

© 2011 MICHAEL KAHN KATELMAN

A META-LANGUAGE FOR FUNCTIONAL VERIFICATION

BY

MICHAEL KAHN KATELMAN

DISSERTATION

Submitted in partial fulfillment of the requirements
for the degree of Doctor of Philosophy in Computer Science
in the Graduate College of the
University of Illinois at Urbana-Champaign, 2011

Urbana, Illinois

Doctoral Committee:

Professor José Meseguer, Chair and Director of Research
Professor Arvind, Massachusetts Institute of Technology
Associate Professor Grigore Roşu
Professor Josep Torrellas

ABSTRACT

This dissertation perceives a similarity between two activities: that of coordinating the search for simulation traces toward reaching verification closure, and that of coordinating the search for a proof within a theorem prover. The programmatic coordination of simulation is difficult with existing tools for digital circuit verification because stimuli generation, simulation execution, and analysis of simulation results are all decoupled. A new programming language to address this problem, analogous to the mechanism for orchestrating proof search tactics within a theorem prover, is defined wherein device simulation is made a first-class notion. This meta-language for functional verification is first formalized in a parametric way over hardware description languages using rewriting logic, and subsequently a more richly featured software tool for Verilog designs, implemented as an embedded domain-specific language in Haskell, is described and used to demonstrate the novelty of the programming language and to conduct two case studies. Additionally, three hardware description languages are given formal semantics using rewriting logic and we demonstrate the use of executable rewriting logic tools to formally analyze devices implemented in those languages.

This dissertation is dedicated to MIPS Technologies, Inc., in appreciation for having provided me with the opportunity to closely observe the functional verification effort of the 74K microprocessor.

ACKNOWLEDGMENT

I entered the doctoral program during August of 2004, and ultimately defended on July 18, 2011. Summarizing concisely the nearly seven years during which I have been part of the Department of Computer Science at UIUC is, of course, impossible. However, I am profoundly grateful to have been able to participate in some way in the long and important tradition of scholarly work, and to have been able to experience so many of the special things that come from being an academic.

Not the least of those things are all of the friendly, smart, and eccentric people who I met and shared the experiences with, and whose creativity was always thought provoking and entertaining. I am also incredibly grateful for the freedom I was afforded to pursue my own research agenda, and feel that it enriched my experience it immeasurably. In addition, I enjoyed especially the many seminars and courses I attended, books and research articles I learned from, research visits at other institutions, and conferences in far-away places (Svalbard!).

A few people deserve special thanks in making all the incredible experiences noted above possible. My friend and advisor, José Meseguer, of course being foremost among them for his technical, emotional, and financial support, and for being so committed to the ideals of scholarly pursuit. Arvind and his research group at MIT could not have been more hospitable and welcoming, or more generous with their time. So much so that I inflicted myself upon them during three separate extended stays. I am also extremely grateful to the other members of my dissertation committee, Josep Torrellas and Grigore Roşu, who were always kind and helpful in pursuing this work.

I wish that I knew how to properly thank all of the people, places, and things with which I associate the best aspects of the past seven years:

(apologies for omissions) José Meseguer, Arvind, Josep Torrellas, Grigore Roşu, Tanya Crenshaw, Joe Hendrix, Sean Keller, Ralf Sasse, Camilo Rocha,

Musab AlTurki, Kyungmin Bae, Beatriz Alarcón, Raúl Gutiérrez, Felix Schernhammer, Santiago Escobar, Peter Ölveczky, Francisco Duran, Narciso Martí-Oliet, Fredrik Kjølstad, Azadeh Farzan, Nana Arizumi, Traian Şerbănuţă, Patrick Meredith, Chucky Ellison, Andrei Ştefănescu, Dennis Griffith, Michael Ilseman, Edgar Pek, Pavithra Prabhakar, David Nelson, Alexandre Duchâteau, Jonas Eckhardt, Tobias Mühlbauer, Steven Lauterburg, Rajesh Kumar Karmani, Vijay Ganesh, Nirav Dave, Myron King, Kermin Elliot Fleming, Michael Pellauer, Murali Vijayaraghavan, Abhinav Agarwal, Rays Jiang, Andrew Colombi, Jeff Green, Mark-Oliver Stehr, Susie Heo, Samuel Kamin, Barış Aktemur, Howard Katelman, Susan Katelman, John Katelman, Joseph Katelman, Madeline Katelman, Dav Zimak, Brendan Kiburg, Nicholas Rizzolo, The Graybeards.

TABLE OF CONTENTS

CHAPTER 1	INTRODUCTION	1
CHAPTER 2	PRELIMINARIES	9
2.1	Rewriting Logic	9
2.2	Haskell	16
CHAPTER 3	RELATED WORK	20
3.1	Functional Verification	20
3.2	Rewriting Logic Semantics	25
CHAPTER 4	FORMALIZATION	28
4.1	Overview	28
4.2	Parameterization	30
4.3	Analogy	31
4.4	\mathcal{R}_{HDL}	33
4.5	Problem	34
4.6	\mathcal{R}_{META}	35
4.7	\mathcal{R}_{IR}	36
4.8	\mathcal{R}_{STRAT}	39
4.9	Example	39
CHAPTER 5	IMPLEMENTATION ARCHITECTURE	42
5.1	VlogMetaLang	43
5.2	Skeleton	45
5.3	VlogMetaLang.Core	46
5.4	VlogMetaLang.Syntax	48
5.5	VlogMetaLang.Data	49
5.6	VlogMetaLang.Util	51
5.7	VlogMetaLang.Strategy	52
5.8	VlogMetaLang.SMT	54
5.9	Example	54

CHAPTER 6	CAPABILITIES	59
6.1	Some Utilities	61
6.2	Coordination of Multiple Simulations	63
6.3	Feedback	68
6.4	Backtracking	72
6.5	Breadth-First	76
6.6	Symbolic Execution	79
6.7	Combined Concrete and Symbolic Simulation	81
CHAPTER 7	CASE STUDIES	85
7.1	I ² C Bus-Master Controller	85
7.2	Microprocessor	91
CHAPTER 8	SEMANTICS: VERILOG	102
8.1	Disclaimer	102
8.2	Contributions	102
8.3	Concepts	103
8.4	Semantics: Configuration	109
8.5	Semantics: Equations and Rules	112
8.6	Examples	119
CHAPTER 9	SEMANTICS: PRODUCTION RULE SETS	126
9.1	Introduction	126
9.2	Mathematical Semantics: \mathcal{M}_{PRS}	128
9.3	Rewriting Logic Semantics: \mathcal{R}_{PRS}	138
9.4	Relative Correctness of \mathcal{M}_{PRS} and \mathcal{R}_{PRS}	151
9.5	Automated Hazard and Deadlock Freedom Analysis	162
9.6	Speed-Independent and Quasi-Delay-Insensitive Circuits	168
9.7	Conclusion	170
CHAPTER 10	SEMANTICS: BTRS	174
10.1	Syntax	175
10.2	Caveats	179
10.3	Example: Single-Element Queue	180
10.4	Semantics Overview	181
10.5	Expression Evaluation	184
10.6	Action Evaluation	189
10.7	Semantics	192
10.8	Discussion	193
10.9	Example: A Deadlocking Completion Buffer	194
CHAPTER 11	CONCLUSION	199

APPENDIX A	IMPLEMENTATION DETAILS	201
A.1	start	201
A.2	concretize	203
A.3	simulate	205
REFERENCES	214

CHAPTER 1

INTRODUCTION

The International Technology Roadmap for Semiconductors (ITRS) [41] is a biennial report on technical challenges confronting the global semiconductor industry, representing a consensus view of major industry associations from Asia, Europe, and the United States. Topics ranging from device physics to embedded software are discussed and analyzed to identify research directions and potential solutions, speculating over a fifteen-year time span. The topic of this dissertation is the part of the digital circuit design process called *functional verification*, wherein a device is evidenced to coincide with its high-level design specification, and about which the 2009/2010 ITRS makes the following statements.

“Implied needs are in: (1) verification, which is a bottleneck that has now reached crisis proportions . . .”

“... due to the growing complexity of silicon designs, functional verification is still an unresolved challenge, defeating the enormous effort put forth by armies of verification engineers and academic research efforts.”

“Multiple sources report that in current development projects verification engineers outnumber designers, with this ratio reaching two to one for the most complex designs.”

[41, Design]

The above quotations demonstrate the practical importance, and indeed the urgency of, concerted research efforts aimed at improving functional verification: effecting verification closure faster, at a higher quality, and with fewer engineering resources. Many aspects of the functional verification process warrant attention from researchers, however in this dissertation, we

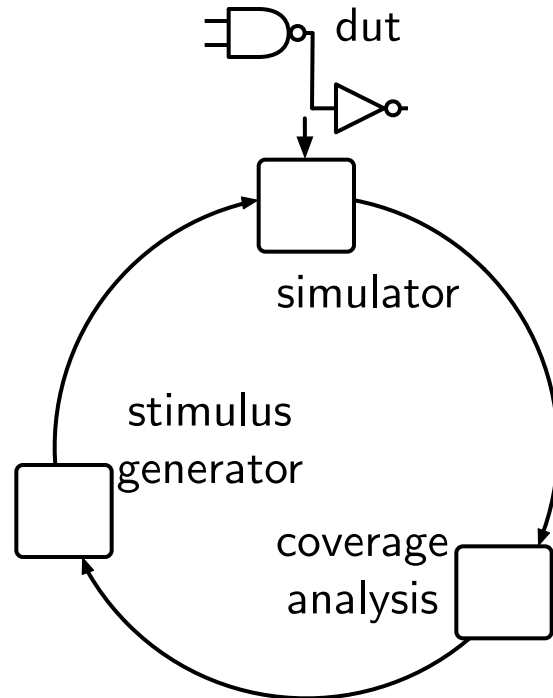


Figure 1.1: Coverage Closure Feedback Loop.

address just one, *coverage closure*. This is the iterative process of generating stimuli, running simulations, and assessing various coverage metrics (*e.g.* see [6]), as depicted in Figure 1.1.

Janick Bergeron, a highly regarded engineer who has written extensively on functional verification (*e.g.* [5, 6]), recently said about the coverage closure process:

Something that is challenging and time-consuming is an ideal candidate for automation. In [contemporary verification practice], the Holy Grail is the automation of the feedback loop between the coverage metrics and the constraint solver.

[7, July 5, 2010]

Fully automating this feedback loop is an unreasonable goal, as doing so would imply an efficient automated algorithm for reaching coverage closure; indeed, this is why it is described above as being “the Holy Grail”. However,

it is wrong to view automation as something that must be done in full, or not at all; as this dissertation will demonstrate, many opportunities for partial automation of the coverage closure feedback loop exist and can be taken advantage of.

The main contribution of this dissertation is the development of a programming language that allows for the entire coverage closure feedback loop to be coordinated programmatically and in a uniform way. As a result, verification engineers can easily construct programs that partially automate this feedback loop, doing so in a way that is general-purpose, tailored for a particular device, or even tailored for an individual coverage goal. The ratios of stimulus generation, simulation, and analysis are made completely flexible, so that a verification engineer can target different strategies. Strategies with a high amount of simulation per loop iteration may be useful in early stages of design when many coverage goals are still outstanding, for example; whereas in later stages of the verification process, tighter iterations of the loop may be used to generate highly targeted simulations for coverage goals that have proved difficult to discharge.

The design of our programming language is based on a similarity perceived between two activities: that of coordinating the search for simulation traces toward reaching coverage closure, and that of coordinating the search for a proof within a theorem prover. Theorem proving systems traditionally orchestrate the search for a proof through the use of a meta-language, an idea that originates with the programming language ML, which was originally developed with the express purpose of facilitating interaction with the LCF theorem prover [29, 30]. This dissertation extends that idea and applies it to simulation-based functional verification, yielding *a meta-language for functional verification*.

Our meta-language is called `fvml` and is largely characterized by making simulation a first-class data type that is manipulated by the programmer. This expresses the view that a simulator serves a purpose similar to that of the core theorem prover within the context of the analogy above. As a first-class concept provided by `fvml` directly, simulation traces become values of an appropriate data type and a set of operations are provided to manipulate values of that data type, such as fine-grained control over advancing simulation. `fvml` is developed in this dissertation both formally and at the level of an implementation, and its utility is justified with pedagogical examples and the

presentation of larger case studies.

An example of a specific case where our meta-language may be brought to bear is as an alternative to highly targeted testing, such as when a directed test is created to discharge a difficult coverage goal. Targeted stimulus generation consumes a disproportionate amount of time in contemporary methodology. According to Tom Borgstrom, Program Director of the Verification Group at Synopsys:

Today it is not uncommon to go from 0% to 80% coverage in just a few days after the [constrained-random] testbench is up & running.

What about the remaining 20%?

Today, one of the long poles in verification is coverage convergence – the process where verification engineers analyze the coverage generated by constrained-random tests, identify gaps or “coverage holes”, and adjust the verification environment to try to fill the gaps. If you think this sounds laborious, repetitive and time-consuming you’d be correct. I’ve spoken to chip designers who say a third of their overall chip development schedule is spent in this iterative, largely manual, coverage convergence phase of verification.

[9, July 6, 2010]

Filling coverage holes that are part of the “remaining 20%” typically means either modifying constraints in a random testbench or simply writing a directed test. As the above quotation suggests, this is typically tedious work with substantial manual intervention by the verification engineer: viewing waveforms, determining possible changes to the test to better target the unfilled coverage goal, and then making those modifications. The alternative that `fvml` provides is the ability to write a program operating in a tight feedback loop attempting to find an appropriate stimulus and removing the need, as much as possible, for intervention by the engineer.

An example that we will employ in a subsequent chapter to illustrate this idea is a program that solves a digital circuit version of a “maze”. Instead of analyzing the circuit manually, determining the exact way out of the maze, and

then writing a directed test navigating that path, our meta-language allows a verification engineer to write a program that solves mazes in a general way through a backtracking-based search. This is an efficient, targeted strategy, is easier to write than a directed test because the internal structure of the maze does not need to be known, and is also resilient to changes in the configuration of the path of the maze.

An important side note that must be mentioned has to do with symbolic simulation. Our meta-language, as described above, is constructed around the idea that simulation becomes a first-class concept; however, simulation need not be limited in the usual way to just concrete stimuli. Advances in SMT solving technology make the use of general capabilities for symbolic simulation highly desirable, even in the context of a simulation-based verification methodology and without attempting formal verification. As one concrete example, expounded upon in a subsequent chapter, incorporating symbolic simulation provides the essential infrastructure to implement targeted testing strategies like the one described in [34], which serves as the basic idea underlying the popular Magellan tool from Synopsys (*e.g.* see [93, 92]), and can be constructed in just a few lines of code in our meta-language.

Regarding the meta-language for functional verification, this dissertation makes a number of contributions, derived largely from our set of works [53, 51, 52]. The specific contributions of this dissertation to functional verification are as follows:

1. It identifies a problem with contemporary functional verification methodology that is of substantial practical importance: the inability to effectively orchestrate the coverage closure feedback loop programmatically.
2. It proposes, as a way of remedying this problem, a programming language, or meta-language, for functional verification where simulation becomes a first-class concept.
3. It defines a formal semantics of such a meta-language, called `fvml`, as an embedded domain-specific language within rewriting logic, and which is, in addition, parametric on the hardware description language (HDL) used to develop the device-under-test.
4. It specifies an implementation of such a meta-language specialized for testing Verilog devices, called `vlogml`, as an embedded domain-specific

language in Haskell.

5. It provides a set of pedagogical `vlogm1` programs that demonstrate the novel capabilities made possible through our meta-language.
6. It provides two case studies that demonstrate `vlogm1` programs analyzing more substantial devices, including a bus-master controller and a simple microprocessor.

Implicit on the approach to functional verification and the meta-language features of `vlogm1` just summarized, there are two additional ideas that are developed in this dissertation:

- *parameterizability*: The meta-language is formalized in such a way that it is really a *parametric* language $[\mathcal{L}]m1$, where \mathcal{L} is the HDL of the device-under-test. For $\mathcal{L} = \text{Verilog}$, we obtain $[\text{Verilog}]m1$, or `vlogm1` for short. Similar instantiations can be developed for a variety of HDLs. This is important because it separates in a modular way the language used for *verification* purposes from the language used for *design*.
- *semantics-based*: That is, not only is the semantics of the parametric meta-language $[\mathcal{L}]m1$ based on the formal system of rewriting logic, but the HDL \mathcal{L} itself is not provided as a standard compiler or simulator (which would make it in general impossible to be used as a *symbolic* execution engine), but as an *executable formal specification* in rewriting logic.

The importance of being *semantics-based* and of directly using a *formal description* of the chosen HDL is that much greater confidence can be imparted in the tool supporting the instantiation of $[\mathcal{L}]m1$ when used for functional verification purposes. This is because what the tool is executing are the *formal definitions* of the semantics of the chosen HDL, which can be directly inspected and criticised at a high level, as opposed to a low-level implementation of the HDL which may be difficult to understand and debug and may be even inaccessible due to proprietary reasons.

For these reasons, our research on functional verification fits within the broader formal framework of the *rewriting logic semantics project* [77, 78], where rewriting logic is used to develop formal executable specifications of programming languages as rewrite theories, and such specifications are then

used to analyze programs in the given programming language in a variety of ways, such as through simulation, testing, model checking, static analysis, and theorem proving. Our $[\mathcal{L}]m1$ metalanguage approach specializes this general idea by: (i) restricting the focus to HDLs, and (ii) focusing on simulation-based functional verification as opposed to, say, model checking or theorem proving verification.

For all of the above reasons, a second important contribution of this thesis, largely derived from [55, 47, 49, 73], is the development of *formal semantic definitions in rewriting logic* for several important hardware design paradigms.

7. Verilog. This is one of the most widely used languages today to design digital circuits, and our formalization is, as far as we know, the most comprehensive of formalization effort to date, though many language features are omitted. In its capacity as an executable semantics all possible program behaviors can be explored and we have used this feature to give evidence as to the existence of bugs in widely-used simulators, some of which have since been fixed by the tool authors.
8. Production Rule Sets. This is a language used in the design of asynchronous digital circuits. We have given both a mathematical semantics and rewriting logic semantics and proved their equivalence through a strong bisimulation result. Both of the semantics also clarify the concept of hazardous execution, a crucial correctness property, and have used the executable semantics to automatically prove/disprove the existence of hazards, as well as deadlocks.
9. BTRS. This is simplified version of the Bluespec hardware description language that views hardware design from the perspective of a set of guarded atomic actions. As a rigorous structural semantics for BTRS already exists [21], the rewriting logic semantics is almost straightforward. This speaks however to one of the ideas of the rewriting logic semantics project, which is the suitability of rewriting logic as a semantic framework. We also demonstrate how to use the executable semantics to find deadlocks in BTRS programs.

The dissertation is organized into chapters as follows:

- *Chapter 1.* Identifies the problem motivating this dissertation and explains at a high-level the means through which we address it.

- (*Chapter 3*). Discusses related work, defines mathematical notation particular to this dissertation, and enumerates, via appropriate references, other needed background material such as in rewriting logic and the Haskell programming language.
- *Chapter 4*. This chapter presents the formalization of `fvml` within rewriting logic.
- *Chapter 5*. This chapter describes `vlogml`, an implementation of `fvml` in Haskell and specialized for analyzing Verilog devices.
- *Chapter 6*. This chapter presents several pedagogical examples of `vlogml` programs demonstrating the novel capabilities of our meta-language. The device that is analyzed is a digital circuit implementing a sort of maze.
- *Chapter 7*. This chapter presents two larger case studies, also using the above `vlogml`. One, an I²C bus-mastering controller, and the second, a small microprocessor.
- *Chapter 8*. This chapter presents a rewriting logic semantics for a substantial portion of Verilog.
- *Chapter 9*. This chapter presents a rewriting logic semantics for production rule sets, a language used in the design of asynchronous digital circuits.
- *Chapter 10*. This chapter presents a rewriting logic semantics for BTRS, a simplified form of Bluespec.
- (*Chapter 11*). This chapter presents some final thoughts and conclusions.

CHAPTER 2

PRELIMINARIES

This chapter presents some basic information that may help orient a reader unfamiliar with rewriting logic, Maude, or Haskell. It is not meant to be comprehensive, rather, it simply provides a few of the “basics”.

2.1 Rewriting Logic

This section reviews the basics of rewriting logic, Maude, and the rewriting logic semantics project.

2.1.1 Syntax, Proof Theory, and Model Theory

To precisely define our meta-language we employ *rewriting logic* [74]. One reason is that rewriting logic has been shown to be well-suited to defining the formal semantics of programming languages, which will be relevant at two levels: (1) at the level of the functionality provided by our meta-language, and (2) at the level of the design language. Our meta-language is made generic by being *parameterizable* on a formalization of the design language semantics. Then, using the fact that rewriting logic is *reflective* [16, 17], we can achieve the “meta” aspects of the testbench language in an elegant way.

Formally, rewriting logic is defined by its deduction system and model theory, as with other logics. It is parameterized by a suitable equational logic, *e.g.* unsorted, many-sorted, order-sorted, membership, so that a *rewriting logic specification* is defined as a triple (Σ, E, R) with (Σ, E) a signature and set of axioms of the underlying equational logic, and R a set of appropriately defined *rewrite rules*.

The structure of the rules can vary slightly with the underlying equational logic and whether or not one considers *conditional* rules. The essential idea

in all cases is that a rewrite rule specifies an ordered pair of patterns (s, s') with the intuition that any instance, say $\theta(s)$, of s , where θ instantiates the parameters of the patterns s and s' and is called a *substitution*, can be dynamically transformed into a corresponding instance $\theta(s')$ of s' . A rewrite rule (s, s') is typically written more suggestively as

$$s \longrightarrow s'.$$

Common presentations either define rewrites directly on *terms* formed from the symbols of Σ and a set of variables, or else on E -equivalence classes of terms formed by deduction in the underlying equational theory. In the case of the Maude rewriting logic language and tool [15], the underlying equational logic is *membership equational logic* [76] and rewrite rules are defined at the term level with conditions constructed as a conjunction of atomic rewrite, equation, and membership formulae.

For expository purposes going forward, we will consider only the unsorted, unconditional case, and define rewriting on equivalence classes of terms. The presentation we will give largely follows that of [75]. Under these assumptions we will first define what a rewriting logic specification is and then define the logic's deduction system and model theory.

A rewriting logic specification $\mathcal{R} = (\Sigma, E, R)$ is any triple with (Σ, E) an (unsorted) equational logic specification consisting of an equational signature Σ and a set of equations, E , of the form $t = t'$, with $t, t' \in T_\Sigma(X)$, the set of well-formed terms involving the symbols of Σ and with variables drawn from some fixed set X ; and where the rewrite rules are just any set $R \subseteq T_{\Sigma, E}(X)^2$, where $T_{\Sigma, E}(X)$ denotes the set of E -equivalence classes of terms defined by equational deduction from E . For a given term $t \in T_\Sigma(X)$, we consider the E -equivalence class of t , $[t] \in T_{\Sigma, E}(X)$, to be defined such that $[t] \stackrel{def}{=} \{t' \in T_\Sigma(X) \mid E \vdash t' = t\}$.

Deduction in this variant of rewriting logic establishes sequents of the form

$$(\Sigma, E, R) \vdash [t] \longrightarrow [t']$$

from the finite application of the following inference rules:

1. reflexivity: for each $[t] \in T_{\Sigma, E}(X)$

$$\frac{\cdot}{[t] \longrightarrow [t]}$$

2. congruence: for each $n \in \mathbb{N}$ and $f \in \Sigma$ of arity n

$$\frac{[t_1] \longrightarrow [t'_1] \quad \cdots \quad [t_n] \longrightarrow [t'_n]}{[f(t_1, \dots, t_n)] \longrightarrow [f(t'_1, \dots, t'_n)]}$$

3. replacement: for each $([t(x_1, \dots, x_n)], [t'(x_1, \dots, x_n)]) \in R$

$$\frac{[w_1] \longrightarrow [w'_1] \quad \cdots \quad [w_n] \longrightarrow [w'_n]}{[t(w_1, \dots, w_n)] \longrightarrow [t'(w'_1, \dots, w'_n)]}$$

4. transitivity

$$\frac{[t_1] \longrightarrow [t_2] \quad [t_2] \longrightarrow [t_3]}{[t_1] \longrightarrow [t_3]}$$

The model theory of rewriting logic is given by the notion of an \mathcal{R} -system (e.g. see [75]). Again, there is some variation depending on the underlying equational logic and the form of the rewrite rules. Consider an unsorted, unconditional rewrite specification $\mathcal{R} = (\Sigma, E, R)$. An \mathcal{R} -system consists of (1) a category \mathcal{S} , (2) for each $n \in \mathbb{N}$ and $f \in \Sigma$ of arity n , a functor $f_{\mathcal{S}} : \mathcal{S}^n \longrightarrow \mathcal{S}$, and (3) for each $([t], [t']) \in R$ a natural transformation $r_{\mathcal{S}} : t_{\mathcal{S}} \Rightarrow t'_{\mathcal{S}}$ where $t_{\mathcal{S}}, t'_{\mathcal{S}}$ are functors defined in the natural, inductive way from the $f_{\mathcal{S}}$ functors. Additionally, the $f_{\mathcal{S}}$ functors must satisfy the property that for any equation $t(x_1, \dots, x_n) = t'(x_1, \dots, x_n) \in E$, $t_{\mathcal{S}} = t'_{\mathcal{S}}$. The semantics of Maude system modules are essentially given by the initial \mathcal{R} -system corresponding to the rewrite specification \mathcal{R} given by the module.

Detailed accounts of rewriting logic can be found in [74, 75, 11]. For membership equational logic, see [76]. Reflection in rewriting logic and a number of equational logics, including membership equational logic, is addressed in [16, 17]. For details on the Maude rewriting logic language, see [15] and the explanations below.

2.1.2 Maude

Maude [15] is an executable rewriting logic language with an assortment of built-in automated formal analysis tools, including invariant checking via breadth-first search [15, Ch. 12] and linear-temporal logic model checking [15, Ch. 13]. The underlying equational logic used is membership equational logic [76], and also supported is the use of conditional rewrite rules. The basic unit of organization within Maude source code is a *module*, of which the two main types are *functional modules* [15, Ch. 4], which correspond to equational specifications (empty set of rewrite rules), and *system modules* [15, Ch. 6], which (typically, though not necessarily) contain a non-empty set of rules, and therefore correspond to full rewrite theories.

An example functional module that uses quite a few features of Maude is the following module axiomatizing sets of elements from a space of four elements.

```
fmod SET is

  sort Elem .

  ops a b c d : -> Elem .

  sort Set .
  subsort Elem < Set .
  op empty : -> Set .
  op __ : Set Set -> Set [assoc comm id: empty] .

  vars X : Elem .
  vars XS : Set .

  eq X X = X .

endfm
```

Notice that two sorts have been declared using the keyword `sort`, `Elem`, for elements of the sets that we are defining, and `Set`, for the sets themselves. The keyword `op` or `ops` introduces new symbols into the signature of the specification with the number of arguments given and implying appropriately defined axioms on the domain and co-domain of that symbol to enforce the

sort constraints.

Subsorting, which is implied by the use of membership equational logic, is allowed through the `subsort` keyword. In this case, we use subsorting to conflate an element and the singleton set containing that element. The juxtaposition operator `_` is annotated with *equational attributes* [15, §4.4.1], which Maude can use to perform rewriting modulo. Our set constructor, corresponding to set union, is above being declared associative, commutative, and having as its identity element the empty set, `empty`. Lastly, we define an equation corresponding to the idempotency property of sets; equations are introduced with the keyword `eq`.

As a result, when the above module is loaded into Maude, it can correctly deduce that the set $\{a, d, a, b, a, c\}$ is the same as $\{a, b, c, d\}$. To see this, we can execute the `reduce` command [15, §4.9] at the interactive prompt:

```
Maude> reduce a d a b a c .
reduce in SET : a d a b a c .
rewrites: 2 in 0ms cpu (0ms real) (~ rewrites/second)
result Set: a b c d
```

System modules, unlike functional modules, allow one to use the keyword `rl` to introduce rewrite rules. For example, we can extend the above module in such a way that we allow `b` to be rewritten to `a`.

```
mod GOODBYE-B is
  including SET .

  rl b => a .

endm
```

Now, when we reduce the earlier set, we still get the same result, as `reduce` does not apply rules.

```
Maude> reduce a d a b a c .
reduce in GOODBYE-B : a d a b a c .
rewrites: 2 in 0ms cpu (0ms real) (~ rewrites/second)
result Set: a b c d
```

However, using the command `rewrite`, our newly added rules is applied and we get the expected result.

```
rewrite in GOODBYE-B : a d a b a c .
rewrites: 4 in 0ms cpu (0ms real) (~ rewrites/second)
result Set: a c d
```

Modules, therefore, define rewriting logic specifications, perhaps with an empty set of rules, which for functional modules, is of course always the case. These modules are expected to satisfy certain properties in order to guarantee that they are sufficiently executable within the Maude interpreter and that soundness is guaranteed from the analyses performed by some of Maude's built-in tools. These properties are called *admissibility requirements* in the context of Maude. In particular, the equational part of a specification is typically expected to be confluent and terminating modulo the equational axioms [15, §4.6], and the rewriting rules should be coherent with respect to the equations [15, §6.3].

There are a number of built-in and add-on tools to perform formal analysis of rewriting logic specifications written in Maude. In particular, Maude has built-in support for breadth-first search [15, Ch. 12] and linear-temporal-logic model checking [15, Ch. 13]. As a very simple application of Maude's `search` command, we can observe that from the set $\{b, d\}$ we can reach a set through rewriting that contains a , but we cannot reach a set that contains c . The first case comes from executing

```
Maude> search b d =>* a X:Set .
search in GOODBYE-B : b d =>* a X:Set .

Solution 1 (state 1)
states: 2 rewrites: 1 in 0ms cpu (0ms real)
(~ rewrites/second)
X:Set --> d

No more solutions.
states: 2 rewrites: 1 in 0ms cpu (0ms real)
(~ rewrites/second)
```

And, the second case simply searches for a set with c as an element, ultimately finding no solution.

```
Maude> search b d =>* c X:Set .
search in GOODBYE-B : b d =>* c X:Set .

No solution.
states: 2  rewrites: 1 in 0ms cpu (0ms real)
(~ rewrites/second)
```

2.1.3 Rewriting Logic as an Executable Semantics Definitional Framework

The idea of the rewriting logic semantics project [77, 19, 78] is to use rewriting logic as the mathematical foundation for giving formal semantic definitions to programming languages. This idea has both practical and conceptual advantages over other frameworks. For example, it unifies denotational and operational-style semantics in a useful way, and with executable rewriting logic languages such as Maude [15], it is possible to get an interpreter from the semantic definition without any additional work, a very practical benefit. In addition to execution, there are more traditional formal analysis tools that apply to *any* language formalized within rewriting logic, and which may be available to a user once a language is formalized, for example, reachability analysis to search for failures of invariants is made available, as is LTL model checking.

Given a programming language \mathcal{L} , the semantics of \mathcal{L} is given as a *rewriting logic specification* $\mathcal{R}_{\mathcal{L}} = (\Sigma, E, R)$ where the signature Σ characterizes the *syntax* of \mathcal{L} , the equations E characterize the *deterministic* aspects of \mathcal{L} , and the rewrite rules R characterize the *concurrent* aspects of \mathcal{L} . For example, a simple concurrent language might use rules for every load or store of a variable, but equations for all other features, such as the processing of if-statements, sequential composition, and while loops, all of which would not be observable between threads. In Maude notation, this could be expressed (roughly) as follows


```
eq eval(if true  then S1 else S2, M) = eval(S1, M) .
eq eval(if false then S1 else S2, M) = eval(S2, M) .
eq eval(S1 ; S2, M) = eval(S2, eval(S1, M)) .
rl eval(X := V, M) => M[V / X] .
```

Conceptually, `eval` is an evaluation function for a *program* and a *state*, where a state is just a *map* from variable identifiers to values (M). Note that the first three lines are *equations*, *i.e.* elements of E , whereas the fourth line where we define what it means to do a variable assignment, is a *rule*, *i.e.* an element of R .

The canonical approach to rewriting logic semantics is given in [77]. Many traditional operational-semantic styles, such as big-step and small-step structural operational semantics and reduction semantics, can also be embedded into rewriting logic in an elegant way, as shown in [19]. An extensive definition of Scheme using the rewriting logic semantics approach can be found in [20]; many other examples of language definitions are cited in [77].

2.2 Haskell

Haskell is a functional language based on a polymorphic lambda calculus, noted especially for its non-strict semantics, type classes, purity, and its use of “monads”. Of course, while it is impossible to provide a comprehensive overview of the language in this section, we do attempt to impart some of the syntax as well as a particular aspect of Haskell that `vlogml` relies on, called the “state monad”.

In some sense, Haskell is very similar to other contemporary functional programming languages. A programmer will typically begin by defining some algebraic data types, for example

```
data Tree a = LEAF a | BRANCH (Tree a) (Tree a)
```

defines a polymorphic data type, `Tree a`, for binary trees. The symbol `a` is a type *variable*, and can be instantiated with any type, for example

```
BRANCH (LEAF 1) (BRANCH (LEAF 2) (LEAF 3))
```

represents a value of type `Tree Int`, since the leaves contain integer values 1 – 3. The tokens `LEAF` and `BRANCH` are called *constructors*.

Once a data type is defined, a programmer can write functions that expect arguments with that type, for example we can write a (polymorphic) function that determines the height of a tree

```
height (LEAF _)      = 0
height (BRANCH x y) = 1 + max (height x) (height y)
```

This function uses pattern matching on the constructor symbols to distinguish the two relevant cases. The “_” acts as a place holder when we do not need the argument, such as in this case where we simply need to know that we are at a leaf node. As usual in functional programming, recursion is used heavily.

Haskell is statically typed and the type of the `height` function can be automatically inferred, but one can give a type explicitly if desired, for example to aid in readability.

```
height :: Tree a -> Int
height (LEAF _)      = 0
height (BRANCH x y) = 1 + max (height x) (height y)
```

Note that the `->` symbol is used to construct function types, in this case a polymorphic function from the (polymorphic) type representing trees to the integer type. Haskell uses *currying* notation, so that a function of two arguments would be given as

```
f :: a -> b -> c
```

The above concepts cover only the very basics, demonstrating the syntax for introducing new types, defining functions, and the syntax for ascribing a type to a function. Of course, Haskell has a multitude of other features, as well as many predefined data types and so forth. The goal is just to provide some basic familiarity so that when we introduce the concepts crucial in the implementation or use of `vlogml`, they are somewhat grounded.

One such concept is Haskell’s “state monad”. It consists of a type and a set of “monadic” and other operations on that type, and is used to simulate *stateful* computations. Let us start with just the type, which is defined as

```
data State s a = STATE (s -> (a,s))
```

where $(,)$ is the type constructor for pairs. The `State` type constructor therefore takes two arguments, the first one being the type variable `s`, intuitively corresponding to *state* of the computation, and the second one being `a`, which will correspond to the *result* of the computation. The single constructor, `STATE`, simply boxes a function of type

$$s \rightarrow (a, s)$$

That is, a value of type `State s a` is essentially just a function taking some initial state to a resulting value and the updated state.

Using Haskell’s “do”-notation, one can write code that appears much like imperative code. Consider the type `State Int Bool`, essentially just

$$\text{Int} \rightarrow (\text{Bool}, \text{Int})$$

corresponding to a stateful computation whose backing state is a single integer and which returns a boolean result. To demonstrate the do-notation, let us write a simple function that first stores a value into the state, say 11, second, retrieves the value from the state, and third checks whether the value retrieved is greater than 10, returning true if it is, and false if not. This is done as

```
stateGreaterThanTen :: State Int Bool
stateGreaterThanTen = do
  put 11
  x <- get
  return (x > 10)
```

The `do` construct effectively says to execute the following statements in order, one after the other, carrying along the state. The functions `put` and `get` are pre-defined with `State`, they swap-in a new state and retrieve the current state, respectively. In addition, the do-notation hides the use of the “monadic” operations pre-defined with `State`. These operations ensure that the sequencing of operations occurs in the expected way. `stateGreaterThanTen` is therefore a constant function that returns the value `(True, 11)`.

In `vlogml`, the main data type is a “state monad” of a slightly more complex variety. The particulars of this data type are described in the following section, which should also help to further clarify how “stateful” computations are used in Haskell.

Finally, it is important to emphasize that, unlike some other functional languages, these stateful computations maintain referential transparency. That is, they obey the expected properties of mathematical functions.

CHAPTER 3

RELATED WORK

Broadly speaking, this dissertation makes contributions in two distinct areas, the first being the functional verification of digital hardware, and second being the formal specification of the semantics of hardware design languages. This chapter correspondingly splits related work across a pair of sections, with Section 3.1 covering related work on functional verification, paying special attention to the way in which different kinds of languages may be used to address the verification burden, and with Section 3.2 addressing related work concerning the semantics of hardware description languages.

3.1 Functional Verification

Perhaps surprisingly, the use of *programming language techniques* to address inefficiencies and issues of quality in the functional verification process has not been extensively investigated within the academic research community. However, functional verification of hardware as a whole is a well-established field. Most academic work on functional verification concerns methodology and algorithmics, for example addressing scalability of formal verification (*e.g.* [12]) or automatically generating assertions (*e.g.* [97]). Industrial players, on the other hand, have put substantial effort into the design of programming languages used for functional verification, especially the *hardware verification languages* *e* [39], OpenVera [91], and SystemVerilog [38].

The lack of a “research community” around language-based approaches to functional verification means that there is no commonly understood narrative that explains the state of the art. As such, the narrative we present here represents a rather unique way of characterizing the effect of programming languages on contemporary functional verification. It centers around a categorization of relevant programming languages into a set of four categories that

reflect methodology (simulation versus formal verification), and the extent to which the language was designed with functional verification as a goal. The four categories are given according to the primary *intended purpose* for which they were designed

1. *hardware verification through simulation* (e.g. SystemVerilog);
2. *hardware verification through formal proof* (e.g. reF^lECT);
3. *hardware design* (e.g. Verilog);
4. *primary usage falls outside the hardware domain* (e.g. C++).

Indeed, the lack of a cohesive research community in this area also means that the language situation is exceptionally complicated with regards to terminology and concepts, and it bears warning that many relevant concepts are commonly used in an imprecise way; even the notion of language is somewhat difficult to ascertain exactly. For example, SystemVerilog is discussed below as a language of class (1) above, while Verilog is discussed as a member of class (3). At the same time, though, Verilog is a proper sub-language of SystemVerilog. The distinction we endeavor to make is in the *intended* use of the language features that are most closely associated with the language name, but it should be noted that this intention is based simply on contemporary usage amongst the loose collection of professional engineers and researchers who make up the design verification engineering community.

The languages of class (3) predate and led to the development of the languages of the first two classes, and so we begin with class (3). The two most prominent languages that occupy this category are Verilog [36] and VHDL [37], which are also the two most widely-used design languages. There are two kinds of effects that design languages have on functional verification: a *direct* kind and an *indirect* kind.

The direct effect is that design languages are often used not only for design, but also to construct a testbench which will not be synthesized into hardware. One of the main reasons that design languages are attractive in this second role is that one avoids any interoperability problems that arise from having two different languages that need to work together in concert. A second reason is that designers and verification engineers are often the same person, and mastering one language is simpler than mastering multiple

languages. Both of these are valid and understandable advantages of making dual use of a design language for verification.

The indirect effect is more subtle but widely understood in the context of software. Higher-level languages lead to more natural designs and fewer bugs. As an example, one of the reasons modern programmers often prefer Java over a language such as C is that the language completely removes the possibility of certain kinds of memory bugs. In the context of hardware design, a good, modern example of this is the Bluespec language [8], whose embrace of high-level programming concepts reduces the verification burden; for example, through promotion of a richer type system that is statically checked.

Of course, the main downside of using a design language for verification purposes is that verification-specific language features are often omitted or added as an afterthought. In addition, when verification-specific features are included, an unfortunate side-effect often occurs where an ill-defined subset of the language becomes synthesizable into hardware, and many legal programs are rejected by synthesis tools. This can cause confusion and may not even be consistent across synthesis tools. Indeed, one often hears engineers speak of a “synthesizable subset” of Verilog, which has exactly those problems mentioned.

Let us now return to class (1) and class (2) languages. The need for verification-specific language features, separate from those used for design, led to the development of class (1) languages. This class consists primarily of the *hardware verification languages e* [39], OpenVera [91], and SystemVerilog [38].

SystemVerilog was largely based on OpenVera, and so the two languages share essential language features related to functional verification. In particular, the two languages add support for general-purpose programming concepts which, unlike design-oriented languages, are not meant to be synthesized and only run during simulation. The most apparent example is object-oriented programming, which is seen to increase productivity and correctness when developing verification code. In addition, these languages add several important features highly tailored for functional verification, including built-in support for constrained-randoms, functional coverage, and assertions.

The other hardware verification language mentioned, *e*, has a reputation as a very thoughtfully developed language for verification purposes, and is especially noted for its inclusion of aspect-oriented programming [58] to

allow for different testing scenarios to be pieced together easily. Indeed, OpenVera subsequently was extended with aspects, due to their success in *e*. *e* also supports object-oriented programming, constrained-randoms, functional coverage, and assertions.

Compared to `fvml`, the language for functional verification developed in this dissertation, the existing class (1) languages just described are markedly different. Conceptually, all treat a testbench as an *environment* in which the device under test operates, rather than as a programmatic way of orchestrating multiple simulations; that is, none consider simulation as a first-class concept. In addition, none allows for the use of symbolic simulation during testing.

As far as we are aware, the only language whose primary intended purpose falls into class (2) is one used internally by Intel and based around the *reFlect* language described in [32]. *reFlect* augments the λ -calculus with a quote/anti-quote mechanism to get, as its name suggests, *reflection* capabilities. The emphasis on reflection in the language suggests the possibility of simulation as a first-class notion, and so at first glance may appear to duplicate our offering.

The two languages are, however, entirely different, with the difference being in intended *use*. The intended use of *reFlect* has always been formal verification of hardware using *theorem proving* techniques, and mostly uses reflection to get a first-class representation of the device for reasoning purposes. Our language, on the other hand, looks to investigate the usefulness of combining a first-class notion of *simulation* (which is not the same as the device being simulated) with declarative programming and automated constraint solving as a way to enhance contemporary *simulation-based* methodology, as opposed to targeting a formal verification regime. Therefore, we do not claim that our contribution is the first to think of employing meta-level ideas in the context of hardware verification. What we *do* claim is that it is the first to conceptualize and investigate the utility of the idea of providing simulation as a first-class notion at the language-level, a meta-level idea, as a way to improve the productivity and quality of simulation-based functional verification engineering efforts.

That said, it is worth noting that *reFlect* could have been used as a framework for testing the ideas we have set forth. We could have embedded `vlogm1` into *reFlect* instead of Haskell, for example, or given our formal definition of `fvml` within the framework described in [32], instead of as we did, using rewriting logic. These would have been legitimate choices, and [32]

does indeed provide an elegant framework, but would have been subject of course to some trade-offs. In particular, the semantic framework capabilities of rewriting logic are crucial to the parametric character of our language formalization (see Chapter 4).

As a final note regarding class (2) languages, we should mention ACL2 [54], although it does not fit into this class especially well because it was not designed with the express purpose of being used for hardware verification. As we noted above, though, the situation with related work in the area does not lend itself well to rigid interpretation. Be that as it may, ACL2 has been used extensively to prove parts of microprocessors correct, most notably at AMD [89]. Again, this is quite different from our offering, which emphasizes simulation-based verification.

Class (4) languages commonly include C++ [40] and Python [87], among other general-purpose programming languages. The use of these languages is sensible for a variety of reasons, especially in the development of checking programs that are specialized to assess the correctness of a device's output, which varies wildly from device to device and is often most easily expressed in general-purpose programming languages. Of course, these languages are less useful for generating stimuli and monitoring coverage than the hardware verification languages, and compared to `fvml` they do not provide any specific help in the orchestration of multiple simulations.

The above languages are the most relevant work related to `fvml`. Of course, a vast array of additional work related to many different aspects of functional verification has also been done, and is relevant insofar as these works endeavor toward the same high-level goal: to ease the burden of functional verification and produce devices with fewer bugs. They are different from our work in that they address some aspect of functional verification other than the general problem of designing a language for simulation-based testbench development; although, in some cases there are interesting implications pertaining to our particular problem. We mention a few in the remainder of this section, just to give a flavor of some of the areas that have been researched.

Intel has developed their own formal verification environment [44, 32] that is used as part of the overall functional verification process. AMD does something similar using ACL2 [89]. Synopsys sells an automatic property checking (and stimuli generation) tool called Magellan [34, 93, 92], whose underlying ideas are also used in non-hardware contexts [63]. Private com-

panies such as Jasper Design Automation sell formal verification tools, and formal-methods-based tools, although detailed information about such proprietary tools is obviously quite limited. Substantial work on the generation of random stimuli (*e.g.* [61, 1]), equivalence checking (*e.g.* [86, 13]), coverage metrics (*e.g.* [14]), automatic assertion generation (*e.g.* [97]), and reuse exist. We also mention the work of [59, 60] which was an early debugging tool that allowed user control over symbolic simulation, though it was not designed for use in hardware verification and was interactive, rather than controlled through the use of a meta-level programming interface.

3.2 Rewriting Logic Semantics

The rewriting logic semantics project, including its goals, ideas, and accomplishments, is reviewed in detail in three papers [77, 19, 78]. The essential idea of the project is to define programming languages, process calculi, and the like in a mathematically rigorous and unambiguous way, and to develop language-agnostic analysis tools. These ideas come out in various ways in this dissertation; for example, to check deadlock of BTRS programs. The remainder of this subsection reviews related work regarding the three hardware description languages that we have formalized as part of this dissertation.

Regarding Verilog, in [28], Michael Gordon presents a formal semantics for a simplified version of Verilog called V. V does not deal with many of the features of Verilog that our definition does (such as value sizing). Additionally, it uses new terminology rather than that of the standard. While the syntax described in the paper is formal, the semantics, as presented, is primarily in English language form. Additionally, the semantics presented is not executable, making it more difficult to ask questions about what output a given program should produce.

Gordon Pace and Jifeng He present a brief formal semantics of Verilog in [84]. While completely formal, the definition they present does not cover several major features of Verilog, such as non-blocking assignments or handling the intricacies of bitvectors. Additionally, their semantics is not executable.

In [90], Hisashi Sasaki presents a semantics for Verilog in terms of abstract state machines. Executability of the definition is not discussed, and it does not capture the inherent nondeterminism of Verilog, which we feel is one of

the most important issues to understanding Verilog.

In [105], Huibiao Zhu, Jifeng He, and Jonathan Bowen present an algebraic semantics of Verilog, which they use to derive a denotational semantics. Their semantics cover a smaller subset of the language than He’s earlier work in [84]: not even net assignments are covered.

Of the definitions that we have found, ours covers the largest number of constructs in the Verilog language, including importantly its inherent nondeterminism. Our definition is also far from complete; for example we do not define the semantics of tasks. However, we believe that our semantics is the most complete to-date and covers many of the most widely used language features. Additionally, to the best of our knowledge only our semantics is executable, allowing for experimentation with Verilog programs.

The work on production rule sets 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.

The first topic is the semantics of production rule sets. Of course, the current work improves upon our own earlier efforts in [55, 47]. The current efforts, including both the mathematical 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 other goal, such as Martin’s synthesis method [65, 67]. However, we believe that a clear and rigorous semantic reference is itself an important goal, and ultimately can reduce fragmentation and misunderstanding when undertaking problems that rely on having a semantics.

Martin has also used his semantics in an auxiliary way to prove that the scope of possible circuits under delay-insensitivity is limited [66] and that quasi delay-insensitive circuits are Turing-complete [64]. The semantics from [65, 67] was also examined in [85] in order to clarify the relationship between production rule sets and corresponding physical circuit implementations. We have also used an earlier version of the semantics to prove properties about the relationship of devices under different timing assumptions [55].

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 [47], 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.

Regarding hazard freedom, it is known that (proprietary) tools based on Monte Carlo methods exist and are used in practice. In addition, we know of a few other works that attempt to exhaustively prove the absence of hazards in asynchronous circuits (though not necessarily designed directly for production rule sets).

The methods developed in [4] 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. [96] uses the modern program analysis technique of abstract interpretation to reason about hazards in asynchronous circuits. [102] uses a standard symbolic model checker to verify hazard free operation of speed-independent circuits, and an older tool called `prlint` [18] 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 [57], and a number of works, *e.g.* [80, 12, 88, 103, 104] 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.

Regarding the Bluespec semantics, our work is based on a previous, structural-operational semantics (SOS), given in [21].

CHAPTER 4

FORMALIZATION

This chapter presents a formalization of two things: a problem, the high-level problem addressed in this dissertation, and a proposed solution to that problem, which constitutes the main contribution of this dissertation. More specifically, we formalize two notions:

1. *verification closure*, expressed here in a simplified fashion as the problem of generating a set of simulation traces satisfying a collection of coverage goals.
2. *meta-language for functional verification*, `fvml`, which we see as an effective mechanism for attaining verification closure.

Both of the above items will be formalized within the mathematical framework of rewriting logic [74, 75, 10], which provides two important benefits. First, it will allow the meta-language to be parameterized over a hardware design language, and thereby made generic with respect to the language chosen to implement the device-under-test. Second, it allows us to structure our meta-language as an embedded domain-specific language, where general purpose programming features are had through rewriting logic itself.

4.1 Overview

Our formalization effort involves four separate rewriting logic specifications,

$$(1) \mathcal{R}_{HDL} \quad (2) \mathcal{R}_{META} \quad (3) \mathcal{R}_{IR} \quad (4) \mathcal{R}_{STRAT}$$

organized according to the relationships depicted in Figure 4.1, wherein $\bar{\mathcal{R}}_{HDL}$ denotes a suitable meta-representation of \mathcal{R}_{HDL} as an object-level term, as

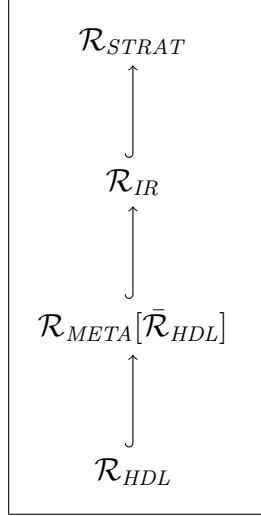


Figure 4.1: Structure of \mathcal{R}_{HDL} , \mathcal{R}_{META} , \mathcal{R}_{IR} , and \mathcal{R}_{STRAT} .

described in detail below in Section 4.6. The first three specifications

- (1) \mathcal{R}_{HDL} (2) \mathcal{R}_{META} (3) \mathcal{R}_{IR}

are provided *to* the user and together comprise the *domain-specific* part of `fvml`. The new language features, which consist of four “*inference rules*”, are exported through \mathcal{R}_{IR} . \mathcal{R}_{HDL} and \mathcal{R}_{META} are mostly hidden from the user, but are essential to formalize the semantics of the operations provided by \mathcal{R}_{IR} . \mathcal{R}_{HDL} is a *rewriting logic semantics* for the hardware description language used to implement the device-under-test, and \mathcal{R}_{META} is used to represent rewriting logic proofs, a *meta-level concept*, at the object level. Together \mathcal{R}_{HDL} and \mathcal{R}_{META} allow us to consider and manipulate an appropriate notion of *device simulation* as a first-class concept in our meta-language.

The fourth specification

- (4) \mathcal{R}_{STRAT}

is created *by* the user and constitutes a program that has been written in our meta-language, `fvml`. \mathcal{R}_{STRAT} employs the domain-specific features of \mathcal{R}_{IR} , as well as the general-purpose programming facilities of rewriting logic, to construct a *program*, or “*strategy*”, in the `fvml` language that exercises the device-under-test.

The essential point to understand is that the *general-purpose* programming facilities of rewriting logic are added to the *domain-specific* operations provided by \mathcal{R}_{IR} , including \mathcal{R}_{HDL} and \mathcal{R}_{META} , to form our *embedded domain-specific language* `fvml`. That is, roughly speaking:

$$\text{fvml} \stackrel{\text{def}}{=} [[(1) \mathcal{R}_{HDL} \quad (2) \mathcal{R}_{META} \quad (3) \mathcal{R}_{IR}] + [\textit{rewriting logic}]]$$

And, additionally, it is crucial to note that \mathcal{R}_{STRAT} is just a program written in `fvml`. Each of the four specifications, \mathcal{R}_{HDL} , \mathcal{R}_{META} , \mathcal{R}_{IR} , and \mathcal{R}_{STRAT} , as well as their respective purpose and structure, is detailed in subsequent sections.

4.2 Parameterization

A crucial assumption we make in our formalization is the existence of a suitable \mathcal{R}_{HDL} , which is taken to be a rewriting logic semantics of a programming language used to implement digital hardware, also called a *hardware description language*. The advantage of structuring our meta-language formalization this way is that \mathcal{R}_{HDL} becomes a *parameter* of it, and thereby `fvml` is made applicable to *any* digital circuit designed in a programming language that can be formalized within rewriting logic. If we are interested in using our meta-language to exercise, for example, an Ethernet MAC designed in Verilog, then the assumption is that we possess an \mathcal{R}_{HDL} formalizing the semantics of Verilog; `fvml` itself does not need to be changed to accommodate Verilog.

Substantial evidence exists which validates this \mathcal{R}_{HDL} assumption. This dissertation, in particular, provides rewriting logic semantics for three languages used to design digital hardware in Chapters 8 – 10. Many other programming languages, covering a range of paradigms, have been formalized in this way, including substantial portions of Java [26], Scheme [71], and C [24], just to name three. [77, 19, 78] provide comprehensive accounts of work contributing to this project, appropriately called “the rewriting logic semantics project”.

4.3 Analogy

An analogy with theorem proving is helpful to understand the meta-language and the arrangement of the rewriting logic specifications depicted in Figure 4.1. The source of the analogy is the combination of ML+LCF [29]. ML was originally conceived as a meta-language for orchestrating proof-construction in LCF while searching for a particular proof of interest. It acknowledged a very similar problem to what we view as a serious hindrance to simulation-based verification: that to effectively investigate the search space, in one case proofs and in the other simulations, one needs a *programmatic* way to generate and interrogate the items in that search space as first-class objects. The various relationships with our meta-language are summarized in Table 4.2; \mathfrak{D} denotes a device, and \mathcal{Q} denotes a set of coverage goals. For readers not intimately familiar with the construction of that theorem proving system, it may be best to quickly read through the LCF analogy and then return to it for a second reading after going through the content in the remainder of this chapter.

ML+LCF	\iff	fvml	
LCF	\iff	\mathcal{R}_{IR} (and \mathcal{R}_{META})	(1)
ML	\iff	rewriting logic	(2)
$\Gamma \vdash \varphi$	\iff	\mathcal{R}_{HDL} and $\mathfrak{D}, \mathcal{Q}$	(3)
Γ	\iff	\mathcal{R}_{HDL}	(4)
φ	\iff	$\mathfrak{D}, \mathcal{Q}$	(5)
Tactic	\iff	\mathcal{R}_{STRAT}	(6)
$\frac{\begin{array}{c} \vdots \\ \vdots \\ \dots \\ \vdots \end{array}}{\Gamma \vdash \varphi}$	\iff	$\{\mathcal{R}_{HDL} \vdash s_{\mathfrak{D},0} \longrightarrow s'_j\}_{j < m}$	(7)

Figure 4.2: Analogy between ML+LCF and fvml.

The following describes the relationships, as marked in the above table, in more detail:

1. the core inference rules provided by LCF are analogous to the simulation-generating rules of \mathcal{R}_{IR} ;
2. as an embedded language, the general-purpose programming facilities of ML are analogous to those inherited from rewriting logic in fvml;
3. the target judgment for theorem proving is best considered, for the purposes of the analogy, in terms of its components;

4. \mathcal{R}_{HDL} can be seen as a set of axioms from which reasoning takes place;
5. The goal of the reasoning in `fvml` is to witness to a particular set of coverage goals, \mathcal{Q} , with respect to a device, \mathcal{D} ;
6. in order to attain this goal, a user writes a program, or tactic;
7. the eventual result of the enterprise, upon success, is a witness of the goal: a proof in the case of `ML+LCF`, and a set of simulation traces covering \mathcal{Q} in the case of `fvml` ($s_{\mathcal{D},0}$ denotes the initial state of the device, and s'_j some future state).

Consider wanting to prove a theorem about group theory in `LCF` for example, that

$$\Gamma_{GROUP} \vdash x \cdot 1 = x$$

The first thing that we need is Γ_{GROUP} , an axiomatization of group theory. This set of axioms naturally corresponds to \mathcal{R}_{HDL} ; it is just that the theorems that follow from \mathcal{R}_{HDL} can be viewed as simulations. `LCF` must ensure that any deductions used to arrive at $x \cdot 1 = x$ are sound, which is accomplished by only allowing proofs to be generated through the core set of inference rules provided by the `LCF` prover interface; in `fvml` the corresponding deduction rules are given by \mathcal{R}_{IR} . Finally, \mathcal{R}_{STRAT} corresponds to the user's `ML` program, which is simply an attempt by the user to program a strategy to find a proof of $\Gamma_{GROUP} \vdash x \cdot 1 = x$. \mathcal{R}_{META} does not have a particularly natural analogue within the theorem proving context, though it roughly corresponds to the data-types used to represent proofs.

In the remainder of this chapter, we first formalize the verification closure problem using concepts from rewriting logic. With this formalization in mind, our meta-language for functional verification is developed in such a way that it clearly addresses verification problems of that type. The formalization of the meta-language is split into subsections, one for each of the four rewriting logic specifications noted above. Finally, we provide a simple example demonstrating a program in this language and at the rewriting logic level, using the syntax of Maude.

4.4 \mathcal{R}_{HDL}

\mathcal{R}_{HDL} denotes a rewriting logic specification capturing the semantics of a programming language suitable for digital hardware design. The main purpose of it within our meta-language formalization is to provide a straightforward way to speak about *simulation traces*. Both of our formalization problems ask questions about simulation traces: *does a particular trace satisfy a coverage goal? what strategy should be used to generate simulation traces?* In the theorem proving analogy, \mathcal{R}_{HDL} is Γ , the set of axioms from which judgments $\Gamma \vdash \varphi$ are established.

The existence of \mathcal{R}_{HDL} is a crucial assumption made by our formalization effort. This assumption allows the meta-language to be made generic in the sense that it is applicable to any hardware device, so long as that device is designed in a language that can be formalized in rewriting logic; and we have yet to come across one that is not amenable to formalization in rewriting logic. Some familiar examples of languages that have been given (sometimes partial) semantics using rewriting logic include Verilog, Production Rule Sets, and Bluespec. Chapters 8 – 10 develop corresponding \mathcal{R}_{HDL} 's for portions of these three hardware description languages.

A small set of concrete capabilities are assumed to be provided by \mathcal{R}_{HDL} . These capabilities are needed for the formalization of the meta-language as well as the formalization of the simulation-based verification problem we aim to help address with it, and are specified in this section.

Definition 4.4.1 (\mathcal{R}_{HDL}). In addition to being a rewriting logic semantics for an appropriate hardware design language, we assume that \mathcal{R}_{HDL} comes endowed with the following:

- *Config*: a sort used to denote terms capturing an entire program *configuration*, everything needed to continue with simulation.
- *Input*: a sort used to denote terms that capture the notion of an input stimulus.
- *inputOf* : $Config \longrightarrow Input$: an equationally-defined operator assumed to yield the remaining pool of input stimuli from a given configuration.

□

We will often speak of a *device*, which is simply used to mean a program given in the language formalized by \mathcal{R}_{HDL} . Additionally, we will assume that to each device we can associate an initial configuration. This initial configuration serves as the term from which \mathcal{R}_{HDL} judgments are derived; and, through these judgments, we obtain a suitable notion of simulation, as detailed in Section 4.5 below.

Definition 4.4.2 ($c_{\mathfrak{D},0}$). Let \mathfrak{D} be a device and X a set of variables, denumerable for each sort in \mathcal{R}_{HDL} . By

$$c_{\mathfrak{D},0} \in T_{\mathcal{R}_{HDL}, Config}(X),$$

we denote a symbolic term with variables in X giving the initial configuration of \mathfrak{D} . If we want to distinguish the remaining input pool of $c_{\mathfrak{D},0}$, we write $c_{\mathfrak{D},0}[i]$, where $i \in T_{\mathcal{R}_{HDL}, Input}(X)$ is the result of applying *inputOf* to $c_{\mathfrak{D},0}$. If \mathfrak{D} is understood, it can be omitted, as in c_0 or $c_0[i]$. \square

4.5 Problem

Utilizing a rewriting logic specification \mathcal{R}_{HDL} of the kind described above in Section 4.4, this section defines a precise, albeit oversimplified, notion of the verification closure problem, the problem that our meta-language, *fvml*, aims to address. The purpose is simply to provide a suitable frame of reference to understand the kind of goal that programs in our meta-language strive to attain.

To begin with, we require a formal definition of single-step rewriting, from which we obtain our definition of *simulation*. This notion is the central concept our meta-language is concerned with.

Definition 4.5.1 ($\longrightarrow_{\mathcal{R}}^1$). Let \mathcal{R} be a rewriting logic specification. Associated to \mathcal{R} is a binary relation, the *single-step rewrite relation*

$$\longrightarrow_{\mathcal{R}}^1 \subseteq T_{\mathcal{R}}(X)^2,$$

defined such that for any $t, t' \in T_{\mathcal{R}}(X)$, $(t, t') \in \longrightarrow_{\mathcal{R}}^1$ if and only if there exists a derivation of $\mathcal{R} \vdash t \longrightarrow t'$ with exactly one use of the inference rule “replacement” [74]. It is well-known that a general derivation of a rewrite

proof can always be represented as a sequential composition of single-step rewrites [74]. To denote that $(t, t') \in \longrightarrow_{\mathcal{R}}^1$, we often write

$$t \longrightarrow_{\mathcal{R}}^1 t'.$$

□

Definition 4.5.2 (Simulation). Let \mathfrak{D} be a device. A *simulation*, or *simulation trace*, of \mathfrak{D} is a finite sequence of configurations $(c_{\mathfrak{D},0}[i], \dots, c_{m-1}) \in T_{\mathcal{R}_{HDL}, Config}(X)$ such that for all $0 \leq j < m - 1$, $c_j \longrightarrow_{\mathcal{R}_{HDL}}^1 c_{j+1}$. □

Our notion of verification closure requires that a set of simulations be generated covering a set of predicates on simulations. These predicates are considered only in the abstract and are taken to be “coverage goals”. Standard notions of functional coverage, such as state and transition coverage, will usually be easy to formulate. Coverage metrics that are more syntax-oriented, such as statement coverage, could be more difficult.

Definition 4.5.3. Let \mathfrak{D} be a device. A *coverage goal* of \mathfrak{D} is a predicate on the set of simulations of \mathfrak{D} . □

Definition 4.5.4. Let \mathfrak{D} be a device and \mathcal{Q} a set of coverage goals of \mathfrak{D} . A finite set, S , of simulations, where each simulation $s \in S$ is of the form:

$$c_{\mathfrak{D},0}[i], \dots, c_{m-1}$$

with i an input stimulus and $m \geq 0$ the length of the simulation, is a $\mathfrak{D}, \mathcal{Q}$ -*cover* if and only if for each goal $Q \in \mathcal{Q}$, there exists a $s \in S$ such that $Q(s)$ holds. □

4.6 \mathcal{R}_{META}

\mathcal{R}_{META} provides the basic capabilities needed to *generate*, as *object-level terms*, simulation traces. As was described above in Section 4.5, simulation of a hardware device can be viewed as a sequence of single-step rewrites, defined according to the relation $\longrightarrow_{\mathcal{R}_{HDL}}^1$. And this relation is, in turn, defined in terms of the proofs derivable from \mathcal{R}_{HDL} . A potential problem arises in that a suitable formal framework in which we can manipulate $\longrightarrow_{\mathcal{R}_{HDL}}^1$, a

meta-logical notion of rewriting logic, is needed at the *object level*; \mathcal{R}_{META} provides a solution to that problem.

\mathcal{R}_{META} effectively captures parts of the meta-theory of rewriting logic through *reflection* to the object level [16, 17]. It may be suitably defined in a variety of ways, and we do not fix one here; any reasonable definition should provide enough functionality to satisfy our needs. As a concrete example, we consider taking Maude’s META-LEVEL module [15, §14], which contains a rich collection of meta-level functions for rewriting logic, as our \mathcal{R}_{META} .

In particular, META-LEVEL provides enough functionality so that $\longrightarrow_{\mathcal{R}_{HDL}}^1$ can easily be used to generate simulation traces. A sort `Module` is included that may be used to meta-represent \mathcal{R}_{HDL} as an object-level term denoted $\bar{\mathcal{R}}_{RTL}$, and similarly a sort `Term` is available that may be used to meta-represent, for a device \mathfrak{D} , $c_{\mathfrak{D},0}$ as object-level term, $c_{\mathfrak{D},0}^-$. META-LEVEL also provides an operation

```
metaRewrite : Module Term Bound ~> ResultPair
```

such that the term `metaRewrite`($\bar{\mathcal{R}}_{RTL}, c_{\mathfrak{D},0}^-, 1$), when the partial operation is successful, is equal to a pair (\bar{c}', \cdot) whose first component is the meta-representation of a term, c' , of sort *Config* that is reachable in one step of rewriting from $c_{\mathfrak{D},0}$; that is, which is guaranteed to satisfy

$$c_{\mathfrak{D},0} \longrightarrow_{\mathcal{R}_{HDL}}^1 c'$$

4.7 \mathcal{R}_{IR}

\mathcal{R}_{IR} provides a small set of simple operations from which complex testing programs in our meta-language, `fvml`, are created. Our running analogy equates \mathcal{R}_{IR} to LCF, both of which provide a small set of “inference rules” from which their respective artifacts of interest are generated. In the case of LCF, these are formal proofs, whereas in \mathcal{R}_{IR} the rules are used to manage the generation of simulations; though, within our formalization, rewriting logic proofs and simulation are of course one and the same.

\mathcal{R}_{IR} is parameterized by a rewriting logic semantics, \mathcal{R}_{HDL} , of a programming language used to design hardware and which is of the form specified in

Section 4.4. \mathcal{R}_{HDL} and \mathcal{R}_{META} are then included as part of \mathcal{R}_{IR} , and $\bar{\mathcal{R}}_{RTL}$ is an object-level term defined equal to a constant. Together, these are used to define a set of terms corresponding to *simulations*.

Definition 4.7.1. \mathcal{R}_{IR} defines a sort *Simulation*. A term of sort *Simulation* is a non-empty list of terms of sort *Config*, axiomatized in the usual way with an associative binary append constructor (*e.g.* see [15, §9.12.1]). The inference rules of \mathcal{R}_{IR} , defined below, generate terms of sort *Simulation*, where it is guaranteed that each adjacent pair of elements is related by $\longrightarrow_{\mathcal{R}_{HDL}}^1$; this may also be enforced at the sort level, through membership axioms, if desired. \square

A term of sort *Simulation* is variously denoted in the following ways:

- As an indexed list of configurations, for example,

$$c_0, \dots, c_{m-1}$$

where for all $0 \leq j < m$, $c_j \in T_{\mathcal{R}_{HDL}, Config}(X)$. Although not explicitly enforced at the sort level, it will always be the case that $c_j \longrightarrow_{\mathcal{R}_{HDL}}^1 c_{j+1}$ in the simulations we construct.

- A more stylized version with an arrow, \rightsquigarrow , which is indicative of change. This arrow distinguishes only configurations of interest, as in

$$c_0 \rightsquigarrow c_{m-1} \quad \text{or} \quad c_0 \rightsquigarrow c' \rightsquigarrow c_{m-1},$$

so that, unlike the first notation, no ellipsis is used;

- A second stylized version, similar to the above version, but where both the endpoints and input stimulus are distinguished; as in

$$c_0 \overset{i}{\rightsquigarrow} c_{m-1}$$

with $i \in T_{\mathcal{R}_{HDL}, Input}(X)$ and c_0 assumed to be of the form $c_0[i]$.

Each of the core operations provided through \mathcal{R}_{IR} 's interface yields a term of sort *Simulation*. Four operators are provided: *identity*, *substitution*, *advance simulation*, and *transitivity*. Below, each operation is given an associated “inference rule”, which it operates in accordance with. For the

most part, premises correspond with inputs, and conclusions correspond with the operation's output.

The “substitution” rule requires a sort, which will be denoted $Subst$, for terms that will represent substitutions. We assume that this sort characterizes mappings from (sorted) variables to terms of an appropriate sort; or rather, an appropriate object-level meta-representation of these concepts. The parameterized MAP module in Maude [15, §9.13] can be used for this purpose, for example.

Definition 4.7.2. \mathcal{R}_{IR} provides an interface defined by the following operations:

1. $rule_I : Config \longrightarrow Simulation$: given a configuration, $rule_I$ produces an identity simulation according to the following rule:

$$\frac{\cdot}{c \rightsquigarrow c}$$

2. $rule_S : Simulation \times Subst \longrightarrow Simulation$: given a simulation and a substitution, produces a new substituted simulation, with variables instantiated according to the substitution:

$$\frac{c \overset{i}{\rightsquigarrow} c' \quad \rho : X \longrightarrow T_{\mathcal{R}_{HDL}}(X)}{\rho(c) \overset{\rho(i)}{\rightsquigarrow} \rho(c')}$$

3. $rule_A : Simulation \longrightarrow Simulation$: given a simulation, extends it by one simulation “step” through rewriting (the implementation of $rule_A$ requires the use of \mathcal{R}_{META} , the second premise, which, in this case, does *not* correspond to an argument of $rule_A$):

$$\frac{c \rightsquigarrow c' \quad c' \xrightarrow{\mathcal{R}_{HDL}^1} c''}{c \rightsquigarrow c''}$$

4. $rule_T : Simulation \times Simulation \longrightarrow Simulation$: given two compatible simulations, the two are composed by transitivity:

$$\frac{c \rightsquigarrow c' \quad c' \rightsquigarrow c''}{c \rightsquigarrow c''}$$

□

4.8 \mathcal{R}_{STRAT}

\mathcal{R}_{STRAT} differs from the above rewriting logic specifications in that it is not really part of the meta-language itself. It is analogous to an ML program that programmatically works through some portion of the space of proofs searching for one establishing, for example, the group theory theorem $x \cdot 1 = x$. In the case of \mathcal{R}_{STRAT} , instead of using the inference rules of LCF to generate a proof, the rules of \mathcal{R}_{IR} are used to coordinate the generation of simulation traces toward verification closure; that is, toward covering a set of coverage goals.

4.9 Example

```
1 fmod CHECK-GOAL is
2   protecting R-IR .
3   --- auxiliary sorts, ops, meta-variables, etc. omitted
4
5   sort Goal .
6
7   op check-goal : Goal Simulation -> Bool .
8   eq check-goal(GOAL, SIM) = ...
9
10 endfm
```

Figure 4.3: Some Auxiliary Infrastructure for the Example in Figure 4.4.

The purpose of this section is to demonstrate what is essentially the most basic program one can construct using `fvml`, a sort of “Hello World!”. In the context of functional verification, perhaps the most simple program one would want to construct is a program that performs *directed testing*, where a device, a concrete input stimulus, and testing goal are given and used to generate a simulation that is then checked against the testing goal.

One implementation of directed testing in `fvml` is shown in Figure 4.4, with some auxiliary functionality provided by another module shown in Figure 4.3, both of which use the syntax of Maude. In order to emphasize the essential


```

1 fmod R-STRAT is
2   protecting R-IR .
3   protecting CHECK-GOAL .
4
5   --- auxiliary sorts, ops, meta-variables, etc. omitted
6
7   op directed-test : Config Input Goal -> Bool .
8   eq directed-test(CO, INPUT, GOAL) =
9     check-goal(GOAL, full-simulation(CO[INPUT])) .
10
11  op full-simulation : Simulation -> Simulation
12  eq full-simulation(SIM :: C) =
13    if inputOf(C) == nil
14      then SIM :: C
15      else full-simulation(ruleA(SIM :: C)) fi .
16
17 endfm

```

Figure 4.4: A Directed Testing Strategy.

ideas, some sorts, operators, meta-variables and so forth are not spelled out in detail; however, any construct whose intent is not immediately clear from context is explained.

Directed testing essentially requires that we have two pieces of functionality: (1) the ability to generate a simulation from a given device implementation and input stimulus, and (2) some means through which the simulation is assessed within the overall verification plan, such as its contribution to functional coverage goals or success or failure against assertions. `fvml` does not take a position regarding (2) and leaves it to the user to construct using the general-purpose programming facilities provided by our embedded language. However, as a language whose intent is to orchestrate the generation of simulations, it makes (1) straightforward.

Figure 4.3 indicates the functionality corresponding to (2) above. We assume in this example that the user has provided an appropriate set of terms (the sort `Goal`) corresponding to criteria against which simulations are assessed, as well as a function, `check-goal`, that is able ascertain whether or not a simulation satisfies some property from this space. Of course, more

complex mechanisms of assessing simulations might be more appropriate in practice, for example, ones that return more elaborate results than simple booleans.

Figure 4.4 implements the functionality corresponding to (1) above, namely the part that generates a simulation from a device and an input stimulus. The exact structure of a configuration and an input stimulus is, as described above in Section 4.4, defined by \mathcal{R}_{HDL} . The equationally-defined function `full-simulation` simply takes the initial configuration and input stimulus and recursively applies the `fvml` operation $rule_A$ (here denoted `ruleA`), until the input stimulus is exhausted. To do this, we assume that the list constructor for simulations is the Maude operator `_::_`, that `Config` is a sub-sort of `Simulation`, and that the formalization \mathcal{R}_{HDL} of the hardware description language in which the device was built has an operator, `nil`, representing an empty stimulus. `nil` is used as the terminating case for the recursion and marks that the simulation should end. Finally, the top-level function `directed-test` aggregates the two parts, first generating a simulation and then using the functionality of the `CHECK-GOAL` module to summarize the result.

In addition to demonstrating the construction of the most basic `fvml` program possible, the example in this section emphasizes that while `fvml` takes a clear position on the conception of simulation-based verification as an activity that revolves around the orchestrated generation of simulations, it takes no position on the exact manner in which the generated simulations are assessed during the verification process. Appropriately, therefore, it provides for that aspect of the process the general-purpose programming language facilities of the language into which `fvml` is embedded.

CHAPTER 5

IMPLEMENTATION ARCHITECTURE

This chapter describes the *architecture* of an embedded domain-specific language in Haskell for a concrete realization of our meta-language, `fvml`, defined in the previous chapter. This architecture is, in addition, specialized for the Verilog hardware description language, though many of the pieces would apply to other design languages. Our current implementation, `vlogml`, which largely follows this architecture, is open source and available at [46].

By using the term “architecture”, our intention is to make a distinction between what is presented here and the source code that comprises our current implementation and which can actually be compiled and executed on a computer [46]; though, we refer to both using the name `vlogml`. The reason for this is to be able to focus on the more interesting conceptual aspects of an implementation without becoming mired in ancillary implementation detail, and also to take some other liberties where appropriate so that ideas can be conveyed as clearly as possible. Without this distinction, we find it easy to miss the forest for the trees, so to speak. Of course, the reader is invited to consult the source code at any time [46].

`vlogml` is defined as an embedded domain-specific language in Haskell. This means that the domain-specific functionality of our meta-language formalized in Chapter 4 is implemented within Haskell as a set of data types and functions, and that as a result we also gain seamless use of Haskell’s general-purpose programming facilities, which together yield the full-fledged language. As for any programming language, our expectation regarding `vlogml` is that it will provide a strong foundation upon which additional, rich functionality can be built and distributed. For example, one might envision a library targeting constrained-random simulation, providing functionality to support that paradigm; for example, providing a language of constraints, the necessary solver technology to generate random values that respect the constraint language, and so forth. Note that distributing such functionality

as a *library*, as opposed to becoming directly part of the *language* syntax and semantics, is a substantial difference between our language and a language such as SystemVerilog.

The remainder of this chapter is organized as follows.

- Section 5.1 outlines the main components of the topmost `vlogml` module, called `VlogMetaLang`, which must be imported to create a `vlogml` testbench program.
- Section 5.2 defines a “skeleton” testbench program that we will assume throughout the `vlogml` examples given in this dissertation and which imports `VlogMetaLang`.
- Sections 5.3 – 5.6 provide additional details about the `vlogml` architecture, broken down into a set of Haskell modules that capture different conceptual pieces of `vlogml`. The correspondence of these modules with `fvm1` is emphasized.
- Sections 5.7 and 5.8 describe two modules that are outside of the `vlogml` core, yet build on it to provide a richer experience. Section 5.7 adds monadic “strategy” functionality and Section 5.8 integrates an SMT solver.
- Section 5.9 provides a concrete example testbench in `vlogml` which relies on the functionality made available by our architecture.

The complete source code of `vlogml` is freely available [46] and distributed under the terms of the GNU General Public License, version 3 (GPLv3). As we noted above, liberties have been taken in this chapter to aid in conveying the essential concepts and ideas of an implementation, leading to some mismatching between what is presented here and that source code.

5.1 VlogMetaLang

`VlogMetaLang` exports a core set of data types and functions, essentially corresponding to the functionality provided by the \mathcal{R}_{IR} part of the formalization (see Section 4.7), and the necessary parts of \mathcal{R}_{HDL} and \mathcal{R}_{META} (sections 4.4 and 4.6), specialized of course for the Verilog case. In particular, a data

```
module VlogMetaLang.Syntax
module VlogMetaLang.Data
module VlogMetaLang.Util
module VlogMetaLang.Core
```

Figure 5.1: VlogMetaLang Interface.

type representing *simulation* of Verilog devices is provided, as are functions to manipulate values of this type by advancing (symbolic) simulation, or querying the waveform history of the simulation, and so forth. From these core operations and data types, we expect that users will build-out richer functionality, such as adding the ability to resolve symbolic contexts to concrete values (see Section 5.8).

The functionality of `VlogMetaLang` is partitioned across four sub-modules, as shown in Figure 5.1, the details of which are described in Sections 5.4 – 5.3 next; `VlogMetaLang` simply does aggregation. Briefly, the purpose of each module is as follows.

- `VlogMetaLang.Syntax`: This module contains data types representing various aspects of Verilog syntax, such as expressions and statements, and provides helpful functions aiding in the construction of these values.
- `VlogMetaLang.Data`: This module exports additional data type declarations that are essential for `vlogml`, in particular its (*first-class notion of simulation*), but also other important data types, such as those representing input stimuli.
- `VlogMetaLang.Util`: This module exports useful utility functions operating on the data types defined in `VlogMetaLang.Data`. For example, one function that is provided yields the waveform history from a simulation.
- `VlogMetaLang.Core`: This module exports three functions, representing the core `vlogml` language features and are used to generate simulations. *These functions correspond to the operations $rule_I$, $rule_S$, and $rule_A$ from \mathcal{R}_{IR} .*

5.2 Skeleton

```
1 import VlogMetaLang
2
3 main :: IO ()
4 main = reportResult =<< strategy =<< start dutrc
```

Figure 5.2: Skeleton Testbench.

This section describes a `vlogm1` program “skeleton” that will provide appropriate scaffolding for future `vlogm1` example programs. It imports the module `VlogMetaLang` from the previous section to gain access to the core `vlogm1` data types and functions. Assuming this skeleton program will be useful because it captures the portion of a `vlogm1` program that will remain constant throughout our examples. It is displayed in Figure 5.2.

As shown in the figure, the skeleton program includes `VlogMetaLang` and is constructed as the monadic composition of three operations. The compositionality is made apparent by considering the types of each of these operations, which are

```
start      :: DeviceRC  -> IO Simulation
strategy   :: Simulation -> IO a
reportResult :: a        -> IO ()
```

`strategy` and `reportResult` contain a type variable `a` that may be instantiated with any concrete type, but must be instantiated to the same type for both functions so that the composition produces a result that is well-typed.

The intended purpose of each of the composed functions is as follows:

- **start**: takes as an argument a value of type `DeviceRC` that contains essential configuration information needed for bootstrapping and essentially returns a value of type `Simulation` representing the initial state of the device, which is an “empty” simulation. This operation requires the use of I/O.

- **strategy**: takes the empty simulation as an argument and uses it as a starting point for its testing regime, which represents the body of the program. It contains the logic for orchestrating and driving the generation of device simulation runs, all of which stem from this initial simulation.
- **reportResult**: accepts the result of **strategy** and generates a report to the terminal, or saved to disk, or whatever is convenient, explaining the result of having run the program.

The skeleton program assumes, in addition, that we are in possession of a data value named `dutrc` that is of type `DeviceRC`. This value specifies necessary configuration information for the device-under-test, as described above, and again later in Section 5.3.

5.3 VlogMetaLang.Core

```

start      :: DeviceRC      -> IO Simulation
concretize :: Substitution  -> Simulation -> Simulation
simulate  :: (Input a) => a -> Simulation -> Simulation

```

Figure 5.3: VlogMetaLang.Core Interface.

Let us begin with `VlogMetaLang.Core` and then return to some of the other modules in `VlogMetaLang` upon which it relies. `VlogMetaLang.Core` provides the essential operations of our meta-language, those used to *construct simulation values* and which are crucial to constructing the **strategy** function from our skeleton program. Three operations are provided in total, as shown in Figure 5.3; they correspond to the operations $rule_I$, $rule_S$, and $rule_A$ from \mathcal{R}_{IR} , respectively.

In the subsequent sections 5.4 – 5.6, we provide some additional details about the `vlogml` data types noted in Figure 5.3. At a high-level, however, the purpose of the data types should be clear: `Simulation` is the type that represents our first-class notion of Verilog simulation, the type `Substitution` is used to represent mappings from symbolic variables to values, and the *type*

class (**Input a**) represents things that can be used as input stimulus. Device configuration information that is used needed for bootstrapping are provided by values of type `DeviceRC`.

```
start :: DeviceRC -> IO Simulation
```

The function `start` corresponds to $rule_I$ from \mathcal{R}_{IR} and is the mechanism through which a value of type `Simulation` is initially obtained. The argument to the function must be a value of type `DeviceRC`, which contains essential *configuration information* about the device-under-test. It includes, for example, the locations of files containing the Verilog source code for the device, as well as additional information needed during parsing and elaboration. `start` returns an initial, “empty”, *simulation* of the device. I/O is required to read the files containing the device source code.

```
concretize :: Substitution -> Simulation -> Simulation
```

The function `concretize` corresponds to $rule_S$ from \mathcal{R}_{IR} . Its first argument is a substitution, a mapping from symbolic variables (see Section 5.4) to Verilog expressions, and its second argument is a (symbolic, presumably; though not necessarily) simulation. The result of applying the function is a new (possibly symbolic) simulation where symbolic variables have been substituted for according to the mapping given by the first argument.

```
simulate :: (Input a) => a -> Simulation -> Simulation
```

The function `simulate` corresponds to $rule_A$ from \mathcal{R}_{IR} . The type signature for the function is slightly complicated by the use of a *typeclass* making it polymorphic in the first argument, which is used to represent an input stimulus (see Section 5.4). Essentially, the function takes as arguments a value of some type that can be converted to an input stimulus and a (possibly symbolic) simulation, and returns a new (possibly symbolic) simulation that is the result of advancing the given simulation according to the stimulus.

5.4 VlogMetaLang.Syntax

```
data Literal
data Identifier
data Variable
data Expression
data Statement
data Process
...
...
data Program
```

Figure 5.4: (Partial) VlogMetaLang.Syntax Interface.

VlogMetaLang.Syntax exports data types for parts of the Verilog syntax, as well as utilities to aid in the construction of values of these types. For example, they are used to construct queries about simulations. A partial listing of the VlogMetaLang.Syntax interface, exporting several data types, is shown in Figure 5.4.

- **Literal, Identifier *etc.***: The intended purpose of each data type is for the most part clear from the name of the data type; for example, `Literal` is used to represent Verilog-style literals such as `4'b1010`.
- **Variable**: One exception is the `Variable` data type. It is used to represent symbolic variables created by the user during testing. Symbolic variables do not have a direct counterpart in the syntax of Verilog, which is why it is a special case. The reason `vlogm1` includes symbolic variables alongside Verilog syntax is that the context in which these variables are used is essentially the same as that of regular identifiers.

Construction via Quasi-Quoting: As described at the outset of this section, VlogMetaLang.Syntax also exports functions to aid in the construction of values of the above types. For the purposes of this dissertation, we will use “quasi-quoting” to construct values of the above types. For example, consider the following expression in the syntax of Verilog

```
x + 1 >> 2
```

where `x` is assumed to be an identifier that is declared in the Verilog source code and is in scope. The corresponding value of type `Expression` is constructed in `vlogml` as

```
[expr| x + 1 >> 2 |]
```

In general in `vlogml`, by surrounding a Verilog expression with `[expr|...|]`, a corresponding value of type `Expression` is create. This particular syntax is in accordance with the current incarnation of Haskell’s quasi-quote library [62].

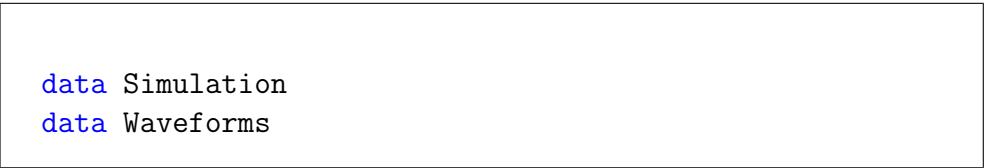
Symbolic variables are created by allowing a slightly extended syntax within `[expr|...|]`. An identifier that is surrounded with `<...>` and prefixed by an explicit bit-width, as in:

```
[expr| <32'y> + 1 >> 2 |]
```

which creates a symbolic variable, `y`, that is 32-bits wide.

Similarly to the construction of expressions, quasi-quoting syntax is used to construct other elements of Verilog syntax. Statements are constructed with `[stmt|...|]`, processes with `[proc|...|]`, identifiers with `[iden|...|]`, and (symbolic) variable construction, outside the context of an expression, with `[var|...|]`.

5.5 VlogMetaLang.Data



```
data Simulation
data Waveforms
```

Figure 5.5: `VlogMetaLang.Data` Interface (Part 1).

`VlogMetaLang.Data` is responsible for providing all of the core data types that are not closely associated with Verilog syntax, as those are provided by `VlogMetaLang.Syntax`. The relevant portion of the interface is split across Figures 5.5 – 5.7 for explanatory purposes.

Figure 5.5 shows the exportation of two especially important data types:

- **Simulation:** This is the data type representing a simulation. It contains enough information to continue simulation, such as the state of the stratified event queue described in the Verilog Standard [36], as well as a waveform history and a few additional pieces of information. To construct a simulation in `vlogml`, one must use the functions given in Section 5.3; the data type constructors are not exported.
- **Waveform:** This data type represents a full waveform history for a device. Essentially, it provides a simulation-time-indexed map of the value of every node in the device. `VlogMetaLang.Util` provides a utility to obtain the waveforms associated with a simulation (see Section 5.6).

```

data DeviceRC = DeviceRC {
    sources :: [FilePath]
    , top    :: Identifier
    , ... -- additional fields
}

newtype Substitution
    = Substitution (Map Variable Expression)

```

Figure 5.6: `VlogMetaLang.Data` Interface (Part 2).

Figure 5.6 shows the exportation of two data types, one that is used for basic device configuration, and the second a representation of the concept of substitution, which is used in various contexts in `vlogml`.

- **DeviceRC:** This is a record type containing basic device configuration information. It has fields for the list of Verilog source files and an identifier noting the top-level module name, among other items.
- **Substitution:** This is a data type representing substitutions. Specifically, it is used to map variables to values of type `Expression`. The declaration uses `newtype` instead of `data`, which is usual practice in Haskell for renamed types [23].

Figure 5.7 shows the exportation of a data type and a type class that are used in `vlogml` to construct input stimuli.

```
data Input =
    CycleBased [Map Identifier Expression]
    | ... -- additional constructors
class Input a where
    toInput :: a -> Input
```

Figure 5.7: VlogMetaLang.Data Interface (Part 3).

- **Input**: This is the data type that `vlogml` uses internally to represent input stimulus and drive simulation. For the purposes of this dissertation, we show just one kind of input that is supported, cycle-based. A list of maps from identifiers to expressions are given, one for each clock cycle, assigning values to the device inputs.
- **(Input a)**: This is a type class allowing users to create specialized input stimuli tailored for their device and then use it for simulation; any algebraic data type may be used. See Section 5.9 for an example.

5.6 VlogMetaLang.Util

```
waves :: Simulation -> Waveforms
clock  :: Simulation -> Int
```

Figure 5.8: (Partial) VlogMetaLang.Util Interface.

Since the user is not provided the constructors for `Simulation`, some functions are needed to retrieve needed information from values of this type.

- **waves**: This function returns an entire waveform history for the current simulation.
- **clock**: This function returns the current simulation time, as an integer.

```

type (Strategy a)
startM      ::      DeviceRC -> Strategy ()
simulateM   :: (Input a) => a -> Strategy ()
simulateRandM :: (Input a) => a -> Strategy ()
queryM      ::      Expression -> Strategy Bool
concretizeM ::      Substitution -> Strategy ()

runStrat    :: (Strategy a) -> Simulation -> IO (a,Simulation)
evalStrat   :: (Strategy a) -> Simulation -> IO a
execStrat   :: (Strategy a) -> Simulation -> IO Simulation

```

Figure 5.9: VlogMetaLang.Strategy Interface.

5.7 VlogMetaLang.Strategy

The `VlogMetaLang.Strategy` library is used as the basis for almost all of the `vlogml` examples presented throughout this dissertation. It exports a polymorphic type

```
type Strategy a = StateT Simulation IO a
```

that is an instance of Haskell’s state-transformer monad, `StateT` (see [23]).

Intuitively, `(Strategy a)` is intended to represent stateful computations where the backing state is a simulation, and which furthermore allows the use of I/O. For example, I/O can be used for random number generation or to invoke an SMT solver, as we do in Section 5.8. The main benefit of the `(Strategy a)` type is that it removes the need to explicitly pass simulations between functions. Indeed, this is the usual reason to employ a state monad.

`VlogMetaLang.Strategy` then provides a set of functions from which these strategies may be constructed. In particular, monadic versions of the functions from `VlogMetaLang.Core`:

```

startM      ::      DeviceRC -> Strategy ()
simulateM   :: (Input a) => a -> Strategy ()
concretizeM ::      Substitution -> Strategy ()

```

Figure 5.9 notes two additional functions that are exported by the `VlogMetaLang.Strategy` library and are used in examples presented later. The first is a modified version of `simulateM` that performs random simulation.

```
simulateRandM :: (Input a) => a -> Strategy ()
```

This function works by substituting random values for any variables that occur in the input. `simulateM`, on the other hand, keeps the variables and performs symbolic simulation.

The second function that will be used accepts as its first argument a Verilog expression, which must be of scalar (boolean) type, and returns a `Bool`.

```
queryM :: Expression -> Strategy Bool
```

This function evaluates the argument expression according to the current state of the simulation, if it evaluates to a single-bit value equal to one, then the value `True` is returned; if it evaluates to any other value, concrete or symbolic, then the value `False` is returned.

In order to finish a strategy computation, some method is required to get out of the monad. State monads are usually unravelled as functions of the form

```
s -> (a,s)
```

where `s` denotes the type of the backing state, and `a` is the result of the computation. The function `runStrat` unravels values of type `(Strategy a)` to return a function of this type, but which is also capable of doing I/O, for the reasons described at the outset of the section:

```
runStrat :: (Strategy a) -> Simulation -> IO (a,Simulation)
```

If just the result of the computation is desired, or just the modified `Simulation` is desired, the specialized functions

```
evalStrat :: (Strategy a) -> Simulation -> IO a  
execStrat :: (Strategy a) -> Simulation -> IO Simulation
```

may be used. These correspond to the functions `runStateT`, `evalStateT`, and `execStateT` which are part of the standard Haskell distribution [23].

```
smt :: (Maybe Int) -> Expression -> IO (Maybe Substitution)
```

Figure 5.10: `VlogMetaLang.SMT` Interface.

5.8 `VlogMetaLang.SMT`

This section describes a library currently provided by `vlogml` and used in some later examples that allows a user to interface with an SMT solver. A user can take advantage of this library to automatically resolve symbolic simulations to interesting concrete ones. The underlying solving is done by STP [27], an SMT solver for bit-vectors and arrays which is well suited to Verilog.

The library currently just exports one function, `smt`, which essentially accepts as input a Verilog expression with scalar type and then attempts to find a satisfying assignment to the symbolic variables contained in that expression. The exact signature of the function is

```
smt :: (Maybe Int) -> Expression -> IO (Maybe Substitution)
```

The first argument specifies an optional timeout value, in seconds, after which STP aborts. If the expression is satisfiable and the solver is able to determine a satisfying assignment before the timeout is reached, a substitution for that satisfying assignment is returned; otherwise, if no such satisfying assignment exists or the search is aborted due to a timeout, the value `Nothing` is returned.

5.9 Example

This section employs the above functionality to construct an example `vlogml` program, exemplifying one of the most simple verification paradigms: directed testing. However simple, directed testing should, if the meta-language is accomplishing all of the goals we have set for it, be supported in a straightforward and simple way; this section demonstrates that this is so for `vlogml`.

The device-under-test is the Verilog module presented in Figure 5.11, which is meant to be indicative of a maze: at each clock cycle, the device

```

1  module maze(clk, i);
2
3  input      clk;
4  input      i;
5  reg    [2:0] loc;
6
7  initial loc = 0;
8
9  always @(posedge clk)
10     case (loc)
11         0 : loc <= i ? 1 : 0;
12         1 : loc <= i ? 0 : 2;
13         2 : loc <= i ? 3 : 0;
14         3 : loc <= i ? 0 : 4;
15         4 : loc <= i ? 5 : 0;
16         5 : loc <= i ? 6 : 7;
17         6 : $display("FAILURE");
18         7 : $display("SUCCESS");
19     endcase
20 endmodule

```

Figure 5.11: Example Verilog Module: “Maze”.

“moves” from the current location to some new location based on the single-bit input *i*. The goal is to “walk” from the initial location, represented by the value 0, to the location represented by the value 7, a successful exit; the location given by the value 6 represents an inescapable dead-end.

Three separate tasks need to be covered by the program: specification of the directed stimulus, use of the stimulus to generate a simulation, and checking that the simulation successfully navigates the maze. Ideally, for a directed test of the maze device, a stimulus will be given as a list of 1’s and 0’s, as in

[1, 0, 1, 0, 1, 0]

where each element of the list represents the intended value of *i* during the clock cycle corresponding to its position in the list.

Specialized stimuli are implemented within `vlogml` by first creating an


```

1 data MazeInput =
2   Concrete [Int]
3
4 instance Input MazeInput where
5   toInput (Concrete xs) = CycleBased (map mkSubst xs)
6   where mkSubst x = ... -- i maps to x

```

Figure 5.12: Specialized Input for the Maze.

appropriate data type, and then creating a corresponding instance of the type class `Input`, described above in Section 5.5. In the case of the maze and the kind of directed stimuli described above, the most straightforward implementation is by creating an algebraic data type, say `MazeInput`, with a constructor taking a list of integers as an argument. As a concrete implementation, we assume the code presented in Figure 5.12:

- (*lines 1 – 2*): Declare a data type, `MazeInput`, representing input stimuli for the maze. The constructor `Concrete` captures the stimuli as lists of integers, representing cycle-by-cycle values for the input `i`.
- (*lines 4 – 5*): Add an appropriate instance of `toInput` to convert maze input into the internal format supported for simulation. The exact specification of the conversion is omitted for reasons described above in Section 5.5 (see [46]).

Having the `MazeInput` data type and assuming the skeleton program from Section 5.2, the main `vlogml` program can be implemented straightforwardly, as shown in Figure 5.13. The first part, `stimulus` (*line 2*), specifies the particular stimulus we will use to generate a simulation:

- (*line 2*): A concrete stimulus that will successfully navigate the maze.

The second part of the program, the function `strategy` (*lines 5 – 7*), simply runs simulation with the defined stimulus and subsequently calls a function to assess the result; it is implemented using the `Strategy` combinators described above in Section 5.7.

- (*line 6*): Run simulation using the concrete stimulus defined by `stimulus`.
- (*line 7*): Check condition for a successful exit from the maze.

```

1  --- directed stimulus
2  stimulus = Concrete [1, 0, 1, 0, 1, 0]
3
4  --- strategy logic
5  strategy = evalStrat $ do
6    simulateM stimulus
7    checkResult
8
9  --- determine result of strategy
10 checkResult = do
11   out <- queryM [expr| loc == 7 |]
12   sim <- get
13   if out
14     then return (Just sim)
15     else return Nothing

```

Figure 5.13: Directed Testing in `vlogml`.

The third part of the program checks the resulting simulation to determine if a successful exit from the maze has occurred. This is specified by the function `checkResult` (*lines 10 – 15*).

- (*line 11*): Check condition for a successful exit from the maze;
- (*line 12*): Bind the resulting simulation to the variable `sim`;
- (*lines 13 – 15*): Upon successful exit from the maze, return the simulation trace, otherwise return `Nothing`, indicating failure.

Note that in the particular case of this example, the concrete type corresponding to the variable `a` above in Section 5.2 is `(Maybe Simulation)`.

Let us consider now directed stimulus within a broader context via a pair of examples that are more realistic than the maze. First, consider a microprocessor, where a directed test would typically take the form of an assembly program; for example, see Section 7.2. Just as in the above example, to support assembly programs in `vlogml` one would first create a data type for them and then an appropriate instance of the `Input` type class. This allows for assembly programs that can be simulated to be built directly via the data-type’s constructors.

However, building an assembly program directly out of the constructors for an algebraic data type, while relatively natural, can also be improved upon. For example, one can provide other means of generating values of this data type that are closer to the usual syntax of assembly programs; for example, through quasi-quoting, or through parsing assembly files directly and converting them into the internal representation.

As a second example, the I²C bus-mastering case study (Section 7.1) will operate at a granularity of “Wishbone transactions”, which are defined according to the Wishbone Interface [25]. Again, one starts by constructing an appropriate data type and an instance of the `Input` type class for Wishbone transactions, followed by some convenient way for the user to construct values of this type.

CHAPTER 6

CAPABILITIES

This chapter seeks to demonstrate, with concrete `vlogm1` examples, a variety of novel capabilities of our meta-language. By novel, we mean verification strategies that are easy to effect in our meta-language but not with existing functional verification tools and that, in addition, operate over the coverage closure feedback loop described at the outset of the dissertation in Chapter 1 and allow for its automation in unique ways. Therefore, the examples presented are a significant part of justifying the meta-language’s existence.

Each of the examples presented applies a strategy aimed at solving, in the way described above in Section 5.9, the maze device from that same section, which is reprinted in this chapter for convenience; see Figure 6.1. The examples operate over this small device so that they may be presented completely to emphasize the capabilities of the meta-language, rather than becoming buried in the complexity of the device being analyzed. Case studies applying `vlogm1` to more substantial devices, specifically a bus-master controller and a small microprocessor, are described in Chapter 7.

Aside from Section 6.1, which contains some maze-specific `vlogm1` utility functions, each of the sections of this chapter describes a specific maze-solving strategy and an implementation of that strategy in `vlogm1`. In addition, we attempt to draw broader conclusions from the strategy about the meta-language’s capabilities. All of the examples are distributed along with `vlogm1` and may be independently verified [46]; however, for the same reasons of clarity that we described at the beginning of Chapter 5, some of the details differ between what is presented here and what is given as executable code in [46], though all are functionally the same.

```

1 module maze(clk, i);
2
3   input      clk;
4   input      i;
5   reg    [2:0] loc;
6
7   initial loc = 0;
8
9   always @(posedge clk)
10    case (loc)
11      0 : loc <= i ? 1 : 0;
12      1 : loc <= i ? 0 : 2;
13      2 : loc <= i ? 3 : 0;
14      3 : loc <= i ? 0 : 4;
15      4 : loc <= i ? 5 : 0;
16      5 : loc <= i ? 6 : 7;
17      6 : $display("FAILURE");
18      7 : $display("SUCCESS");
19    endcase
20 endmodule

```

Figure 6.1: Example Verilog Module: “Maze” (Re-Printed).

6.1 Some Utilities

This section defines a small set of maze-specific utilities that will be used throughout the following examples. As just one example, the `MazeInput` type from Section 5.9 is here extended to allow for symbolic input stimuli. The various utilities are set out in Figures 6.2 – 6.5.

```
1 data MazeInput =
2   Concrete [Int]
3   | Symbolic Int
4
5 instance Input MazeInput where
6   toInput (Concrete xs) = ...
7   toInput (Symbolic j) = ...
```

Figure 6.2: Specialized Input for the Maze (Symbolic and Concrete).

Figure 6.2: `MazeInput`. The data type representing maze stimuli is extended to account for symbolic stimuli as follows:

- (*line 3*): The constructor `Symbolic` is used to represent input stimuli where, for the number of clock cycles given by the argument, a fresh symbolic variable is generated and assigned to the maze input.
- (*line 7*): The `(Input a)` instance is extended to account for the additional constructor.

```
1 instance (Input a) => Input [a] where
2   where toInput xs = ...
```

Figure 6.3:

Figure 6.3: `(Input [a])`. In one situation, we will want to generate stimuli that are composed partially from concrete stimuli and partially from symbolic stimuli. This instance allows us to do so simply by giving a list

of values of type `MazeInput`. For example, with the `(Input a)` declaration shown in the figure, the following list may be used as a valid input stimulus representing two cycles of concrete simulation followed by eight cycles of symbolic stimulus.

```
xs :: [MazeInput]
xs = [Concrete [0,1], Symbolic 8]
```

```
1 checkOutOfMaze :: Strategy Bool
2 checkOutOfMaze = queryM [expr| loc == 7 |]
```

Figure 6.4:

Figure 6.4: `checkOutOfMaze`. This function returns a boolean indicating, with respect to the current simulation context, whether it has successfully navigated the maze; a successful exit being characterized by the expression `(loc == 7)`.

```
checkOutOfMazeSMT :: Strategy (Maybe Substitution)
checkOutOfMazeSMT = do
  exp <- evalM [expr| loc == 7 |]
  lift (smt timeout exp)

timeout = Just 10
```

Figure 6.5:

Figure 6.5: `checkOutOfMazeSMT`. This function attempts to resolve a symbolic simulation context to a concrete simulation that successfully exits the maze. If the solver succeeds, a substitution is returned that maps the simulation's symbolic variables accordingly. The function `lift` is a standard Haskell function, that is here used to take a value of type `(IO a)` to a value of type `(Strategy a)`. A timeout of ten seconds is imposed for the SMT solver to complete its work, which, in the case of the maze device and our examples, is more than sufficient.

6.2 Coordination of Multiple Simulations

This section demonstrates by example the essential novelty of our meta-language over a traditional tool set, which is its ability to easily orchestrate multiple simulations, and the feedback obtained from those simulations, together as a single testing strategy. The particular strategy under consideration is as follows: one hundred trials of purely random simulation are executed serially, with each trial running for ten clock cycles. After each trial, the resulting simulation is analyzed to determine if the maze was successfully navigated, and if so, the effort is halted and overall success is reported, otherwise the next trial is then started.

Section 6.2.1 first demonstrates an implementation of this strategy using existing tools, followed by the `vlogml` implementation in Section 6.2.2, and, finally, concluding with a comparative analysis of the two in Section 6.2.3.

6.2.1 Solution 1: SystemVerilog, VCS, and bash

A straightforward way to accomplish the stated strategy with traditional tools is by constructing two programs and employing the functionality of a simulator. The role of the first program is to generate stimulus for a single simulation run; this program is then compiled with a simulator and used by the second program, whose role is to coordinate the one hundred separate trials.

The first program is written in SystemVerilog as shown in Figure 6.6, which has strong support for constrained-random stimulus generation. This program can be compiled with a simulator such as VCS to produce an executable, say `simv`, which is then called by the second program. This second program is the shell script presented in Figure 6.7.

Program 1: SystemVerilog. The essential code of the program in Figure 6.6 are lines 17 – 22, which have the following meaning:

- (*line 17*): Execute the following block statement ten times. The block drives the maze input for a single clock cycle each time it is executed.
- (*line 19*): Wait for the next clock cycle, given by the positive edge of the signal `clk`.


```

1 module testbench;
2
3   reg clk, i;
4   Bit x = new;
5
6   // device-under-test
7   maze m(clk, i);
8
9   // clock generation
10  always #5 clk = ~clk;
11
12  // stimulus generation
13  initial
14  begin
15    clk = 0;
16    i   = 0;
17    repeat (10)
18    begin
19      @(posedge clk);
20      x.randomize;
21      i = #1 x.val;
22    end
23  end
24 endmodule

```

Figure 6.6:

- (line 20): Generate a new, single-bit random value. This depends on the definition of a data type, `Bit`, not shown in the figure, that allows for the randomized generation of a single bit. It can be defined as

```

class Bit;
  rand bit val;
endclass

```

- (line 21): Drive the maze input, the signal `i`, with the randomly generated bit.

```

1  #! /bin/bash
2
3  for j in {1..100} ; do
4    ./simv +ntb_random_seed=$j | grep -q SUCCESS
5    if [ $? -eq 0 ] ; then
6      echo "succeeded at $j."
7      exit
8    fi
9  done ; echo "failed all 100."

```

Figure 6.7:

Program 2: Bash Script. The coordination of the one hundred separate trials is left to the second program, the bash script presented in Figure 6.7; it assumes that the first program has been compiled with VCS and that the resulting executable is named `simv`.

- (*line 3*): Establish a loop that will execute one hundred iterations of the body, given in lines 4 – 8. The loop counter is assigned to the variable `j`.
- (*line 4*): Run a single trial of random simulation by calling `simv` with an new random seed, in this case, `j`. The output is matched, using the standard utility `grep`, for a successful exit.
- (*lines 5 – 8*): Check the output of analyzing the trial. If the maze was navigated successfully, report this fact to the terminal and exit.
- (*lines 9*): Upon failing all one hundred trials, report failure the terminal and exit.

6.2.2 Solution 2: `vlogm1`

The benefit of our meta-language is that the entire strategy can be effected as a single program that is much clearer, simpler, and more manageable. Along with the scaffolding assumed to be provided by the skeleton program of Section 5.2 and the utilities defined in Section 6.1, the `vlogm1` program

effecting our high-level strategy is implemented as shown in Figures 6.8 – 6.10. As per the skeleton, two functions must be defined to complete a working `vlogml` program: `strategy` and `reportResult`. A third function, `trial`, upon which `strategy` relies, is described separately for the sake of clarity.

```
1 trial = do
2   simulateRandM (Symbolic 10)
3   checkOutOfMazeCond
```

Figure 6.8: Single Random Simulation Trial.

Figure 6.8: `trial`. The first part of the program defines a function, `trial`, that mimics the SystemVerilog part of the traditional testbench above in Section 6.2.1.

- (*line 2*): Run random simulation for ten clock cycles. Recall that `simulateRandM` converts symbolic values in the input into concrete random values and then performs simulation.
- (*line 3*): Check the condition that we have successfully navigated out of the maze, returning a boolean.

```
1 strategy sim = aux 0 (evalStrat trial sim)
2
3 aux 100 simv = return Nothing
4 aux j    simv = do
5   result <- simv
6   if result
7     then return (Just j)
8     else aux (j+1) simv
```

Figure 6.9: Management of One Hundred Random Trials.

Figure 6.9: `strategy`. The meta-level logic that was earlier captured in the bash script is captured in our `vlogml` program with the function `strategy`, and an auxiliary function, shown in 6.9.

- (*line 1*): Call the auxiliary function, `aux`, with the first argument, representing the trial number, initialized to zero, and the second argument a function that uses `trial` and the initial simulation provided by the skeleton to produce a random simulation of the device each time it is called.
- (*line 3*): All one hundred trials have failed, return `Nothing`, which indicates failure.
- (*line 4*): Start trial `j`, where the current trial index is less than one hundred.
- (*line 5*): Run the random trial, binding the result of the call to `checkOutOfMazeCond`, which is called as part of executing `simv`, to the variable `result`.
- (*lines 6 – 8*): If the maze was successfully navigated, return that fact along with the current trial number; otherwise, increase the trial counter and perform the next trial.

```

1 reportResult (Nothing) = putStrLn "failed all 100"
2 reportResult (Just j) = putStrLn ("succeeded at " ++ show j)

```

Figure 6.10: Print Result of the Test.

Figure 6.10: `reportResult`. The overall success or failure of the strategy is reported to the terminal via the function `reportResult`.

- (*line 1*): `Nothing` is returned by `strategy` when the test failed; report this fact to the user.
- (*line 2*): `(Just j)` is returned by `strategy` when the test succeeds at the iteration indicated by `j`; report this fact to the user.

6.2.3 Comparison

Consider again the coverage closure feedback loop depicted in Chapter 1 as Figure 1.1. There we asserted that existing tools treat each of the three components, stimuli generation, simulation, and coverage analysis, as distinct pieces that are largely decoupled. Section 6.2.1 demonstrates this fact concretely, and is suggestive of the limitations imposed by this decoupling.

Indeed, the mapping between the components of Figure 1.1 and the pieces described in Section 6.2.1 is straightforward: stimuli generation is accomplished with the SystemVerilog program from Figure 6.6; simulation is accomplished through the compilation of SystemVerilog source code, together with the device-under-test, with VCS; and coverage analysis is accomplished with the utility `grep`. These pieces are more aptly described as being “cobbled together”, than as being strongly coupled and mutually reinforcing.

There are many reasons for this; we provide just two examples. One important observation is that the result of previous simulation runs cannot affect the stimulus generation mechanism, as SystemVerilog provides no straightforward way of receiving or analyzing information about previous simulation runs. This is crucial. As both the result of analysis and stimulus construction are available together withing `vlogm1`, incorporating feedback to drive stimuli generation from previous simulation results is easy. This capability is used to striking effect in the following section.

A second deficiency made apparent is simply the lack of availability of good tools that engineers may use to programmatically analyze coverage. The bash script used to coordinate the strategy relied on `grep`, a woefully inadequate tool for the task. In `vlogm1` on the other hand, with a full waveform history and the general purpose programming facilities of Haskell at one’s disposal, many possibilities are opened up. However, it should be noted that some deficiencies still remain in `vlogm1` in this regard; for example, syntactic coverage metrics are not necessarily apparent from waveforms.

6.3 Feedback

The `vlogm1` program presented in this section demonstrates how feedback from previous simulations can be used directly to *calculate* what stimuli should be generated in future runs. It is a modification of the program presented in

the previous section where the `trial` function takes an additional argument, namely, feedback from the previous simulation, which it then used as part of calculating the stimulus to be used during the next simulation. Specifically, instead of ten clock cycles of random simulation, the first two cycles of simulation are determined by the following algorithm:

- If the final location in the maze at the end of the previous simulation was greater than 3, then the first two clock cycles use the stimulus (`Concrete [0,0]`).
- If the final location in the maze at the end of the previous simulation was less than or equal to 3, then the first two clock cycles use the stimulus (`Concrete [1,0]`).

Afterward, the trial is completed by executing eight clock cycles of random simulation.

6.3.1 `vlogml`

The program is structured similarly to the one above in Section 6.2. The main difference is that a new function is substituted in the place of the function `checkOutOfMazeCond` that instead checks two conditions: successful exit from the maze, and whether the final location is greater than 3. This function is called `checkConditions`.

```
1 checkConditions = do
2   x <- checkOutOfMazeCond
3   y <- checkGT3Cond
4   return (x,y)
5
6 checkGT3Cond = queryM [expr| loc > 3 |]
```

Figure 6.11:

Figure 6.11: `checkConditions`. This function operates on the current simulation and checks two conditions, returning the result of these checks as a pair of boolean values.

- (*line 2*): Check the condition for the simulation having successfully navigated the maze. Bind the result to `x`, a boolean.
- (*line 3*): Check the condition for the simulation being such that its current location is greater than 3. Bind the result to `y`, a boolean.
- (*line 4*): Return the result of the checks.

```

1 trial gt3Cond = do
2   if gt3Cond
3     then simulateM [0,0]
4     else simulateM [1,0]
5   simulateRandM (Symbolic 8)
6   checkConditions

```

Figure 6.12:

Figure 6.12: `trial`. This function generates the next simulation trial. Unlike the corresponding function from Section 6.2 above, this function accepts a single argument. This argument is calculated by the `checkConditions` function with respect to the previously executed simulation.

- (*line 2*): Consider the final location in the maze obtained during the previous simulation, which is calculated by `checkConditions`, and passed to this function by the `aux` function (see below).
- (*line 3*): If the final location of the previous simulation was greater than 3, perform simulation for two clock cycles with the stimulus `[0,0]`.
- (*line 4*): If the final location of the previous simulation was less than or equal to 3, perform simulation for two clock cycles with the stimulus `[1,0]`.
- (*line 5*): Perform random simulation for the remaining eight clock cycles.
- (*line 6*): Call `checkConditions`, which is used in `aux` (see below).

```

1 strategy sim = aux 0 (\x -> evalStrat (trial x) sim) False
2
3 aux 100 simv gt3 = return Nothing
4 aux j   simv gt3 = do
5   (oom,gt3') <- simv gt3
6   if oom
7     then return (Just j)
8     else aux (j+1) simv gt3'

```

Figure 6.13:

Figure 6.13: strategy. This is the function, part of the skeleton program, that runs one hundred iterations of the trial simulation and handles the propagation of the feedback condition being used to partially determine the stimulus used during the immediately subsequent simulation.

- (*line 1*): Call `aux` with appropriately initialized values, including wrapping `trial` in such a way that the feedback condition may be conveyed as an argument.
- (*line 3*): All one hundred trials have failed, return `Nothing`, which indicates failure.
- (*line 4*): Start trial `j`, where the current trial index is less than one hundred.
- (*line 5*): Run one trial, providing the “greater-than-3” condition from the previous trial as an argument. The result, the pair of conditions from `checkConditions`, are bound to the variables `oom`, for “out-of-maze”, and `gt3'`, the greater-than-3 condition for the new trial.
- (*lines 6 – 8*): If the maze was successfully navigated, return that fact along with the current trial number; otherwise, increase the trial counter and perform the next trial.

6.3.2 Comparison

The condition that the final state of the previous simulation is greater than three, and the concrete stimuli choices, are of course contrived, as is the entire

maze device. However, the program demonstrates concisely one of the most important advantages of our meta-language: the ability to programmatically, in a completely general and user-specified way, use information from previous simulation runs as feedback that affects the calculation of future stimuli. Doing something similar with existing tools would be extremely awkward; for example, in a SystemVerilog-oriented methodology, one would probably have to write external functions in C that are then called.

6.4 Backtracking

The example presented in this section is interesting because it solves the maze problem in a very straightforward way, and yet would be extremely awkward to effect with other existing tools, either alone or in combination. The `vlogml` code is presented across Figures 6.14 – 6.16 and efficiently solves the maze using a simple backtracking strategy that coordinates multiple simulations, and uses feedback obtained from those simulations, all at once.

The program performs a systematic search of the reachable state space. At any given location in the maze, the program first checks if the current location is the target location representing a successful exit from the maze, in which case the search abruptly ends and success is signaled. Second, the program determines the current location and triggers a backtracking operation if it has been visited previously. If the current location has not been visited previously and is not the exit of the maze, it causes the program to walk systematically in all possible directions and repeat.

6.4.1 `vlogml`

Figure 6.14: `btSearch`. This function performs the basic backtracking logic described above. Its first argument is a list of previously visited locations and its second argument an escape continuation (more on this below). In addition, it is implemented within Haskell’s continuation monad [23] as this is needed to elegantly handle escaping the search; it also requires lifting functions such as `evalM`.

- (*line 2*): Evaluate the current location in the maze, binding the result, a value of type `Expression`, to the variable `x`.

```

1 btSearch prevs exit = do
2   x <- evalMM [expr| loc |]
3   when (x == [expr| 7 |]) (exit (x:xs))
4   if x 'elem' prevs
5     then return prevs
6     else do
7       prevs' <- branch (x:prevs ) exit 1
8       prevs'' <- branch ( prevs') exit 0
9       return prevs''
10
11 evalMM = lift . evalM

```

Figure 6.14:

- (*line 3*): Check the current location in the maze against the location indicating a successful exit, `[expr| 7 |]`. If the maze has been successfully navigated, use the escape continuation to immediately exit.
- (*line 4*): If the exit condition is not met, then check if the current location has been visited previously.
- (*line 5*): If the current location has been visited, return. As we will see below with the `branch` function, this triggers backtracking.
- (*lines 6 – 9*): If the current location has not been visited previously, systematically search both possible branches from the current simulation by calling `branch`, detailed next.

```

1 branch prevs exit i = do
2   backtrackSim <- get
3   simulateMM (Concrete [i])
4   btSearch prevs exit
5   put backtrackSim
6
7 simulateMM = lift . simulateM

```

Figure 6.15:

Figure 6.15: branch. This function performs the operation corresponding to the third case described above, where the current location is both newly seen and not the exit of the maze. It advances simulation one step, moving in the “direction” given as the third argument; the first two arguments are just as for `btSearch`.

- (*line 2*): Save the current simulation context for the purposes of backtracking.
- (*line 3*): Advance simulation according to the given input.
- (*line 4*): Recursively call `btSearch` on the newly generated branch. If this branch provides a successful exit from the maze, the control flow will be modified using the escape continuation.
- (*line 5*): This statement is executed only when the branch fails to find a way out of the maze. In such an event, it simply backtracks.

```
1 testbench = do
2   seen <- runContT (callCC aux) return
3   sim <- get
4   if [expr| 7 |] 'elem' seen
5     then return (Just sim)
6     else return Nothing
7
8 aux exit = btSearch [] exit
```

Figure 6.16:

Figure 6.16: strategy. This function is part of the skeleton and is constructed in two parts. The first binds an escape continuation so that the search for a simulation out of the maze can be ended as soon as one is found. The second checks the result of the search.

- (*line 2*): Call `btSearch`, with the arguments appropriately initialized. `aux` initializes the list of previously seen states to the empty list, and `callCC` provides the escape continuation.

- (*line 3*): Bind the simulation resulting from the backtracking search to the variable `sim`.
- (*lines 4 – 6*): Check for success and return an appropriate result.

6.4.2 Comparison

One of the things that our meta-language excels at, and which we mean to emphasize with the above example, is as an alternative to traditional directed testing. Directed testing is an extremely labor-intensive process typically reserved for when more automated means of testing, usually constrained randoms, fail to attain some needed coverage goal. It is a reality of contemporary verification practice and its impact on verification and design cycle is significant, as the following quote, recalled from the Chapter 1, notes:

Today it is not uncommon to go from 0% to 80% coverage in just a few days after the [constrained-random] testbench is up & running.

What about the remaining 20%?

Today, one of the long poles in verification is coverage convergence – the process where verification engineers analyze the coverage generated by constrained-random tests, identify gaps or “coverage holes”, and adjust the verification environment to try to fill the gaps. If you think this sounds laborious, repetitive and time-consuming you’d be correct. I’ve spoken to chip designers who say a third of their overall chip development schedule is spent in this iterative, largely manual, coverage convergence phase of verification.

[9, July 6, 2010]

Roughly speaking, traditional directed testing is an iterative process that repeatedly cycles through the following tasks until, for example, the target coverage goal has been discharged: create/modify stimulus, run simulation to generate waveforms, analyze waveforms. The process is inefficient because, even when the dependence between iterations of this loop can be described

in an algorithmic way, the limitations of existing tools force the first and third steps to be done manually and undertaken directly by an engineer. It is a result of the current decoupling of stimulus generation, simulation, and analysis.

Of course, the main purpose of our meta-language and `vlogml` is to enable a verification engineer to, in a straightforward way, write a program that orchestrates the entire feedback loop. Depending on the task at hand, different strategies will be called for. In the case of directed testing, the strategies will be highly-targeted and specialized searches for a simulation that satisfies, for example, a particular coverage goal. This is exactly what the backtracking strategy demonstrates.

6.5 Breadth-First

This section presents a `vlogml` program where a set of random simulation traces are generated and analyzed “breadth-first”. Starting with one hundred copies of the initial state of the device, as a value of type `Simulation`, we iterate over each copy extending simulation one cycle with a random stimulus and checking to see if we have successfully exited from the maze. If any of the one hundred simulations has exited, the program returns successfully; otherwise, the simulations are extended one more cycle of random simulation. At a depth of ten clock cycles, the program aborts and reports failure.

6.5.1 `vlogml`

The program is presented piece-wise as three functions, `singleStep`, `depth`, and `strategy`, which are given as Figures 6.17 – 6.19, respectively. `singleStep` extends a given simulation one clock cycle, `depth` manages the application of the single step of simulation over the one hundred individual simulation instances, and `strategy` simply initializes `depth` and wraps it appropriately to account for the assumed skeleton program.

Figure 6.17: `singleStep`. This function extends a given simulation one clock cycle, using random simulation.

- (*line 2*): Perform one cycle of random simulation.

```

1 singleStep = do
2   simulateRandM (Symbolic 1)
3   checkOutOfMaze

```

Figure 6.17: One Randomized Simulation Step.

- (*line 3*): Determine if the additional cycle of simulation has resulted in a successful exit from the maze.

```

1 depth :: Int -> [Simulation] -> Strategy (Maybe Simulation)
2 depth 10 sims = return Nothing
3 depth j sims = do
4   xs <- mapM (runStrat singleStep) sims
5   case find fst xs of
6     Just (_,sim) -> return (Just sim)
7     Nothing      -> do
8       let snds = snd . unzip
9           depth (j+1) (snds xs)

```

Figure 6.18: Breadth-First Logic

Figure 6.18: depth. The main logic controlling the breadth-first strategy is implemented according to this function. The function's first argument denotes the current depth and the second is the list of one hundred simulations being operated on. The function returns either a simulation that has successfully navigated the maze, or `Nothing`.

- (*line 2*): A depth of ten has been reached without finding a successful simulation trace. In this case we return `Nothing`, signaling that the strategy has failed.
- (*line 3*): Start a new iteration where the simulation depth is increased by one clock cycle.
- (*line 4*): Extend the one hundred simulations by a single step, recording for each simulation a pair containing the result of the success check

and the resulting simulation. Therefore, the type of the variable `xs` is deduced to be

```
xs :: [(Bool,Simulation)]
```

- (*line 5*): Search the resulting list for a simulation that successfully exited the maze, as indicated by the first component of the elements of `xs`.
- (*line 6*): One of the simulations succeeded, return it and end the search.
- (*lines 7 – 9*): None of the simulations succeeded, gather all of them using the function `snds` and call `depth` recursively.

Figure 6.19: strategy. This function initializes `depth` and wraps its execution so as to fit within the framework provided by the skeleton program in Section 5.2. Its single argument is the initial simulation provided by the call to `start` in the skeleton program.

```
strategy sim = evalStrat $ depth 0 (replicate 100 sim)
```

Figure 6.19: Breadth-First Logic

- (*line 1*): Finally, `strategy` is defined simply by calling `depth` with appropriate `Call depth` with an initial depth of zero and one hundred copies of the initial simulation, which are constructed using the standard Haskell function `replicate`. 6.19.

6.5.2 Comparison

The breadth-first strategy demonstrates the ability in our meta-language to generate simulations in a fine-grained and incremental manner. Indeed, the backtracking strategy does the same, but there our intention was to focus on backtracking, whereas the breadth-first strategy clearly emphasizes the coordination of multiple simulations, all of which are being generated incrementally. Traditional simulators make stopping and restarting simulations

cumbersome, and in most cases infeasible; one traditional method, using a scan-chain, has many well-known drawbacks, and would be horribly inefficient for fine-grained simulation generation.

6.6 Symbolic Execution

Although not a benefit exclusive to `vlogml`, its ability to uniformly handle both concrete and *symbolic* simulation and, via the `VlogMetaLang.SMT` library (see Section 5.8), to query an SMT solver, allows for some very interesting testing programs to be created. This section simply introduces the symbolic simulation and SMT capabilities with an extremely simple example: ten clock cycles of symbolic simulation are executed, and then the SMT solver is called in an attempt to resolve a concrete simulation that successfully exits the maze. The following section demonstrates a more interesting example that combines symbolic simulation and concrete simulation and makes crucial use of the meta-level features of `vlogml`.

6.6.1 `vlogml`

The implementation is split across two functions, `strategy` and `checkResult`, which are given as Figures 6.20 and 6.21, respectively. `strategy` contains the high-level logic and `checkResult` handles applying the result from the SMT solver.

```
1 strategy = evalStrat $ do
2   simulateM (Symbolic 10)
3   x <- checkOutOfMazeSMT
4   checkResult x
```

Figure 6.20: Symbolic Simulation and SMT Call.

Figure 6.20: `strategy`. This function takes as its only argument the initial simulation provided by the call to `start` in the skeleton program from Section 5.2.

- (*line 2*): Execute ten cycles of symbolic simulation.
- (*line 3*): Invoke the SMT solver and bind the result, which is of type (`Maybe Substitution`), to the variable `x`. A substitution being returned successfully then yields a concrete stimulus leading out of the maze.
- (*line 4*): Call `checkResult` to apply the result from the SMT solver appropriately (see below).

```

1  checkResult (  Nothing) = return Nothing
2  checkResult (Just subst) = do
3    sim <- get
4    let sim' = concretize subst sim
5    return (Just sim')
```

Figure 6.21: Concretization of a Symbolic Simulation.

Figure 6.21: `checkResult`. The function takes a single argument, which is the result from the SMT solver and, if the solver was successful, returns a concrete simulation that successfully exits the maze.

- (*line 1*): If the SMT solver fails to find a substitution, return `Nothing`.
- (*line 2*): The SMT solver succeeded in finding a substitution, which, through pattern matching, is bound to the variable `subst`.
- (*line 3*): Bind the result of symbolic simulation to a variable `sim`.
- (*lines 4 – 5*): Apply `subst` to the symbolic simulation using `concretize` and return the result.

6.6.2 Comparison

No widely available Verilog simulator that we are aware of allows for the user to apply symbolic simulation directly, let alone resolve the resulting symbolic context to concrete values using technology such as an SMT solver. For VHDL, there is a symbolic simulator available [83], but is not built to be controlled in the same fine-grained manner as in `vlogml`.

6.7 Combined Concrete and Symbolic Simulation

This section presents a `vlogm1` program implementing a combined random/symbolic strategy. The idea for the strategy comes from [34] and serves as the foundational idea underlying the popular Magellan tool [93, 92]; it is also known as “hybrid concolic testing” [63]. In addition to combining random and symbolic simulation, the program also makes crucial use of an SMT solver and the meta-level features of `vlogm1`.

The core of the strategy operates according to the following four steps:

1. Random simulation is run for some number of cycles, after which the resulting simulation is checkpointed.
2. Symbolic simulation is run for some number of cycles.
3. An SMT solver is applied to the simulation obtained after the combined random/symbolic simulation. If the solver succeeds, the algorithm finishes successfully.
4. If the solver fails, the simulation checkpointed in step (1) is returned to and, starting from this simulation, the algorithm returns to step (1).

6.7.1 `vlogm1`

The implementation is split across essentially three separate functions named `performHybridSimulation`, `checkResult`, and `strategy`, which are given as Figures 6.22 – 6.24, respectively. The first of the three performs steps (1) and (2) above, returning the checkpointed simulation for backtracking purposes, if needed. The second, performs steps (3) and (4), except for the actual call to the SMT solver. The third puts the two pieces together and performs the SMT call.

Figure 6.22: `performHybridSimulation`.

- (*line 2*): Perform random simulation for five clock cycles.
- (*line 3*): Bind the resulting simulation to the variable `failSim`, needed we are required to backtrack.

```

1 performHybridSimulation = do
2   simulateRandM (Symbolic 5)
3   failSim <- get
4   simulateM      (Symbolic 5)
5   return failSim

```

Figure 6.22: Combined Concrete and Symbolic Simulation Trial.

- (*line 4*): Perform symbolic simulation for five clock cycles.
- (*line 5*): Return failSim.

```

1 checkResult (Nothing  ) failSim = do
2   put failSim
3   aux (j+1)
4 checkResult (Just subst) failSim = do
5   sim <- get
6   let sim' = concretize subst sim
7   return (Just sim')

```

Figure 6.23: End of Trial Logic.

Figure 6.23: checkResult. The second function we define, `checkResult`, is assumed to receive the result of the SMT solver as its first argument and the previous result of random simulation as its second argument.

- (*line 1*): SMT solving failed.
- (*line 2*): Restore the previous result of random simulation, before symbolic simulation was attempted.
- (*line 3*): Start again. In terms of the algorithm as described above, this denotes a return to step (1).
- (*lines 4 – 7*): SMT solving succeeded, get the symbolic simulation and apply the substitution returned by the SMT solver to it. Return the result.

```

1 strategy = aux 0
2
3 aux 100 = return Nothing
4 aux j    = do
5   failSim <- performHybridSimulation
6   x <- checkOutOfMazeSMT
7   checkResult x failSim

```

Figure 6.24: Algorithm Similar to [34].

Figure 6.24: strategy.

- (*line 1*): Call auxiliary function with iteration counter initialized to 0.
- (*line 3*): After one hundred trials, abort and signal failure.
- (*line 4*): Perform another trial.
- (*line 5*): Perform steps (1) and (2) above, binding to `failSim` the random simulation needed for backtracking.
- (*line 6*): Invoke the SMT solver to resolve the variables in the symbolic simulation to values that successfully exit the maze.
- (*line 7*): Using the result of the SMT solver and the backtracking strategy, perform step (4) of the algorithm as described above.

6.7.2 Comparison

One very noteworthy feature of this testbench is that it mimics the essential strategy of Synopsys' Magellan tool, which is also based on [34]. This is a powerful result, because not only is Magellan expensive, in monetary terms, but by virtue of being a separate executable, to the user it is essentially a black box that is not very customizable. The fact that the core strategy of Magellan can be mimicked in just a few lines of code in `vlogm1`, for free and in a way completely customizable by a user, demonstrates in a powerful way the new opportunities available to a verification engineer by taking up and using `vlogm1`. Of course, it bears mentioning that Magellan, in addition to the core strategy, is also thought (it is proprietary, so one cannot say for sure)

to include many complex heuristics that are not mimicked in the testbench developed in this section.

CHAPTER 7

CASE STUDIES

Whereas the goal of the previous chapter was simply to demonstrate the capacity of our meta-language, through `vlogm1`, for novel testing programs, the purpose of this chapter is to show `vlogm1` effectively applied to more substantial devices. To do so, we selected two devices, each between one thousand and two thousand lines of Verilog, that we develop testing programs for in `vlogm1`.

1. The first device [33] is an implementation of a serial bus-master controller for the I²C protocol [82]. It supports advanced features such as multi-mastering and clock stretching, which we exercise with a `vlogm1` program, uncovering a potential bug.
2. The second device is a small microprocessor that we developed. Two `vlogm1` programs are then presented demonstrating, first, a targeted strategy that is resilient to organizational changes to a memory, and, second, the use of an SMT solver to automatically resolve partial assembly programs to interesting stimuli.

7.1 I²C Bus-Master Controller

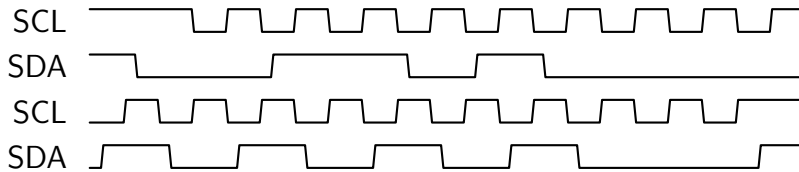
The I²C-bus protocol is defined by [82]. The particular bus-mastering controller that we worked with is given a brief overview of next, we then go into some detail on the arbitration and clock stretching mechanisms of I²C, as these are crucial to understanding the coverage goal that we targeted. Finally, we detail a `vlogm1` strategy that exposes a potential bug in the controller that we were working with.

7.1.1 The Device

Our case study uses an open-source controller with support for multi-mastering and clock stretching [33]. It is designated as an “OpenCores Certified” project, meaning that it is relatively mature, having a testbench, documentation, and in many cases, and indeed in this particular one, proved on FPGA. The implementation consists of three modules, and is roughly one thousand lines of Verilog source code.

I²C is a serial bus consisting of two bidirectional open-drain lines, **SCL**, used for clocking, and **SDA**, used for data. The open-drain design means that any device, master or slave, pulling down a line will override any other device driving it high. The bus was developed by Philips Electronics and is used to drive a wide variety of low-speed peripherals, such as cellphone displays.

A mastering device can initiate two types of transactions, read and write. All transactions begin by sending a “start bit” (negative edge on **SDA** with **SCL** high), then send a sequence of bytes, each separated from the previous one by an acknowledge bit, and finally a “stop bit” (positive edge on **SDA** with **SCL** high) to end the transaction. The first byte always consists of a seven bit slave address followed by a read/write bit, indicating the type of transaction. For example, the timing diagram

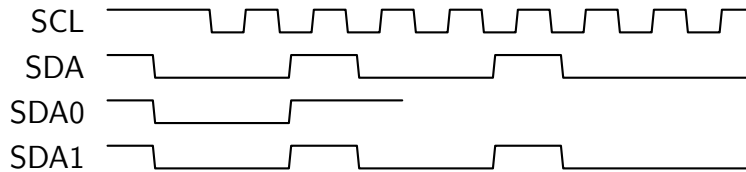


corresponds to a write transaction to slave address $011_0100 = 0x34$, sending a single byte of data, $1010_1010 = 0xAA$.

7.1.2 Arbitration and Clock Stretching

As a multi-mastering bus, I²C requires an arbitration process to determine which master will get control of the bus when two or more want it simultaneously. After initiating a transaction, each mastering device monitors the level of **SDA**. If while trying to keep **SDA** high it instead, due to the open-drain design, finds **SDA** to be low, then the controller assumes another mastering device is pulling it down and aborts its transaction, thereby losing the arbitration.

The following timing diagram shows an example of the arbitration process, where one mastering device tries to initiate a transaction to address 011_0100, and a second device simultaneously initiates a transaction to 010_0100.



The signals `SDA0` and `SDA1` are what each controller attempts to drive `SDA` toward. The device addressing `011_0100` loses arbitration at the third bit, and stops trying to drive `SDA`.

Clock stretching works similarly to arbitration, but on the `SCL` line. Any slave device taking part in a transaction can essentially force the mastering device to stop and wait in the middle of a transaction it is directing. It does this by holding the `SCL` line low. When the slave releases `SCL` the master may resume the transaction.

7.1.3 I²C Stimulus Specification

The case study connects up a simple multi-master system, with two mastering controllers connected to the bus, denoted `m0` and `m1`. At a high level, the strategy aims to explore subtle timing variations during arbitration. To evoke an arbitration, `m0` will send data byte `0000_0010` and `m1` will send data byte `0000_0011`. Based on the arbitration mechanism described above, one would expect `m0` to win arbitration and the slave to receive the data value `0000_0010`.

Before describing the logic of the testbench, we first develop the data types and associated instance of (`Input a`) type class that are used to drive the multi-mastering configuration. The first data type, `I2Ctransaction` yields transactions on a single master, and the second, `I2Cx2`, groups two lists of such transactions together, which are then used to drive the two controllers concurrently. These are shown as Figures 7.1 – 7.3.

Figure 7.1: `I2Ctransaction`.


```
data I2CTransaction =
    Read Int [Int]
  | Write Int [Int]
  | Delay Int
```

Figure 7.1:

- (*line 2*): This constructor is used to initiate a read transaction. The first argument is the address of the slave device to read from, and the second argument is a list of addresses within the device to read from.
- (*line 3*): This constructor is used to initiate a write transaction. The first argument is the address of the slave device to write to, and the second argument is a list of values to communicate to the device, typically both internal addresses and data.
- (*line 4*): This constructor is used to cause the device to idle. The argument is the number of clock cycles to idle.

```
data I2Cx2 = I2Cx2 {
    ctrlr0 [I2CTransaction]
  , ctrlr1 [I2CTransaction]
}
```

Figure 7.2:

Figure 7.2: `I2Cx2`. This is a record data type that contains a list of transactions for each of the two mastering controllers that are part of our device setup. We label the two controllers `ctrlr0` and `ctrlr1`.

```
instance Input I2Cx2 where
    toInput x = ...
```

Figure 7.3:

Figure 7.3: (Input a). The implementation of this type class is quite involved, but is straightforward in the sense that one must simply conform to the Wishbone interconnection interface [25], which is used by the I²C-bus mastering controller that we are testing [33]. Therefore, we omit the definition from this dissertation, though it is available in full at [46].

7.1.4 vlogm1 Program

The high-level strategy that we implement to test the multi-mastering capabilities of the I²C-bus controller [33] is to initiate write transactions on the devices *almost* concurrently, with some small amount of delay between the two. A number of simulation trials are run, each time increasing the delay an additional clock cycle. The testing program assumes the skeleton from Section 5.2, and is structured as shown in Figure 7.4, where the `trial` function is called repeatedly with different small delays, up to a threshold of three.

```
1 strategy sim = aux 0 (\j -> evalStrat (trial j) sim)
2
3 aux 5 f = return Nothing
4 aux j f = do
5     success <- f j
6     if success
7         then return (Just j)
8         else aux (j+1) f
```

Figure 7.4:

```
1 trial j = do
2     let c0 = [ Write 777 [0,2]]
3         c1 = [Delay j, Write 777 [0,3]]
4     simulateM (I2Cx2 c0 c1)
5     queryM [| slave.data == 2 || slave.data == 3 |]
```

Figure 7.5:

Figure 7.5: `trial`. This function performs a simulation trial. Its single argument is the amount of delay that should exist between the initiation of write transactions between the two mastering controllers.

- (*line 2*): This is the input stimulus for `ctrlr0`. It is just a write of the two bytes, 0 and 2, to a device with identifier 777.
- (*line 3*): This is the input stimulus for `ctrlr1`. It consists of a delay, relative to the argument of `trial`, and a write of the two bytes, 0 and 3, also to the device identified by the value 777.
- (*line 4*): Perform simulation using the above stimulus.
- (*line 5*): Check the result of the slave device, `slave`, which is identified by value 777, and ensure that one of the writes successfully completed.

7.1.5 Result

To be correct, the slave should receive either a value of 2 or a value of 3, though because of how the arbitration process works we would expect it to get 2. However, for `trial 3`, the slave gets a value of 0, which seems to be a bug in the device; for `trial 2` or `trial 4`, a value of 2 gets successfully transmitted.

The problem seems to stem from the logic controlling a signal called `clk_en`, which, when asserted, causes the internal state machine of the controller to go to the next state. The inserted delays typically cause a clock stretching event to occur, but the logic controlling `clk_en` gives preference to stepping the state machine instead of the stretching event. By changing this priority, the device operates correctly.

7.1.6 Discussion

What is interesting about the above `vlogm1` program is that it executes a set of distinct simulations, systematically testing different amounts of delay in the start of the write transaction on the second controller. To do this with existing tools would require either a script along the lines of the one from Section 6.2, or to write one large test with multiple resets of the device. In

the first case, we run into the undesirable situation of splitting the test into two distinct parts, and in the second case it becomes more difficult to debug errors because one must search through a combined, larger simulation result, rather than the natural individual simulations that one desires.

7.2 Microprocessor

The second case study that we present operates on a small microprocessor, with multiple possible configurations of the data memory. Two `vlogm1` programs are presented, one that aims to demonstrate the writing of a highly-targeted strategy that is much more resilient to design changes than usual directed tests, and a second example that demonstrates how an SMT solver can be used to automatically resolve partially given assembly program stimuli to concrete assembly programs that effect interesting microprocessor conditions.

7.2.1 Overview

The microprocessor under consideration is a single-cycle design, with eight architected registers, `R0` – `R7`, and separate instruction and data memories. It supports eight instructions, including arithmetic, conditional branching, and load and store from data memory, and each instruction is 32-bits. The data memory is two-way banked.

To execute a program on the microprocessor, one first asserts reset, followed by flashing a sequence of instructions, one per clock cycle, on a 32-bit port connected to the module. As the instructions are being transmitted and written into the instruction memory, a wire named `exec` must be held low. At the point that `exec` is asserted, it is assumed that program transmission has ended, and the microprocessor then starts executing the program starting with the instruction at memory location zero.

7.2.2 Assembly Language

The microprocessor under consideration has a small instruction set consisting of eight instructions. To construct assembly programs for this microprocessor within `vlogm1`, we employ the data types shown in Figures 7.6 – 7.9 and

must, in addition, construct an appropriate instance of the (`Input a`) type class, as shown in Figures 7.10 – 7.11.

```
data Register
data Instruction
data Assembly
```

Figure 7.6:

Figures 7.6 – 7.9.

- **Register:** This data type is an enumeration of the eight architected registers supported by the microprocessor instruction set. It is defined as shown in Figure 7.7.

```
data Register = R0 | R1 | R2 | R3 | R4 | R5 | R6 | R7
```

Figure 7.7:

- **Instruction:** This data type defines the eight instructions supported by the microprocessor instruction set, plus a symbolic “wildcard” instruction, given by a constructor `ANY`, which can represent any of the eight concrete instructions. It is defined as shown in Figure 7.8. The instructions, in the order given in the figure, represent: moving an fixed value (immediate) to a register, addition, bit-wise exclusive-or, rightward shift, load word, store word, branch on not-zero, halt execution.
- **Assembly:** This data type defines assembly programs, which in the case of our simple microprocessor, are essentially just lists of assembly instructions. Figure 7.9 presents the concrete definition.

Figures 7.10 – 7.11. To convert assembly programs into a value of type `Input`, we define a set of “packing” functions that yield bit-accurate representations of instructions and then use these functions to generate bit-vectors

```

1 data Instruction =
2   -- rdest  immediate
3   IMM Register Int
4
5   -- rdest  rsrc1  rsrc2
6   ADD Register Register Register
7   | XOR Register Register Register
8   | SFR Register Register Register
9
10  -- rdest  raddr
11  | LDW Register Register
12  | STW Register Register
13  | BNZ Register Register
14
15  --
16  | HALT
17  | ANY

```

Figure 7.8:

```

data Assembly = Assembly [Instruction]

```

Figure 7.9:

that are flashed on the microprocessor’s input ports during the bootstrap sequence.

Example Programs. Two very simple example assembly programs are shown in Figures 7.12 and 7.13. They demonstrate how assembly program stimuli may be specified in a natural way within a `vlogm1` program, similar to the kind of syntax accepted by many assemblers.

- (*Figure 7.12*): This first program consists of four assembly instructions. The first two instructions put the constant one into registers R0 and R1, respectively; the third instruction adds these two registers, placing the result in register R2; finally, the fourth instruction halts the microprocessor.

```

registerPack R0 = 0
registerPack R1 = 1
registerPack R2 = ...

assemblyPack (IMM rdest x) =
  ors [shiftL opcodePack 9, shiftL rdestPack 6, x]
  where opcodePack = 6
         rdestPack = registerPack rdest
assemblyPack (ADD rdest rsrc1 rsrc2) = ...

```

Figure 7.10:

```

instance (Input Assembly) where
  toInput (Assembly xs) = CycleBased (foldr f [] ys)
  where ys = zip xs [0..]
        f (j,inst) = ...

```

Figure 7.11:

- (*Figure 7.13*): The second program is the same as the first, except that the second instruction is modified to be a *symbolic* value representing any instruction. As an example, if simulation was run on this program, we could query the SMT solver for a concrete instruction (more specifically, its microprocessor-level representation) such that the value in register R2 at the end of the program was, say, two. In this case, it could return the same instruction used in the first program.

7.2.3 vlogm1 Example 1

As the first demonstration of applying `vlogm1` to our microprocessor, we will consider a coverage goal that has to do with the organization of the data memory. Recall from above that the data memory is organized into multiple banks. Our goal will be to write a program such that two consecutive stores to the data memory write to different words in the same bank. In addition, we will develop a `vlogm1` program that attains this coverage goal in such a way that it is robust with respect to changes in the number of banks and how

```
Assembly
[ IMM R0 1
, IMM R1 1
, ADD R2 R0 R1
, HALT
]
```

Figure 7.12:

```
Assembly
[ IMM R0 1
, ANY
, ADD R2 R0 R1
, HALT
]
```

Figure 7.13:

the banks are organized.

The strategy employed iteratively runs simulation of an assembly program that performs two stores to the data memory, each time increasing the stride between the addresses stored up to some threshold. The implementation of the program assumes the skeleton program from Section 5.2 and is otherwise constructed from four functions:

- **program**: This function generates a small assembly program containing a pair of store instructions at a stride given by the argument.
- **checkGoal**: This function checks the property corresponding to the coverage goal. Although its implementation is straightforward, without a comprehensive `vlogml` library yet for describing functional coverage goals, it is somewhat verbose. Therefore, the definition of `checkGoal` is omitted here (see [46]).
- **trial**: This function performs a single simulation trial, using the function `program` to generate the stimulus and to maintain a stride counter, and then checks the coverage goal condition.

- `iterate`: This function iteratively calls `trial` with different strides and manages the exit conditions.

```

1 program j = Assembly
2   [ IMM R0 1
3     , IMM R1 (1+j)
4     , IMM R2 63
5     , STW R2 R0
6     , STW R2 R1
7     , HALT
8   ]

```

Figure 7.14:

Figure 7.14: program. A call to `program j` generates an assembly program whose meaning is determined as follows.

- (*line 2*): `R0` will be used as the first store address.
- (*line 3*): `R1` will be used as the second store address, which is offset from the first address by the amount given in the argument of the function.
- (*line 4*): `R2` will be used as the source of the value that is stored to data memory.
- (*line 5*): Execute the store to the first address.
- (*line 6*): Execute the store to the second address.
- (*line 7*): Halt.

```

1 trial j = do
2   simulateM (program j)
3   checkGoal

```

Figure 7.15:

Figure 7.15: `trial`.

- (*line 2*): Run a simulation of the microprocessor using the assembly program generated through calling `program`.
- (*line 3*): Check to see if the simulation successfully effected the coverage goal, meaning that two consecutive stores to the data memory occurred at different addresses in the same bank.

```
1 iterate 10 f = return Nothing
2 iterate j f = do
3   success <- f j
4   if success
5     then return (Just (program j))
6     else iterate (j+1) f
```

Figure 7.16:

Figure 7.16: `iterate`. The first argument of `iterate` is the current stride value and the second argument is a wrapped version of `trial` that will be called (see definition of `strategy` next).

- (*line 2*): At a stride of ten, the strategy fails, returning `Nothing`.
- (*line 3*): Call `trial` with the current stride and bind the result of `checkGoal` to the variable `success`.
- (*line 5*): The current stride was successful, return the associated assembly program.
- (*line 6*): The current stride failed, increment the stride value and proceed with the next iteration.

```
strategy sim = iterate 0 (\j -> evalStrat (trial j) sim)
```

Figure 7.17:

Figure 7.17: strategy. This function simply calls `iterate` with appropriately initialized values. Note that the second argument binds a function that maps strides to simulation trials that return a result with respect to the initial simulation provided by `start` in the skeleton program from Section 5.2.

7.2.4 Discussion

What is interesting about the above `vlogm1` program is that it is both highly targeted and resilient to changes in the organization of the data memory's internal banking scheme. A traditional directed test only accomplishes the first, and upon regression in a changed organization, would need to be considered directly by an engineer and rewritten.

7.2.5 `vlogm1` Example 2

The second `vlogm1` program we develop for the microprocessor is one that uses symbolic simulation and an SMT solver to automatically generate part of an assembly program. In many instruction sets, multiple instructions are needed to generate certain bit-patterns within a register, encouraging the use of complicated bit-manipulations that lead to subtle bugs. The `vlogm1` program that we describe mitigates this problem simply by stating the bit-pattern desired and invoking an SMT solver. As a result, it generates part of an assembly program that yields exactly the constant desired in register R2.

```
1 strategy sim = evalStrat aux sim
2
3 aux = do
4   simulateM program
5   x <- solveSim
6   checkResult x
```

Figure 7.18:

Figure 7.18: strategy and aux. These functions stage the high-level logic of the program, which in turn is constructed from three component functions, `program`, `solveSim`, and `checkResult`, elaborated on below.

- (*line 1*): Call `aux` appropriately wrapped and with the initial simulation obtained from the call to `start` in the skeleton program.
- (*line 4*): Perform simulation of the assembly program shown in Figure 7.19. As indicated above, this results in a symbolic simulation as one of the instructions in the program is left unspecified via the `ANY` constructor.
- (*line 5*): Invoke the SMT solver with an appropriately formed query to generate the needed instruction.
- (*line 6*): Consider the result obtained from the SMT solver and provide, if successful, an appropriate concrete simulation.

```
1 program = Assembly
2   [ IMM R0 2
3     , ANY
4     , ADD R2 R0 R1
5     , HALT
6   ]
```

Figure 7.19:

Figure 7.19: program.

- (*line 2*): Load the value 2 into register R0.
- (*line 3*): Leave the second instruction symbolic.
- (*line 4*): Add registers R0 and R1 and place the result in register R2.
- (*line 5*): Halt.

```

1 solveSim = do
2   x <- evalM [| r2 == 65 |]
3   lift (smt Nothing x)

```

Figure 7.20:

Figure 7.20: solveSim.

- (*line 2*): Evaluate the expression that indicates a successful simulation. The constant that we attempt to put into the register R2 is 65, which overflows the 6-bit immediate field of IMM and therefore requires multiple instructions to effect.
- (*line 3*): Invoke the SMT solver with the above-bound evaluated expression.

```

1 checkResult ( Nothing) = return Nothing
2 checkResult (Just subst) = do
3   sim <- get
4   let sim' = concretize subst sim
5   return (Just sim')

```

Figure 7.21:

- (*line 1*): If the SMT solver fails to find a substitution, return Nothing.
- (*line 2*): The SMT solver succeeded in finding a substitution, which, through pattern matching, is bound to the variable `subst`.
- (*line 3*): Bind the result of symbolic simulation to a variable `sim`.
- (*lines 4 – 5*): Apply `subst` to the symbolic simulation using `concretize` and return the result.

7.2.6 Discussion

What this example demonstrates is an advanced use of symbolic simulation to generate a more complex data type, namely, assembly programs. It is

interesting too because it invokes the solver for a case that one would expect to succeed, generating a specific constant value in a register, but when done by hand could be error prone. This kind of situation is what we envision to be the best use of SMT solving: to discharge tedious constraints, but ones that are not computationally extremely difficult.

CHAPTER 8

SEMANTICS: VERILOG

This chapter addresses the semantics of Verilog [36], perhaps the most widely used language for digital circuit design since the early 1990s. A formal executable semantics for a substantial subset of Verilog is developed, and its utility in exploring the space of executions for small programs is demonstrated. A rewriting logic semantics not only lends further evidence to the suitability of rewriting logic as a semantics framework [77], but is also important for practical reasons: many Verilog-based tools exist, including several formal verification tools from which users expect strong guarantees about the behavior of their programs. Having a formal executable semantics provides a standard by which inconsistencies in specific tools can be uncovered. In fact, this work has uncovered several such inconsistencies.

8.1 Disclaimer

The purpose of this disclaimer is to state clearly what our semantics is *not*, so that the actual contributions made by our effort are not obscured. First, our semantics is *not comprehensive*, though we believe it to be more comprehensive than any other formal semantics of Verilog attempted to date. Second, our semantics is *not definitive*, though it is based on careful readings of the standard [36] – the *only* definitive characterization of Verilog –, discussions with long-time developers of Verilog simulators, and experiments using well-regarded simulators to interpret small Verilog programs.

8.2 Contributions

Verilog is immense; the official standards document [36] is almost 600 pages long, and our formalization effort presented here is therefore necessarily

incomplete and selective in its coverage. Our formalization is, however, the most extensive of any such effort attempted to date, so far as we are aware; this is the first contribution of our semantics. Both “synthesizable” and “behavioral” aspects of the language are handled, in a uniform way, so that the intent of the standard is followed as closely as possible. Our belief is that, because of this, support for additional constructs not currently covered can be incorporated in a straightforward way.

The second contribution of our formal semantics is that it is *executable*, which of course is one of the main distinguishing features of rewriting logic semantics [77]. Executability is had via Maude and yields the usual analysis possibilities, including simulation and state-space search. As Verilog is highly concurrent and nondeterministic, this latter ability is possibly the most useful in that it provides the unique possibility, of all tools that we are aware of, to attempt to answer the question of whether a particular behavior is legal for a given program; usual simulators will only give you a single (hopefully) legal execution.

Lengthy discussions about the possible behaviors of small Verilog abound online, many of which might have been completely settled with a formal executable semantics as the one presented here. Given that many of these discussions are had among tool developers, one potential benefit is the development of more correct and consistent tools for Verilog, some of which are expected to provide very strong, formal guarantees about Verilog programs.

Finally, we describe in detail a set of example programs that demonstrate some of the most important, and commonly misunderstood features of Verilog. These include aspects of the language having to do with concurrency, bit-width calculations, and special handling of different kinds of assignments. Several of the examples exposed bugs in mature, widely-used simulators, and were subsequently fixed. This is our third contribution.

8.3 Concepts

This section introduces some of the syntactic and semantic concepts that are crucial for reasoning about Verilog programs and understanding our formal semantics, which is detailed in Sections 8.4 and 8.5. Figures 8.1 and 8.2 present two small examples that we will reference throughout this section for


```

1  module flipflop(clk, in, out);
2      input      clk;
3      input  [15:0]  in;
4      output [15:0]  out;
5      reg    [15:0]  r;
6
7      assign out = r;
8      always @(posedge clk)
9          r <= in;
10
11  endmodule

```

Figure 8.1: D-Type Flip-Flop Example

illustration purposes.

Figure 8.1 shows a Verilog module having the functionality of a D-type flip-flop. A module is a unit of design that allows for code reuse and abstraction. It is similar in spirit to modules from several software programming languages. While the example is simple, it illustrates some important features of Verilog.

Verilog has two main classes of nodes: variables and nets. Variables represent the notion of state, requiring memory of some kind, while nets abstract the idea of wires that carry information from one area of a design to another. The `input` and `output` keywords declare which variables are inputs and outputs to the module. Module inputs and outputs are automatically assumed to be of net type; registered outputs can be explicitly specified by adding the keyword `reg` after `output`. For nodes that are not ports, the keyword `wire` specifies a net type, and the keyword `reg` specifies a variable type. The nodes `clk`, `in`, and `out` in the flip-flop module are all nets, whereas `r` is a variable.

Both nets and variables most often represent scalars or vectors of four-state logic values. The input `clk` is a scalar since, unless explicit indices are given to index into the vector, scalar is assumed. On the other hand, the node `in` is declared with an explicit indexing range, specifying it to be of length sixteen. According to the Verilog standard, variable data types may only be assigned within procedural blocks, such as the one on lines 8–9 of Figure 8.1, while net types can only be assigned in continuous assignments such as the one on line 7.

Lines 8-9 show a procedural block, in this case an *always* block, introduced with the keyword `always`. An always block denotes a constantly running computation, essentially an infinite loop. Note that Verilog also has a related concept called an initial block, which is introduced with the keyword `initial` and runs once starting at the beginning of simulation.

An initial block can be seen in the example in Figure 8.2. The term *procedural blocks* refers collectively to `always` and `initial` blocks, tasks, and functions. Tasks and functions are not covered in our semantics. An `always` block is constructed with a single statement; in the examples presented in Figures 8.1 and 8.2, and commonly in Verilog designs, this statement has the structured form `@(posedge clk) S'`, with `S'` also a statement. The effect of this statement is to delay evaluation of `S'` until its event control, `(posedge clk)`, is triggered. In the case of `(posedge clk)`, the triggering event is a specific kind of change perceived in the value of `clk`, namely a change from any non-1 value to 1, representing a positive edge.

The assignment on line 7 is a continuous net assignment. Perhaps somewhat counterintuitively, this assignment will be the last action of the module on a given positive edge of `clk`. A continuous assignment is triggered for evaluation whenever any value in its right-hand side changes, which, in the case of the current example, is whenever the value of `r` changes. This can be thought of in terms of hardware as attaching a wire to the output of the register `r`.

As the example in Figure 8.1 illustrates, there are two basic types of assignments at the top level, *continuous* assignments, such as the one on line 7, that allow assignment to net types, and *procedural* assignments, such as the one on line 9, that allow assignment to variable types. Procedural assignments can be broken down further into *blocking* and *non-blocking* assignments.

The module in Figure 8.2 shows an `initial` block and two `always` blocks, which look very similar, yet compute very different results. In the `initial` block, `nb1` is initialized to 0. In the block on lines 9–13, blocking assignments are used (hence the variable names `b1,b2`), while lines 15–19 use non-blocking assignments.

To understand what is going on in this example, let us assume a value for `in`. Let `in` be 1, then, considering the first `always` block on lines 9–13, the value of `b1` will be 1, while that of `b2` will be 2. This is because the assignment of `b1` blocks the statements following it until its completion. The non-blocking

```

1 module procedural_assigns(clk, in);
2   input      clk;
3   input [15:0] in;
4   reg  [15:0] b1, b2;
5   reg  [15:0] nb1, nb2;
6
7   initial nb1 = 0;
8
9   always @(posedge clk)
10  begin
11    b1 = in;
12    b2 = b1 + 1;
13  end
14
15  always @(posedge clk)
16  begin
17    nb1 <= in;
18    nb2 <= nb1 + 1;
19  end
20 endmodule

```

Figure 8.2: Assignment Types Example

assignments in the block on line 15–19 do not block the statements following them. Following the block, `nb1` will contain 1, but `nb2` will also contain 1, because the *previous* value of `nb1`, namely 0, is used in the assignment on line 18.

As a language created to model and design circuits, Verilog is inherently concurrent. Capturing this concurrency, and the resulting non-determinism allowed by the standard, is one of the most important tasks of any formal definition of the language. Many Verilog users, however, learn the language primarily through simulators, many of which are single-threaded and deterministic; or, even if not single-threaded, certainly not capable of enumerating all legal behaviors.

To ease understanding and maintain consistency, we adopt several of the terms used in the Verilog standard. First and foremost is that of the *process*. In a Verilog design, a process is anything that can perform computation. According to the standard [36, §11.2]: “Processes are objects that can be

evaluated, that may have state, and that can respond to changes on their inputs to produce outputs”. Going back to our introductory flipflop example from Figure 8.1, the `always` block on lines 8–9 is one process, while the continuous assignment on line 7 is another. The module itself is also a process. Our formal representation of processes is given in Section 8.4.

While processes are very specific, the terminology of *event* encompasses several different concepts. Except where specifically mentioned, we try to make the event terminology of the standard explicit in the definition, to ease understanding for those familiar with the standard. Every update of a net or variable is an *update event*. The evaluation of a process is an *evaluation event*. This is the only type of event that is not explicitly represented in the definition, which instead merges the concepts of process and evaluation event, effectively treating processes as events.

While Verilog is used to synthesize circuit designs, it was originally, and essentially continues to be, designed for simulation. Because of this, Verilog is sensitive to simulator time. In fact, in addition to the ability to delay statements until a particular condition holds, such as on line 8 of Figure 8.1, it is also possible to delay statements some number of simulator cycles. The syntax for this consists of preceding a statement, say `S`, with, say, `#5`, which means that `S` will be delayed 5 time units.

The most important concept regarding the Verilog semantics is the *stratified event queue* [36, §11.3]. Events are divided into five prioritized categories that determine when they are scheduled for execution with respect to simulation time: *active events*, *inactive events*, *non-blocking assign update events*, *monitor events*, and *future events*. We further add the category of *listening events*, which does not exist in the standard but help clarify the execution of Verilog designs.

Active events are all events that are currently running, *i.e.*, they are not waiting for any specific trigger, and they have not been delayed. *Inactive events* are curious, in that as far as we know they only occur when a statement has been delayed by exactly 0 time units, for example, through a statement such as `(r = #0 1;)`. *Non-blocking assign update events* are generated by executing a non-blocking assignment. *Monitor events* are related the Verilog *monitor* statement, which is essentially a print statement that occurs at the end of every simulator cycle in which its arguments *change*. *Future events* are processes that have been delayed by some non-zero amount, which must still

```

1  module value_size;
2     reg [ 3:0] r1, r2;
3     reg [15:0] out;
4
5     initial
6     begin
7         r1 = 15;
8         r2 = 15;
9         out = r1 + r2;
10    end
11 endmodule

```

Figure 8.3: Value Sizing Example

eventually execute. *Listening events* are those events that are waiting for a particular trigger to occur; they will be promoted to active events/processes as soon as that trigger occurs.

Each type of event, as listed, is promoted to an active event or process only when there are no events before it in the list, except for listening events, which may be promoted as soon as the trigger that they are listening for occurs. For example, inactive events are all, *at the same time*, promoted to active events when there are no more active events or processes to be executed. Similarly, non-blocking assign update events are all simultaneously promoted to active events when there are no more active or inactive events in the given simulation cycle. When all events, except for listening events and future events, have been exhausted, time is advanced to that of the earliest future event. If there are no pending future events the program completes execution.

Verilog has interesting rules for the size of operands. Figure 8.3 shows a simple, but by no means exhaustive, example of this. Despite the fact that both `r1` and `r2` are only four bits wide, the variable `out` is still assigned the value 30 at line 9. For the purposes of the addition, `r1` and `r2` are treated as sixteen bit quantities, because `out` is a sixteen bit quantity. There are many different rules for the sizing of values; the above example only covers one of them (sizing to the left hand side of an expression). All of the rules are covered in our definition. Due to the fact that the standard specifies these rules very clearly, we refer the interested reader to the standard or the full

specification of our definition available at [72].

8.4 Semantics: Configuration

This section, together with the one following, provide an annotated account of our formal semantics. We follow a common convention of rewriting logic semantics: the syntax of terms, called *configurations*, is first defined by means of sorts and operator declarations to represent the entire state of the program, and subsequently semantic equations and rules are added that act on configurations to advance the state of the computation. This section defines configurations, using the program in Figure 8.1 as an example, and Section 8.5 defines the essential equations and rules that act on configurations. The specification is available in its entirety [72].

Configurations are represented as terms of sort `Configuration`, defined as sets of *configuration items*. The most important aspect of a configuration is that it represents the state of Verilog’s event queue. Each stratum of the queue is associated with its own configuration item.

```
op updateEvents : Set{Event} -> ConfigurationItem .
op incativeEvents : Set{Event} -> ConfigurationItem .
op nonBlockingAssignUpdateEvents
    : List{Event} -> ConfigurationItem .
op monitorEvents : Set{Event} -> ConfigurationItem .
op futureEvents : List{Event} -> ConfigurationItem .
```

In addition to the above strata, which are exactly the ones dictated by the standard [36, §11.3], we add a new stratum that holds events sensitive to updates to nodes.

```
op listeningEvents : Set{Event} -> ConfigurationItem .
```

There are various kinds of events that are formalized, including update events, events that are sensitive to update events and execute only once, and events that are sensitive to update events and execute continuously. Terms of sort `TriggerSet` below are just sets of node names.

```

op          updateEvent : Exp BitVector -> Event .
op          listeningEvent : TriggerSet      Computation
                    -> Event .
op continuousListeningEvent : TriggerSet Name Computation
                    -> Event .

```

The second argument to `continuousListeningEvent` is a net driven by a continuous assignment. Continuous listening events are, within our semantics, always associated with continuous assignments, and need to be handled specially; the standard dictates that they cannot generate exactly the same kind of update event as procedural blocking assignments.

In addition, the update events that are held using the `updateEvents` operator will include just pending updates to variables, as opposed to both pending updates and active processes. Instead, we introduce one additional stratum to hold active processes.

```

op activeProcesses : Set{Process} -> ConfigurationItem .

```

Processes are terms created from something we call “computations”, which are used to stage the execution of different classes of constructs, such as expressions, statements, and top-level process designators such as `initial` and `always` blocks. Procedural and continuous assignments get separate process constructors.

```

op          k :      Computation -> Process .
op continuousk : Name Computation -> Process .

```

And, the operators holding the different kinds of computations are given as follows.

```

op top : Top      -> Computation .
op exp : Nat+ Exp -> Computation .
op stmt :      Statement -> Computation .
op stmt : Name Statement -> Computation .

```

As an example of a partial configuration, just giving the state of the event queue, the following term specifies the initial state of the event queue for the flipflop module presented in Figure 8.1.

```

activeProcesses(k(top(always @(posedge clk) r <= in;)))
    updateEvents(empty)
    inactiveEvents(empty)
nonBlockingAssignUpdateEvents(empty)
    monitorEvents(empty)
    futureEvents(empty)
    listeningEvents(
        continuousListeningEvent(r, out, exp(16, r)))

```

The current value of all nodes in the circuit, called the *environment* in our semantics, as well as the current simulation time, are the other two components that are part of a configuration.

```

op env : Env -> ConfigurationItem .
op time : Nat -> ConfigurationItem .

```

The operator `env` is used to hold the environment of the system. It contains a mapping from variable or net names to bitvectors. In the initial state of the flipflop module, the environment is given by the following term.

```

env( clk |-> [0 # 1]
    , in |-> [0 # 16]
    , out |-> [0 # 16]
    , r |-> [0 # 16])

```

A bitvector is a pair consisting of a value and a bitwidth, separated with a `#`; *e.g.*, `out` is a 16-bit node assigned value 0 above. The operator `time` is used to hold the current simulator time. It always starts at 0.

To summarize, the term given below specifies the initial configuration for the flipflop module presented in Figure 8.1. Note that we have omitted some additional configuration items that are useful in practice, but not usually considered part of the Verilog simulation state. For example, we omit a configuration item to hold the output buffer.


```

activeProcesses(k(top(always @(posedge clk) r <= in;)))
    updateEvents(empty)
    inactiveEvents(empty)
nonBlockingAssignUpdateEvents(empty)
    monitorEvents(empty)
    futureEvents(empty)
    listeningEvents(
        continuousListeningEvent(r, out, exp(16, r))
time(0)
env( clk |-> [0 # 1]
    , in  |-> [0 # 16]
    , out |-> [0 # 16]
    , r   |-> [0 # 16]
    )

```

8.5 Semantics: Equations and Rules

The semantics for initial blocks is very simple. The statements of an initial block must run exactly once. The equation below simply strips off the keyword `initial`, forcing the statements represented by the variable `S` to evaluate. Note that the operator `k` is not mentioned in the equation; in fact, no subterms are mentioned except those we explicitly require. In addition, throughout the remainder of the chapter, all Maude-level variables will be assumed to have the sort of argument of the operator in which it appears in a term, unless explicitly noted otherwise.

$$\text{top(initial } S) = \text{stmt}(S)$$

The semantics of always blocks is very similar, except that the statements of the body of the block must be repeated indefinitely, thus the equation forces the statements `S` to be run, but also schedules another copy of the always block to run after `S` completes. In this case the equation must match the `k` operator to keep the always block from infinitely unrolling rather than executing the statements of the body before unrolling another step.

$$k(\text{top(always } S)) = k(\text{stmt}(S) \rightarrow \text{top(always } S))$$

Procedural assignments generate update events, which go into the stratified event queue. The update events themselves are responsible for actually

updating the environment of the system and waking up any listening processes. The semantics of blocking assignments is such that trailing statements are barred from being executed until after the generated update event completes. The first step is to calculate the value of the right-hand side of the assignment. In the equation displayed next, the right-hand side of the assignment, designated with the variable `Ex`, is pulled out and placed at the top of the computation stack; the size of the expression is also calculated.

```

eq activeProcesses(k(stmt(QEx = Ex ;) -> K) PS) env(Env)
  = activeProcesses(k(exp(expSize(QEx, Env, Ex), Ex)
    -> blockingAssign(QEx) -> K) PS) env(Env) .

```

The rules for evaluating expressions is straightforward and are omitted (see [72] for details). Once the right-hand side is fully evaluated to a value, that is, to something of sort `BitVector`, the actual assignment takes place and sensitive listening events are added to the set of active processes for later evaluation.

```

eq activeProcesses(k(BV -> blockingAssign(X) -> K) PS)
  updateEvents(ES)
  = activeProcesses(PS)
    updateEvents(updateEventList(updateEvent(X,BV), K) ES)

```

In the equation above, the variable `PS` denotes the other processes, `ES` denotes the current set of events, `X` denotes the node name being assigned to, and `K` denotes the rest of the computation. The important thing to note is that the term represented by `K` gets placed in the update event list that is added to `updateEvents`. It contains all remaining statements in the given procedural block. As we will see below, this computation will be run as an active process once the update event gets reflected in the environment; for the time being, however, it is removed from the active processes. The term rooted at `updateEventList` allows us to group any number of update events that must be executed in order. This is useful both for assignments with concatenations on the left-hand side (see [72]), and for scheduling non-blocking assign update events; here, with a blocking assignments, the full power of having a list is not needed.

While very similar in form, we use a rule to define non-blocking assignments. The reason for this is that all of the non-blocking assignments added to the set

```

module netassign;
  wire w;
  reg r;

  assign w = r;

  initial
  begin
    r = 0;
    r = 1;
  end
endmodule

```

Figure 8.4: Net Assignment Example

contained within the term rooted at the `nonBlockingAssignUpdateEvents` symbol are eventually scheduled to execute in one `updateEventList`. This is to facilitate the standard’s mandate that non-blocking assignments in one procedural block complete in order. If an equation were used, non-blocking assignments in *different* procedural blocks would only be allowed to interleave in one order. With a rule, we ensure that non-blocking assignments may be ordered non-deterministically, while still keeping the ordering within one block.

```

rl activeProcesses(k(BV -> nonBlockingAssign(X) -> K) PS)
  nonBlockingAssignUpdateEvents(EL)
=> activeProcesses(k(K) PS)
  nonBlockingAssignUpdateEvents(EL ; updateEvent(X,BV))

```

The variable `EL` above denotes a list of events. A list differs from a set in that the elements are ordered. Note that the term `K` is allowed to continue as `k(K)` appears in the `activeProcesses` on the right hand side of the rule. This is exactly the desired semantics of a non-blocking assignment: the rest of the block is allowed to complete before the update event from the assignment is allowed to make any change to the environment.

There are two equations for governing the semantics of continuous assignment. Note that only one outstanding update event exists for a given driver at any one time. This is the reason for having “continuous” versions of many

constructs. We do not address the case of a net having multiple drivers.

```

eq activeProcesses(      continuousk(X, BV1) PS)
      updateEvents(continuousUpdateEvent(X, BV2) ES)
  = activeProcesses(      PS)
      updateEvents(continuousUpdateEvent(X, BV1) ES) .

eq activeProcesses(continuousk(X, BV) PS)
      updateEvents(ES)
  = activeProcesses(      PS)
      updateEvents(continuousUpdateEvent(X, BV) ES)
      [otherwise] .

```

The variables `BV1` and `BV` in the first and second equations, respectively, are the results of assignment computations. By the time a bitvector becomes the sole remaining argument, the computation has been completely evaluated. The first equation will replace any pending update event to the same net with an update containing the current value of the assignment right-hand side computation. This case is handled with the first equation. Gordon [31] refers to this idea as *cancelling*.

The second equation is an “otherwise” equation [15, §4.5.4] that is only applied if the first equation does not match because there is not already a pending `continuousUpdateEvent` to the node represented by `X` in the equation.

Net and variable lookup and updating is performed through rules in rewriting logic. The reason for this is that the Verilog standard states that it is legal for a simulator to execute any outstanding active update event. The value of a given net or variable, when referenced, is simply found in the environment. Note that we must mention the `activeProcesses` term because the environment exists at the top level of the configuration. The rules for other kinds of operands, specifically bit and part selections, are similar (see [72]). In addition, we have simplified the rule slightly here, removing the part that handles bit-width calculations. This was done for readability.

```

rl activeProcesses(k(exp(N, X) -> K) PS) env(X |-> BV, M)
  => activeProcesses(      BV -> K) PS) env(X |-> BV, M) .

```

Net and variable updating occurs when an update event executes. Update events are generated by the assignments as explained above. Update events

must modify the environment, wake up any process that is currently waiting as a listening event, and alert any monitor event that the value has changed.

```

rl updateEvents(updateEventList(updateEvent(X, BV1) ; EL, K) ES1)
  env(X |-> BV2, ENV)
  listeningEvents(ES2)
=> updateEvents(updateEventList(EL, K) ES1)
  env(X |-> BV1, ENV)
  listeningEvents(sense(X, BV2, BV1, ES2))

```

Here the various ES's represent sets of events. The term rooted at `updateEventList` groups several update events as a list. This allows us to ensure that non-blocking assignments occur in program text order within a procedural block, as well as allowing for an easy and clear representation of concatenation-form assignments (see [72]). The operator `sense` is responsible for waking up the proper listening events and deciding if any monitor event need execute in the current simulator cycle. It uses the previous and current value of the variable or net that is updated to make its determinations.

The last equation here schedules the term bound to `K` from the update event list to execute when all update events in the update event list have been exhausted. For non-blocking assignments, the term bound to `K` will be `empty`, meaning that nothing is scheduled.

```

eq activeProcesses(PS)
  updateEvents(updateEventList(empty, K) ES)
= activeProcesses(
  k(K) PS)
  updateEvents(ES) .

```

The semantics for delays and triggers is fairly straightforward. Delays simply move the current active process to the future event set with a simulator time equal to the current time added to the time of the delay, if the delay is non-zero. If the delay is zero, the rest of the active processes are moved to the set of inactive events set. Triggers add the rest of the current active process to the set of listening events. The equations for each of these follow.

```

eq time(N)
  futureEvents(EL)
  activeProcesses(      k(stmt(# NzN S) -> K)   PS)
= futureEvents(futureEvent(N + NzN, stmt(S) -> K) ; EL)
  time(N)
  activeProcesses(PS) .

eq inactiveEvents(ES)
  activeProcesses(      k(stmt(# 0 S) -> K)   PS)
= inactiveEvents(inactiveEvent(stmt(S) -> K) ES)
  activeProcesses(PS) .

```

Here, in the equation for non-zero delays, `NzN` is a non-zero natural number, while `N` is any natural number. `N` is the current time, while `NzN` is the delay. As expected, the rest of the current process, `K`, is added to the future event set, as well as the delayed statement `S`. The first argument to the `futureEvent` operator is the simulator cycle in which the event should be scheduled to run as an active process.

For wait statements, the process associated with the wait is added to the listening events set. Here, `SL` is the sensitivity list; it is maintained as the first argument to `listeningEvent` so that the equationally defined function `sense` can decide which listening events must be scheduled when an update event executes.

```

eq listeningEvents(EL)
  activeProcesses(      k(stmt(@(SL)      S) -> K)   PS)
= listeningEvents(listeningEvent(SL, stmt(S) -> K) ; EL)
  activeProcesses(PS) .

```

Lastly, we present the rules for scheduling the main simulator loop. The general idea is to continue with the next set of events in the list when all the previous sets are `empty`.

Active processes and update events are allowed to run at any time, and do so via the equations and rules detailed above. The first set of events activated when active processes and update events are exhausted is the set of inactive events. The operator `activate` schedules each individual inactive event as an active process by wrapping the associated computation with the `k` operator. In the equation, the variable `NES` denotes specifically a *non-empty* set of events.

```
rl activeProcesses(empty)
  updateEvents(empty)
  inactiveEvents(NES)
=> activeProcesses(activate(NES))
  updateEvents(empty)
  inactiveEvents(empty) .
```

When there are no active processes, update events, or inactive events, the non-blocking assign update events are activated simultaneously by moving them to the update event set.

```
rl activeProcesses(empty)
  updateEvents(empty)
  inactiveEvents(empty)
  nonBlockingAssignUpdate(NEL)
=> activeProcesses(empty)
  updateEvents(updateEventList(NEL, empty))
  inactiveEvents(empty)
  nonBlockingAssignUpdate(empty) .
```

The variable NEL denotes a non-empty list of events. The events in the non-blocking assign update event set are added to one update event list. The continuation argument is empty, as there is nothing to continue after a non-blocking assignment changes the state of the program, as mentioned above

A similar rule exists for waking up monitor events. Note that the monitor event is primed for its next execution by being wrapped with a special operator called `futureMonitorEvents`.

```

rl activeProcesses(empty)
    updateEvents(empty)
    inactiveEvents(empty)
    nonBlockingAssignUpdateEvents(empty)
        monitorEvents(monitorEvent(TrS, B, Format, ExL) ES1)
        futureMonitorEvents(ES2)
=> activeProcesses(if B
                    then k(display(true, Format, ExL, nilEL))
                    else emptyProcesses fi)
    updateEvents(empty)
    inactiveEvents(empty)
    nonBlockingAssignUpdateEvents(empty)
        monitorEvents(ES1)
        futureMonitorEvents(ES2)
    monitorEvent(TrS, false, Format, ExL)) .

```

Finally, if no monitor events are present, the future events set to execute earliest are made active. The equationally defined operator `awaken` queues up all other events that were delayed until time associated with the variable `N2`, and leaves any events scheduled later as future events.

```

rl activeProcesses(empty)
    updateEvents(empty)
    inactiveEvents(empty)
    nonBlockingAssignUpdateEvents(empty)
        monitorEvents(empty)
        futureMonitorEvents(ES)
        futureEvents(futureEvent(N2, K) ; EL)
        time(N1)
=> activeProcesses(k(K))
    updateEvents(empty)
    inactiveEvents(empty)
    nonBlockingAssignUpdateEvents(empty)
        monitorEvents(ES)
        futureEvents(awaken(N2, EL))
        time(N2) .

```

8.6 Examples

Races. The first example is taken directly from the IEEE Standard [36], and demonstrates the essential source of non-determinism in Verilog; it is shown in Figure 8.5. After the assignment of `q` to 0, a race condition is created


```

module m;

    wire p;
    reg q;

    assign p = q;                                /* Icarus Verilog 0.9.2
    initial                                       0
    begin                                         */
        q = 1;                                    /* VCS D-2009.12
        #1;                                       0
        q = 0;                                    */
        $display(p);
    end
endmodule

```

Figure 8.5: Race Condition Example from the IEEE Standard ([36, §11.5]).

between the continuous assignment, which needs to update the value of `p`, and the display of `p`. It is correct for a simulator to output either 0, if the display is executed first, or 1, if the continuous assignment is executed first; both `iverilog` and `vcs` happen to output 0.

The second example is shown in Figure 8.6 and, like the first example, comes from the IEEE Standard. The purpose of the example is to expose the apparent vagaries of bit-width determinations, and the effect of this determination on the evaluation of expressions.

In the example, the question that arises is about the correct bit-width of the expression `(a + b)`. The IEEE Standard dictates that for addition, the expression is calculated using a bit-width equal to the maximum of the bit-widths of its operands; the same is true for subtraction, but *not for shift-right*, where the operands are self-determined.

Therefore, the first display must print 0 because the addition is calculated using 4-bits and the overflow is lost. The second display must print 8 because the bit-width of unsized integer literals while implementation-dependent, is required to be at least 32 bits [36, §4.8], and therefore the overflow is not lost because the entire expression `(a + b - 0)` is calculated using 32 bits.

Indeed, the determination of bit-widths is quite complex, as even the declared bit-width of an identifier may involve compound expressions whose bit-widths must be determined. Consider the program shown in Figure 8.7, which declares identifiers `x` and `y` using complex range expressions; that is,

```

module m;

    reg [3:0] a,b;
    reg [3:0] x;

    initial
    begin
        a = 4'b1111;
        b = 4'b0001;
        x = (a + b) >> 1;
        $display(x);
        x = (a + b - 0) >> 1;
        $display(x);
    end
endmodule

```

Figure 8.6: Expression Bit-Width Sizing Example ([36, §5.4.2]).

```

module m;

    reg [4'b1111 + 4'b0001 >> 1:0] x;
    reg [4'b1111 + 1 >> 1:0] y;

    initial
    begin
        x = -1;
        y = -1;
        $display("x = %b", x);
        $display("y = %b", y);
    end
endmodule

```

Figure 8.7: Bit-Width Calculation Bug.

```

module m;

    reg x;

    // spot 1
    initial
        x = 0;
    initial
        x = 1;
    begin
        x = 1;
        #1;
        $display("x = %d", x);
    end

    // spot 2

endmodule

```

Figure 8.8:

where the most and least-significant bits are compound expressions and not simple unsized literals. The displays will print a sequence of 1s for each identifier, the length of which indicates the identifier's bit-width.

As the example from Figure 8.6 demonstrated, the bit-width of the expressions $(4'b1111 + 4'b0001)$ and $(4'b1111 + 1)$ are different, with the expression $(4'b1111 + 4'b0001)$ discarding the overflow, in particular. Therefore, the declarations in the program should be interpreted as being equivalent to

<pre> reg [0:0] x; reg [8:0] y; </pre>

meaning that x should be 1-bit and y should be 9-bits. As a result, this example exposes a bug in `iverilog`, which mistakenly determines that x should be 9 bits wide.

One possible way of exposing nondeterminism using a traditional Verilog simulator is by shuffling processes around to different locations in the source code; for example, consider the programs presented in Figures 8.8 and 8.9. According to the semantics of Verilog, the two programs are equivalent, and may legally display either 0 or 1.

However, simply by moving one initial construct below the other in source

```

module m;

    reg [3:0] x;

    // spot 1

    initial
    begin
        x = 1;
        #1;
        $display("x = %d", x);
    end

    // spot 2
    initial
        x = 0;

endmodule

```

Figure 8.9: Nondeterminism in Verilog

order, it is possible to coerce `iverilog` and `vcs` to display different results. However, attempting to glean the entirety of a Verilog program's meaning this way, essentially by tricking the compiler, leaves much to be desired. A fast simulator is an invaluable tool, but this example demonstrates the inadequacy of such tools when answers to fundamental questions are needed.

Consider what occurs starting at the point during simulation when `x` is assigned `4'b1010` and simulation time is 2. If we were to evaluate the event condition `@(x == 4'b1010)` at this point, the event would be triggered. However, that does not necessarily happen, we simply put the process with the event-control back into the queue of runnable processes. If the simulator then chooses to execute the assignment of `x` to `1'b1111` before executing the process with the event control, then when the process is subsequently run, it will see that the event it is waiting on has not yet happened and be de-scheduled, waiting for other changes to `x`.

The other interesting aspect of the program is the continuous assignment. When the assignment of `x` to `4'b1010` occurs, an update is scheduled for `y` at 1 time unit in the future, however, when the assignment of `x` to `4'b1111` occurs subsequently, the earlier scheduled update is *cancelled*. This is noteworthy because it represents a special case of the manipulation of the event queue: it

```

module m;

    reg [15:0] x;

    always @(x[0])
        x = x+1;          /* Icarus Verilog 0.9.2
                          Does Not Exit
    initial             */
    begin               /* VCS D-2009.12
        x = 0;          x =      3
        x = 2;          */
        #1;
        $display("x = %d", x);
    end

endmodule

```

Figure 8.10:

is one of very few occasions where a pending event is removed.

Therefore, while at time 2 it is legal for a simulator to trigger the first `always` construct, the one displaying `ding!`, it is not legal for a simulator to trigger the other `always` construct displaying `ding! ding!` at time 3. This is the behavior seen with `iverilog`; of course, it is also legal for the `always` construct to not be triggered at time 2, which is the behavior had through running `vcs`; therefore, both `iverilog` and `vcs` produce correct, although different, results.

The final example, presented as Figure 8.10, demonstrates a common misconception about Verilog, namely, that an `always` construct such as

```

always @(x[0])
    x = x+1;

```

is continually sensing changes to `x`. In fact, sensitivity to changes in `x` only occurs after the assignment to `x` on the following line, when control reaches the `@(x[0])` again. Therefore, it is not correct for a simulator to loop infinitely in the above example.

The exact nature of the error exhibited by `iverilog` is somewhat subtle. It effectively makes the following program transformation.

```

assign x1 = x[0];
always @(x1)
    x = x+1;

```

The transformed program can loop infinitely, and so in some sense the problem exhibited by `iverilog` is that this transformation is not semantics-preserving.

CHAPTER 9

SEMANTICS: PRODUCTION RULE SETS

9.1 Introduction

Asynchronous digital circuits have been employed to design low-power, high-performance microprocessors, *e.g.* [69], as well as in emerging applications such as systems-on-chip (SOCs), *e.g.* [68], soft-error-tolerant systems, *e.g.* [42], and nano-electronics, *e.g.* [70]. The critical property that makes asynchronous circuits advantageous in these applications is enormous immunity to both intrinsic and extrinsic timing variation. 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 [65]. 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.

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*. 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, it provides a set of mathematical tools for proving properties about asynchronous circuits.

The second issue that this paper addresses is the problem 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 checks for deadlock and hazard freedom, can be made completely automatic. Executability is had in this paper through a semantic formalization in *rewriting logic* [74, 77] which, through the rewriting logic engine Maude [15], offers various automated and semi-automated analysis possibilities.

The remainder of the paper is organized as follows. Section 9.2 defines \mathcal{M}_{PRS} , the “mathematical” formal semantics of production rule sets, which improves upon an earlier effort [56] through the use of a more familiar operational style. Section 9.3 defines \mathcal{R}_{PRS} , an executable semantics of production rule sets in rewriting logic, improving upon [48] through its extremely close, almost identical, matching with the mathematical semantics. Section 9.4 establishes the relative correctness of \mathcal{M}_{PRS} and \mathcal{R}_{PRS} . Specifically, a strong bisimulation between transition systems induced by the semantics is proved. As corollaries to strong bisimulation, we get relative correctness with respect to deadlocks and hazards as well. Section 9.5 demonstrates how to use the Maude tool to check deadlock and hazard freedom automatically, and applies this analysis to several small circuits. A pair of optimizations are developed, and their efficacy examined. Section 9.6 looks briefly at two timing assumptions other than delay-insensitivity, namely speed-independence and quasi-delay-insensitivity.

9.2 Mathematical Semantics: \mathcal{M}_{PRS}

This section revisits our work in [56], providing a revised “mathematical” semantics for production rule sets for the delay-insensitive case. We refer generally to the formalization given in this section as \mathcal{M}_{PRS} . Compared to [56], \mathcal{M}_{PRS} 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 “mathematical” 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 semantics which follows in Section 9.3.

The mathematical 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 [55]. A clear formal semantics is also crucial to facilitating communication between, and a common understanding amongst, practitioners, as well as for helping newcomers to the field understand essential concepts.

Section 9.2.1 deals with the syntax of production rule sets. Section 9.2.2 introduces the semantics informally and works through a small example. For simplicity, hazards are omitted from the discussion in that section. Section 9.2.3 formalizes the semantics in detail, including hazards, and Section 9.2.4 returns to the example circuit and works through an execution that generates a hazard. Section 9.2.5 contains a discussion about production rule sets in the context of two somewhat similar formalisms for concurrency, guarded commands and communicating sequential processes.

9.2.1 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 9.2.1 (Syntax of Production Rule Sets). Let Y denote a denumerable set of variables used to specify node names. A *production rule* is a triple

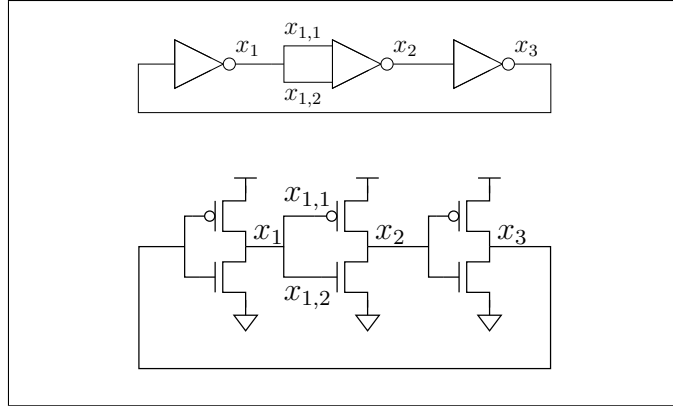


Figure 9.1: Gate-level and CMOS-level Specification of a 3-Inverter Ring Oscillator.

(g, x, d) , with g , the *guard*, being a boolean expression involving variables from Y , $x \in Y$ is the *transition variable*, and $d \in \{\uparrow, \downarrow\}$ 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

$$g \mapsto x d$$

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

$$\begin{aligned} g_1 &\mapsto x_1 d_1 \\ &\vdots \\ g_m &\mapsto x_m d_m \end{aligned}$$

9.2.2 Example

The purpose of this section is to give, by way of an example, an informal introduction to the dynamic behavior of production rule sets; that is, their *semantics*. The example circuit we consider is shown in Figure 9.1. It is known as a 3-inverter ring oscillator.

Digital ring oscillators are typically amongst 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.

Figure 9.1 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. It should be noted that in modern CMOS technologies (65nm 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 production rule set corresponding to the 3-inverter ring oscillator depicted in Figure 9.1 is

$$\begin{array}{cccccc} \neg x_3 \mapsto x_1 \uparrow & x_1 \mapsto x_{1,1} \uparrow & x_1 \mapsto x_{1,2} \uparrow & \neg x_{1,1} \mapsto x_2 \uparrow & \neg x_2 \mapsto x_3 \uparrow & \\ x_3 \mapsto x_1 \downarrow & \neg x_1 \mapsto x_{1,1} \downarrow & \neg x_1 \mapsto x_{1,2} \downarrow & x_{1,2} \mapsto x_2 \downarrow & x_2 \mapsto x_3 \downarrow & \end{array}$$

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

$$\begin{array}{l} x_1, x_{1,1}, x_{1,2}, x_3 \mapsto 0 \\ x_2 \mapsto 1 \end{array}$$

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

Informally, the σ' 's reachable from σ are had 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

$$\begin{array}{ccc} \neg x_3 \mapsto x_1 \uparrow & & \neg x_{1,1} \mapsto x_2 \uparrow \\ \neg x_1 \mapsto x_{1,1} \downarrow & \neg x_1 \mapsto x_{1,2} \downarrow & x_2 \mapsto x_3 \downarrow \end{array}$$

However, note that while all of these production rules have guards that evaluate to true in the current state, only the rule $\neg x_3 \mapsto x_1 \uparrow$ 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 $\neg x_3 \mapsto x_1 \uparrow$ is said to be enabled, whereas, for example, the rule $x_2 \mapsto x_3 \downarrow$ 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 σ 's reachable from σ ; namely $\sigma' = \sigma$ and the σ' defined by

$$\begin{array}{l} x_1, x_2 \mapsto 1 \\ x_{1,1}, x_{1,2}, x_3 \mapsto 0 \end{array}$$

From the above σ' , where x_1 has switched, there are four σ'' '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.

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 9.2.4 to expand the example execution steps so that hazards are accounted for according to the semantics.

9.2.3 Mathematical Semantics

At a high level, our goal is to define a binary relation between program states, denoted \longrightarrow_P , that corresponds to one step of concurrent execution, relative to a production rule set P . $s \longrightarrow_P s'$ 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 \longrightarrow_P -chains

$$s_1 \longrightarrow_P s_2 \longrightarrow_P s_3 \longrightarrow_P \cdots$$

subject to a form of fairness, described later.

Fix a production rule set P . $Variable_P \subseteq Y$ denotes the set of all variables occurring in P . A *state* (with respect to P) is a pair

$$(\sigma : Variable_P \longrightarrow Level, H \subseteq Variable_P)$$

where $Level \stackrel{def}{=} \{0, \mathbf{X}, 1\}$. The set of all states (with respect to P) is denoted $State_P$.

The σ component of a state (σ, H) serves the familiar purpose of specifying values for all nodes in the circuit, with the \mathbf{X} value meaning that a *hazard* has been expressed at that node. This direct expression of hazards was first introduced in [56]. Hazards come in two basic varieties, *interference* and *instability*. The purpose of the set H is to record the origination of potential hazards, which is only expressed when a node ultimately takes on the value \mathbf{X} ; this expression can happen an arbitrary number of computation steps in the future.

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 $\sigma : Variable_P \longrightarrow Level$, we define a set

$$Interference_{P,\sigma} \subseteq Variable_P$$

such that $y \in Interference_{P,\sigma}$ iff there exists $g_1 \mapsto y \uparrow, g_2 \mapsto y \downarrow \in P$ such that $\sigma(g_1) = \sigma(g_2) = 1$.

The evaluation of a boolean expression in the three-valued mapping, such as $\sigma(g_1)$ above, extends the usual meaning of \neg, \wedge, \vee on $\{0, 1\}$ according to the following equivalences:

$$\neg \mathbf{X} = \mathbf{X} \quad \mathbf{X} \wedge 0 = 0 \quad \mathbf{X} \wedge 1 = \mathbf{X} \quad \mathbf{X} \vee 1 = 1 \quad \mathbf{X} \vee 0 = \mathbf{X} \quad \mathbf{X} \wedge \mathbf{X} = \mathbf{X} \quad \mathbf{X} \vee \mathbf{X} = \mathbf{X}.$$

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, $(\sigma, H) \longrightarrow_P (\sigma', H')$, and is captured by

a set

$$Instability_{P,\sigma,\sigma'} \subseteq Variable_P.$$

To define this set, we first need an auxiliary notion, called *enablement*. Given a valuation σ , $Enabled_{P,\sigma} \subseteq Variable_P$ is defined so that $y \in Enabled_{P,\sigma}$ 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 $\sigma(g) = 1$.

Given enablement, $y \in Instability_{P,\sigma,\sigma'}$ iff $y \in Enabled_{P,\sigma}$, $y \notin Enabled_{P,\sigma'}$, 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.

$$Hazard_{P,\sigma,\sigma'} \subseteq Variable_P$$

is defined such that $y \in Hazard_{P,\sigma,\sigma'}$ iff any of the following conditions are met:

- $y \in Interference_{P,\sigma'}$;
- $y \in Instability_{P,\sigma,\sigma'}$;
- there exists a $g \mapsto yd \in P$ such that $\sigma'(g) = \mathbf{X}$.

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

$$Action_P \stackrel{def}{=} \{\mathbf{skip}\} \cup \{x := v \mid x \in Variable_P, v \in Level\}$$

Given a set of actions $A \subseteq Action_P$ and a variable $x \in Variable_P$, we denote the subset of *x-actions* of A as

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

Definition 9.2.2 (Mathematical Semantics of Production Rule Sets). Let

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

be a production rule set. The evaluation relation

$$\longrightarrow_{\subseteq} (P \times State_P) \times Action_P$$

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

$$\frac{\cdot}{\langle g \mapsto x \uparrow, (\sigma, H) \rangle \longrightarrow x := 1} \sigma(g) = 1 \quad \frac{\cdot}{\langle g \mapsto x \downarrow, (\sigma, H) \rangle \longrightarrow x := 0} \sigma(g) = 1$$

$$\frac{\cdot}{\langle g \mapsto xd, (\sigma, H) \rangle \longrightarrow x := \mathbf{X}} x \in H$$

$$\frac{\cdot}{\langle g \mapsto xd, (\sigma, H) \rangle \longrightarrow \mathbf{skip}}$$

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

$$\frac{\langle r_1, (\sigma, H) \rangle \longrightarrow a_1 \quad \dots \quad \langle r_m, (\sigma, H) \rangle \longrightarrow a_m}{(\sigma, H) \longrightarrow_P (\sigma[a_1, \dots, a_m], H[\sigma, a_1, \dots, a_m])}$$

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

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

and the set H of possible hazards is updated to the set $H[\sigma, a_1, \dots, a_m]$ such that for all $y \in Variable_P$, $y \in H[\sigma, a_1, \dots, a_m]$ iff

- $y \in Hazard_{P, \sigma, \sigma[a_1, \dots, a_m]}$, or
- $y \in H$ and $\sigma(y) = \sigma[a_1, \dots, a_m](y)$.

Note the fact that a pair $\langle g \mapsto xd, (\sigma, H) \rangle$ 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 $j \in \mathbb{N}$, $\xi(j) \rightarrow_P \xi(j+1)$ 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 $i > j$, $y \in \text{Enabled}_{P,\sigma_i}$ or $y \in H_i$, but y never switches; that is, $\sigma_i(y) = \sigma_j(y)$. This condition is the aforementioned fairness constraint.

9.2.4 Example, with Hazards

Let us return to the example of Section 9.2.2 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 9.2.2, with the obvious exception that we now account for hazards. So, our initial state $s_0 = (\sigma_0, H_0)$ has σ_0 defined by

$$\begin{aligned} x_1, x_{1,1}, x_{1,2}, x_3 &\mapsto 0 \\ x_2 &\mapsto 1 \end{aligned}$$

and $H_0 = \emptyset$.

Going to $s_1 = (\sigma_1, H_1)$, we let x_1 switch. Therefore, σ_1 is given by

$$\begin{aligned} x_1, x_2 &\mapsto 1 \\ x_{1,1}, x_{1,2}, x_3 &\mapsto 0 \end{aligned}$$

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 σ_3 be such that $x_{1,2}$ switches, but

$x_{1,1}$ does not; these are the only two currently enabled rules. That is, σ_3 is

$$\begin{aligned}x_1, x_{1,2}, x_2 &\mapsto 1 \\x_{1,1}, x_3 &\mapsto 0\end{aligned}$$

The interesting aspect of this state change is that H_3 becomes *non-empty*. It is straightforward to check that $x_2 \in Hazard_{P, \sigma_2, \sigma_3}$, and therefore that $x_2 \in H_3$. Since there are production rules $\neg x_{1,1} \mapsto x_2 \uparrow, x_{1,2} \mapsto x_2 \downarrow \in P$, both with guards that are true in σ_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

$$\begin{aligned}x_1, x_{1,2} &\mapsto 1 \\x_{1,1}, x_3 &\mapsto 0 \\x_2 &\mapsto X\end{aligned}$$

and $H_4 = \{x_3\}$.

9.2.5 Concurrency in Production Rule Sets

Although the language of production rule sets shares certain features with both *guarded commands* [22] and *communicating sequential processes* (CSP) [35], 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 (\uparrow) or pulled down toward logical 0 (\downarrow)

$$\begin{aligned}x &\mapsto z \downarrow \\y &\mapsto w \downarrow \\\neg y &\mapsto w \uparrow\end{aligned}$$

A reasonable candidate translation into the language of guarded commands would be the statement

```

do    $x \rightarrow z := 0$ 
      [ ]  $y \rightarrow w := 0$ 
      [ ]  $\neg y \rightarrow w := 1$ ,
od

```

Similarly, one might reasonably attempt to view the above production rule set as the following CSP parallel command

```

*[  $x \rightarrow z := 0$  ] ||
*[  $y \rightarrow w := 0$  ] ||
*[  $\neg y \rightarrow w := 1$  ]

```

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:

```

 $z = 1, w = 1;$ 
 $z = 1, w = 0;$ 
 $z = 0, w = 1;$ 
 $z = 0, w = 0.$ 

```

In a single step of computation, neither the guarded command statement nor the parallel command can change both z and w to 0. 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.

9.3 Rewriting Logic Semantics: \mathcal{R}_{PRS}

The purpose of this section is to translate the mathematical semantics of Section 9.2 into an *executable* one using rewriting logic [74], a formalism which has been shown [77] to be well suited for exactly this task. The particular notation used is that of the rewriting logic tool Maude [15]. As we will see, the rewriting logic semantics mimics closely the mathematical semantics. The rewriting logic theory described in this section will be referred to as \mathcal{R}_{PRS} .

Using the concrete notation of the Maude tool gives us executability directly. It 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 9.5 (the entire Maude specification is available at [50]).

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 QID module [15, §9.2] provides our variables: strings of characters preceded by a single quote.

```
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 9.2, the corresponding terms using Maude notation are very similar. The production rule $\neg y \mapsto w \uparrow$ becomes the term `[not 'y -> '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

SET module ([15, §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 [15, §8.3].

```

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 SYNTAX module, the production rule set

$$\begin{array}{l}
 x \mapsto z \downarrow \\
 y \mapsto w \downarrow \\
 \neg y \mapsto w \uparrow
 \end{array}$$

becomes a term of sort `ProductionRuleSet`, written in the notation of Maude as

$$\begin{array}{l}
 [\quad 'x \rightarrow 'z \ -] \\
 [\quad 'y \rightarrow 'w \ -] \\
 [not \ 'y \rightarrow 'w \ +]
 \end{array}$$

Continuing from the start of Section 9.2.3, we define an operator which takes a production rule set P as an argument and returns the set of `Variable` terms corresponding to the set $Variable_P$ defined in Section 9.2.3. Recall that $Variable_P$ 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 $g \mapsto xd$).

```

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

```

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 [50] for details).

Unlike the set $Variable_P$, which was specified according to an equationally defined function, the sets $Level$ and $State_P$ will be given entirely new sorts. Recall that $Level = \{0, X, 1\}$ and that for a production rule set P a state is a pair (σ, H) with $\sigma : Variable_P \rightarrow Level$ and $H \subseteq Variable_P$. The σ component is defined using Maude's MAP module [15, §9.13.1].

```

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

```

Note that the parameter P of $State_P$ 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 [15, §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 $State_P$ is harmless.

As an example of what **AUX-SEMANTICS-3** provides, suppose that we have a state (σ, H) for the above production rule set where

$$\begin{aligned}
\sigma(x) &= 0 \\
\sigma(y) &= 0 \\
\sigma(z) &= 1 \\
\sigma(w) &= X
\end{aligned}$$

and $H = \{y, z\}$; the representation of (σ, H) as a term of sort **State** is written in the Maude notation as

$$((\text{'x} \mid\text{-> } 0, \text{'y} \mid\text{-> } 0, \text{'z} \mid\text{-> } 1, \text{'w} \mid\text{-> } X), (\text{'y} , \text{'z}))$$

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.

```

fmod 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

```

We are now in a position to handle the primary definitions having to do with hazards: $Interference_{P,\sigma}$, $Instability_{P,\sigma,\sigma'}$, and $Hazard_{P,\sigma,\sigma'}$; all of which are predicates on $Variable_P$. Consider $Interference_{P,\sigma}$. We declare an equationally defined function that takes two arguments, one a term of sort `ProductionRuleSet` corresponding to P , and the second a term of sort `Map{Variable,Level}` corresponding to σ , and returns a term of sort $2^{Variable}$ corresponding to $Interference_{P,\sigma}$.

Recall how $Interference_{P,\sigma} \subseteq Variable_P$ was defined: for all $y \in Variable_P$, $y \in Interference_{P,\sigma}$ 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.

`InterfPred` determines if a variable satisfies the interference property and `InterfFilter` filters the set $Variable_P$ by applying `InterfPred` to every variable in P one-by-one.

```

fmod AUX-SEMANTICS-5 is pr AUX-SEMANTICS-4 .
  ... --- meta-variable declarations omitted

  op InterfPred :
    Variable ProductionRuleSet Map{Variable,Level} -> Bool .
ceq 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 [otherwise] .

  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 9.2 so that for all $y \in Variable_P$, $y \in Instability_{P,\sigma,\sigma'}$ if and only if $y \in Enabled_{P,\sigma}$, $y \notin Enabled_{P,\sigma'}$, and $\sigma(y) = \sigma(y')$. Recall that $Enabled_{P,\sigma} \subseteq Variable_P$ was defined so that for all $y \in Variable_P$, $y \in Enabled_{P,\sigma}$ 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 [50]).


```

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
/\ [G+ -> Y +] P' := P
/\ Sigma(G+) == 1 .
ceq EnabledPred(Y, P, Sigma) = true
if Sigma(Y) /= 0
/\ [G- -> Y -] P' := P
/\ 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 )
/\ not EnabledPred(Y, P, Sigma')
/\ Sigma(Y) == Sigma'(Y) .
eq InstPred(Y, P, Sigma, Sigma') = false [owise] .

op Instability-{_,_,_} :
    ProductionRuleSet Map{Variable,Level}
    Map{Variable,Level} -> 2^Variable .
eq Instability {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 X values.

```

fmod 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 **skip**. The restriction operator on a set of actions picks those non-skip actions with a particular variable given as the first component.

```

fmod 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

fmod 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

```

Rewrite rules are used to mimic the effect of the four inference rules

$$\frac{\cdot}{\langle g \mapsto x \uparrow, (\sigma, H) \rangle \longrightarrow x := 1} \sigma(g) = 1 \quad \frac{\cdot}{\langle g \mapsto x \downarrow, (\sigma, H) \rangle \longrightarrow x := 0} \sigma(g) = 1$$

$$\frac{\cdot}{\langle g \mapsto xd, (\sigma, H) \rangle \longrightarrow x := X} x \in H$$

$$\frac{\cdot}{\langle g \mapsto xd, (\sigma, H) \rangle \longrightarrow \mathbf{skip}}$$

```

mod AUX-SEMANTICS-10 is pr AUX-SEMANTICS-9 .
  ... --- meta-variable declarations omitted

  op <_,_> : ProductionRule State -> [Action] .

  cr1 < [G -> Y +], (Sigma,H) > => (Y := 1)
    if Sigma(G) == 1 .
  cr1 < [G -> Y -], (Sigma,H) > => (Y := 0)
    if Sigma(G) == 1 .
  cr1 < [G -> Y D], (Sigma,H) > => (Y := X)
    if Y in H .
  rl < [G -> Y D], (Sigma,H) > => skip .
endm

```

Notice that the $\langle _, _ \rangle$ constructor yields a term of *kind* `[Action]` (see [15, §3.5]), but without a proper sort. This will be crucial when we define the rewrite rule corresponding to \longrightarrow_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* `Action`.

The top-level rewrite rule that ultimately gives us \longrightarrow_P relies on rewriting logic equivalents for $\sigma[a_1, \dots, a_m]$ and $H[\sigma, a_1, \dots, a_m]$. We start with $\sigma[a_1, \dots, a_m]$, which was defined in Section 9.2 according to

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

```

fmod AUX-SEMANTICS-11 is pr AUX-SEMANTICS-9 .
... --- meta-variable declarations omitted

op _[_] :
  Map{Variable,Level} 2^Action -> Map{Variable,Level} .
eq empty [A] = empty .
eq (Y |-> V , Sigma) [A] = sigma'(Y, V, A) , (Sigma[A]) .

op sigma' : Variable Level 2^Action -> Entry{Variable,Level} .
ceq sigma'(Y, V, A) = Y |-> 1
  if (Y := 1) == A | Y .
ceq sigma'(Y, V, A) = Y |-> 0
  if (Y := 0) == A | Y .
ceq sigma'(Y, V, A) = Y |-> X
  if (Y := X) == (A | Y) or | (A | Y) | > 1 .
eq sigma'(Y, V, A) = Y |-> V [owise] .
endfm

```

$H[\sigma, a_1, \dots, a_m]$ is similarly straightforward. Following the definition from Section 9.2, $H[\sigma, a_1, \dots, a_m] \subseteq \text{Variable}_P$ such that for all $y \in \text{Variable}_P$, $y \in H[\sigma, a_1, \dots, a_m]$ iff

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

```

fmod AUX-SEMANTICS-12 is pr AUX-SEMANTICS-11 .
... --- meta-variable declarations omitted

op HPred : Variable ProductionRuleSet State 2^Action -> Bool .
ceq HPred(Y, P, (Sigma,H), A) = true
if HazardPred(Y, P, Sigma, Sigma[A]) .
ceq HPred(Y, P, (Sigma,H), A) = true
if Y in H
/\ Sigma(Y) == (Sigma[A])(Y) .
eq HPred(Y, P, (Sigma,H), A) = false [owise] .

op _[_,_]'[_] :
  2^Variable Map{Variable,Level} 2^Action
  ProductionRuleSet -> 2^Variable .
eq H [Sigma, A] {P} = HFilter(Variable-{P}, P, (Sigma,H), A) .

op HFilter :
  2^Variable ProductionRuleSet State
  2^Action -> 2^Variable .
eq HFilter(empty , P, (Sigma,H), A) = empty .
eq HFilter((Y,YS), P, (Sigma,H), A) =
  if HPred(Y, P, (Sigma,H), A) then Y else empty fi ,
  HFilter(YS, P, (Sigma,H), A) .
endfm

```

Finally, we give a conditional rewrite rule that captures the earlier top-level inference rule

$$\frac{\langle r_1, (\sigma, H) \rangle \longrightarrow a_1 \quad \dots \quad \langle r_m, (\sigma, H) \rangle \longrightarrow a_m}{(\sigma, H) \longrightarrow_P (\sigma[a_1, \dots, a_m], H[\sigma, a_1, \dots, a_m])}$$

```

mod SEMANTICS is pr AUX-SEMANTICS-12 .
  pr AUX-SEMANTICS-10 .

  ... --- meta-variable declarations omitted
  vars A : 2^Action .

  sort Configuration .
  op _[_] : State ProductionRuleSet -> Configuration .

  op mkActs : ProductionRuleSet State -> [2^Action] .
  eq mkActs(empty, (Sigma,H)) = empty .
  eq mkActs(R P , (Sigma,H)) =
    < R, (Sigma,H) > , mkActs(P, (Sigma,H)) .

  crl (Sigma,H) {P} => (Sigma[A],H[Sigma,A] {P}) {P}
    if mkActs(P, (Sigma,H)) => A .
endm

```

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 \longrightarrow (from rewriting logic) to define \longrightarrow_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

$$\text{mkActs}(P, (\text{Sigma}, H)) \Rightarrow A$$

serves the purpose of the *multiple* premises of the inference rule

$$\frac{\langle r_1, (\sigma, H) \rangle \longrightarrow a_1 \quad \dots \quad \langle r_m, (\sigma, H) \rangle \longrightarrow a_m}{(\sigma, H) \longrightarrow_P (\sigma[a_1, \dots, a_m], H[\sigma, a_1, \dots, a_m])}$$

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 `mkActs` is only kinded, the variable `A` has *sort* `2^Action`. This ensures that all of the individual production rules are rewritten to actions in the condition, before a computation step is

taken.

9.4 Relative Correctness of \mathcal{M}_{PRS} and \mathcal{R}_{PRS}

This section establishes a *strong bisimulation* between two transition systems: one induced by the mathematical semantics of production rule sets, \mathcal{M}_{PRS} , and the other induced by the executable semantics given in rewriting logic, \mathcal{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; in particular, in the model checking results presented later in Section 9.5.

The strong bisimulation result is obtained as follows. First, we define a function, \mathbf{cast}_P , that maps states in the mathematical semantics to corresponding states in the rewriting logic semantics. Subsequent to this, we make explicit the transition systems associated to both the semantics; and finally we show that the two are strongly bisimilar via \mathbf{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 two semantics.

We introduce a number of mathematical conventions that are used throughout this section. First, we assume that \mathcal{R}_{PRS} is specified as

$$\mathcal{R}_{PRS} = (\Sigma_{PRS}, E_{PRS}, R_{PRS}).$$

Use of the sort name `Configuration` is overloaded to stand also for the set

$$T_{\Sigma_{PRS}/E_{PRS}, \text{Configuration}}$$

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

9.4.1 \mathbf{cast}_P

Fix a production rule set P . Our bisimulation relation is defined by a function \mathbf{cast}_P taking each state $(\sigma, H) \in \text{State}_P$ to a corresponding term in the rewriting logic specification \mathcal{R}_{PRS} ; specifically, $\mathbf{cast}_P((\sigma, H))$ will be a term

of sort `Configuration`. That is, applying the overloading of `Configuration` specified above, `castP` is a function

$$\text{cast}_P : \text{State}_P \longrightarrow \text{Configuration}.$$

`castP` is defined at the top by calling two functions, one that recurses over the structure of an element of `StateP`, yielding a term of sort `State`, and a second recursing over the structure of a production rule set and yielding a term of sort `ProductionRuleSet`; specifically,

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

To simplify the presentation note that we have used “`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. `cast` will also name all similar functions converting other types of objects in the mathematical semantics into terms in the rewriting logic semantics.

The definition of `cast` functions is largely routine. For most constructs in the mathematical 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 sets of things 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 `SET` module [15, §9.12.2] employs *equational attributes* [15, §4.4.1] and associate-commutative rewriting for associativity and commutativity, and for idempotency an explicit equation is used.

$$\text{cast}(\{r_1, \dots, r_m\}) = \text{cast}(r_1) \cdots \text{cast}(r_m)$$

Similarly, valuation functions of the form $\sigma : Variable_P \longrightarrow Level$ can be viewed as sets of pairs, and are cast accordingly into

$$\text{cast}(\{(y_1, v_1), \dots, (y_m, v_m)\}) = \text{cast}(y_1) \mapsto \text{cast}(v_1) , \dots , \text{cast}(y_m) \mapsto \text{cast}(v_m)$$

By way of summarizing, consider the production rule set corresponding to a single nand-gate

$$P = \{x_1 \wedge x_2 \mapsto y \downarrow, \neg x_1 \vee \neg x_2 \mapsto y \uparrow\}$$

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

$\begin{aligned} &('x1 \mapsto 1 , 'x2 \mapsto 1, 'y \mapsto 1, \text{empty}) \\ &\{ [\quad 'x1 \quad \text{and} \quad 'x2 \quad \rightarrow 'y -] \\ &\quad [(\text{not } 'x1) \quad \text{or} \quad (\text{not } 'x2) \rightarrow 'y +] \} \end{aligned}$

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

Lemma 9.4.1. *Let P be a production rule set.*

$$\text{cast}_P : State_P \longrightarrow Configuration$$

is injective.

Proof. Exercise. □

9.4.2 Strong Bisimulation

Having defined cast_P we are now in a position to state our main result establishing the relative correctness of \mathcal{M}_{PRS} and \mathcal{R}_{PRS} . For notational convenience and symmetry, for a given production rule set P , clear from context, we use $\longrightarrow_{\mathcal{M}}$ to denote the relation \longrightarrow_P defined according to the mathematical semantics \mathcal{M}_{PRS} . Similarly, we use $\longrightarrow_{\mathcal{R}}$ to denote the *one step* rewriting relation induced by \mathcal{R}_{PRS} on terms of sort **Configuration**.

Theorem 9.4.1. *Let P be a production rule set. Consider the transition systems*

$$T_{\mathcal{M}} \stackrel{def}{=} (State_P, \longrightarrow_{\mathcal{M}})$$

$$T_{\mathcal{R}} \stackrel{def}{=} (Configuration, \longrightarrow_{\mathcal{R}})$$

$cast_P$, when seen as a relation, is a strong bisimulation between $T_{\mathcal{M}}$ and $T_{\mathcal{R}}$.

The following lemma will be useful in the proof of the above theorem, which is given afterward.

Lemma 9.4.2. *Let P be a production rule set. For all $g \mapsto xd \in P$, $(\sigma, H) \in State_P$, and $a \in Action_P$, we have*

$$\langle g \mapsto xd, (\sigma, H) \rangle \longrightarrow a$$

according to \mathcal{M}_{PRS} , if and only if

$$\mathcal{R}_{PRS} \vdash cast(\langle g \mapsto xd, (\sigma, H) \rangle) \longrightarrow cast(a)$$

Proof. By cases on a :

- (**skip**): It is enough to show that both $\langle g \mapsto xd, (\sigma, H) \rangle \longrightarrow \mathbf{skip}$, with respect to \mathcal{M}_{PRS} , and $cast(\langle g \mapsto xd, (\sigma, H) \rangle) \longrightarrow cast(\mathbf{skip})$, with respect to \mathcal{R}_{PRS} , hold unconditionally.

That $\langle g \mapsto xd, (\sigma, H) \rangle \longrightarrow \mathbf{skip}$ holds unconditionally with respect to \mathcal{M}_{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)

$$\frac{\cdot}{\langle g \mapsto xd, (\sigma, H) \rangle \longrightarrow \mathbf{skip}}$$

Similarly, with respect to \mathcal{R}_{PRS} ,

$$\begin{aligned} \mathcal{R}_{PRS} \vdash cast(\langle g \mapsto xd, (\sigma, H) \rangle) = \\ \langle [cast(g) \rightarrow cast(x) cast(d)], (cast(\sigma), cast(H)) \rangle \\ \longrightarrow \mathbf{skip} = cast(\mathbf{skip}) \end{aligned}$$

due to the rewriting rule

`cr1 < [G -> Y D], (Sigma,H) > => skip .`

- $(y := 1)$: (\Rightarrow) Suppose $\langle g \mapsto xd, (\sigma, H) \rangle \longrightarrow y := 1$ holds with respect to \mathcal{M}_{PRS} ; we show that also $\text{cast}(\langle g \mapsto xd, (\sigma, H) \rangle) \longrightarrow \text{cast}(y := 1)$ with respect to \mathcal{R}_{PRS} .

Clearly, $\langle g \mapsto xd, (\sigma, H) \rangle \longrightarrow y := 1$ may only hold with respect to \mathcal{M}_{PRS} according to the rule

$$\frac{\cdot}{\langle g \mapsto x \uparrow, (\sigma, H) \rangle \longrightarrow x := 1} \sigma(g) = 1$$

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

$$\begin{aligned} & \text{cast}(\langle g \mapsto xd, (\sigma, H) \rangle) \\ &= \langle [\text{cast}(g) \rightarrow \text{cast}(x) \uparrow], (\text{cast}(\sigma), \text{cast}(H)) \rangle \end{aligned}$$

and matches the left-hand side of the rewrite rule

`cr1 < [G -> Y +], (Sigma,H) > => (Y := 1)
if Sigma(G) == 1 .`

As an exercise, we leave that $\sigma(g) = 1$ implies also the condition of the rewriting rule: $\text{cast}(\sigma)(\text{cast}(g)) == 1$. This yields that $\text{cast}(\langle g \mapsto xd, (\sigma, H) \rangle) \longrightarrow \text{cast}(x) := 1 = \text{cast}(y := 1)$, as needed.

(\Leftarrow) Suppose $\mathcal{R}_{PRS} \vdash \text{cast}(\langle g \mapsto xd, (\sigma, H) \rangle) \longrightarrow \text{cast}(y := 1)$.

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

`cr1 < [G -> Y +], (Sigma,H) > => (Y := 1)
if Sigma(G) == 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 $\text{cast}(\sigma)(\text{cast}(g)) == 1$ implies that $\sigma(g) = 1$, which is left as an exercise, the rule

$$\frac{\cdot}{\langle g \mapsto x \uparrow, (\sigma, H) \rangle \longrightarrow x := 1} \sigma(g) = 1$$

applies to $\langle g \mapsto xd, (\sigma, H) \rangle = \langle g \mapsto x \uparrow, (\sigma, H) \rangle$ which ultimately yields the desired result that $\langle g \mapsto xd, (\sigma, H) \rangle \longrightarrow (x := 1) = (y := 1)$.

- The remaining cases are similar.

□

We are now in a position to give a proof of Theorem 9.4.1.

Proof of Theorem 9.4.1. ($T_{\mathcal{R}}$ simulates $T_{\mathcal{M}}$): Let $(\sigma, H), (\sigma', H') \in \text{State}_P$ be such that

$$(\sigma, H) \longrightarrow_{\mathcal{M}} (\sigma', H')$$

According to Definition 9.2.2, there exist actions a_1, \dots, a_m such that: (a) for each a_j , $1 \leq j \leq m$, $\langle r_j, (\sigma, H) \rangle \longrightarrow a_j$, and (b)

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

Therefore, it is sufficient to prove that

$$\text{cast}_P((\sigma, H)) \longrightarrow_{\mathcal{R}} \text{cast}_P((\sigma[a_1, \dots, a_m], H[\sigma, a_1, \dots, a_m]))$$

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

```

crl (Sigma,H) {P} => (Sigma[A],H[Sigma,A] {P}) {P}
  if mkActs(P, (Sigma,H)) => A .

```

and Lemma 9.4.2. 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 `mkActs` according to its definition, yielding

$$\begin{aligned} \text{mkActs}(\text{cast}(r_1) \dots \text{cast}(r_m), (\text{cast}(\sigma), \text{cast}(H))) = \\ \langle \text{cast}(r_1), (\text{cast}(\sigma), \text{cast}(H)) \rangle, \dots, \langle \text{cast}(r_m), (\text{cast}(\sigma), \text{cast}(H)) \rangle \end{aligned}$$

and which, according to Lemma 9.4.2, rewrites to

$$\text{cast}(a_1), \dots, \text{cast}(a_m)$$

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 $(\text{Sigma}', H')\{P\}$ with

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

and

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

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

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

and so it remains to be proved that $H' = \text{cast}(H[\sigma, a_1, \dots, a_m])$, which is left as an exercise.

($T_{\mathcal{M}}$ simulates $T_{\mathcal{R}}$): Let $(\sigma, H) \in \text{State}_P$ and let c' be a term of sort `Configuration` such that

$$\text{cast}_P((\sigma, H)) \longrightarrow_{\mathcal{R}} c'$$

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

$$\begin{aligned} \text{crl } (\text{Sigma}, H) \{P\} => (\text{Sigma}[A], H[\text{Sigma}, A] \{P\}) \{P\} \\ \text{if } \text{mkActs}(P, (\text{Sigma}, H)) => A \end{aligned}$$

c' 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^{Action} reachable via rewriting from the term

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

Appealing to Lemma 9.4.2, and to Lemma 9.4.1 about the injectivity of cast , it is enough to show that, letting $P = \{r_1, \dots, r_m\}$, A is of the form

$$\text{cast}(a_1), \dots, \text{cast}(a_m)$$

with $a_1, \dots, a_m \in \text{Action}_P$, 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

$$\mathcal{R}_{PRS} \vdash \langle \text{cast}(r_j), (\text{cast}(\sigma), \text{cast}(H)) \rangle \longrightarrow \text{cast}(a_j),$$

and therefore by Lemma 9.4.2 also that $\langle r_j, (\sigma, H) \rangle \longrightarrow a_j$. Then applying the rule

$$\frac{\langle r_1, (\sigma, H) \rangle \longrightarrow a_1 \quad \dots \quad \langle r_m, (\sigma, H) \rangle \longrightarrow a_m}{(\sigma, H) \longrightarrow_P (\sigma[a_1, \dots, a_m], H[\sigma, a_1, \dots, a_m])}$$

we get that $(\sigma, H) \longrightarrow_{\mathcal{M}} (\sigma[a_1, \dots, a_m], H[\sigma, a_1, \dots, a_m])$. Again, it remains to be shown that the $_[_]$ operators, for each component of the state, correspond, which follows according to a straightforward induction. \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 $\mathcal{A} = (A, \longrightarrow \subseteq A \times A)$ and an element $a \in A$, we let $\text{Reach}_{\mathcal{A}}(a) \subseteq A$ denote the set of reachable states from a ; *i.e.*

$$\{a' \in A \mid a \longrightarrow^* a'\}.$$

The relative correctness of \mathcal{M}_{PRS} and \mathcal{R}_{PRS} with respect to these correctness properties will fall out through instances of the following corollary to Theorem 9.4.1.

Corollary 9.4.1. *Let P be a production rule set and $s_0 \in \text{State}_P$. For any pair of predicates*

$$Q_{\mathcal{M}} \subseteq \text{State}_P \text{ and } Q_{\mathcal{R}} \subseteq \text{Configuration}$$

such that $s \in Q_{\mathcal{M}}$ if and only if $\mathbf{cast}_P(s) \in Q_{\mathcal{R}}$, then

$$\text{Reach}_{T_{\mathcal{M}}}(s_0) \subseteq Q_{\mathcal{M}} \text{ if and only if } \text{Reach}_{T_{\mathcal{R}}}(\mathbf{cast}_P(s_0)) \subseteq Q_{\mathcal{R}}$$

Proof of Corollary 9.4.1. (\Rightarrow): Suppose that $\text{Reach}_{T_{\mathcal{M}}}(s_0) \subseteq Q_{\mathcal{M}}$; we demonstrate by induction on $T_{\mathcal{R}}$ reachability derivations (these transition systems have finite carriers) that also

$$\text{Reach}_{T_{\mathcal{R}}}(\mathbf{cast}_P(s_0)) \subseteq Q_{\mathcal{R}}.$$

The induction hypothesis asserts that, for a reachable configuration

$$c \in \text{Reach}_{T_{\mathcal{R}}}(\mathbf{cast}_P(s_0))$$

both of the following conditions hold: (1) $c \in \mathbf{cast}_P(\text{State}_P)$, and for the unique $s \in \text{State}_P$, guaranteed by Lemma 9.4.1, such that $\mathbf{cast}_P(s) = c$, (2) $s \in \text{Reach}_{T_{\mathcal{M}}}(s_0)$. This implies that $s \in Q_{\mathcal{M}}$ and therefore that $c \in Q_{\mathcal{R}}$.

The induction hypothesis clearly holds for $c_0 = \mathbf{cast}_P(s_0)$. Now, let

$$\mathbf{cast}_P(s) = c \in \text{Configuration}$$

be such that it has both properties of the induction hypothesis and suppose that $c \rightarrow_{\mathcal{R}} c'$ with $c' \in \text{Configuration}$. It follows from Theorem 9.4.1 and the induction hypothesis that there exists a $s' \in \text{State}_P$ such that $s \rightarrow_{\mathcal{M}} s'$ and $\mathbf{cast}_P(s') = c'$. This implies that $c \in \mathbf{cast}_P(\text{State}_P)$ and that $s' \in \text{Reach}_{T_{\mathcal{M}}}(s_0)$.

(\Leftarrow): This direction follows similarly, but without the need of Lemma 9.4.1. □

9.4.3 Hazard Freedom

Hazard freedom essentially asserts the impossibility of reaching a state where any node takes the value \mathbf{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 $\sigma(y) = \mathbf{X}$.

Definition 9.4.1. Let P be a production rule set and $\sigma : \text{Variable}_P \rightarrow \text{Level}$.

We say that \mathcal{M}_{PRS} exhibits hazard freedom with respect to P and σ if and only if for all

$$(\sigma', H') \in \text{Reach}_{T_{\mathcal{M}}}((\sigma, \emptyset))$$

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

In rewriting logic, we can give an equationally-defined function, **Hazard!**, which is essentially the characteristic function of Hazard!_P .

```

op Hazard! : Configuration -> Bool [frozen] .
eq Hazard!(((Y |-> X, SIGMA), H){P}) = true .
eq Hazard!((SIGMA, H){P}) = false [owise] .

```

Definition 9.4.2. Let P be a production rule set and $\sigma : \text{Variable}_P \rightarrow \text{Level}$. We say that \mathcal{R}_{PRS} exhibits hazard freedom with respect to P and σ if and only if for all

$$c' \in \text{Reach}_{T_{\mathcal{R}}}(\text{cast}_P((\sigma, \emptyset)))$$

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

Proposition 9.4.1. Let P be a production rule set and $\sigma : \text{Variable}_P \rightarrow \text{Level}$. \mathcal{R}_{PRS} exhibits hazard freedom with respect to P and σ if and only if \mathcal{M}_{PRS} exhibits hazard freedom with respect to P and σ .

Proof. According to Corollary 9.4.1, 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. \square

9.4.4 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 9.2.3 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 either

- $\sigma(y) \neq 0$ 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, $Switchable_{P,\sigma,H} \subseteq Variable_P$, similar to $Enabled_{P,\sigma}$, but also with a parameter $H \subseteq Variable_P$. For a variable $y \in Variable_P$ and a state $(\sigma, H) \in State_P$, $y \in Switchable_{P,\sigma,H}$ if and only if either $y \in Enabled_{P,\sigma}$ or $y \neq X$ and $y \in H$.

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

Definition 9.4.3. Let P be a production rule set and $\sigma : Variable_P \rightarrow Level$. We say that \mathcal{M}_{PRS} exhibits deadlock freedom with respect to P and σ if and only if for all

$$(\sigma', H') \in Reach_{T_{\mathcal{M}}}((\sigma, \emptyset))$$

$Switchable_{P,\sigma',H'} \neq \emptyset$.

Above in Section 9.3 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 `EnabledPred`. We define a similar notion for $Switchable_{P,\sigma,H}$, called `SwitchPred`.

```

op SwitchPred : Variable ProductionRuleSet
  Map{Variable,Level} 2^Variable -> Bool .
ceq SwitchPred(Y, P, Sigma, H) = true
  if Sigma(Y) /= 1
  /\ [G+ -> Y +] P' := P
  /\ Sigma(G+) == 1 .
ceq SwitchPred(Y, P, Sigma, H) = true
  if Sigma(Y) /= 0
  /\ [G- -> Y -] P' := P
  /\ Sigma(G-) == 1 .
ceq SwitchPred(Y, P, Sigma, H) = true
  if Sigma(Y) /= X
  /\ Y in H .
eq SwitchPred(Y, P, Sigma, H) = false [owise] .

```

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

Definition 9.4.4. Let P be a production rule set and $\sigma : \text{Variable}_P \rightarrow \text{Level}$. We say that P exhibits *deadlock freedom with respect to P and σ* if and only if for all

$$(\text{SIGMA}', \text{H}')\{P\} \in \text{Reach}_{T_{\mathcal{R}}}(\text{cast}_P((\sigma, \emptyset)))$$

`Switchable`- $\{P, \text{SIGMA}', \text{H}'\} \neq \text{empty}$.

Proposition 9.4.2. Let P be a production rule set and $\sigma : \text{Variable}_P \rightarrow \text{Level}$. \mathcal{R}_{PRS} exhibits *deadlock freedom with respect to P and σ* if and only if \mathcal{M}_{PRS} exhibits *deadlock freedom with respect to P and σ* .

Proof. Similar to that for preservation of hazard freedom. □

9.5 Automated Hazard and Deadlock Freedom Analysis

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 [50].

Section 9.5.1 briefly describes each of the asynchronous circuits we are subjecting to analysis. Section 9.5.2 describes the commands necessary to check hazard and deadlock freedom using Maude and the semantics \mathcal{R}_{PRS} presented in Section 9.3. These results demonstrate the need for some optimizations, which are described in Section 9.5.3, followed in Section 9.5.4 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.

9.5.1 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 [50].

- **3InverterRing** (12 production rules): A ring oscillator is typically the first circuit used to demonstrate the viability of a new process technology.
- **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.

9.5.2 Experiments, without Optimizations

The hazard and deadlock freedom analyses are performed using Maude's `search` command [15, §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 that invariant.

As described above in Section 9.4.3, the invariant we want to check for hazard freedom is the negation of `Hazard!`, or alternatively, that no reachable state satisfies `Hazard!`. We use this second formulation, which is accomplished in Maude with the following command,

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

The term `initialC` equals `castP((σ , \emptyset))` where σ 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 command is (The reason why the search command cannot use the `=>!` modality is because empty sets of actions can always produce idle transitions.)

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

Applying these checks to each of the circuits described above in Section 9.5.1 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 Figure 9.2. Clearly, some form of simplification/optimization is needed to reduce memory consumption and gain tractability even for the very small circuits we are considering.

9.5.3 Performance Optimizations

Two optimizations to the rewriting logic semantics, \mathcal{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 it goes from being memory bound to being computation bound.

Circuit Name	Size	Hazard Freedom	Deadlock Freedom
3InverterRing	12rl	MEM	MEM
ClosedBuffer	26rl	MEM	MEM
Toggle	28rl	MEM	MEM
PCHBAndFixed	66rl	MEM	MEM
1BitFullAdderFixed	118rl	MEM	MEM
PCHBAndToggle	130rl	MEM	MEM

Figure 9.2: Hazard and Deadlock Freedom Analysis Results, *without* Optimizations 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 set memory limit (4GiB).

Out-of-control memory usage is primarily due to the condition of the top-level rewrite rule in \mathcal{R}_{PRS} , which, recalling from Section 9.3, is

$$\text{mkActs}(P, (\text{Sigma}, I)) \Rightarrow A.$$

Suppose that we have the production rule set

$$\begin{aligned} & [\quad 'x \rightarrow 'z \ -] \\ & [\quad 'y \rightarrow 'w \ -] \\ & [\text{not } 'y \rightarrow 'w \ +] \end{aligned}$$

Applying `mkActs` to this set yields the following term of *kind* $[2^{\text{Action}}]$

$$\begin{aligned} & \langle [\quad 'x \rightarrow 'z \ -], (\text{Sigma}, I) \rangle , \\ & \langle [\quad 'y \rightarrow 'w \ -], (\text{Sigma}, I) \rangle , \\ & \langle [\text{not } 'y \rightarrow 'w \ +], (\text{Sigma}, I) \rangle \end{aligned}$$

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

$$\begin{aligned} & 'z := 0 , \\ & \text{skip} , \\ & 'w := X \end{aligned}$$

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

$$\begin{aligned} & \langle [\quad 'x \rightarrow 'z \ -], (\text{Sigma}, I) \rangle , \\ & \qquad \qquad \qquad \text{skip} , \\ & \langle [\text{not } 'y \rightarrow 'w \ +], (\text{Sigma}, I) \rangle \end{aligned}$$

and then

$$\begin{aligned} & \quad \quad \quad 'z := 0 , \\ & \quad \quad \quad \text{skip} , \\ & \quad < [\text{not } 'y \rightarrow 'w +], (\text{Sigma}, I) > \end{aligned}$$

does not need to be considered separately from first rewriting

$$\begin{aligned} & \quad \quad \quad 'z := 0 , \\ & < [\quad 'y \rightarrow 'w -], (\text{Sigma}, I) > , \\ & < [\text{not } 'y \rightarrow 'w +], (\text{Sigma}, I) > \end{aligned}$$

followed by

$$\begin{aligned} & \quad \quad \quad 'z := 0 , \\ & \quad \quad \quad \text{skip} , \\ & < [\text{not } 'y \rightarrow 'w +], (\text{Sigma}, I) > \end{aligned}$$

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 \mathcal{R}_{PRS} , become bound to the variable `A`. Accomplishing this reduction is done through a small modification to the inference rules for \longrightarrow in Definition 9.2.2, so that, for example,

$$\frac{\cdot}{\langle g \mapsto x \uparrow, (\sigma, H) \rangle \longrightarrow x := 1} \sigma(g) = 1$$

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 σ and H during a transition when, for example, $\sigma(x) = 1$ and one of the updating actions is $x := 1$.

9.5.4 Experiments, with Optimizations

The result of applying each optimization in isolation, as well as the aggregate effect of applying both in tandem, is conveyed in the second table listed in of

Circuit Name	Size	list opt	switch opt	all opts
3InverterRing	12rl	no – states – 682ms – 1, 561, 117 rewrites	no – states – 296, 363ms – 174, 533, 736 rewrites	no N/A states – 5ms – 13, 709 rewrites no
ClosedBuffer	26rl	MEM	MEM	N/A states – 1, 648ms – 4, 343, 371 rewrites no
Toggle	28rl	no – states – 356, 559ms – 802, 141, 445 rewrites	MEM	N/A states – 454ms – 1, 224, 675 rewrites no
PCHBAndFixed	66rl	TIME	MEM	N/A states – 638, 570ms – 1, 552, 737, 662 rewrites
1BitFullAdderFixed	118rl	TIME	MEM	TIME
PCHBAndToggle	130rl	TIME	MEM	TIME

Figure 9.3: Hazard Freedom Analysis Results, *with* Optimizations. 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 set memory limit (4GiB).

Figure 9.3 for hazard freedom, and conveyed through Figure 9.4 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; though of course \mathcal{R}_{PRS} was designed for conceptual clarity, above all.

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.

Circuit Name	Size	list opt	switch opt	all opts
3InverterRing	12rl	TIME	MEM	no N/A states – 1,315,793ms – 2,517,712,268
ClosedBuffer	26rl	MEM	MEM	rewrites TIME
Toggle	28rl	TIME	MEM	no N/A states – 73,891ms – 177,563,797
PCHBAndFixed	66rl	TIME	MEM	rewrites no N/A states – 794,175ms – 1,886,088,552
1BitFullAdderFixed	118rl	TIME	MEM	rewrites TIME
PCHBAndToggle	130rl	TIME	MEM	TIME

Figure 9.4: Deadlock Freedom Analysis Results, *with* Optimizations. 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 set memory limit (4GiB).

9.6 Speed-Independent and Quasi-Delay-Insensitive Circuits

The primary objective of this paper is to improve upon our semantics work in [55, 47] for the unrestricted, or *delay-insensitive* case. With that said, it is also worthwhile to look briefly at analysis statistics for asynchronous circuits under two other timing assumptions, *speed-independence* [81, 79] and *quasi-delay-insensitivity* [65, 68], which are considered practical for developing large-scale devices. This section presents those results, which are based on an implementation of these timing assumptions in Maude that was developed in accordance with [55].

Of course, a very desirable avenue for future work is to give a similar treatment to these cases that is given in this paper for the delay-insensitive case. And, as speed-independence and quasi-delay-insensitivity simply constrain the set of possible behaviors exhibited in the delay-insensitive case, the infrastructure developed in this paper could be used directly in such an endeavor. However, this is a very substantial undertaking and outside the scope of the current work.

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 (see [55, 70] for more details).

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 [64]. 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 [55].

Figures 9.5 and 9.6 present the results of analyzing the circuits from Section 9.5.1 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 Figure 9.7.

Due to the result from [55] cited above, this implies hazard-freedom in the quasi-delay-insensitive case as well.

9.7 Conclusion

This paper improves upon our earlier work in [55, 47], providing a cleaner formal semantics of production rule sets for the delay-insensitive case, including both a mathematical 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 mathematical semantics is perhaps best suited as the foundation for proving meta-theorems about production rule sets, such as we did in [55]. 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 [47].

A number of challenges remain, some rather daunting. Firstly, the speed-independence and quasi delay-insensitivity cases from [55] 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 [2, 3] allows for a rewriting-based approach to continue to be used, and perhaps even build directly on our work here.

Circuit Name	Size	delay-insensitive	speed-independence
3InverterRing	12rl	no - states - 2ms - 10,944	yes 12 states - 9ms - 55,254
ClosedBuffer	26rl	rewrites no - states - 372ms - 2,054,504	rewrites yes 20 states - 93ms - 513,522
Toggle	28rl	rewrites no - states - 118ms - 766,143	rewrites yes 28 states - 63ms - 392,887
PCHBAndFixed	66rl	rewrites no - states - 74,692ms - 415,399,458	rewrites yes 681 states - 9,647ms - 55,564,688
1BitFullAdderFixed	118rl	rewrites -	rewrites -
PCHBAndToggle	130rl	-	-

Circuit Name	Size	quasi delay-insensitive
3InverterRing	12rl	yes 17 states - 18ms - 100,313
ClosedBuffer	26rl	rewrites yes 59 states - 551ms - 2,561,312
Toggle	28rl	rewrites yes 139 states - 502ms - 2,648,172
PCHBAndFixed	66rl	rewrites yes 2,679 states - 83,471ms - 409,224,700
1BitFullAdderFixed	118rl	rewrites -
PCHBAndToggle	130rl	-

Figure 9.5: Hazard-freedom model checking results. Maude 2.5, Intel Xeon X5570 (2.93GHz, 8MB L3), 24GB RAM, 64-bit Linux, kernel version 2.6.18. “-” means timed out or exhausted memory resources.

Circuit Name	Size	delay-insensitive	speed-independence
3InverterRing	12rl	no - states - 1,497ms - 9,051,904 rewrites	yes 12 states - 9ms - 55,242 rewrites yes 20 states - 97ms - 513,502 rewrites yes 28 states - 63ms - 392,859 rewrites no - states - 10,395ms - 55,523,752 rewrites -
ClosedBuffer	26rl	-	-
Toggle	28rl	-	-
PCHBAndFixed	66rl	-	-
1BitFullAdderFixed	118rl	-	-
PCHBAndToggle	130rl	-	-

Circuit Name	Size	quasi delay-insensitive
3InverterRing	12rl	yes 17 states - 17ms - 100,296 rewrites yes 59 states - 555ms - 2,561,253 rewrites yes 139 states - 501ms - 2,648,033 rewrites no - states - 87,629ms - 408,664,483 rewrites -
ClosedBuffer	26rl	-
Toggle	28rl	-
PCHBAndFixed	66rl	-
1BitFullAdderFixed	118rl	-
PCHBAndToggle	130rl	-

Figure 9.6: Deadlock-freedom model checking results. Maude 2.5, Intel Xeon X5570 (2.93GHz, 8MB L3), 24GB RAM, 64-bit Linux, kernel version 2.6.18. “-” means timed out or exhausted memory resources.

Circuit Name	Size	speed-independence
3InverterRing	12rl	yes 6 states - 1ms - 9, 514 rewrites
ClosedBuffer	26rl	yes 10 states - 5ms - 37, 596
Toggle	28rl	rewrites yes 12 states - 5ms - 42, 546
PCHBAndFixed	66rl	rewrites yes 114 states - 668ms - 4, 270, 606
1BitFullAdderFixed	118rl	rewrites yes 1, 800 states - 117, 925ms - 453, 471, 253
PCHBAndToggle	130rl	rewrites yes 2, 844 states - 76, 436ms - 298, 696, 957
		rewrites

Figure 9.7: Hazard-freedom model checking results, “no wires” optimization for speed-independence. Maude 2.5, Intel Xeon X5570 (2.93GHz, 8MB L3), 24GB RAM, 64-bit Linux, kernel version 2.6.18.

CHAPTER 10

SEMANTICS: BTRS

This chapter develops an executable semantics in rewriting logic for a language called BTRS [21], which is a simplified form of a more feature-rich hardware description language called Bluespec [8]. Bluespec is based on *guarded atomic actions*, or *rules*, and aims to realize the productivity and correctness benefits of a modern high-level language; what sets it apart is that it attempts to do this in the context of digital design, where the evolution of languages has languished. The Bluespec compiler can be used as part of a synthesis chain generating efficient hardware from Bluespec source.

The syntax of BTRS and a big-step operational semantics [43] are developed in [21], and it is upon that formalization that our executable one in rewriting logic is based. Our aim in developing our formalization in rewriting logic is that it mimics the one from [21] extremely closely, so that the correctness of the rewriting logic specification is straightforward.

In addition to the formalization of BTRS in rewriting logic, this chapter makes a few additional contributions. First, a couple of quite small bugs in the operational rules from [21] are discovered and fixed. Second, an explicit rule, both in the style of [21] as well as in rewriting logic, is developed for driving the execution of a BTRS program. Third, we present a small case study involving a version of a completion buffer that, in some situations but not all situations, may deadlock; we demonstrate how to use the execution and formal analysis tools provided by our rewriting logic specification and the rewriting logic tool Maude to expose this deadlock, something testing could miss.

$m ::=$	<code>Module name</code>	
	<code>[Register r v]</code>	// Regs w/ initial values
	<code>[Rule R a]</code>	// Rules
	<code>[ActMeth g λx.a]</code>	// Action method
	<code>[ValMeth f λx.e]</code>	// Value method
$a ::=$	<code>r := e</code>	// Register update
	<code> if e then a</code>	// Conditional action
	<code> a a</code>	// Parallel composition
	<code> a ; a</code>	// Sequential composition
	<code> a when e</code>	// Guarded action
	<code> (t = e in a)</code>	// Let action
	<code> m.g(e)</code>	// Action methcall g of m
$e ::=$	<code>r</code>	// Register Read
	<code> c</code>	// Constant Value
	<code> t</code>	// Variable Reference
	<code> e op e</code>	// Primitive Operation
	<code> e ? e : e</code>	// Conditional Expression
	<code> e when e</code>	// Guarded Expression
	<code> (t = e in e)</code>	// Let Expression
	<code> m.f(e)</code>	// Value Methcall f of m
$op ::=$	<code>&& ...</code>	// Primitive operations

Figure 10.1: BTRS Grammar (verbatim copy of [21, Figure 1]).

10.1 Syntax

The grammar defining BTRS syntax is shown in Figure 10.1; it is a verbatim reprint of [21, Figure 1]. Each of the relevant syntactic categories, *expressions*, *actions*, and so forth, is mapped to an associated sort in our rewriting logic semantics; for example, expressions will become terms of sort `Expression`.

All of the sorts used in the rewriting logic specification of BTRS syntax, save for one, are declared in the module `SYNTAX-SKELETON`, displayed next. `Program` is commented out by necessity since, as we will see, it has a dependency that cannot yet be resolved. In addition, a few sorts are included in the rewriting logic semantics that have no analogous production in the grammar from Figure 10.1, but are nevertheless convenient and natural; for example, `Rule`.


```
fmod SYNTAX-SKELETON is
  sort Identifier .
  sort Literal .
  sort PrefixOp .
  sort InfixOp .
  sort Expression .
  sort Action .
  sort Rule .
  sort Register .
  sort Method .
  sort Component .
  sort Module .
  --- sort Program (TBD)
endfm
```

Terms of each of these sorts are added through the notion of an *extending* module importation in Maude [15, §8.1.2]. This is a common technique in rewriting logic semantics [77, 19, 78], as it enhances modularity; for example, we avoid having to fix a set of identifiers, values, or operators. See Section 10.3 for an example.

One of the main goals stated at the outset of this chapter was to develop a rewriting logic semantics for BTRS that is “evidently correct”; that is, representationally so near to the original semantics in [21] that its correctness is essentially evident. In what follows, each segment of the rewriting logic specification is accompanied by the associated portion of [21, Figure 1] from which it was derived, so that the two specifications may be easily compared.

The syntax of expressions deviates only slightly from the original BTRS grammar in Figure 10.1. A constructor is added for prefix expressions.

```

    e ::= r           // Register Read
        || c          // Constant Value
        || t          // Variable Reference
        || e op e     // Primitive Operation
        || e ? e : e  // Conditional Expression
        || e when e   // Guarded Expression
        || (t = e in e) // Let Expression
        || m.f(e)     // Value Methcall f of m

fmod EXPRESSION is extending SYNTAX-SKELETON .
  subsort Identifier Literal < Expression .
  op __      : PrefixOp  Expression          -> Expression .
  op ___     : Expression InfixOp  Expression -> Expression .
  op _?:_    : Expression Expression Expression -> Expression .
  op _when_  : Expression Expression          -> Expression .
  op _=in_   : Identifier Expression Expression -> Expression .
  op _._(_)  : Identifier Identifier Expression -> Expression .
endfm

```

Actions effect state changes, similar to how statements do in procedural languages, though as we will see below, with substantial differences. The only deviation in the rewriting logic specification is the addition of a second dot for action-method invocation. This is simply to disambiguate it from value-method invocation and to suppress a warning from Maude.

```

    a ::= r := e      // Register update
        || if e then a // Conditional action
        || a | a      // Parallel composition
        || a ; a      // Sequential composition
        || a when e   // Guarded action
        || (t = e in a) // Let action
        || m.g(e)     // Action methcall g of m

fmod ACTION is extending SYNTAX-SKELETON .
  op _:=_      : Identifier Expression      -> Action .
  op if_then_  : Expression Action         -> Action .
  op _|_       : Action Action             -> Action .
  op _;_       : Action Action             -> Action .
  op _when_    : Action Expression         -> Action .
  op _=_in_    : Identifier Expression Action -> Action .
  op _.._(_)   : Identifier Identifier Expression -> Action .
endfm

```

Modules are essentially named packages of different “components”: register declarations, method definitions, and rules. Although the original BTRS grammar does not associate individual productions to these components, we find it convenient in the rewriting logic semantics to give each type of component its own sort. We start with the individual components and then move on to full modules.

```

    m ::= Module name
        [Register r v] // Regs w/ initial values
        [Rule R a]     // Rules
        [ActMeth g λx.a] //Action method
        [ValMeth f λx.e] //Value method

fmod COMPONENT is extending SYNTAX-SKELETON .
  op Register__ : Identifier Literal      -> Register .
  op Rule__     : Identifier Action       -> Rule .
  op ActMeth_\_ : Identifier Identifier Action -> Method .
  op ValMeth_\_ : Identifier Identifier Expression -> Method .

  subsort Register Rule Method < Component .
endfm

```

With these module components defined, we simply instantiate the parameterized SET module from the Maude prelude [15, §9.12.2] with an appropriate view and add a constructor symbol to get modules.

```

view Component    from TRIV to SYNTAX-SKELETON is
  sort Elt to Component .
endv
view Module       from TRIV to SYNTAX-SKELETON is
  sort Elt to Module .
endv
fmod PROGRAM is extending SYNTAX-SKELETON .
  protecting SET{Component}
    * ( op _',_ to __ ) .
  protecting SET{Module}
    * ( sort Set{Module} to Program
      , op _',_ to __ ) .

  op Module__ : Identifier Set{Component} -> Module .
endfm

```

The *syntax* of BTRS, according to our rewriting logic semantics, is then just the collection of all of the above modules. Specific identifiers, literal values, and the basic operators are defined on a by-need basis. An example is presented in the following section, to demonstrate how this idea works.

```

fmod SYNTAX is
  including EXPRESSION .
  including ACTION .
  including COMPONENT .
  including PROGRAM .
endfm

```

10.2 Caveats

The grammar of BTRS allows for some programs that we will assume to be outside the scope of what our semantics considers. These limitations mostly have to do with how identifiers are used. For example, modules are assumed to be uniquely named and, therefore, assuming $m1$, $f0$, and $f1$ are terms of

sort `Identifier`, the following BTRS program falls outside the scope of what is considered.

```
Module m1
  (Register f0 0)
Module m1
  (Register f1 0)
```

Similarly, all register names should be unique, even across separate modules. Therefore, the following modification to the above BTRS program addresses the module naming problem, but since the register names now collide it still falls outside the scope of what is considered.

```
Module m1
  (Register f0 0)
Module m2
  (Register f0 0)
```

10.3 Example: Single-Element Queue

```
Module fifo
  Register vf0 false
  Register f0
  ActMeth enq(x) = (vf0 := true | f0 := x) when !vf0
  ActMeth deq() = (vf0 := false) when vf0
  ValMeth first() = f0 when vf0
```

Figure 10.2: BTRS Single-Element Queue (verbatim from [21, Figure 3])

Consider the BTRS module shown in Figure 10.2, which implements a single-element queue. Some syntactic liberties have been taken in the FIFO module. Register `f0` is not provided with an initial value, and both `deq` and `first` omit their argument binding. In addition, a non-binary operator, `!`, is used.

The single-element queue is specified in Maude by first extending the `SYNTAX` module with all necessary literals, identifiers, and operator symbols.

```

fmod SYNTAX-EXT is extending SYNTAX .
  including NAT .

  sort Boolean .
  subsort Nat Boolean < Literal .
  ops fifo vf0 f0 x : -> Identifier .
  ops enq deq first : -> Identifier .
  op ! : -> PrefixOp .
  ops True False : -> Boolean .
endfm

```

With the additional syntax, the specification of the queue is straightforward.

```

fmod EXAMPLE is including SYNTAX-EXT .
  op FIFO : -> Module .
  eq FIFO =
    Module fifo
      (Register vf0 False)
      (Register f0 0)
      (ActMeth enq \ x . (vf0 := True | f0 := x) when ! vf0)
      (ActMeth deq \ x . ((vf0 := False) when vf0))
      (ValMeth first \ x . (f0 when vf0)) .
    endfm

```

10.4 Semantics Overview

The semantics of BTRS expression and action evaluation are defined in [21, Figure 4] through a set of operational-style inference rules. As a convenience for the reader, the content of that figure is reprinted in a set of figures apportioned across Sections 10.5 and 10.6, according to subject. Section 10.5 details expression evaluation and Section 10.6 details action evaluation. Our aim is to mimic these rules as closely as possible in rewriting logic.

In addition to expression and action evaluation, the semantics of BTRS must specify how changes to program state are driven by repeatedly selecting any ready BTRS-level rule and executing its body; this is detailed in Section 10.7.

Some essential infrastructure is required throughout the semantics specification. One of the essential concepts that is needed is that of a mapping

from identifiers to values, something we call a *store*. Stores are implemented via Maude’s parameterized MAP module [15, Ch. 9.13.1].

```
fmod SEMANTICS-SKELETON-1 is including SYNTAX .
  protecting MAP{Identifier,Literal}
  * ( sort Map{Identifier,Literal} to Store
    , op _[_] to _(_) ) .
endfm
```

The general forms given in [21] for expression evaluation and action evaluation are $\langle S, U, B \rangle \vdash e \rightarrow v$ and $\langle S, U, B \rangle \vdash a \rightarrow U'$, respectively; where the S, U, U', B are all stores, e is an expression, v is either a program value or a special term, **NR**, denoting a “not-ready” condition, and a is an action. Corresponding operators are introduced according to the following module.

```
fmod SEMANTICS-SKELETON-2 is including SEMANTICS-SKELETON-1 .
  sort EvaluationState .

  op <_,_,_> : Store Store Store -> EvaluationState .
  op _|-(_)_->> : EvaluationState Program Expression ~> Literal .
  op _|-(_)_->> : EvaluationState Program Action ~> Store .
  op NR : -> [Literal] .
endfm
```

Unlike the \vdash symbol used in [21], our $|-$ operator makes explicit the BTRS program in which the evaluation is taking place. The $|-$ operator differs in another way with \vdash : it is a term that stands for any result that, with \vdash , goes syntactically after the \rightarrow in the relation. If this result, when it exists, is unique, then we can employ Maude-level equations to define $|-$; if it is not, we must use Maude-level rules to define the $|-$ operators. It so happens that equations suffice. The operators are partial (as indicated by the $\sim>$ arrow), and therefore defined at the kind level, both to support the **NR** “result”, and because not all actions can be meaningfully evaluated with respect to a given evaluation state and program; for example, because of the failure of a `_when_` guard.

In addition, note that literals are employed as the result of successfully evaluating an expression. For this to be valid it is assumed that data values and the space of program literals are isomorphic. **NR** is an error value and is

treated in the usual fashion of Maude error terms, meaning that it is a term in the same *kind* of sort `Literal`, but does not have a *sort*.

Operator evaluation is handled via Maude's meta-level [15, §14]. Each operator will be mapped to a meta-level representation of another operator with the intended semantics. There is one such lookup table for `PrefixOps` and a separate lookup table for `InfixOps`. As with the operator symbols themselves, the intention of these tables is that they are populated on an as-needed basis through module extensions.

```
fmod SEMANTICS-SKELETON-3 is including SYNTAX .
  including META-LEVEL ...necessary renamings omitted
  including QID .

  op prefixOpLookup : PrefixOp -> Qid .
  op infixOpLookup : InfixOp -> Qid .
endfm
```

Booleans are required so that conditional and guarding expressions and actions can be evaluated.

```
fmod SEMANTICS-SKELETON-4 is including SYNTAX .
  sort Boolean .
  ops True False : -> Boolean .
  subsort Boolean < Literal .
endfm
```

Two forms of substitution are defined on stores: one which does a single substitution for a given identifier and value, and a second one that updates a store from a *set of mappings* provided by a second store. In either case, identifiers previously mapped have their values updated, and identifiers without previous mappings are in some sense newly inserted.

```
fmod SEMANTICS-SKELETON-5 is including SEMANTICS-SKELETON-1 .
  ... variable declarations omitted

  op _[_/_] : Store Literal Identifier -> Store .
  eq S[V / I] = insert(I, V, S) .

  op _[_] : Store Store -> Store .
  eq S[empty] = S .
  eq S[(I |-> V) , S'] = (S[V / I])[S'] .
endfm
```


The topmost level of our semantics will rewrite pairs containing a BTRS program and a store mapping each device register to its current value; such pairs are called *configurations*.

```
fmod SEMANTICS-SKELETON-6 is including SEMANTICS-SKELETON-1 .
  sort Configuration .
  op <_,_> : Program Store -> Configuration .
endfm
```

The basic infrastructure for defining the BTRS semantics is then just the aggregation of the above modules.

```
fmod SEMANTICS-SKELETON is
  including SEMANTICS-SKELETON-1 .
  including SEMANTICS-SKELETON-2 .
  including SEMANTICS-SKELETON-3 .
  including SEMANTICS-SKELETON-4 .
  including SEMANTICS-SKELETON-5 .
  including SEMANTICS-SKELETON-6 .
endfm
```

10.5 Expression Evaluation

Expression evaluation is defined in [21, Figure 4] through a set of operational-style rules. Our rewriting logic semantics simply attempts to follow these rules in a very close way. For convenience, these operational-style rules are reprinted in Figure 10.3.

For each of the operational-style rules shown in Figure 10.3 we will have a corresponding equation in rewriting logic. The remainder of this section presents those equations, one-by-one and side-by-side with the appropriate operational-style rule from Figure 10.3. The multiple printings are simply intended to serve as a convenience to the reader; the aggregated rules allowing for easy reference of the entire space of rules, and the embedded rules serving to justify the equational definition as well as demonstrate the small representational distance between the two semantic formalizations.

We begin with the “reg-read” rule and its corresponding equation in rewriting logic

Expression Rules:	
reg-read	$\langle S, U, B \rangle \vdash r \rightarrow (U++S)(r)$
const	$\langle S, U, B \rangle \vdash c \rightarrow \underline{c}$
variable	$\langle S, U, B \rangle \vdash t \rightarrow B(t)$
op	$\frac{\langle S, U, B \rangle \vdash e_1 \rightarrow v_1, v_1 \neq \text{NR} \quad \langle S, U, B \rangle \vdash e_2 \rightarrow v_2, v_2 \neq \text{NR}}{\langle S, U, B \rangle \vdash e_1 \text{ op } e_2 \rightarrow v_1 \underline{\text{op}} v_2}$
tri-true	$\frac{\langle S, U, B \rangle \vdash e_1 \rightarrow \text{true}, \langle S, U, B \rangle \vdash e_2 \rightarrow v}{\langle S, U, B \rangle \vdash e_1 ? e_2 : e_3 \rightarrow v}$
tri-false	$\frac{\langle S, U, B \rangle \vdash e_1 \rightarrow \text{false}, \langle S, U, B \rangle \vdash e_3 \rightarrow v}{\langle S, U, B \rangle \vdash e_1 ? e_2 : e_3 \rightarrow v}$
e-when-true	$\frac{\langle S, U, B \rangle \vdash e_2 \rightarrow \text{true}, \langle S, U, B \rangle \vdash e_1 \rightarrow v}{\langle S, U, B \rangle \vdash e_1 \text{ when } e_2 \rightarrow v}$
e-when-false	$\frac{\langle S, U, B \rangle \vdash e_2 \rightarrow \text{false}}{\langle S, U, B \rangle \vdash e_1 \text{ when } e_2 \rightarrow \text{NR}}$
e-let-sub	$\frac{\langle S, U, B \rangle \vdash e_1 \rightarrow v_1, \langle S, U, B[v/t] \rangle \vdash e_2 \rightarrow v_2}{\langle S, U, B \rangle \vdash t = e_1 \text{ in } e_2 \rightarrow v_2}$
e-meth-call	$\frac{\langle S, U, B \rangle \vdash e \rightarrow v, v \neq \text{NR}, \quad m.f = \langle \lambda t.e_b \rangle, \langle S, U, B[v/t] \rangle \vdash e_b \rightarrow v'}{\langle S, U, B \rangle \vdash m.f(e) \rightarrow v'}$

Figure 10.3: Expression Evaluation Rules (verbatim from [21, Figure 4]).

$\langle S, U, B \rangle \vdash r \rightarrow (U++S)(r)$
becomes
<code>ceq < S, U, B > -(P) R ->> = (S[U])(R)</code>
<code>if \$hasMapping(S[U], R) .</code>

The condition in the above equation is needed to distinguish terms of sort `Identifier` that are being used as registers, as opposed to local bindings. The distinction that is made is whether the variable has a mapping in the S component of an evaluation state, meaning it is a register, or if it has a mapping in the B component of the evaluation state, meaning it is a local binding. The caveats explained in Section 10.2 allow us to use this mechanism soundly. Aside from the condition that ensures the identifier is, indeed, a register, and using `_[]` for `++`, the two definitions are almost identical.

Constants, defined in via rule “const”, are handled next. Recall that we

identify program values with program literals.

$$\langle S, U, B \rangle \vdash c \rightarrow \underline{c}$$

becomes

$$\text{eq } \langle S, U, B \rangle \vdash \text{-(P) } C \text{ ->> } = C \text{ .}$$

For the “variable” rule, a condition is added having the same purpose as the one introduced for the “reg-read” rule.

$$\langle S, U, B \rangle \vdash t \rightarrow B(t)$$

becomes

$$\begin{aligned} \text{ceq } \langle S, U, B \rangle \vdash \text{-(P) } T \text{ ->> } &= B(T) \\ \text{if } \$\text{hasMapping}(B, T) &\text{ .} \end{aligned}$$

Compared to the earlier semantics equations, and the ones that follow, handling operators requires the most sophisticated use of Maude’s built-in features, specifically the meta-level [15, §14]. Maude’s meta-level is used to reify the meaning of each operator; that is, going from *op* to op in the “op” rule.

$$\frac{\begin{array}{l} \langle S, U, B \rangle \vdash e_1 \rightarrow v_1, v_1 \neq \text{NR}, \\ \langle S, U, B \rangle \vdash e_2 \rightarrow v_2, v_2 \neq \text{NR} \end{array}}{\langle S, U, B \rangle \vdash e_1 \text{ op } e_2 \rightarrow v_1 \underline{\text{op}} v_2}$$

becomes

$$\begin{aligned} \text{ceq } \langle S, U, B \rangle \vdash \text{-(P) } E1 \text{ OP } E2 \text{ ->> } &= \text{evalInfixOp}(V1, OP, V2) \\ \text{if } V1 := \langle S, U, B \rangle \vdash \text{-(P) } E1 \text{ ->>} & \\ \wedge V1 \neq \text{NR} & \\ \wedge V2 := \langle S, U, B \rangle \vdash \text{-(P) } E2 \text{ ->>} & \\ \wedge V2 \neq \text{NR} &\text{ .} \end{aligned}$$

$$\begin{aligned} \text{op evalInfixOp : Literal InfixOp Literal } \sim &\text{> Literal .} \\ \text{eq evalInfixOp}(L1, OP, L2) = \text{downTerm} & \\ \text{infixOpLookup}(OP) [\text{upTerm}(L1), \text{upTerm}(L2)] & \\ , \text{NR} &\text{ .} \end{aligned}$$

`evalInfixOp` first constructs the meta-level representation of a term that captures the intended semantics of *OP/op*. The crucial part of the meta-level

term construction is done through the table `infixOpLookup`, which takes `OP/op` as an argument and returns the meta-level representation of `op` as its result. Finally, `downTerm` brings the meta-level representation of `v1 op v2` to the object-level.

Although [21] does not explicitly handle unary prefix operators, their semantics is evident from the above rule.

```

ceq < S, U, B > |-(P) OP1 E1 ->> = evalPrefixOp(OP1, V1)
  if V1 := < S, U, B > |-(P) E1 ->>
  /\ V1 =/= NR .

op evalPrefixOp : PrefixOp Literal ~> Literal .
eq evalPrefixOp(OP1, L1) =
  downTerm(prefixOpLookup(OP1) [upTerm(L1)], NR) .

```

The conditional operator is defined with a pair of equations, just as it is defined via a pair of rules, “tri-true” and “tri-false”, in [21]. Note that it is this rule that necessitates the inclusion of Booleans in our semantic infrastructure.

$$\frac{\langle S, U, B \rangle \vdash e_1 \rightarrow true, \langle S, U, B \rangle \vdash e_2 \rightarrow v}{\langle S, U, B \rangle \vdash e_1 ? e_2 : e_3 \rightarrow v}$$

$$\frac{\langle S, U, B \rangle \vdash e_1 \rightarrow false, \langle S, U, B \rangle \vdash e_3 \rightarrow v}{\langle S, U, B \rangle \vdash e_1 ? e_2 : e_3 \rightarrow v}$$

becomes

```

ceq < S, U, B > |-(P) E1 ? E2 : E3 ->> = V
  if True := < S, U, B > |-(P) E1 ->>
  /\ V := < S, U, B > |-(P) E2 ->> .
ceq < S, U, B > |-(P) E1 ? E2 : E3 ->> = V
  if False := < S, U, B > |-(P) E1 ->>
  /\ V := < S, U, B > |-(P) E3 ->> .

```

As with conditional expressions, guarded expression evaluation in the two systems is nearly identical.

$$\frac{\langle S, U, B \rangle \vdash e_2 \rightarrow true, \langle S, U, B \rangle \vdash e_1 \rightarrow v}{\langle S, U, B \rangle \vdash e_1 \text{ when } e_2 \rightarrow v}$$

$$\frac{\langle S, U, B \rangle \vdash e_2 \rightarrow false}{\langle S, U, B \rangle \vdash e_1 \text{ when } e_2 \rightarrow \text{NR}}$$

becomes

```

ceq < S, U, B > |-(P) E1 when E2 ->> = V
  if True := < S, U, B > |-(P) E2 ->>
  /\ V := < S, U, B > |-(P) E1 ->> .
ceq < S, U, B > |-(P) E1 when E2 ->> = NR
  if False := < S, U, B > |-(P) E2 ->> .

```

Local bindings are held separately in the third component of what we have termed the evaluation state; that is, the component that is denoted B to suggest *bindings*. As expounded upon in Section 10.8, this rule must be slightly modified in translation so that variable names match appropriately.

$$\frac{\langle S, U, B \rangle \vdash e_1 \rightarrow v_1, \langle S, U, B[v/t] \rangle \vdash e_2 \rightarrow v_2}{\langle S, U, B \rangle \vdash t = e_1 \text{ in } e_2 \rightarrow v_2}$$

becomes

```

ceq < S, U, B > |-(P) (T = E1 in E2) ->> = V2
  if V1 := < S, U, B > |-(P) E1 ->>
  /\ V2 := < S, U, B[V1 / T] > |-(P) E2 ->> .

```

The equation for method invocation demonstrates why we have included the BTRS program as an argument of the $|-$ operators. We use AC-matching [15, §4.8] to pick the appropriate method from the appropriate module.

$$\frac{\langle S, U, B \rangle \vdash e \rightarrow v, v \neq \text{NR}, \quad m.f = \langle \lambda t.e_b \rangle, \langle S, U, B[v/t] \rangle \vdash e_b \rightarrow v'}{\langle S, U, B \rangle \vdash m.f(e) \rightarrow v'}$$

becomes

```

ceq < S, U, B > |-(P) M . F(E) ->> = V'
  if (Module M ((ValMeth F \ T . EB) CS)) P' := P
  /\ V := < S, U, B > |-(P) E ->>
  /\ V /= NR
  /\ V' := < S, U, B[V / T] > |-(P) EB ->> .

```

Action Rules:	
reg-update	$\frac{\langle S, U, B \rangle \vdash e \rightarrow v, v \neq \text{NR}}{\langle S, U, B \rangle \vdash r := e \rightarrow U[v/r]}$
if-true	$\frac{\langle S, U, B \rangle \vdash e \rightarrow \text{true}, \langle S, U, B \rangle \vdash a \rightarrow U'}{\langle S, U, B \rangle \vdash \text{if } e \text{ then } a \rightarrow U'}$
if-false	$\frac{\langle S, U, B \rangle \vdash e \rightarrow \text{false}}{\langle S, U, B \rangle \vdash \text{if } e \text{ then } a \rightarrow U}$
a-when-true	$\frac{\langle S, U, B \rangle \vdash e \rightarrow \text{true}, \langle S, U, B \rangle \vdash a \rightarrow U'}{\langle S, U, B \rangle \vdash a \text{ when } e \rightarrow U'}$
par	$\frac{\langle S, U, B \rangle \vdash a_1 \rightarrow U_1, \langle S, U, B \rangle \vdash a_2 \rightarrow U_2}{\langle S, U, B \rangle \vdash a_1 \mid a_2 \rightarrow U_1 \uplus U_2}$
seq	$\frac{\langle S, U, B \rangle \vdash a_1 \rightarrow U_1; \langle S, U_1, B \rangle \vdash a_2 \rightarrow U_2}{\langle S, U, B \rangle \vdash a_1 ; a_2 \rightarrow U_2}$
a-let-sub	$\frac{\langle S, U, B \rangle \vdash e \rightarrow v, \langle S, U, B[v/t] \rangle \vdash a \rightarrow U'}{\langle S, U, B \rangle \vdash t = e \text{ in } a \rightarrow U'}$
a-meth-call	$\frac{\langle S, U, B \rangle \vdash e \rightarrow v, v \neq \text{NR}, m.g = \langle \lambda t.a \rangle, \langle S, U, B[v/t] \rangle \vdash a \rightarrow U'}{\langle S, U, B \rangle \vdash m.g(e) \rightarrow U'}$

Figure 10.4: Action Evaluation Rules (verbatim from [21, Figure 4]).

10.6 Action Evaluation

Equations for action evaluation continue along in a similar vein. First is “reg-update”.

	$\frac{\langle S, U, B \rangle \vdash e \rightarrow v, v \neq \text{NR}}{\langle S, U, B \rangle \vdash r := e \rightarrow U[v/r]}$
becomes	
ceq	$\langle S, U, B \rangle \vdash (P) R := E \rightarrow U[V / R]$
if	$\langle S, U, B \rangle \vdash (P) E \rightarrow U$
	$\wedge V \neq \text{NR} .$

Conditional actions are defined in both systems very similarly, just as were conditional expressions.

Merge Functions:

$$\begin{aligned}
U_1 \uplus U_2 &= \text{error if } \exists r. \{r \mapsto v_1\} \in U_1 \wedge \{r \mapsto v_2\} \in U_2 \\
&\quad \text{otherwise } U_1 \cup U_2 \\
\{\}(x) &= \perp \\
S[v/t](x) &= v \text{ if } t = x \\
&\quad \text{otherwise } S(x)
\end{aligned}$$

Each action rule gives a list of register updates given an environment $\langle S, U, B \rangle$ where S represents the register state, U is the observable updates, and B represents the local bindings. NR represents the “not-ready” value and can be stored in a binding, but *not* assigned to a register. The strictness of method calls is enforced by checking that parameter values are not NR. Initially U and B are empty and S contains the value of all registers. One can think of $++$ as list concatenation. If the system gets stuck because no rule is applicable, it is assumed that an empty U is returned.

Figure 10.5: Auxiliary Definitions (verbatim from [21, Figure 4]).

$$\frac{\langle S, U, B \rangle \vdash e \rightarrow \text{true}, \langle S, U, B \rangle \vdash a \rightarrow U'}{\langle S, U, B \rangle \vdash \text{if } e \text{ then } a \rightarrow U'}$$

$$\frac{\langle S, U, B \rangle \vdash e \rightarrow \text{false}}{\langle S, U, B \rangle \vdash \text{if } e \text{ then } a \rightarrow U}$$

becomes

```

ceq < S, U, B > |-(P) if E then A ->> = U'
  if True := < S, U, B > |-(P) E ->>
  /\ U'   := < S, U, B > |-(P) A ->> .
ceq < S, U, B > |-(P) if E then A ->> = U
  if False := < S, U, B > |-(P) E ->> .

```

Instead of propagating a special value when a guarded action’s condition is false, we simply omit an equation for this case. This meets the intention of the operational-style system, as described in Figure 10.5, which is taken directly from [21].

$$\frac{\langle S, U, B \rangle \vdash e \rightarrow \text{true}, \langle S, U, B \rangle \vdash a \rightarrow U'}{\langle S, U, B \rangle \vdash a \text{ when } e \rightarrow U'}$$

becomes

```

ceq < S, U, B > |-(P) A when E ->> = U'
  if True := < S, U, B > |-(P) E ->>
  /\ U'   := < S, U, B > |-(P) A ->> .

```

The equations defining the semantics of parallel composition require an additional function corresponding to the \uplus operation [21, Figure 4] (see Figure 10.5). Note that rewriting logic equations differ rather substantially from the original operational-style rules (see *errata*, Section 10.8)

$$\frac{\langle S, U, B \rangle \vdash a_1 \rightarrow U_1, \langle S, U, B \rangle \vdash a_2 \rightarrow U_2}{\langle S, U, B \rangle \vdash a_1 \mid a_2 \rightarrow (U_1 \uplus U_2)}$$

becomes (see *errata*)

```

ceq < S, U, B > |-(P) A1 | A2 ->> = U[U']
  if U1 := < S[U], empty, B > |-(P) A1 ->>
  /\ U2 := < S[U], empty, B > |-(P) A2 ->>
  /\ U' := U1 uplus U2 .

op _uplus_ : Store Store ~> Store .
ceq U1 uplus U2 = U1 , U2
  if not overlap(U1, U2) .

op overlap : Store Store -> Bool .
eq overlap(((R |-> V1) , U1), ((R |-> V2) , U2)) = true .
eq overlap(U1, U2) = false [owise] .

```

In contrast to the definition of parallel composition, the definition of sequential composition is straightforward and follows [21] directly.

$$\frac{\langle S, U, B \rangle \vdash a_1 \rightarrow U_1; \langle S, U_1, B \rangle \vdash a_2 \rightarrow U_2}{\langle S, U, B \rangle \vdash a_1 ; a_2 \rightarrow U_2}$$

becomes

```

ceq < S, U, B > |-(P) A1 ; A2 ->> = U2
  if U1 := < S, U , B > |-(P) A1 ->>
  /\ U2 := < S, U1, B > |-(P) A2 ->> .

```

Local bindings are handled exactly as for the expression case.

$$\frac{\langle S, U, B \rangle \vdash e \rightarrow v, \langle S, U, B[v/t] \rangle \vdash a \rightarrow U'}{\langle S, U, B \rangle \vdash t = e \text{ in } a \rightarrow U'}$$

becomes

```

ceq < S, U, B > |-(P) (T = E in A) ->> = U'
  if V := < S, U, B > |-(P) E ->>
  /\ U' := < S, U, B[V / T] > |-(P) A ->> .

```


The same is true for method invocation.

$$\frac{\langle S, U, B \rangle \vdash e \rightarrow v, \quad v \neq \text{NR}, \quad m.g = \langle \lambda t.a \rangle, \quad \langle S, U, B[v/t] \rangle \vdash a \rightarrow U'}{\langle S, U, B \rangle \vdash m.g(e) \rightarrow U'}$$

becomes

```
ceq < S, U, B > |-(P) M .. G(E) ->> = U'
  if (Module M ((ActMeth G \ T . A) CS)) P' := P
  /\ V := < S, U, B > |-(P) E ->>
  /\ V /= NR
  /\ U' := < S, U, B[V / T] > |-(P) A ->> .
```

10.7 Semantics

In addition to the equations defining expression and action evaluation, all that is needed for a complete semantics is a rewriting-logic-level rule driving the execution of a BTRS program. This is accomplished by using AC-matching to select *any rule* from the program, executing its body, and modifying the register state with any updates that result from executing it.

```
mod RULE-EXECUTION is extending SEMANTICS-SKELETON .
  ... variable declarations omitted
  var U' : Store .

  crl < P, S > => < P, S[U'] >
    if (Module M ((Rule R A) CS)) P' := P
    /\ U' := < S, empty, empty > |-(P) A ->> .

endm
```

One subtle aspect of the above rule is the handling of rules that are not enabled; for example, because the condition of a **when** embedded into the body of the rule evaluates to false. By declaring U' to be of *sort* **Store**, as opposed to its associated kind, we ensure that only rules that are enabled to execute actually do. Recall that the $|-$ operator is declared only to have kind **Store**, and so only takes sort **Store** as a “special” case.

Finally, the semantics of BTRS is just the aggregation of the above modules.

```

mod SEMANTICS is
  including SEMANTICS-SKELETON .
  including EVAL-EXPRESSION      .
  including EVAL-ACTION          .
  including RULE-EXECUTION       .
endm

```

10.8 Discussion

We briefly explain the two very minor *errata* in the SOS rules from [21], were noted above. The local binding rule for expressions exhibits a small typo where it is written $B[v/t]$; v is unbound. Therefore, the original rule

$$\frac{\langle S, U, B \rangle \vdash e_1 \rightarrow v_1, \langle S, U, B[v/t] \rangle \vdash e_2 \rightarrow v_2}{\langle S, U, B \rangle \vdash t = e_1 \text{ in } e_2 \rightarrow v_2}$$

should instead be

$$\frac{\langle S, U, B \rangle \vdash e_1 \rightarrow v_1, \langle S, U, B[v_1/t] \rangle \vdash e_2 \rightarrow v_2}{\langle S, U, B \rangle \vdash t = e_1 \text{ in } e_2 \rightarrow v_2}$$

The second *erratum* is slightly more significant. The rule for parallel composition (verbatim from [21])

$$\frac{\langle S, U, B \rangle \vdash a_1 \rightarrow U_1, \langle S, U, B \rangle \vdash a_2 \rightarrow U_2}{\langle S, U, B \rangle \vdash a_1 \mid a_2 \rightarrow (U_1 \uplus U_2)}$$

$$U_1 \uplus U_2 = \begin{cases} \text{error} & \text{if } \exists r. \{r \mapsto v_1\} \in U_1 \wedge \{r \mapsto v_2\} \in U_2 \\ U_1 \cup U_2 & \text{otherwise} \end{cases}$$

should only take into account collisions from register assignments that *occur during the evaluation of a_1 and a_2* , and not those that were already part of U . One way of fixing this is to recast the rule in a way similar to our specification in rewriting logic.

$$\frac{\langle S[U], \emptyset, B \rangle \vdash a_1 \rightarrow U_1, \langle S[U], \emptyset, B \rangle \vdash a_2 \rightarrow U_2, \text{Dom}(U_1) \cap \text{Dom}(U_2) = \emptyset}{\langle S, U, B \rangle \vdash a_1 \mid a_2 \rightarrow (U[U_1])[U_2]}$$

10.9 Example: A Deadlocking Completion Buffer

When used in conjunction with a rewriting logic engine such as Maude [15], the executable semantics serves not only as a correct-by-construction BTRS simulator (relative to the correctness of the given semantics in rewriting logic), but also as a multipurpose formal tool for analyzing BTRS programs. Indeed, Maude provides the ability to simulate BTRS programs, to prove and disprove invariants, to prove and disprove the existence of deadlock, and to perform full linear temporal logic (LTL) model checking, among other abilities (see Chapter 2). As an example utilizing these abilities in the context of BTRS, this section demonstrates a deadlock in a completion buffer.

A completion buffer is a structure that maintains ordering among operations whose results may be generated out of order. Our BTRS implementation is based on a completion buffer that was part of a Bluespec circuit for processing a stream of IP lookup requests; the latency of each request can vary based on how deep into a hierarchical set of tables one needs to look to process it. This deadlock that gets exposed was injected into the example knowingly and does not occur in the original device. Nevertheless, it is an instructive example.

Each operation begins by calling an action-method `getTokenAct` to reserve an entry in the buffer; when the operation completes its calculation, it puts the result into the buffer and signals its completion with a call to an action-method `done`. Results are taken from the buffer and space is ultimately freed by a call to an action-method `finishedAct`. The entire BTRS source is available at [45].

Corresponding value-methods `getTokenVal` and `finishedVal` return an `id` for the reserved buffer entry and the next operation's result, respectively. These are essentially the same set of functions from the original Bluespec source, aside from the splitting of `getToken` and `finished` into action and value parts.

First, we extend the syntax of BTRS with necessary identifiers, operator symbols, *etc.* In the module given below we omit all but the operator symbols and literals, as everything else will be clear from context.

```

mod CBUFFER-EXT is extending SEMANTICS .
  including NAT .

  ... identifiers omitted

  sort EntryStatus .
  ops Free Out Done : -> EntryState .
  subsort Nat EntryStatus < Literal .

  ops +mod8 == /= && : -> InfixOp .
  ops isOut isDone : -> PrefixOp .

  ... operator table definitions omitted

endm

```

Each entry in the completion buffer has, in addition to a data field, a current status which is characterized by the sort `EntryStatus`. The operator `+mod8` does 3-bit addition, `==` and `/=` are equality and inequality predicates for 3-bit values, and `&&` is logical-and on the sort `Boolean`. `isOut` and `isDone` are predicates on sort `EntryStatus`.

We will start with the register declarations and then move on to describe each action method individually. This is an 8-entry buffer having a data field and a status field for each entry, as well as head and tail pointers, `i` and `o`. `i` points to the next available entry and `o` points to the last entry freed.

```

mod CBUFFER is
  including CBUFFER-EXT .

  op CBUFFER : -> Module .
  eq CBUFFER =
    Module CBuffer
      (Register buff0 0)
      (Register buff1 0)
      (Register buff2 0)
      ... repeated to buff7
      (Register valid0 Free)
      (Register valid1 Free)
      (Register valid2 Free)
      ... repeated to valid7
      (Register i 1)
      (Register o 0)

    ... continues

```

A place in the completion buffer is reserved through a call to `getTokenAct`. The code for `getTokenVal` that returns the entry id is omitted.

```

(ActMeth getTokenAct \ dummy .
  (idx = (i +mod8 1)
    in (( (i := idx)
      | (if (i == 0) then (valid0 := Out))
      | (if (i == 1) then (valid1 := Out))
      | (if (i == 2) then (valid2 := Out))
      ...repeated to t == 7
    )
    when (idx /= o))
  ))

```

When a computation is finished, it places its result into the buffer entry it reserved, using the token as an argument. Note that for analysis purposes, since we are concerned with deadlock, we have used a data abstraction that places a fixed value into the buffer entry.

```
(ActMeth done \ t .
  (if (t == 0) then
    ((valid0 := Done) | (buff0 := 777)) when (isOut valid0)))
| (if (t == 1) then
  ((valid1 := Done) | (buff1 := 777)) when (isOut valid1)))
| (if (t == 2) then
  ((valid2 := Done) | (buff2 := 777)) when (isOut valid2)))
...repeated to t == 7
```

Results are pulled from the completion buffer by calling `finishedAct` and `finishedVal` (omitted).

```
(ActMeth finishedAct \ dummy .
  (idx = (o +mod8 1)
  in (( (o := idx)
    | (if (idx == 0) then ((valid0 := Free)
      when (isDone valid0)))
    | (if (idx == 1) then ((valid1 := Free)
      when (isDone valid1)))
    | (if (idx == 2) then ((valid2 := Free)
      when (isDone valid2)))
    ...repeated to idx == 7
  ) when (i /= o))))
```

To test for deadlock in this system, we construct a set of rules, one for each combination of method and possible method argument, and then ask Maude if a deadlocked term is reachable from the initial state.

```
mod TESTBENCH is including CBUFFER .
  op TESTBENCH : -> Module .
  eq TESTBENCH =
    Module Testbench
      (Rule r0 (CBuffer .. getTokenAct(0)))
      (Rule r1 (CBuffer .. finishedAct(0)))
      (Rule r2 (CBuffer .. done(0)))
      (Rule r3 (CBuffer .. done(1)))
      (Rule r4 (CBuffer .. done(2)))
      ...repeated to done(7)
    ... continues
```

The initial state is given by a term `initialC` defined as

```

... continued from above

op SIGMA : -> Store .
eq SIGMA =
    (buff0 |-> 0)
    , (valid0 |-> Free)
    , (buff1 |-> 0)
    , (valid1 |-> Free)
    , (buff2 |-> 0)
    , (valid2 |-> Free)
    ...repeated to buff7, valid7
    , (i |-> 1)
    , (o |-> 0) .

op initialC : -> Configuration .
eq initialC = < CBUFFER TESTBENCH, SIGMA > .
endm

```

We can run the following command in Maude [15, §6.4.3] to search for a terminated (deadlocked) state.

```
search [1] initialC =>! C:Configuration .
```

Subsequently, Maude notifies us of a deadlock in the buffer. The deadlock comes from an incorrect guard in the definition of the function `finishedAct`, (`i /= o`), which fails when the buffer is full. Depending on the particular usage pattern, this bug may or may not become manifest during use. That is, it may or may not be found during testing.

The buffer can be made correct, in the sense that the above deadlock is removed, by changing the guard to (`idx /= i`). If the above command is run with this guard, Maude instead reports the following

```

Maude> search [1] initialC =>! C:Configuration .
No solution.
states: 3833  rewrites: 10058295 in 7481ms cpu ...

```

indicating that no deadlock state can be reached, which is exactly what is desired.

CHAPTER 11

CONCLUSION

This dissertation addresses a specific problem in contemporary functional verification practice, namely the difficulty of automating the coverage closure feedback loop, and also contributes substantially to the rewriting logic semantics project [77, 19, 78]. The way in which we address the automation problem is through the design of a programming language where simulation becomes a first-class concept. As a result, verification engineers are able to write both general-purpose and specialized programs to discharge coverage goals, which is perhaps the most time-consuming and difficult part of functional verification.

In addition to identifying the problem and proposing a high-level solution in the form of a meta-language about simulation, we have advanced the idea with a set of additional contributions. Specifically, we formalized the language precisely within rewriting logic, implemented a tool that allows programs in the meta-language to be constructed and executed on a computer, demonstrated a broad set of novel capabilities made possible through the language, and we demonstrated applications of the language to more substantial devices, including a bus-master controller and a simple microprocessor. As a result, we were able to uncover a subtle timing bug in the multi-mastering capabilities of the bus-master controller, which had been tested previously with some rigor.

Regarding the rewriting logic semantics project, besides the above-mentioned formalization in it of our meta-language for functional verification, we have also formalized a substantial portion of Verilog, as well as formalized completely two much smaller languages, namely, production rule sets, which are used to design asynchronous circuits, and BTRS, which is a simplified version of the Bluespec hardware description language. In doing so, we were able to find bugs in a widely-used open-source Verilog simulator, clarify greatly the semantics of production rule sets, and, in the case of BTRS, demonstrate

once again the suitability of rewriting logic as semantic framework, as well as uncover a couple of small oversights in the original SOS specification.

Digital hardware design is an extremely complex and multi-faceted engineering process, and many challenges must still be addressed so that the current pace of innovation in digital electronics can continue into the future. This dissertation considers just one aspect of the process, carefully chosen however to address a problem that is especially serious with regards to posing an impediment to future progress. Our belief is that the best solutions must attempt to balance the abilities of the engineers involved with algorithms and other tools available to an engineer. That is why we have looked into the problem of how to design a more suitable programming language for orchestrating simulation-based functional verification. A related belief is that *formal semantics matters*, and can be key not only for traditional formal methods such as model checking or theorem proving, but also for testing-based functional verification of hardware, as our rewriting logic based approach to the metalanguage $[\mathcal{L}]ml$, and the semantics of specific HDL's has shown in practice.

Although this dissertation makes substantial progress, much work still must be done to provide more feature rich and efficient tool support and to integrate other aspects of the verification process when there is a thoughtful way of doing so. For example, a more sophisticated formal definition of what verification closure means and the kind of coverage that is relevant to hardware design is an open problem that is crucially important to simulation-based functional verification. A related issue is how to incorporate formal verification where appropriate, integrating the two approaches both in terms of tools and definition of verification closure.

Regarding the formal semantics work. The experimental results for production rule sets indicate that additional optimizations, based on abstractions perhaps, or other means must be developed to rein-in the state-space explosion problem. A formal-statistical approach may ultimately be the most appropriate, since such techniques are much more scalable. For the Verilog, BTRS, and even production rule sets, an important avenue of future work is also coming up with a general methodology through which symbolic simulation can be achieved in such a way that the resulting simulations are amenable to solving with modern SMT solver technologies.

APPENDIX A

IMPLEMENTATION DETAILS

The following subsections provide details on how the three functions `start`, `concretize`, and `simulate` from Chapter 5 are implemented. They reflect the vast majority of the implementation work in `vlogm1`. In total, the three functions represent approximately ten thousand lines of Haskell and C++ code, whereas the other operations that we will present are implemented in just a few lines of code.

A.1 `start`

Internally, the structure of the implementation of `start` is reminiscent of a compiler. Verilog is first parsed and then goes through a series of canonicalization and optimization passes until it is converted into a representation that can be easily simulated, specifically, a value of type `Simulation`.

The sequence of steps that are executed when `start` is called can be broken-down as follows.

- **parse:** This function reads Verilog source code from files on disk, parses the source code, and generates values of a first intermediate representation. In the actual implementation [46], the most difficult part is parsing, which is handled with source code from the Icarus Verilog simulator [101]. One small difficulty is that the parser populates C++ object instances, which must first be converted to C structs and subsequently marshalled over Haskell’s foreign function interface to generate values of Haskell data types.
- **pretty:** The first intermediate representation is rather inconvenient for processing purposes in Haskell, due to the different paradigms emphasized by C++ and Haskell. Therefore, this function converts

the first intermediate representation into one more suitable for use in Haskell. This second intermediate representation simply consists of the data types exported by `VlogMetaLang.Syntax`.

- **canonicalize**: This phase in the processing pipeline consists of a number of passes that operate over the second intermediate representation, transforming it in various ways that make later operations easier. As just one example, case statements are turned into a cascade of conditional statements. Some operations not exported by `VlogMetaLang.Syntax` and not part of Verilog are used during this phase, such as labels and `gotos`. Additional examples of transformations are elaborated on below.
- **codegen**: The data value resulting from the previous phase has essentially removed all uses of structured syntax from the Verilog program and has been partitioned into basic blocks. The final phase of the pipeline converts each block into the internal “instruction set” used during simulation, which is stack-based, and initializes a value of type `Simulation`, which is the final result of `start`.

Transformation Examples: Two examples of transformations performed during the `canonicalize` phase are presented. The first transformation converts port connections to continuous assignments. Consider a module `m` with interface

```
module m(i1, i2, o);
  input i1;
  input i2;
  output o;
```

and an instance of `m` in another module declared as follows:

```
m inst(x, y, z);
```

The canonicalization pass adds to the module containing this instance declaration the following three continuous assignments:

```
assign inst.i1 = x;
assign inst.i2 = y;
assign z = inst.o;
```

The second example simplifies structured delay statements, which are of the form `<delay> <statement>`, by pulling the statement body out and sequentially composing it with a simple delay having an empty statement body. The canonicalization phase contains a number of passes that effectively eliminate the structured nature of delay controls so that. Consider

```
#5 x = 0;
```

which, as indicated above, is syntactically of the form `<delay> <statement>`. The transformation under consideration converts the above statement into the following sequential block containing two statements:

```
begin
  #5;
  x = 0;
end
```

A.2 concretize

The `concretize` function corresponds to $rule_S$ from \mathcal{R}_{IR} . Therefore, its main purpose is to perform a substitution on the symbolic state of a simulation. In addition, however, `concretize` also performs constant folding and propagation, as opportunities for these simplifications are typically made possible after the substitution is made. An example is given next to clarify the process.

Consider, for example, simulating a Verilog device containing the following process:

```
always @(posedge clk)
  count <= count + 1;
```

which contains uses of two identifiers, `clock` and `count`. Values of type `Simulation` contain a set of mappings that determine the current value of each identifier and variable that is relevant to the simulation. Let us assume in this case that we are in possession of a simulation of this device where the current values of these identifiers are given by the following mappings to

expressions:

$$\begin{aligned}\text{clk} &\mapsto 1 \\ \text{count} &\mapsto x + 1\end{aligned}$$

where x is a symbolic variable. The simulation contains a separate mapping yielding the current value of each symbolic variable; in this case we assume that it has not yet been instantiated, meaning that it maps to itself.

$$x \mapsto x$$

If one uses `concretize` to apply a substitution σ to the above simulation, with $\sigma(x) = 0$, the first operation that is performed is to modify the mapping for symbolic variables referenced by σ . Therefore, the combined mappings get updated to

$$\begin{aligned}\text{clk} &\mapsto 1 \\ \text{count} &\mapsto x + 1 \\ x &\mapsto 0\end{aligned}$$

Constant propagation and folding are then applied structurally. In the case of the example, constant propagation first yields a combined set of mappings

$$\begin{aligned}\text{clk} &\mapsto 1 \\ \text{count} &\mapsto 0 + 1 \\ x &\mapsto 0\end{aligned}$$

and constant folding then simplifies the value of `count` to

$$\begin{aligned}\text{clk} &\mapsto 1 \\ \text{count} &\mapsto 1 \\ x &\mapsto 0\end{aligned}$$

A.3 simulate

`simulate` functions as a special kind of simulator for Verilog devices. Its type signature is,

```
simulate :: (Input a) => a -> Simulation -> Simulation
```

where `a` denotes any time which can be transformed into a proper input stimulus. The usual way in which a user interfaces with a simulator is quite different, roughly speaking the interface is

```
f :: Input -> Device -> Waveforms
```

Our notion of “simulation”, while containing waveform information, also contains enough information to continue simulation from where the waveforms leave off, and is therefore something altogether different.

Therefore, one of the main distinguishing features of `simulate` is that *simulations are treated as first class data values*. This allows a complete simulation run to be constructed piecewise, with different parts of the input decided at different times. The second main distinguishing feature of `simulate`, which is not evident from its type signature, is that it fully supports symbolic simulation, that is, input stimulus is allowed to contain symbolic variables.

To understand the implementation of `simulate` within `vlogml`, we walk through the main components of `simulate`’s implementation, which in many ways matches quite closely the conceptual description given in the Verilog Standard [36].

```
1  type Simulate a = State Simulation a
2  simulate_ :: Input -> Simulate ()
3  delta    ::          Simulate Bool
4  epsilon  ::          Simulate Bool
5  eval     :: Event -> Simulate ()
```

Figure A.1:

Overview. The implementation of `simulate` within `vlogml` will be described at the level of the functions named in Figure A.1.

```
1 simulate i  sim = evalState (simulate_ i') sim
2   where i' = toInput i
```

Figure A.2:

- **(Simulate a)**: This type synonym is an instance of Haskell’s (rather, GHC [95], the most widely used Haskell compiler) **State** monad [94] (see [98, 99, 100] for underlying concepts), where the backing state is a simulation. One of the advantages of using a state monad is that it provides for a clean implementation of computations that are intuitively “stateful”. In the case of **simulate**, a monadic implementation allows us to pass the simulation being modified without explicitly binding it as an argument for every function involved in simulation, of which there are many.
- **simulate_**: This is the main entry point for simulation internally. As shown in Figure A.4, **simulate** essentially just calls this function, with some additional code to resolve its type with the monadic type of **simulate_**. **evalState** is a standard function that unwraps state monads [94].
- **delta**: This function is called to perform a “delta cycle”, which is defined in **vlogml** as progressing simulation until, and including, the next point that internal simulation time is increased, causing the earliest scheduled events from the future stratum of the event queue to be promoted to zero-time events. In the terminology of the Verilog Standard [36], it corresponds to the combination of a “simulator cycle” (processing all zero-time events) and a clock update.
- **epsilon**: This function is called to perform an “epsilon cycle”, which does one of two things: evaluate all pending active events, or move pending zero-time events to active status in the event queue.
- **eval**: This function takes an event as an argument and evaluates the body of that event, which is a list of simple instructions, such as reading a current value of a source identifier, or performing an addition.

Queue Management. As described in the Verilog Standard [36], simulation proceeds by repeatedly processing events from a *stratified event queue* comprised of five strata named as follows: (1) active, (2) inactive, (3) non-blocking, (4) monitor, and (5) future. We now describe how this stratified queue is represented in `vlogm1` within a simulation and how the functions `simulate_`, `delta`, and `epsilon` are used to manage the queue in Standard-conforming manner. To do this, Figures A.3 – A.7 are presented in detail.

```

1  data ZeroTimeQueues = ZeroTimeQueues {
2      activeNotOrdered :: [Event]
3      , activeOrdered   :: [Event]
4      , inactive        :: [Event]
5      , nonblocking     :: [Event]
6  }
7
8  data Queues = Queues {
9      zeroTime :: ZeroTimeQueues
10     , future  :: (Map Int ZeroTimeQueues)
11 }

```

Figure A.3:

Figure A.3: Queues. These are the data types used to represent the stratified event queue defined in the Verilog Standard [36].

- (*lines 1 – 6*): These are the strata containing events that will execute before the next update of the clock; hence, the “zero-time” designation. Active events are the highest priority, followed by inactive events, followed by non-blocking assignment update events. `vlogm1` does not currently support monitors, which is why the record does not contain a field for that stratum. In addition, `vlogm1` partitions the active queue into ordered and unordered lists of events to help avoid subtle event ordering issues that arise when non-blocking events are made active.
- (*lines 8 – 11*): The entire event queue is then represented as a record containing the current value of the zero-time strata, and a mapping from integers, representing time in the future, to an updated zero-time queue in which events that occur in the future are placed.


```

1  simulate_ i = do
2    insertInput i
3    whileM delta

```

Figure A.4:

Figure A.4: `simulate_`

- (*line 2*): Inject the input stimulus, given as an argument, into the simulation instance. The injection process uses some of the same machinery as the `start` function to generate new code blocks from the input and then propagate events that reference these newly generated blocks into the event queues.
- (*line 3*): Continually execute `delta` cycles until the input stimulus is consumed. `delta` returns a boolean, and `whileM` is function we have defined that executes the given function until it returns false.

```

1  delta = do
2    whileM epsilon
3    x <- clockTick
4    case x of
5      (Just t) -> return True
6      Nothing  -> return False

```

Figure A.5:

Figure A.5: `delta`

- (*line 2*): Continually execute `epsilon` cycles until there are no remaining zero-time events.
- (*line 3*): If there are future events, update the simulation clock to the time of the earliest future event and substitute its zero-time events for the current, empty, zero-time events; decrement the time values stored in the future stratum. In this case, the amount of time added to the

simulation clock is returned. If the stimulus has been used up, the value `Nothing` is returned; this is generated from a special flag used internally.

- (*line 4 – 6*): If there continue to be events that may be executed and the stimulus has not been used up, which would set the internal flag mentioned above, return `True`, otherwise return `False`, indicating that the stimulus has been used up entirely.

```
1  epsilon = do
2    xs <- getActiveEvents
3    case xs of
4      x:xs -> eval x >> return True
5      ( []) -> updateZeroTimeQueues
```

Figure A.6:

Figure A.6: `epsilon`

- (*line 2*): Retrieve a list of all active events. This is the concatenation of the ordered and unordered lists.
- (*line 4*): If there is at least one pending active event, evaluate it. The `eval` function is explained below.
- (*line 5*): If there are no pending active events, management of the zero-time queues is invoked. The management process is dictated by the Verilog Standard [36] and is described next.

Figure A.7: `updateZeroTimeQueues`

- (*line 2*): Get the list of inactive events and generate a boolean indicating if the list is non-empty. The result of this check is bound to the variable `s0`.
- (*line 3*): Get the list of non-blocking events and generate a boolean indicating if the list is non-empty. The result of this check is bound to the variable `s1`.

```

1  updateZeroTimeQueues = do
2    s0 <- notNull <$>   getInactiveEvents
3    s1 <- notNull <$> getNonblockingEvents
4    cast (s0,s1) of
5      (True ,   _) -> do
6        activateInactiveEvents
7        return True
8      (False,True ) -> do
9        activateNonblockingEvents
10       return True
11      (False,False) -> return False

```

Figure A.7:

- (*lines 5 – 7*): There are pending inactive events, promote all of them to active status and return `True`, indicating that there are still zero-time events to be processed.
- (*lines 8 – 10*): There are no pending inactive events, but there *are* pending non-blocking assignment update events; promote all of them to active status and return `True`.
- (*line 11*): There are no pending inactive events and no pending non-blocking events. Therefore, there are no pending zero-time events at all and we return `False` to indicate this fact, which is used above in `delta` to initiate the clock tick procedure.

Event Evaluation. The purpose of what is described above is as a structure to hold pending events and to maintain the appropriate ordering constraints among them. Evaluation of an event is handled through the `eval` function. To describe how this function is implemented, a few additional data types are needed; a few of these are simplified for the following discussion to just the most essential information.

Figure A.8

- **Instruction:** This data type represents the internal instruction set of simulation used within `vlogml` and is what is primarily emitted during

```
1  data Instruction
2  data Target
3  data TargetId
4  data Event
```

Figure A.8:

the code generation phase of `start`. It is a stack-based instruction set, and includes various operations, a few of which are shown in Figure A.9 and elaborated on below.

```
1  data Instruction =
2      InstValue Expression
3      | InstAdd
4      | InstGoto TargetId
5      | ... -- additional instructions
```

Figure A.9:

- **Target** and **TargetId**: For our purposes here, a target can be considered just a labelled list of instructions; that is, it contains a value of type `[Instruction]`. Each value of type `Simulation` contains a mapping from “target identifiers”, given by the type `TargetId`, to targets.
- **Event**: For our purposes here, an event can be considered as just a target identifier which provides a location from which to start executing instructions.

```
1  eval    ::      Event -> Simulate ()
2  execute :: [Instruction] -> Simulate ()
```

Figure A.10:

Figure A.10: eval and execute

- `eval`: This function does some internal setup for symbolic simulation and other things that are not described here; however, its main purpose is to use the target identifier of the given event to look up the associated target and call `execute` to process the associated instructions.
- `execute`: This function interprets the internal simulation instruction set. A few examples of how instructions are processed are given next.

Execution Examples. The terminal case of `execute` is given by the empty list of instructions, from which we simply return `unit`.

```
execute [] = return ()
```

In addition, there is a separate case for each instruction type. We next explain the implementation of the instructions shown above in Figure A.9. The implementation assumes monadic stack operations `push` and `pop`, a function that performs Verilog-compliant addition, as well as a function to look up targets, `lookupTarget`.

Immediate values are processed according to the following definition

```
execute ((InstValue x):xs) = do
  push x
  execute xs
```

Addition pops two values from the top of the stack, yielding expressions `x1` and `x2`, performs addition, and then pushes the result onto the stack.

```
execute ((InstAdd):xs) = do
  x1 <- pop
  x2 <- pop
  push (vlogAdd x1 x2)
  execute xs
```

The implementation of `vlogAdd` is slightly involved. As `x1` and `x2` are possibly symbolic expressions, a new expression is first created representing the addition; this expression is then subjected to constant folding to simplify the result.

For an unconditional branch, or “goto”, the target associated is looked up and the instruction stream is redirected immediately.

```
execute ((InstGoto j):xs) = do
  ys <- lookupTarget j
  execute ys
```

REFERENCES

- [1] Allon Adir, Eli Almog, Laurent Fournier, Eitan Marcus, Michal Rimon, Michael Vinov, and Avi Ziv. Genesys-Pro: Innovations in Test Program Generation for Functional Processor Verification. *IEEE Design & Test of Computers*, 21(2):84–93, 2004.
- [2] Gul A. Agha, José Meseguer, and Koushik Sen. PMAude: Rewrite-based Specification Language for Probabilistic Object Systems. *Electronic Notes in Theoretical Computer Science*, 153(2):213–239, 2006.
- [3] Musab AlTurki and José Meseguer. PVeStA: A Parallel Statistical Model Checking and Quantitative Analysis Tool. In *4th International Conference on Algebra and Coalgebra in Computer Science (CALCO 2011)*, to appear, 2011.
- [4] Peter A. Beerel, Jerry R. Burch, and Teresa H.-Y. Meng. Sufficient Conditions for Correct Gate-Level Speed-Independent Circuits. In *Proceedings of the International Symposium on Advanced Research in Asynchronous Circuits and Systems*, pages 33–43, 1994.
- [5] Janick Bergeron. *Writing Testbenches: Functional Verification of HDL Models*. Springer, second edition, 2003.
- [6] Janick Bergeron. *Writing Testbenches Using SystemVerilog*. Springer, 2006.
- [7] Janick Bergeron. Automating Coverage Closure. Verification Martial Arts: A Verification Methodology Blog, 2010. <http://www.vmmcentral.org/vmartialarts/2010/07/automating-coverage-closure>.
- [8] Bluespec, Inc., Waltham, Massachusetts. *Bluespec SystemVerilog Version Reference Guide*, October 2009.
- [9] Tom Borgstrom. Getting the last 20%. On Verification: A Software to Silicon Verification Blog, 2010. <http://synopsysoc.org/softwaretosiliconverification/2010/07/getting-the-last-20>.

- [10] Roberto Bruni and José Meseguer. Generalized Rewrite Theories. In *30th International Colloquium on Automata, Languages and Programming (ICALP 2003)*, volume 2719 of *Lecture Notes in Computer Science*, pages 252–266. Springer, 2003.
- [11] Roberto Bruni and José Meseguer. Semantic foundations for generalized rewrite theories. *Theoretical Computer Science*, 360(1-3):386–414, 2006.
- [12] J.R. Burch, E.M. Clarke, D.E. Long, K.L. McMillan, and D.L. Dill. Symbolic model checking for sequential circuit verification. *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, 13(4):401–424, April 1994.
- [13] Pankaj Chauhan, Deepak Goyal, Gagan Hasteer, Anmol Mathur, and Nikhil Sharma. Non-cycle-accurate sequential equivalence checking. In *Proceedings of the 46th Design Automation Conference (DAC 2009)*, pages 460–465. ACM, 2009.
- [14] Hana Chockler, Orna Kupferman, and Moshe Y. Vardi. Coverage metrics for formal verification. *STTT*, 8(4-5):373–386, 2006.
- [15] Manuel Clavel, Francisco Durán, Steven Eker, Patrick Lincoln, Narciso Martí-Oliet, José Meseguer, and Carolyn L. Talcott. *All About Maude - A High-Performance Logical Framework*, volume 4350 of *Lecture Notes in Computer Science*. Springer, 2007.
- [16] Manuel Clavel and José Meseguer. Reflection in conditional rewriting logic. *Theoretical Computer Science*, 285(2):245–288, 2002.
- [17] Manuel Clavel, José Meseguer, and Miguel Palomino. Reflection in membership equational logic, many-sorted equational logic, Horn logic with equality, and rewriting logic. *Theoretical Computer Science*, 373(1-2):70 – 91, 2007.
- [18] James N. Cook. Production Rule Verification for Quasi-Delay-Insensitive Circuits. Master’s thesis, California Institute of Technology, 1993.
- [19] Traian Florin Șerbănuță, Grigore Roșu, and José Meseguer. A rewriting logic approach to operational semantics. *Information and Computation*, 207(2):305–340, 2009.
- [20] Marcelo d’Amorim and Grigore Roșu. An Equational Specification for the Scheme Language. *Journal of Universal Computer Science*, 11(7):1327–1348, 2005.
- [21] Nirav Dave, Arvind, and Michael Pellauer. Scheduling as Rule Composition. In *5th ACM/IEEE International Conference on Formal Methods and Models for Codesign (MEMOCODE 2007)*, pages 51–60. IEEE Computer Society, June 2007.

- [22] Edsger W. Dijkstra. Guarded commands, non-determinacy and a calculus for the derivation of programs. In Friedrich L. Bauer and Klaus Samelson, editors, *Language Hierarchies and Interfaces*, volume 46 of *Lecture Notes in Computer Science*, pages 111–124. Springer, 1976.
- [23] Simon Peyton Jones (Editor). Haskell 98 Language and Libraries: The Revised Report, 2002.
- [24] Chucky Ellison and Grigore Roşu. A Formal Semantics of C with Applications. Technical Report <http://hdl.handle.net/2142/17414>, University of Illinois at Urbana-Champaign, November 2010.
- [25] Richard Herveille *et. al.* *Wishbone B4: WISHBONE System-on-Chip (SoC) Interconnection Architecture for Portable IP Cores*, (Revision B.4). OpenCores Organization, 2010.
- [26] Azadeh Farzan, Feng Chen, José Meseguer, and Grigore Roşu. Formal Analysis of Java Programs in JavaFAN. In Rajeev Alur and Doron Peled, editors, *16th International Conference on Computer Aided Verification (CAV 2004)*, volume 3114 of *Lecture Notes in Computer Science*, pages 501–505. Springer, 2004.
- [27] Vijay Ganesh and David L. Dill. A Decision Procedure for Bit-Vectors and Arrays. In Werner Damm and Holger Hermanns, editors, *19th International Conference on Computer Aided Verification (CAV 2007)*, volume 4590 of *Lecture Notes in Computer Science*, pages 519–531. Springer-Verlag Berlin Heidelberg, 2007.
- [28] Michael J. C. Gordon. The Semantic Challenge of Verilog HDL. In *Proceedings of the Tenth Annual IEEE Symposium on Logic in Computer Science (LICS-95)*, pages 136–145. IEEE Computer Society, 1995.
- [29] Michael J. C. Gordon, Robin Milner, Lockwood Morris, Malcolm C. Newey, and Christopher P. Wadsworth. A Metalanguage for Interactive Proof in LCF. In *Fifth Annual ACM Symposium on Principles of Programming Languages (POPL 1978)*, pages 119–130, January 1978.
- [30] Michael J. C. Gordon, Robin Milner, and Christopher P. Wadsworth. *Edinburgh LCF*, volume 78 of *Lecture Notes in Computer Science*. Springer, 1979.
- [31] Mike Gordon. The Semantic Challenge of Verilog HDL. In *Tenth Annual IEEE Symposium on Logic in Computer Science (LICS 1995)*, pages 136–145. IEEE Computer Society, 1995.
- [32] Jim Grundy, Thomas F. Melham, and John W. O’Leary. A reflective functional language for hardware design and theorem proving. *Journal of Functional Programming*, 16(2):157–196, 2006.

- [33] Richard Herveille. I2C Controller Core. <http://opencores.org/project,i2c>.
- [34] Pei-Hsin Ho, Thomas R. Shiple, Kevin Harer, James H. Kukula, Robert F. Damiano, Valeria Bertacco, Jerry Taylor, and Jiang Long. Smart Simulation Using Collaborative Formal and Simulation Engines. In Ellen Sentovich, editor, *Proceedings of the 2000 IEEE/ACM International Conference on Computer-Aided Design (ICCAD 2000)*, pages 120–126. IEEE, 2000.
- [35] C. A. R. Hoare. Communicating sequential processes. *Communications of the ACM*, 21(8):666–677, 1978.
- [36] Institute for Electrical and Electronics Engineers (IEEE). *1364-2005 IEEE Standard for Verilog Hardware Description Language*, 2006.
- [37] Institute for Electrical and Electronics Engineers (IEEE). *1076-2008 IEEE Standard VHDL Language Reference Manual*, January 2009.
- [38] Institute for Electrical and Electronics Engineers (IEEE). *1800-2009 IEEE Standard for SystemVerilog – Unified Hardware Design, Specification, and Verification Language*, 2009.
- [39] Institute for Electrical and Electronics Engineers (IEEE). *1647-2011 IEEE Standard for the Functional Verification Language e*, August 2011.
- [40] International Organization for Standardization, Geneva, Switzerland. *ISO/IEC 14882:2011 Information Technology – Programming Languages – C++*, September 2011.
- [41] International Technology Roadmap for Semiconductors. <http://www.itrs.net/>, 2009.
- [42] Wonjin Jang and Alain J. Martin. A Soft-error-tolerant Asynchronous Microcontroller. In *13th NASA Symposium on VLSI Design*, 2007.
- [43] Gilles Kahn. Natural Semantics. In Franz-Josef Brandenburg, Guy Vidal-Naquet, and Martin Wirsing, editors, *4th Annual Symposium on Theoretical Aspects of Computer Science (STACS 1987)*, volume 247 of *Lecture Notes in Computer Science*, pages 22–39. Springer, 1987.
- [44] Roope Kaivola, Rajnish Ghughal, Naren Narasimhan, Amber Telfer, Jesse Whittemore, Sudhindra Pandav, Anna Slobodová, Christopher Taylor, Vladimir Frolov, Erik Reeber, and Armaghan Naik. Replacing Testing with Formal Verification in Intel Core i7 Processor Execution Engine Validation. In *21st International Conference on Computer Aided Verification (CAV 2009)*, volume 5643 of *Lecture Notes in Computer Science*, pages 414–429. Springer, 2009.

- [45] Michael Katelman. An executable formal semantics of BTRS. <https://www.ideals.illinois.edu/handle/2142/28349>, 2011.
- [46] Michael Katelman. Source Code for vlogm1 (July 2011). www.illinois.edu, July 2011.
- [47] Michael Katelman, Sean Keller, and José Meseguer. Concurrent Rewriting Semantics and Analysis of Asynchronous Digital Circuits. In Peter Csaba Ölveczky, editor, *8th International Workshop on Rewriting Logic and Its Applications (WRLA 2010)*, volume 6381 of *Lecture Notes in Computer Science*, pages 140–156. Springer-Verlag Berlin Heidelberg, 2010.
- [48] Michael Katelman, Sean Keller, and José Meseguer. Concurrent Rewriting Semantics and Analysis of Asynchronous Digital Circuits. In *8th International Workshop on Rewriting Logic and its Applications (WRLA '10); to appear*, 2010.
- [49] Michael Katelman, Sean Keller, and José Meseguer. Rewriting Semantics of Production Rule Sets. *Journal of Logic and Algebraic Programming (To Appear)*, 2011.
- [50] Michael Katelman, Sean Keller, and José Meseguer. Source Code for an Executable Formal Semantics of Production Rule Sets in Maude, with Examples (JLAP version, November 2011). <https://www.ideals.illinois.edu/handle/2142/28350>, 2011.
- [51] Michael Katelman and José Meseguer. A Strategy Language for Testing Register Transfer Level Logic. Technical Report 12003, Department Of Computer Science, University of Illinois at Urbana-Champaign, 2009. <http://hdl.handle.net/2142/12003>.
- [52] Michael Katelman and José Meseguer. vlogsl: A Strategy Language for Simulation-Based Verification of Hardware. In Sharon Barner, Ian Harris, Daniel Kroening, and Orna Raz, editors, *Hardware and Software: Verification and Testing - 6th International Haifa Verification Conference (HVC 2010)*, volume 6504 of *Lecture Notes in Computer Science*, pages 129 – 145. Springer Berlin / Heidelberg, 2011.
- [53] Michael Katelman, José Meseguer, and Santiago Escobar. Directed-Logical Testing for Functional Verification of Microprocessors. In *6th ACM & IEEE International Conference on Formal Methods and Models for Codesign (MEMOCODE 2008)*, pages 89–100. IEEE Computer Society, 2008.
- [54] Matt Kaufmann, Panagiotis Manolios, and J. Strother Moore. *Computer-Aided Reasoning: An Approach*. Kluwer Academic Publishers, 2000.

- [55] Sean Keller, Michael Katelman, and Alain J. Martin. A Necessary and Sufficient Timing Assumption for Speed-Independent Circuits. In *15th IEEE Symposium on Asynchronous Circuits and Systems (ASYNC 2009)*, pages 65–76. IEEE, May 2009.
- [56] Sean Keller, Michael Katelman, and Alain J. Martin. A Necessary and Sufficient Timing Assumption for Speed-Independent Circuits. In *15th IEEE Symposium on Asynchronous Circuits and Systems (ASYNC '09)*, pages 65–76, May 2009.
- [57] V. Khomenko, M. Koutny, and A. Yakovlev. Logic synthesis for asynchronous circuits based on petri net unfoldings and incremental sat. In *Application of Concurrency to System Design, 2004. ACSD 2004. Proceedings. Fourth International Conference on*, pages 16 – 25, June 2004.
- [58] Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Videira Lopes, Jean-Marc Loingtier, and John Irwin. Aspect-Oriented Programming. In Mehmet Aksit and Satoshi Matsuoka, editors, *Proceedings of the 11th European Conference on Object-Oriented Programming (ECOOP 1997)*, volume 1241 of *Lecture Notes in Computer Science*, pages 220–242. Springer Berlin / Heidelberg, 1997.
- [59] James C. King. A New Approach to Program Testing. In *Programming Methodology, 4th Informatik Symposium*, volume 23 of *Lecture Notes in Computer Science*, pages 278–290. Springer, 1975.
- [60] James C. King. Symbolic Execution and Program Testing. *Communications of the ACM*, 19(7):385–394, 1976.
- [61] Nathan Kitchen and Andreas Kuehlmann. Stimulus Generation for Constrained Random Simulation. In Georges G. E. Gielen, editor, *2007 International Conference on Computer-Aided Design (ICCAD'2007)*, pages 258–265. IEEE, 2007.
- [62] Geoffrey Mainland. Why It’s Nice to be Quoted: Quasiquoting for Haskell. In Gabriele Keller, editor, *Proceedings of the ACM SIGPLAN Workshop on Haskell, (Haskell 2007)*, pages 73–82, 2007.
- [63] Rupak Majumdar and Koushik Sen. Hybrid Concolic Testing. In *ICSE*, pages 416–426. IEEE Computer Society, 2007.
- [64] Rajit Manohar and Alain J. Martin. Quasi-delay-insensitive circuits are Turing-complete. Technical Report CS-TR-95-11, Computer Science Department, California Institute of Technology, 1995.
- [65] Alain J. Martin. Compiling Communicating Processes Into Delay-Insensitive VLSI Circuits. *Distributed Computing*, 1(4):226–234, 1986.

- [66] Alain J. Martin. The limitations to delay-insensitivity in asynchronous circuits. In *Proceedings of the sixth MIT conference on Advanced research in VLSI*, pages 263–278, Cambridge, MA, USA, 1990. MIT Press.
- [67] Alain J. Martin. Synthesis of Asynchronous VLSI Circuits. Technical Report CS-TR-93-28, Computer Science Department, California Institute of Technology, 1991.
- [68] Alain J. Martin and Mika Nyström. Asynchronous Techniques for System-on-Chip Design. *Proceedings of the IEEE*, 94(6):1089–1120, 2006.
- [69] Alain J. Martin, Mika Nyström, and Catherine G. Wong. Three Generations of Asynchronous Microprocessors. *IEEE Design & Test of Computers*, 20(6):9–17, 2003.
- [70] Alain J. Martin and Piyush Prakash. Asynchronous Nano-Electronics: Preliminary Investigation. In *Proceedings of the 2008 14th IEEE International Symposium on Asynchronous Circuits and Systems*, pages 58–68. IEEE Computer Society, 2008.
- [71] Patrick Meredith, Mark Hills, and Grigore Roşu. An Executable Rewriting Logic Semantics of K-Scheme. In Danny Dube, editor, *Workshop on Scheme and Functional Programming (SCHEME 2007)*, Technical Report DIUL-RT-0701, pages 91–103. Laval University, 2007.
- [72] Patrick Meredith, Michael Katelman, José Meeguer, and Grigore Roşu. Formal executable semantics of verilog webpage, 2010. http://fsl.cs.uiuc.edu/index.php/Verilog_Semantics.
- [73] Patrick O’Neil Meredith, Michael Katelman, José Meseguer, and Grigore Roşu. A Formal Executable Semantics of Verilog. In *8th ACM/IEEE International Conference on Formal Methods and Models for Codesign (MEMOCODE 2010)*, pages 179–188. IEEE Computer Society, 2010.
- [74] José Meseguer. Conditional rewriting logic as a unified model of concurrency. *Theoretical Computer Science*, 96(1):73–155, 1992.
- [75] José Meseguer. Rewriting Logic as a Semantic Framework for Concurrency: a Progress Report. In Ugo Montanari and Vladimiro Sassone, editors, *7th International Conference on Concurrency Theory (CONCUR 1996)*, volume 1119 of *Lecture Notes in Computer Science*, pages 331–372. Springer, 1996.
- [76] José Meseguer. Membership Algebra as a Logical Framework for Equational Specification. In *12th International Workshop on Recent Trends in Algebraic Development Techniques (WADT’97)*, volume 1376 of *Lecture Notes in Computer Science*, pages 18–61. Springer, 1997.

- [77] José Meseguer and Grigore Roşu. The rewriting logic semantics project. *Theoretical Computer Science*, 373(3):213–237, 2007.
- [78] José Meseguer and Grigore Roşu. The Rewriting Logic Semantics Project: A Progress Report. In Olaf Owe, Martin Steffen, and Jan Arne Telle, editors, *Proceedings of the 18th International Symposium on Fundamentals of Computation Theory (FCT 2011)*, volume 6914 of *Lecture Notes in Computer Science*, pages 1–37. Springer, August 2011.
- [79] R E Miller. *Switching Theory, Volume II: Sequential Circuits and Machines*. John Wiley & Sons, Inc., 1965.
- [80] B. Mishra and E. Clarke. Hierarchical verification of asynchronous circuits using temporal logic. *Theoretical Computer Science*, 38:269 – 291, 1985.
- [81] David E. Muller and W. S. Bartky. A theory of asynchronous circuits. In *Proceedings of an International Symposium on the Theory of Switching*, pages 204–243. Harvard University Press, 1959.
- [82] NXP Semiconductors. *P C-bus specification and user manual*, revision 03 edition, June 2007.
- [83] Florent Ouchet, Dominique Borrione, Katell Morin-Allory, and Laurence Pierre. High-level symbolic simulation for automatic model extraction. In *Proceedings of the 2009 IEEE Symposium on Design and Diagnostics of Electronic Circuits and Systems (DDECS 2009)*, pages 218–221. IEEE Computer Society, 2009.
- [84] Gordon J. Pace and Jifeng He. Formal reasoning with Verilog HDL. In *Workshop on Formal Techniques for Hardware and Hardware-like Systems*, 1998.
- [85] Karl Papadantonakis. Design Rules for Non-Atomic Implementation of PRS. Technical Report CaltechCSTR:2005.001, California Institute of Technology, 2005.
- [86] Jaehong Park, Carl Pixley, Michael Burns, and Hyunwoo Cho. An Efficient Logic Equivalence Checker for Industrial Circuits. *Journal of Electronic Testing*, 16(1-2):91–106, 2000.
- [87] Python Software Foundation. *Python Programming Language – Official Website*. www.python.org.
- [88] Oriol Roig, Jordi Cortadella, and Enric Pastor. Verification of Asynchronous Circuits by BDD-based Model Checking of Petri Nets. In *In 16th International Conference on Application and Theory of Petri Nets*, volume 935 of *Lecture Notes in Computer Science*, pages 374–391. Springer-Verlag, 1996.

- [89] David M. Russinoff. A Case Study in Formal Verification of Register-Transfer Logic with ACL2: The Floating Point Adder of the AMD Athlon™ Processor. In Warren A. Hunt Jr. and Steven D. Johnson, editors, *Third International Conference on Formal Methods in Computer-Aided Design (FMCAD 2000)*, volume 1954 of *Lecture Notes in Computer Science*, pages 3–36. Springer, 2000.
- [90] Hisashi Sasaki. A formal semantics for verilog-vhdl simulation interoperability by abstract state machine. In *Design, automation and test in Europe (DATE'99)*, pages 73–78. ACM, 1999.
- [91] Synopsys, Inc. *OpenVera Language Reference Manual: Testbench*, Version 1.4.3, September 2005.
- [92] Synopsys, Inc. *Magellan Reference Guide*, Version C-2009.09, September 2009. (Proprietary Documentation).
- [93] Synopsys, Inc. *Magellan User Guide*, Version C-2009.09, September 2009. (Proprietary Documentation).
- [94] The GHC Team. *The Glorious Glasgow Haskell Compilation System Standard Libraries, Version 7.2.1*, August 2011. <http://www.haskell.org/ghc/docs/7.2.1/html/libraries/index.html>.
- [95] The GHC Team. *The Glorious Glasgow Haskell Compilation System User's Guide, Version 7.2.1*, August 2011. <http://www.haskell.org/ghc>.
- [96] Sarah Thompson and Alan Mycroft. Abstract interpretation of combinational asynchronous circuits. *Science of Computer Programming*, 64(1):166 – 183, 2007. Special issue on the 11th Static Analysis Symposium - SAS 2004.
- [97] Shobha Vasudevan, David Sheridan, Sanjay J. Patel, David Tcheng, William Tuohy, and Daniel R. Johnson. GoldMine: Automatic Assertion Generation Using Data Mining and Static Analysis. In *Design, Automation and Test in Europe*, pages 626–629, 2010.
- [98] Philip Wadler. Comprehending Monads. In *LISP and Functional Programming*, pages 61–78, 1990.
- [99] Philip Wadler. The Essence of Functional Programming. In *Principles of Programming Languages (POPL 1992)*, pages 1–14, 1992.
- [100] Philip Wadler. Monads for Functional Programming. In Johan Jeuring and Erik Meijer, editors, *First International Spring School on Advanced Functional Programming Techniques*, volume 925 of *Lecture Notes in Computer Science*, pages 24–52. Springer, 1995.

- [101] Stephen Williams. *Icarus Verilog User Guide*. http://iverilog.wikia.com/wiki/User_Guide.
- [102] Hüsniü Yenigün, Vladimir Levin, Doron Peled, and Peter A. Beerel. Hazard-Freedom Checking in Speed-Independent Systems. In *CHARME*, pages 317–320, 1999.
- [103] Hao Zheng, E. Mercer, and C. Myers. Modular verification of timed circuits using automatic abstraction. *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, 22(9):1138 – 1153, sept. 2003.
- [104] Hao Zheng, Haiqiong Yao, and T. Yoneda. Modular model checking of large asynchronous designs with efficient abstraction refinement. *Computers, IEEE Transactions on*, 59(4):561 –573, April 2010.
- [105] Huibiao Zhu, Jifeng He, and Jonathan P. Bowen. From algebraic semantics to denotational semantics for verilog. In *International Conference on Engineering Complex Computer Systems (ICECCS'06)*, pages 139–151, 2006.