also \(\text{cast}(\langle g \mapsto x \cdot d, (\sigma, H) \rangle) \longrightarrow \text{cast}(y := 1)\) with respect to \(R_{PRS}\).

Clearly, \(\langle g \mapsto x \cdot d, (\sigma, H) \rangle \longrightarrow y := 1\) may only hold with respect to \(S_{PRS}\) according to the rule

\[
\langle g \mapsto x \uparrow, (\sigma, H) \rangle \longrightarrow y := 1 \quad \sigma(g) = 1
\]

which implies that \(\sigma(g) = 1\), and also that \(d = \uparrow\) and \(y = x\). Therefore,

\[
\text{cast}(\langle g \mapsto x \cdot d, (\sigma, H) \rangle) = \langle \text{cast}(g) \to \text{cast}(x) \cdot 1, (\text{cast}(\sigma), \text{cast}(H)) \rangle
\]

and matches the left-hand side of the rewrite rule

\[
crl < [G \to Y +], (\Sigma, H) > \Rightarrow (Y := 1)
\]

if \(\Sigma(\Sigma) = 1\).

That \(\sigma(g) = 1\) implies also the condition of the rewriting rule:

\[
\text{cast}(\sigma)(\text{cast}(g)) = 1
\]

is straightforward. This yields, as needed, that

\[
\text{cast}(\langle g \mapsto x \cdot d, (\sigma, H) \rangle) \longrightarrow \text{cast}(x) := 1 = \text{cast}(y := 1).
\]

(⇐) Suppose \(R_{PRS} \vdash \text{cast}(\langle g \mapsto x \cdot d, (\sigma, H) \rangle) \longrightarrow \text{cast}(y := 1)\).

\[
\text{cast}(y := 1) = \text{cast}(y) := 1, \text{and it is clear that the only way this rewriting can occur is by application of the rule}
\]

\[
crl < [G \to Y +], (\Sigma, H) > \Rightarrow (Y := 1)
\]

if \(\Sigma(\Sigma) = 1\).

Through pattern matching, we get again that \(\text{cast}(y) = \text{cast}(x)\) and therefore that \(y = x\); \(\text{cast}(d) = +\), which implies that \(d = \uparrow\); and that \(\text{cast}(\sigma)(\text{cast}(g)) = 1\). Then, assuming that \(\text{cast}(\sigma)(\text{cast}(g)) = 1\) one can straightforwardly derive that \(\sigma(g) = 1\).

\[
\langle g \mapsto x \uparrow, (\sigma, H) \rangle \longrightarrow x := 1 \quad \sigma(g) = 1
\]
applies to \((g \mapsto x \cdot d, (\sigma, H)) = (g \mapsto x \mapsto 1, (\sigma, H))\) which ultimately yields the desired result that \((g \mapsto x \cdot d, (\sigma, H)) \rightarrow (x \mapsto 1) = (y \mapsto 1)\).

- The remaining cases are similar. \(\square\)

**Proof of Theorem 1.**

\((T_R \text{ simulates } T_M):\) Let \((\sigma, H), (\sigma', H') \in \text{State}_P\) be such that

\[(\sigma, H) \rightarrow_M (\sigma', H')\]

According to Definition 2, there exist actions \(a_1, \ldots, a_m\) such that: (a) for each \(a_j, 1 \leq j \leq m\), \((r_j, (\sigma, H)) \rightarrow a_j\) and (b)

\[(\sigma' = (a_1, \ldots, a_m), H[\sigma, a_1, \ldots, a_m])\]

Therefore, it is sufficient to prove that

\[\text{cast}_P((\sigma, H)) \rightarrow_R \text{cast}_P((\sigma[a_1, \ldots, a_m], H[\sigma, a_1, \ldots, a_m]))\]

In order to show that the above relation holds, we will apply the top-level rewrite rule defined in \(\mathcal{R}_{PRS}\), namely

\[\text{crl} \left\{ \text{Sigma}, \text{H} \right\} \{P\} \Rightarrow \left\{ \text{Sigma}[A], \text{H[Sigma, A]} \right\} \{P\} \]

if \(\text{mkActs}(P, (\text{Sigma}, \text{H})) \Rightarrow A\). and Lemma 2. Expanding \(\text{cast}_P((\sigma, H))\) shows that it matches the left-hand side of this rule:

\[\text{cast}_P((\sigma, H)) = (\text{cast}(\sigma), \text{cast}(H))(\text{cast}(P))\]

For the condition, we first expand \(\text{mkActs}\) according to its definition, yielding

\[\text{mkActs}(\text{cast}(r_1) \ldots \text{cast}(r_m), (\text{cast}(\sigma), \text{cast}(H))) = \langle \text{cast}(r_1), (\text{cast}(\sigma), \text{cast}(H)) \rangle, \ldots, \langle \text{cast}(r_m), (\text{cast}(\sigma), \text{cast}(H)) \rangle \]

and which, according to Lemma 2, rewrites to

\[\text{cast}(a_1), \ldots, \text{cast}(a_m)\]

Therefore, by the above rewrite rule, which is part of \(\mathcal{R}_{PRS}\), we obtain that \(\text{cast}_P((\sigma, H))\) rewrites to a term \(\{\text{Sigma}', \text{H}'\}\{P\}\) with

\[\text{Sigma}' = \text{cast}(\sigma)[\text{cast}(a_1), \ldots, \text{cast}(a_m)]\]

and

\[H' = \text{cast}(H)[\text{cast}(\sigma), \text{cast}(a_1), \ldots, \text{cast}(a_m)](\text{cast}(P))\]

Expanding \(\text{cast}_P((\sigma[a_1, \ldots, a_m], H[\sigma, a_1, \ldots, a_m]))\), one obtains a term

\[(\text{cast}(\sigma[a_1, \ldots, a_m]), \text{cast}(H[\sigma, a_1, \ldots, a_m]))(\text{cast}(P))\]

and so it remains to be proved that \(H' = \text{cast}(H[\sigma, a_1, \ldots, a_m])\), which is straightforward.

\((T_M \text{ simulates } T_R):\) Let \((\sigma, H) \in \text{State}_P\) and let \(\eta\) be a term of sort \(\text{Configuration}\) such that

\[\text{cast}_P((\sigma, H)) \rightarrow_R \eta\]

As there is only a single rewrite rule in \(\mathcal{R}_{PRS}\) that operates on terms of the same kind as the sort \(\text{Configuration}\), namely

\[\text{crl} \left\{ \text{Sigma}, \text{H} \right\} \{P\} \Rightarrow \left\{ \text{Sigma}[A], \text{H[Sigma, A]} \right\} \{P\} \]

if \(\text{mkActs}(P, (\text{Sigma}, \text{H})) \Rightarrow A\). \(\eta\) must be of the form

\[(\text{cast}(\sigma)[A], \text{cast}(H)[\text{cast}(\sigma), A](\text{cast}(P)))(\text{cast}(P))\]

for some term \(A\) of sort \(2^\text{Action}\) reachable via rewriting from the term

\[\text{mkActs}(\text{cast}(P), (\text{cast}(\sigma), \text{cast}(H)))\]

Lemmas 2 and 1 about the injectivity of \(\text{cast}\) imply that, letting \(P = \{r_1, \ldots, r_m\}\), \(A\) is of the form

\[\text{cast}(a_1), \ldots, \text{cast}(a_m)\]

with \(a_1, \ldots, a_m \in \text{Action}_P\), such that for each \(a_j, 1 \leq j \leq m\), \(\text{cast}(a_j)\) is had through rewriting a term of the form

\[\langle \text{cast}(r_j), (\text{cast}(\sigma), \text{cast}(H)) \rangle\]. This establishes, for each \(1 \leq j \leq m\), that \(\mathcal{R}_{PRS} \vdash \langle \text{cast}(r_j), (\text{cast}(\sigma), \text{cast}(H)) \rangle \rightarrow \]

"..."
cast(\(\sigma\), \(a_1\)), and therefore by Lemma 2 also that \(\{t_1, (\sigma, H)\} \rightarrow a_1\). Then applying the rule
\[
\begin{align*}
\langle t_1, (\sigma, H) \rangle & \rightarrow a_1 \\
(\sigma, H) & \rightarrow P \ (\sigma[a_1, \ldots, a_m], H[\sigma, a_1, \ldots, a_m])
\end{align*}
\]
we get that \((\sigma, H) \rightarrow_M (\sigma[a_1, \ldots, a_m], H[\sigma, a_1, \ldots, a_m])\). Again, it remains to be shown that the \(\_\) operators, for each component of the state, correspond, which follows according to a straightforward induction. □

Proof of Corollary 1.

(\(\Rightarrow\)): Suppose that Reach\(_R\)(\(s_0\)) \(\subseteq Q_M\); we demonstrate by induction on \(T_R\) reachability derivations (these transition systems have finite carriers) that also
\[
\text{Reach}_R(\text{cast}_P(s_0)) \subseteq Q_R.
\]

The induction hypothesis asserts that, for a reachable configuration
\[
c \in \text{Reach}_R(\text{cast}_P(s_0))
\]
both of the following conditions hold: (1) \(c \in \text{cast}_P(\text{State}_P)\), and for the unique \(s \in \text{State}_P\), guaranteed by Lemma 1, such that \(\text{cast}_P(s) = c\). (2) \(s \in \text{Reach}_M(s_0)\). This implies that \(s \in Q_M\) and therefore that \(c \in Q_R\).

The induction hypothesis clearly holds for \(c_0 = \text{cast}_P(s_0)\). Now, let
\[
\text{cast}_P(s) = c \in \text{Configuration}
\]
be such that it has both properties of the induction hypothesis and suppose that \(c \rightarrow_R c'\) with \(c' \in \text{Configuration}\). It follows from Theorem 1 and the induction hypothesis that there exists a \(s' \in \text{State}_P\) such that \(s \rightarrow_M s'\) and \(\text{cast}_P(s') = c'\). This implies that \(c \in \text{cast}_P(\text{State}_P)\) and that \(s' \in \text{Reach}_M(s_0)\).

(\(\Leftarrow\)): This direction follows similarly, but without the need of Lemma 1. □

References