Specification and verification of sequential machines in rule-based hardware languages

by

Thomas Bourgeat

B.S., École Normale Supérieure (2013)
M.S, Université Paris Diderot (2015)

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

Doctor of Philosophy

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

February 2023

© Massachusetts Institute of Technology 2023. All rights reserved.

Author . . . .
Department of Electrical Engineering and Computer Science
December 29, 2022

Certified by . . . .
Arvind
Johnson Professor of Computer Science and Engineering
Thesis Supervisor

Certified by . . . .
Adam Chlipala
Professor of Computer Science
Thesis Supervisor

Accepted by . . . .
Leslie A. Kolodziejski
Professor of Electrical Engineering and Computer Science
Chair, Department Committee on Graduate Students

Specification and verification of sequential machines in rule-based hardware languages

by

Thomas Bourgeat

Submitted to the Department of Electrical Engineering and Computer Science
on December 29, 2022, in partial fulfillment of the
requirements for the degree of
Doctor of Philosophy

Abstract

The design of correct hardware is an important concern in the age of information, where more and more companies are designing chips tailored to their workloads. This raises two well-known problems: how to specify what is a correct design, and, once the notion of correctness is set, how to prove that a given design is correct.

Standard practice relies on a mix of techniques. It uses testing to run concrete scenarios to verify a concrete property: "this specific test passed," but this gives only weak overall correctness guarantees. The other technique is to use hardware formal verification, which phrases correctness as custom temporal-logic formulae and checks that a concrete design verifies those properties by solving a large set of corresponding Boolean equations.

This thesis addresses the hardware-verification question from a different angle: we intend to mechanically formalize the specifications and correctness arguments that architects make in their minds when they design machines, and we encode them in an interactive theorem prover. For this, we address three challenges: (1) We build an expressive framework in which we can express both synthesizable designs and abstract specifications, and we connect and navigate between them in the proof assistant. (2) We enforce several strict language restrictions, allowing us to side-step previous difficulties in specifications and proofs. (3) After acknowledging that we need a modular methodology to keep the verification effort under control, we showcase previously ignored difficulties in specifying complex sequential machines modularly. We introduce *generalized* specifications and develop various proof techniques to prove that concrete designs are instances of their modular specifications. We finally apply our methodology to prove modularly the correctness of a family of pipelined processors, independently of its memory.

Thesis Supervisor: Arvind
Title: Johnson Professor of Computer Science and Engineering

Thesis Supervisor: Adam Chlipala
Title: Professor of Computer Science

Acknowledgments

Foremost, I thank my advisors, Professor Arvind and Professor Adam Chlipala. They let me explore many exciting projects and taught me many things during those years. I also thank Professor Srini Devadas for accepting to be on my thesis committee and for teaching me so much about computer architecture security.

This dissertation has been deeply influenced by dear friends. First, I thank Andy Wright who taught me most of what I know about rule-based designs. Many ideas formalized and presented in this dissertation started as open discussions with Andy, and some ideas are explicitly adapted from his PhD thesis. Then, I thank Clément Pit-Claudel, who is to be credited not only for the work we did together on cycle-accurate semantics but also for teaching me various ways to formalize modular semantics in Coq. Finally, I thank Samuel Gruetter with whom I explored using e-graphs for building proof automation.

During my PhD, I also had the pleasure to work with other research groups. I had great pleasure working with Professor Mengjia Yan, Professor Joel Emer, Professor Srini Devadas, and Professor Martin Rinard. They gave me other perspectives on research, giving me new research experiences that were both very valuable and enjoyable. Besides research, being part of the TA crew for Silvina Hanono Wachman, Daniel Sanchez and Arvind (6.s084) was also a great and formative experience.

I also thank the friends and collaborators from the lab: Ana Arduengo, Andres Erbsen, Áron Ricardo Perez-Lopez, Brian Plancher, Chanwoo Chung, Ian Clester, Ilia Lebedev, James Koppel, Jiazheng (Elvis) Liu, John (Jack) Feser, Joonwon Choi, Jules Drean, Muralidaran Vijayaraghavan, Nikola Samardzic, Pau Recort Bascuas, Peter W. Deutsch, Quan Nguyen, Sabrina Neuman, Sally Lee, Sang-woo Jun, Shuotao Xu, Sizhuo Zhang, Stella Lau, Tianhao Huang, Victor Ying, Weon Taek Na, Xuhao Chen and Yuheng Yang. Discussing with all those people was something that I loved during my time at MIT.

I am also thankful that my PhD was not lonely outside the lab, despite Boston being so far from home. For that, I have to thank more friends: Ben Sherman, Benoît Pit-Claudel, Brett Boston, Cecilia Testart, Eric Atkinson, Gita Singh Mithal, Max Dunitz, Min Ho Kim, Reyu Sakakibara, Sara Achour, Twan Koolen and the musician friends from theory jam,

with a special mention to Avery Yen, Govind Ramnarayan and Rio LaVigne!

Thanks also to my friends at home: Antonin Delpeuch, Clément Veyssier, Florent Poittevin, Gaspard Férey, Lisa Blanchard, Marie-Charlotte Allam, Paul Galvan, Paul Melotti, Romain Pszczolinski and Tito Nguyễn. I am very grateful to them: they all made it easy for me to come see them to catch up when I was visiting France (before COVID happened), making themselves available at the last moment, for an afternoon, an evening, or even a few days!

Finally, thanks to my family - maman, papa, Aline, Cédric, Malo and Titouan - for their unwavering support.

Contents

1 Introduction **15**

1.1 Problem statement . . . 16

1.2 Concerns and challenges: our approach to verification . . . 17

1.3 Organization of the dissertation . . . 22

1.4 Contributions . . . 23

2 Related Work **25**

3 Introduction to Fjfj **31**

3.1 Rules . . . 31

3.2 Systems of several rules . . . 35

3.3 Methods . . . 36

3.4 Modules containing rules and methods . . . 39

3.5 Primitive modules and semantics of modules . . . 41

3.6 Formal semantics . . . 48

3.6.1 Syntax . . . 48

3.6.2 Semantics . . . 50

3.6.3 General definition of a new Fjfj module . . . 57

3.7 Discussion . . . 58

3.7.1 Trace semantics versus transition systems . . . 58

3.7.2 Multiple calls to the same value methods . . . 58

4 Correctness and Refinement **61**

4.1 Simulations as refinement . . . 62
4.1.1 State simulation . . . 62
4.1.2 Module simulation . . . 63
4.1.3 Basic properties of simulations . . . 64
4.2 Simple simulations, step-by-step . . . 65
4.2.1 Register file . . . 65
4.2.2 Queues . . . 77
4.3 Refinement theorem . . . 80
4.3.1 Proof of the refinement theorem . . . 80
4.4 Other ways to compare modules . . . 84
4.4.1 Comparison with previous work . . . 84
4.4.2 Trace inclusion versus our notion of simulation . . . 85

5 Specifications 87
5.1 Different kinds of specifications . . . 87
5.1.1 Operational specifications . . . 87
5.1.2 Propositional specifications . . . 90
5.2 Representations and data structures for specification . . . 92
5.2.1 Avoiding low-level simulation relations . . . 93
5.2.2 Canonicity of representation . . . 95
5.2.3 Generalization to handle concurrency and parallelism: GCD example 96
5.3 Processor generalization for modular specification . . . 98
5.3.1 Processor, memory, and system specification . . . 99
5.3.2 Generalizing specifications: recovering modularity . . . 103
5.3.3 Limitations of specification behaviors . . . 106

6 A Full Example: Pipelined Processor 111
6.1 Original and generalized processor specifications . . . 111
6.1.1 Load buffers . . . 115
6.2 Breaking up the generalized processor into smaller parts . . . 118
6.2.1 Frontend . . . 120

6.2.2 Backend . . . 122
6.3 Processor proofs . . . 127
6.3.1 The generalized processor specification is valid . . . 127
6.3.2 Refinement between decomposed generalized specification and generalized specification . . . 133
6.3.3 Frontend refinement . . . 134
6.3.4 Backend refinement . . . 135
6.4 Discussion . . . 140

7 Coquetier: a simplification tactic for our Coq toolbox 143
7.1 Simple examples . . . 144
7.2 Overview of Coquetier . . . 148
7.3 Embedding Coq in egg . . . 150
7.3.1 Egg fundamentals . . . 150
7.3.2 Initial embedding . . . 153
7.3.3 Metrics to find simpler terms . . . 156
7.3.4 Representing types . . . 157
7.3.5 Proving absurd cases . . . 158
7.3.6 Solving existentials . . . 158
7.4 Limitations . . . 159

8 Cycle-Accurate Semantics 161
8.1 Synchronous circuits and motivation for Kôika . . . 162
8.2 Informal introduction to scheduling in Kôika . . . 165
8.2.1 Rules . . . 165
8.2.2 Scheduling . . . 165
8.2.3 Ephemeral history registers (EHRs) . . . 167
8.3 Formal description of Kôika . . . 169
8.3.1 Syntax . . . 169
8.3.2 Semantics . . . 170
8.4 The One-Rule-at-a-Time Theorem . . . 172

8.5 Case study: a cycle-accurate characterization of a small pipelined system . . 176

9 Conclusion **179**

List of Figures

3-1 Syntax of Fjfj 49

3-2 Example to illustrate the mechanics of the formal semantics 50

3-3 Judgment rules for $* \dashv\!\![*]\!\!\mapsto_{\text{value}} *$ 54

3-4 Judgment rules for $* \dashv\!\![*]\!\!\mapsto *$ 55

4-1 Key conditions of simulation relations 64

4-2 Lifting of state simulation from sub-module states to module states. 82

4-3 Trace equivalence and simulations 86

5-1 Architectural sketch of the full-system specification and the processor specification 100

5-2 Architectural sketch of the full-system implementation 101

5-3 Architectural sketch of our generalized processor 104

5-4 Simple proof-decomposition overview 106

5-5 Load/Store reorderings 108

5-6 Simple proof-decomposition overview 109

6-1 Description of a simple processor specification 112

6-2 Generalized processor specification 114

6-3 Load buffer 116

6-4 Architectural sketch of our decomposed processor specification 119

6-5 Frontend specification 121

6-6 Backend specification 123

6-7 Backend implementation 1/2 125

6-8 Backend implementation 2/2 126
6-9 Arrangement of refinements. 127

7-1 Software architecture of Coquetier 148

8-1 Scheduler semantics, with `rule` $rl = a$ 171
8-2 Rule semantics. 173

List of Theorems

Remark (Ambiguity of language) 43
Remark (Rules are non-deterministic) 43
Remark (Embedding values in natural numbers) 44
Remark (Using state in judgment rules) 52
Remark (Guards and partial relations) 53
Remark (Notation clarification) 56

Remark (Refinement is a weak simulation) 63
4.1 Theorem (Reflexivity of $\prec$) 64
4.2 Theorem (Transitivity of $\prec$) 64
4.3 Theorem (Reflexivity of $\sqsubseteq$) 64
4.4 Theorem (Transitivity of $\sqsubseteq$) 64
Remark (Comparing internal data-structures) 67
4.5 Theorem (Register file refinement) 68
4.6 Theorem (Queue refinement) 79
Remark (The implementation queue is a strict refinement of the specification queue) 79
4.7 Lemma (Lifting of expression semantics from implementation to specification) 81
4.8 Lemma (Lifting of action semantics from implementation to specification) . . 81
4.9 Lemma (Lifting state simulation from submodules to modules) 81
4.10 Lemma (Refinement theorem) 83
4.11 Theorem (Generalized Refinement theorem) 83

Remark (Focus on safety) 120

Remark (Conservative stalling) 124
6.1 Theorem (The generalized processor specification is a valid generalization) . 127
Remark (Flushing as a simulation relation) 129
6.2 Theorem (The decomposed processor specification is valid) 133
6.3 Theorem (Frontend refinement) 134
6.4 Theorem (Backend refinement) 135

8.1 Theorem (One-rule-at-a-time theorem) 174
8.2 Lemma (Committing the effect of previous rules) 175

Chapter 1

Introduction

Who needs hardware verification? Companies that we used to think of as primarily software or service companies are now designing very sophisticated hardware for their own products. For example, Google designed their Tensor system on a chip for phones, Apple designed the M1 and M2 cores for laptops and tablets, and Amazon AWS designed their Graviton processors for the cloud. Even Tesla, primarily an automotive company, designed a chip for their cars. This trend can be seen as the real-world materialization of a foundational principle of computer architecture: *the more we know about the typical workload, the more efficient a machine we can design.*

Each of these companies has unique insights into the workloads for their machines. Even when designing a general-purpose processor that can execute an arbitrary program (as in the case of Google Tensor, Apple M2, or Amazon Graviton), these companies have information and influence on what programs and code fragments will likely run on the machines. They can leverage this information to tailor their chips to their expected workloads.

Still, most companies shy away from hardware development when the designs are not guaranteed to sell in the millions. Indeed, the upfront cost of hardware development is high, a significant part of which is spent on *verification.*

1.1 Problem statement

What do people mean by verification? The goal of traditional hardware verification is to acquire confidence, mainly through testing and simulation, that the chip will work in the use cases of the end user. Standard industry practice relies on a mix of techniques but the most heavily used technique is testing. Testers run concrete scenarios – or testbenches – allowing for easy verification of simple concrete properties like "this interesting program returned the expected result." But since complex hardware designs like processors or accelerators are programmable and can often be integrated into different contexts depending on the system stacks, traditional hardware verification requires significant human and compute resources to test that way, without ever guaranteeing complete correctness.

Industry also uses formal hardware-verification techniques for more complete coverage, which usually phrase correctness in terms of custom temporal-logic formulae. The design and the properties are then converted into a large set of Boolean equations, which commonly have millions of variables. The Boolean equations are then solved by SAT/SMT solvers. Those techniques are useful and impressive, but the verification usually is limited to verifying that a temporal property holds for a few cycles, starting from a hand-picked set of initial states. These techniques also highlight the difficulty in phrasing good temporal properties, a.k.a figuring out good, albeit partial, specifications.

In contrast, in the programming-language and software-formal-verification communities, the word *verification* carries a different meaning. It usually refers to a set of *techniques* to establish strong forms of correctness: for example, one could prove the functional correctness of a sorting function and guarantee that the result produced is always sorted, for arbitrary sizes of inputs. Verification projects from those communities commonly tackle infinite input spaces, infinite state spaces, and arbitrarily long compute time, and that is often where especially interesting problems occur!

What we mean by verification: In this thesis, while our subject is the correctness of *computer architecture designs*, we use the word verification to refer to this last form of verification. As is traditional in formal verification, there are two big families of correct-

ness properties: liveness and safety properties (or functional-correctness properties). Safety properties are properties that say things of the form *nothing bad ever happens*, and liveness properties state that *something good eventually happens*.

In most of this thesis we will only focus on safety properties (only in the last chapter about cycle-accurate semantics will we outline our ability to phrase liveness properties). With this restriction in mind, our definition of the correctness of a design will actually always involve two designs. A complex, optimized sequential machines (for example, a pipelined processor) will be said to be correct with respect to a simpler unoptimized state machine if it does not exhibit any behavior that is not a behavior in the simpler machine. Formally, we will define a *weak simulation relation* of the implementation machine by the specification machine. We will see that our definition of a machine admits non-synthesizable and non-deterministic machines. This allows us to cast a very large number of verification tasks within our simulation framework.

An important consequence and limitation of our focus on safety is that *a design that does nothing is always correct* (nothing bad ever happens, as nothing ever happens). This limitation might be disappointing at first, however, our proofs still carry meaning: if something ever happens in the design, it is guaranteed that it is according to the specification. The machine can freeze, but it cannot produce wrong results. This restriction to safety is common in formal methods.

1.2 Concerns and challenges: our approach to verification

Promoting verification techniques that are closer to designer' intuitions: When we learn computer architecture, we are usually first presented idealized designs, e.g., the notion of prediction of the next program counter and the mechanism by which we can correct mispredictions. At that level, there is no circuit, gate, or Boolean logic involved. After a few hours of thinking about such designs, we get an intuitive (and often correct!) understanding of the idealized designs. From there, when we have the right tools, we are

able to implement those architectures correctly (or almost correctly). During this learning process, we manage to convince ourselves (in a sense we proved to ourselves) that the idealized design is correct. Maybe we made mistakes and misunderstood the schemes, or we ignored details, or sometimes we are even plain wrong. However, at some high level and more often than not, we understand in which sense *the architecture we are studying is correct*.

This self-convincing is never based on exhaustively thinking about the finite state machine described by the design, which typically involves exploring billions of states. Instead the proto-proofs that we are building are articulated around manipulating intuitive high-level abstractions, for example "this part of the system is guaranteeing first-in-first-out," which when put together, lead to an argument that the scheme we were taught is doing what it is supposed to do.

Our ultimate goal is to be able to write proofs that are machine-checkable formalizations of the arguments that architects and students are often thinking about when designing machines. This is in contrast with the systematic large-automata constructions done by computers in the current formal hardware-verification methodology (see the related work in chapter 2) that are often hit by state-explosion problems.

Synthesizable vs Verifiable Designs: Now that we have loosely hinted at what *correctness of a design* will mean for this thesis, we can point out an important concern that motivated several of our technical choices. At the heart of the hardware-verification problem is the mismatch between the abstractions at which one usually thinks and writes circuits (a graph of logical `and`, `or` and `not` operators and registers named the Register Transfer Level abstraction) and the abstractions at which one usually thinks about computer-architecture designs: informal abstract transition systems that use non-synthesizable data structures like lists and maps, nondeterminism, etc. So the intuitions about correctness always live at the latter level, while the actual implementation lives at the former.

Our starting point to tackle this mismatch is the Bluespec design methodology. In this methodology, we don't describe a hardware circuit directly, but instead we generate a circuit from a high-level program: a set of nondeterministic atomic transitions on the state elements. Bluespec System Verilog (BSV) is the salient example of a class of languages

that we name rule-based programming languages, because the atomic transitions are named rules. Traditionally, BSV is known as a hardware description language, because once one puts the right restrictions on building blocks and composition operators, BSV programs are, in a sense, guaranteed to be synthesizable to circuits. Interestingly, the programming model does not mandate that the state elements (or more generally the submodules of a system) have to be synthesizable. And while this perspective has not been fully explored previously, we make heavy use of it in this thesis. In a nutshell, the BSV style of programming languages allowed us, with a few modifications, to have a system in which we can have very different kinds of programs coexist and compose. We can simultaneously write programs that are synthesizable designs and non-synthesizable relational programs (we will see that some of those abstract programs really look like mathematical formulae) which can serve as specifications or intermediate specifications of design.

Modular verification to keep verification tractable: Once we have built a setup where we can more easily bridge the gap between the different levels of abstraction that verifying hardware requires, we can introduce custom abstractions in the proof assistant to avoid the state-space-explosion problem. However, we soon face another problem. That is, for many standard sequential machines, like a pipelined processor connected to a simple memory, tackling the proof of full-system correctness of the design monolithically in the proof assistant requires significant effort. It requires global system invariants that are often not so easy to write and, more importantly, are very difficult to maintain when the code of the design changes. For example, replacing a one-element queue by a two-element queue in some stage of the pipeline of a previously proven correct processor, while requiring two minutes for the working architect to judge it as a safe change, will easily require days of manual labor in the previous approaches.

To allow modular proofs, which will provide more robustness to change, one soon realizes that we actually need *modular specifications*. While this may sound obvious, ignoring this simple observation has led to significant difficulties. Let us quickly outline why.

Let us consider a machine composed of a processor and memory, which interacts with the environment using memory-mapped IO (MMIO). On the one hand, our specification

system would be composed of a simple one-instruction-at-a-time processor and a simple idealized memory composed of an infinite array that can be accessed atomically. Together the memory and the processor form what is commonly thought of as the ISA specification. On the other hand our implementation system would include a pipelined processor, and the memory would be more realistic, sustaining more than one outstanding load. From the full-system perspective, we want to prove that the two systems show exactly the same I/O behavior for any program.

To prove that property, one could hope to tackle the problem modularly: first prove that the pipelined processor implements the one-instruction-at-a-time processor specification, then prove that the fancy memory implements the simple memory, and then use some *language-level* refinement principle that would allow us to compose everything together. However, this is not possible. When it comes to the processor, the pipelined processor is actually not a valid implementation of a one-instruction-at-a-time processor. The pipelined processor might emit spurious loads (speculation), and might reorder some instruction and data loads. So the pipelined processor has *more* behaviors than the naive specification does, even though those extra loads don't matter at all when we look at the full-system semantics, i.e., the MMIO trace.

Hence, the common way of specifying the full ISA is in a sense an *invalid* specification for just the processor. It is only valid for the full system, once we hook up a memory. To solve this problem, we need to introduce a specification *just for the processor*: one that is faithfully modeling a superset of all the possible load and store traces that implementation processors can emit. We also aim for the specification to be simpler than the implementation. We call such a specification a *generalized specification*, and these specifications are the core of an entire chapter of this thesis.

Related work: This dissertation fits in a long line of work that aims at describing and reasoning about complex sequential machines in a way that is abstract (to hide the gate-level details of hardware designs) but precise and modular [3, 55, 30, 23, 22, 59, 21, 58, 19, 62]. Similarly to Kami [19], our work is mechanized in Coq and shares the aspiration to study the correctness of systems through refinements between implementations and specifications.

However, both the notions of module and refinement differ in the two systems. Thanks to language restrictions, Fjfj describes the semantics of a module one transition at a time, while Kami [19] requires considering the effect of calling several methods of a module simultaneously. The general related work will be described in the next chapter, while detailed comparisons with Kami [19] will be given throughout the dissertation.

What about actual time in the rule-based methodology: Finally, note that in the rule-based design methodology the designer specifies all the state elements, i.e., registers, and describes the dynamics of the system using a set of atomic rules. Each rule specifies a state transformation. The semantics of a rule-based program is the nondeterministic execution of this set of atomic rules on the state. However, the actual circuit implementation does not show the same nondeterminism, and actually having the circuit executing a single rule per clock cycle would bring about unacceptably poor performance. We still do need rules to execute concurrently within one hardware clock cycle, though in a controlled way that preserves the illusion of atomic execution.

The commercial BSV compiler relies on a static analysis of the rules to do concurrent scheduling of rules. This approach has the consequence that what precisely happens within a clock cycle depends on what the compiler was able to figure out during its static analysis. BSV programmers often think deliberately about static-analysis details and even change their code to nudge the compiler in the right direction to achieve the desired clock-cycle behavior.

In a sense, one of the great strengths of the rule-based methodology (decoupling architectural timing from functional correctness) can feel like it is introducing difficulties when we do care very precisely about time, as time depends on internals of the compiler.

This apparent issue was a first-order concern to us in 2018 when serious security issues related to the observation of time were put in the minds of many computer architects. In the last chapter of this thesis, we will present Kôika, which helped mitigate those concerns: working with rule-based languages is not abandoning any hope to think precisely about time.

Kôika provides a new core calculus maintaining the essence of BSV, preserving all its desirable properties and yet allowing direct control over the atomic actions executed each

clock cycle, without relying on static analysis. Kôika programmers still need to think about the real rule conflicts but not about the compiler's abstraction thereof. Our calculus includes a *deterministic, cycle-accurate* operational semantics, enabling formal reasoning about performance, without removing the ability to prove invariants by induction on sequential executions: the effect of the set of rules completed each cycle is proven to be always explainable in terms of one-rule-at-a-time semantics.

1.3 Organization of the dissertation

First, we will do a tour of the related work in chapter 2. Then, in chapter 3 we will introduce our programming language: Fjfj. Fjfj is a well-behaved small language, a variant on a fragment of the BSV language, for which we define a formal semantics. The chapter first introduces the language and explains the fundamental building block of Fjfj: *module*s. We first go through examples and then describe and explain the mechanized semantics that we implemented in the Coq proof assistant.

In the following chapter 4 of the dissertation, we will define the tools to be able to compare Fjfj modules: the notion of refinement/simulation. While we tried to sprinkle the chapter with examples, this chapter is the driest of the dissertation, and many theorems and their proofs can likely be skipped in the first read.

At this stage, we have built all the programming-language abstractions we need, and we can start doing a bit of computer architecture: in chapter 5, we will explore different variants of specifications for some example modules that are standard building blocks of computer architecture and compare their relative merits. This points out that writing good specifications is often the main difficulty of the verification task.

An important contribution of this chapter is to introduce the idea that to be able to do modular proofs, we will often need to come up with what we call *generalized specifications*. This chapter ends on a proposed hierarchical specification for a fairly large class of pipelined processors. We will also show that our generalized specification, while it does look very general, is actually not general enough. We will also showcase an example of a processor optimization that *could not be proven correct* in our current setup, explain why, and outline

what would need to be done to prove this kind of design correct modularly. This chapter set up all the required concepts to understand the key example of this thesis presented in the next chapter.

In chapter 6, we will present our study of a family of pipelined processors: we will start from a precise description of the generalized specification we introduced in chapter 5 and then hierarchically decompose the design, giving a succession of intermediate specifications all the way to a family of implementation designs. The chapter will also cover the different theorems we prove about those designs and high-level insight into how we prove them. We leave the technical details and the description of our Coq toolbox for the next chapter.

In chapter 7, we will describe our workflow and the different tools we built in Coq, to write mechanized proofs in our framework. There are two completely orthogonal aspects to doing proofs in our mechanized setup. One aspect has little to do with computer architecture and is simply about proof-assistant engineering: we will present some of the proof-automation strategies that we developed, which significantly increased our productivity. Those might be of relevance outside of our specific framework. The presentation of this work is mostly self-contained and almost standalone. It can also be skipped for the computer architect simply interested in learning and using our methodology but not interested in the inner-workings of this automation.

The second aspect of proofs has to do with proof techniques informed by the computer architect's intuitions. We will especially focus on our formalization of the flushing idea from [14].

We will finish the core of the dissertation by presenting our work to formulate time precisely in rule-based languages in chapter 8 and conclude.

1.4 Contributions

This thesis makes the following contributions:

- Definition of a well-behaved rule-based programming language centered around a simple but very flexible notion of modules.

- Formalization of this language in Coq, with proof of the key metatheorems.
- Generalized hierarchical specifications of a processor and memory subsystem unlocking modular proofs and showcasing several previously ignored issues in processor specifications.
- Proofs that our full-system generalized specification, while looking potentially dangerous compared to standard ISA specifications, is actually harmless from the full-system perspective (memory-mapped IO interactions).
- Mechanized proof of correctness for a family of pipelined processors against our hierarchical specification.
- A set of proof techniques for establishing and working with simulation relations in our framework.
- A cycle-accurate semantics and compilation scheme to Verilog for a fragment of our language.

Chapter 2

Related Work

Structural hardware description languages Traditional HDLs like Verilog and VHDL are mainly used to describe interconnections of boxes, i.e., Boolean gates and registers. The main practical problem with such languages is that they provide inadequate type checking and lack precise semantics, which makes verification and design refinement a Herculean task. Hence, leveraging programming-language techniques to ease design in those languages has been an idea around since the 1980s, and it has seen a rebound in popularity more recently. For example, a popular way to make Verilog-style programming more convenient is by embedding it in a language with a good macro facility, which can provide type safety and good combinators for composition [6, 24, 2, 4]. What all those approaches have in common is that they always view designing hardware as a structural problem: programming-languages tools are used to make composition and connections of structural blocks easier, using advanced type systems or hygienic metaprogramming facilities. This line of work does not tackle the difficulty of specifying and implementing complex interactions between sequential machines, which we believe is a core difficulty of hardware design. Instead they focus on good metaprogramming support for generating circuits. As such they rarely provide innovative approaches to verification.

High-level synthesis (HLS) Another approach to hardware design is to transform programs written in software languages like C, Python, MATLAB, etc. into hardware [25, 16, 20, 27]. HLS compilers rely on compiler techniques developed for parallel and vector archi-

tectures starting in the 1980s. In spite of fundamental limitations of this approach (see for example [1]), the commercial appeal is strong enough that many companies have invested significant resources into building better HLS tools [63, 44]. This approach has shown good promise for signal-processing applications but has not been shown to be useful to describe processors and other complex designs. More recent work [35, 34] has explored interesting flexible compilation techniques, and the use of a software-language semantics might open interesting doors for various verification tasks [29]. As far as we know, those approaches have not yet been applied for complete formal functional correctness of complex sequential machines.

Combinational circuit verification/sequential machine verification Let us emphasize the difference between the problem of specification and verification of combinational functions and the specification and verification of sequential machines.

Combinational circuits are hardware implementations of Boolean functions, and as the topic of verification of equivalences between Boolean functions is very well studied, the subject of mechanical verification of combinational function is very advanced. Over the years, research has produced impressive techniques to solve many practical problems [11, 46], and this thesis makes no attempt at tackling that problem. It is however interesting to notes that several of the advanced techniques used also in modern software verification (e.g. [46, 40, 11, 12]) came from the broad hardware-formal-verification community.

Those tools and techniques, very effective for combinational circuits, are also at the root of most mainstream formal-verification techniques in use today for verification of sequential machines. Indeed those techniques are often based on unfolding the combinational function at the heart of the sequential state machine for several cycles to build a single big combinational function modeling the behavior of the sequential machine across several cycles. While they are able to search impressively large spaces, these techniques get hit by combinatorial-explosion problems, and even the most advanced automatic verification techniques have trouble handling processors that have more than a couple instructions in flight.

Variations on the meaning of correctness In most of the literature the notion of correctness relates two designs with respect to a *refinement map*. That is, the correctness of an implementation with respect to a specification is not defined only through the set of behaviors observed through the interfaces of both the implementation and specification, as we can do for our rule-based languages.

Instead a custom predicate named the refinement map describes in which sense the *states* of the two machines are equivalent. In this setup, the meaning of the proof of correctness not only involves the two machines but also involve the predicate used to related the two machines. If one defines a wrong predicate, for example all the states of the implementation machine are related to all the states of the specification machine, we can complete a proof of correctness that has little meaning.

This prompted very interesting work on what constitute a good refinement map. For processors, [56] introduced an intuitive criterion: the state of the implementation, once flushed, should agree with the specification on the architectural state.

This criterion of correctness raises the side question of how to define the notion of flushing properly, and quickly the community realized that there was more than one way to define flushing, which lead to multiple notions of correctness that were not always comparable. Even though this points out a limitation of using refinement maps as definitions for correctness, the idea of thinking about flushing to relate implementation and specification is rich, and we reuse that idea in this dissertation. However, in our case instead of defining correctness through the refinement map, correctness is defined through behaviors and simulation, independently of flushing and refinement maps. The idea of flushing the implementation is used as a tool to prove the simulation relation. Hence if we define an incorrect notion of flushing we simply will not be able to finish our proof.

In the vein of verification work with custom predicates, we can mention several pioneering works [36, 10] that tackled industrial designs, often in the ACL2 theorem-proving system. We can mention other work done in other frameworks (for example, in UCLID5 [54] or in SMV [41]), using various level of automation and usually tackling custom models expressed at various levels of abstraction over synthesizable designs, trading off for complexity of the architectural schemes being proven [14, 13, 42, 43, 9, 8, 5]. Those last two examples are

done in the context of symbolic model checking, but they are very reminiscent of theorem-proving-oriented approaches. Those are very early examples of a continuum of techniques in formal methods: it is typically too simplistic to classify formal methods between theorem proving, automatic verification and model checking.

We mention also [33], which proposes a nice framework to both specify and prove the correctness of accelerators using traditional formal-verification techniques.

BSV verification Most of the research on the BSV [49] programming model has studied the benefits of the language to express common architectural patterns, extensions to the programming model and synthesis. A big focus was to generate good circuits from descriptions much closer to the architect's intuition than traditional RTL descriptions. One of the seminal papers on the question of verification in rule-based languages is [3]: the authors showed, in pen-and-paper proofs, how the rule-based formalism allowed to describe and study the correctness of complex microarchitectural schemes through the properties of corresponding rewriting systems. This paper strongly influenced us in that it showcased that we can, at least on paper, formalize computer architects' intuitions about correctness.

Interestingly, the new doors that the programming model [31, 32] opened for practical verification, while being a pillar of motivation in the very early days of the language in the late 1990s, were not systematically explored until 2010. During this intermediate time, most of the verification work focused on variations of applying standard hardware formal-verification techniques to the generated Verilog. A first line of work [23, 22, 21, 59] looked at using a form of refinement to characterize the correctness of designs. Then, Kami [19, 58] first formally defined a notion of refinement as the central notion for correctness, within Coq. While technically, their notion of refinement is different from ours (among other things the semantics of a module requires to define a notion of substeps), both definitions aspire to a common vision. Related to the notion of correctness defined in [19] are [60] and [18] that focus on refinement in the context of cache coherence.

More recently, [62] has explored model checking for rule-based languages. At the language level, [62] is the closest work from this thesis: we share the tree-like module hierarchy restriction and the restriction of at most one action method called per submodule. However

[62]'s notion of correctness for processors was not based on simulations and refinements. Instead it reasons about custom relations similar to other works exhibiting custom refinement maps.

Timing-accurate semantics and BSV synthesis The work this thesis presents on timing-accurate semantics in rule-based languages can also be seen through the lense of variations of BSV compilation schemes. In that direction we mention [51, 37, 38, 48, 26]. Each explored extensions and variations on how to think about (or compile) what happens in a cycle in a BSV-style language.

Chapter 3

Introduction to Fjfj

Fjfj is a rule-based hardware language centered around *modules* composed of *rules* and *methods*. Rules are spontaneous internal state transitions that the module takes on its own, while methods are the only way to interact with a module.

We first give an introduction to rules, then methods, and then we give a first example of a complete module.

3.1 Rules

Consider the following two rules:

```
Definition increment :=
    (rule
      (begin
           (set v {read r})
           (* We use curly braces for submodule calls:
            we call the method "read" of the
            submodule "r" on no argument *)
           {write r (inc v)}
           (* We use parens to apply
            the purely combinational function "inc" to
```

```
     the value "v" *)
    ).

Definition f_stage :=
  (rule
    (begin
        (set el {first in_queue})
        {deq in_queue}
        {enq out_queue (f el)})).
```

The first rule calls the method `read` of the register `r` and puts the value returned by the method into a local variable `v`, then applies the combinational function `inc` to `v`, and finally writes the result in register `r` (by calling the method `write`). Note the different syntax between calling the method of a submodule (using curly braces) and the syntax to apply a purely combinational function (using parentheses). The second rule looks up the first element of the input queue, dequeues the first element, and enqueues to the output queue the result of applying a pure combinational function `f` to the first element. All these effects are performed atomically: if `in_queue` is empty or `out_queue` is full, the second rule will have no effect at all. In such a situation, we will sometimes say that the rule was *aborted*. Note that the syntax is reminiscent of Lisp, where instead of writing `f(arg)` like in the C language, we write `(f arg)` or `{method module arg}`. The parentheses are part of the calling syntax and cannot be omitted (and no extra parenthesis can be added).

Our next example showcases one way in which Fjfj differs from traditional software languages regarding semantics. Rule `swap` swaps the values of two registers `r` and `s` without using a temporary variable.

```
Definition swap :=
  (rule
    (begin
        {write s {read r}}
        {write r {read s}})).
```

In Fjfj, the `read` method call of registers refers to the values found in the registers just before we considered executing the rule (we say, at the *beginning* of the action). Reciprocally, the effects of stateful operations, i.e., the `write` method call of registers, are *not observable* at all within the same rule (within the action). All register updates happen simultaneously at the end of the action. More generally, for an arbitrary submodule that we interact with through arbitrary methods, we distinguish between two kinds of methods: 1. value methods, like the `read` method of a register or the `first` method of a queue, that can only observe the state of a module at the beginning of the action 2. action methods, like the `write` method of a register or the `enq` and `deq` methods of a queue, that change the internal state of the module (without returning anything) and only change the state atomically, once the action is completed.

For example, in the following snippets, `el` is the head of the queue even though the method `deq` was called earlier: the dequeuing of that head element is performed at the end of the action, so the value method `first` can still observe it anywhere within the action. Similarly, if the register `r` contained 1, the writing of the value 2 in the second example will not be observed by `{read r}`, and so `el` will be 1, not 2:

```
Definition action_at_end :=
    (rule
        (begin
            {deq queue}
            (set el {first queue}))).
```

```
Definition action_at_end2 :=
  (rule
      (begin
          {write r 2}
          (set el {read r}))).
```

Similarly, if the queue were to be empty in the following example, el would *not* be the element 1 that was just enqueued, and the entire rule would do nothing, as it is impossible to get the first element of an empty queue.

```
Definition not_observing_new :=
  (rule
    (begin
      {enq queue 1}
      (set el {first queue}))).
```

Delaying the effects of the action methods of the submodules to the end of each action requires us to clarify the thorny question of the semantics of calling several actions in a rule, e.g. the case where write is called twice on the same register, or the case where one both enqueues and dequeues from a queue. While there are several ways to define the semantics in a way that is both sound and realizable as circuits, we take the opinionated stand to rule out such issues by simply forbidding calling two action methods of the same module in a single rule. That is, we detect dynamically when a rule tries to call two action methods of the same submodule and we abort the rule. Hence, the following rules do nothing.

```
Definition aborts1 :=
  (rule
    (begin
      {write r 3}
      {write s 0}
      {write s 1})).

Definition aborts2 :=
  (rule
    (begin
      {enq queue 1}
      {deq queue})).
```

This restriction does not prevent a program from calling two different modules:

```
Definition not_abort :=
  (rule
```

```
    (begin
        {enq queue1 1}
        {enq queue2 2})).
```

Moreover, aborts are detected dynamically, during execution, and the whole rule is canceled atomically (in the example `aborts1` above, this implies that the write to `r` will not be performed either). In the following example, the rule would need to enqueue twice in the queue, when `r` and `s` both hold the value 0, hence the rule has to abort in this case, otherwise it succeeds.

```
Definition conditional_abort :=
  (rule
      (begin
        (if (= {read r} 0)
            {enq queue 1}
            pass)
        (if (= {read s} 0)
            {enq queue 2}
            pass))).
```

3.2 Systems of several rules

Up to now, we have given intuition for the semantics of systems of single rules, but the approach generalizes to more rules.

```
Definition do_f :=
    (rule
      (begin
          {deq inp}
          (set el {first inp})
          {enq mid (f el)})).
```

```
Definition do_g =
  (rule
      (begin
          {deq mid}
          (set el {first mid})
          {deq out (g el)}).
```

The so-called one-rule-at-a-time (ORAAT) semantics of a collection of rules is to pick a rule nondeterministically, execute it, and commit its results. (In case of an abort, the state does not change.) The process is repeated endlessly, and if one rule calls an action method that causes a submodule to be updated, the next rule will observe the new state of the submodule. The ORAAT semantics need not produce a deterministic answer because the rules are not required to be confluent. In this specific example, the system of two rules describes a simple arithmetic pipeline computing $g \circ f$ and is actually confluent.

3.3 Methods

Until now we have described systems of multiple rules that interact with submodules through the methods of the submodules, but those systems only had rules, and they did not themselves expose methods. There was no way for the environment to interact with our systems. In particular, the system described could not be used as a submodule of a bigger new module.

In this subsection, we explain the way one can *define* methods of a Fjfj module, which constitute the interface of a module. As previously explained, we can define two kinds of methods: (1) value methods are methods that do not make the module take transitions and simply observe part of the state, (2) action methods instruct the module to take transitions, without observing its internal state.

Let's start with building a queue module using only register modules.

```
(* In the reg.v file, there is a construction [mkReg] to build a register module.
   We create two register instances named valid and data *)
Local Instance submodules : instances :=
```

```
#|
reg.mkReg valid;
reg.mkReg data
|#.

Definition enq :=
   (* enq is an action method which takes a single argument el *)
  (action_method (el)
    (if (not {read valid})
        (begin
            {write valid 1}
            {write data el})
        abort)).

 Definition first :=
    (* first is a value method which takes no argument *)
   (value_method ()
        (if {read valid}
         {read data}
         abort)).

 Definition deq :=
   (action_method ()
       (if {read valid}
         {write valid 0}
         abort)).

 Global Instance mkQueue1 : module _ :=
 (* Define a 1-element queue module constructor, mkQueue1.
    It is composed of the value method first
```

```
and two action methods enq and deq *)
module#(vmet [first]
        amet [enq; deq]).
```

We have now defined our first complete module, named `mkQueue1` (in the namespace `QueuePkg`), giving first the submodules (instances) of the module, and then its action methods `enq`, `deq` and its single value method `first`. This module contains no rules. The semantics of methods is different from the semantics of rules in the following ways.

- While rules nondeterministically execute on their own, methods need to be called. A method can be called in three different ways: (i) the caller of the method is itself a defined method of the parent module, (ii) the caller of the method is a rule of the parent module calling the method, or (iii) the method under consideration is a top-level defined method, and the outside world is the caller.

- All methods can be given arguments, and a value method returns a value without updating the state of the module (contrary to rules), while an action method only updates the state of the module without returning values. It means that in the body of a value method, one can only call *value methods* of submodules, while the body of an action method can mention both kinds of methods.

- When a method does not succeed because the method dynamically aborts, the abort propagates to the caller, causing the caller to abort. As a rule has no caller, this is a small difference between rules and methods.

We can now explain the 3 defined methods of the queue. `first` is the only value method. As such, it is not allowed to use any action method of any submodule, so in this case it cannot write to any register, but it can read from all registers. Moreover, when the valid bit is zero, the `first` method aborts (so cannot be called). `enq` is an action method. As such, it is also allowed to use the action methods of the submodules, like the write method of the registers. In this implementation, the implementation of `enq` first checks the valid bit. It avoids aborting only when the valid bit was low, indicating the one-element queue is empty.

In that case, it puts the new element `el` in the data register. The other action method, `deq`, works symmetrically.

When a caller (a parent rule or a parent method) interacts with a module, the interaction, a single atomic step of interaction, can simply be represented as an arbitrary number of value method calls followed by at most one action method call. We denote such an *atomic interaction* as: $(f_1(x_1) \rightarrow y_1 \mid\mid f_2(x_2) \rightarrow y_2 \ldots \ \& \ g(x))$ where f_i are value methods and g is an action method.

Since value methods are just observations and do not affect the state of the module, they commute (*i.e.* , $\mid\mid$ is commutative and associative). By putting the action method at the end, the notation also reflects the fact that in Fjfj a step state update occurs after the observations. Together these two remarks justify the the terminology *atomic interaction*.

3.4 Modules containing rules and methods

Finally, we can put rules and methods together in a complete example. The following module is a pipeline module computing g composed with f:

```
(* We previously defined a one-element queue constructor, mkQueue1,
in a file QueuePkg.v, we now use it to instantiate 3 one-element queues.*)

Local Instance submodules : instances :=
#|
QueuePkg.mkQueue1 inp;
QueuePkg.mkQueue1 mid;
QueuePkg.mkQueue1 out
|#.

Definition enq :=
    (action_method (el)
        {enq inp el}).
```

```
Definition do_f :=
    (rule
        (begin
            {deq inp}
            (set el {first inp})
            {enq mid (f el)})).

Definition deq :=
    (action_method ()
        {deq out}).

Definition first :=
    (value_method ()
        {first out}).

Definition do_g :=
    (rule
        (begin
            {deq mid}
            (set el {first mid})
            {enq out (g el)})).

Global Instance mkPipeline : module _ :=
   (* Define a g of f pipeline module made of:
        - two rules do_f and do_g
        - a first value method
        - two action methods enq and deq *)
    module#(rule [do_f; do_g]
            vmet [first]
            amet [enq; deq]).
```

In this example, we did not give details of the implementation of the queues used as submodules. One valid implementation is the implementation of one-element queues we gave in the previous paragraph, but one could also use a two-element queue or possibly a queue specification made of a potentially unbounded list.

One might wonder what are the basic primitive modules that we can start from. In a traditional setup where we only do design, we would just have registers holding bounded-size data as primitive building blocks. It is known that rule-based programs using only such registers as leaf submodules can be compiled to efficient RTL.

An important aspect of our module system is its flexibility in defining new modules. Fjfj allows us to define new primitive modules with custom semantics: the basic building blocks are not limited to registers. When we introduce new primitive modules, we will lose synthesizability, but we will gain ease of expressivity and better modeling facilities which will allow us to ease verification. Those nonsynthesizable primitive modules will be intermediates used only for verification and not present in the actual implementation design.

This leads to a key principle of our methodology: in Fjfj, we have a substitution principle. If we have proven that a module is a valid implementation of a model, e.g. the one-element queue using two registers versus the ideal list-based model, then we safely substitute the implementation for the specification in any design. We will elaborate on this principle in chapter 4.

3.5 Primitive modules and semantics of modules

In the previous section we gave an intuition for *rules* and *methods* in Fjfj from examples where we gave snippets of syntax. Conceptually, *rules* and *methods* are transitions, so the semantics of a module is the collection of those transition relations.

In this chapter we elaborate on the semantics of modules. We use a special typeface for those relations, *i.e.* for rules, we write $\mathbb{r}$ for the state-transition relation. If a module operates on a state of type T, the state-transition relation of a rule $\mathbb{r}$ is a subset of $T \times T$.

Before we elaborate on the structure of $\mathbb{M}$s and $\mathbb{r}$s, let's discuss what kinds of values are manipulated in Fjfj.

The values manipulated in Fjfj are conceptually bitvectors. As such, semantically we can simply ignore the size and make every bitvector part of a bigger type $\mathbb{N}$ (that we can see as the type of arbitary-sized bitvectors). When the size is semantically important to some function, instead of the size being passed implicitly through the type, it is passed as a value. As pointed out earlier, every method has formally a single argument at the semantics level. However, we can pack and unpack multiple arguments (using a one-to-one correspondence between $\mathbb{N}$ and $\mathbb{N} \times \mathbb{N}$) into a single argument, hence this simplification useful for prototyping is without loss of generality.

The semantics of a module, noted $\mathbb{M}$, is described by the following collection that conceptually represents a Labelled Transition System (LTS):

- A collection of *rules*' transition relations, where each rule is an update relation on state: $\mathbb{r} \subset T \times T$.
- A collection of *value methods*' observation relations, where each method is an observation relation on the state: $v \subset T \times \mathbb{N} \times \mathbb{N}$ which relates a state, an argument, and a returned value. (Or $v \subset T \times \mathbb{N}$ if the method does not take arguments)
- A collection of *action methods*' transition relations, where each method is a state-transformation relation: $a \subset T \times \mathbb{N} \times T$ which relates a state, an argument, and a new state (Or $a \subset T \times T$ if the method does not take arguments)
- With T as the *internal module type*: the type of data that the module manipulates

Not every such relation has to come from a piece of Fjfj syntax. As we already explained, it is sometimes useful to define models of modules not from pieces of Fjfj syntax and to instead directly describe the state transition relations in Coq. Such modules that have semantics without syntax are *primitive modules*, in contrast to *standard modules*.

In the case of a *standard module*, the 4 fields of $\mathbb{M}$ are derived automatically from the pieces of syntax for rules and methods and from the semantics of the submodules. In that case, we need to distinguish r – the syntax of a rule – from $\mathbb{r}$ a state-transition relation corresponding to a rule. One of the objectives of this chapter is to define a function $x \mapsto [\![x]\!]$, which associates a semantics $\mathbb{r} = [\![\mathrm{r}]\!]$ to a piece of rule syntax r. We will do the same for the

state-transition relations corresponding to action methods and value methods, and we will package them into the semantics of a module $\mathbb{M} = [\![\mathrm{M}]\!]$.

In the case of *primitive modules*, one defines the 4 fields of $\mathbb{M}$ directly in the proof assistant, as we will soon see with examples.

Remark (Ambiguity of language). We already overloaded the English words *rules*, *value methods* and *action methods*. Sometimes they refer to a piece of syntax r, sometimes they refers to the corresponding semantics: a relation $\mathbb{r}$. The reader should be able to disambiguate based on the context.

Remark (Rules are non-deterministic). Previous work on formalization and mechanization of rule-based languages considered the state transitions of rules and methods to be *functions* instead of *relations*. That is, traditionally rules and methods had deterministic effects in a given state and the nondeterminism only came from the nondeterministic choice of which rule to execute. In Fjfj we are considering the possibility of having non-deterministic rules and methods. Hence, contrary to standard rule-based languages, we can conveniently write Fjfj specification modules which have no rules but are nondeterministic modules. The 4 fields of $\mathbb{M}$ sometimes will be derived from actual pieces of syntax for rules and methods and submodules (*i.e.* see subsection 3.6.3 for the actual definition of $[\![\mathrm{M}]\!]$), but for the *primitive modules*, they are directly defined as three sets and a type.

As a preliminary example, suppose we have a value method whose semantics relation is the following single set $v = \{(t, 0, 2)\}$, then calling the value method on the module is possible if and only if the module is in state t and the argument passed to v is 0. In that case, the value method returns the value 2.

The partial definition of this relation is crucial to keep in mind: it highlights that the domain of the relation is encoding the abort semantics. Every rule and action method is a partial transition relation: for example in our first complete module `mkPipeline`, when the `mid` queue is empty, it is impossible to dequeue from it. Hence, the `do_g` rule of our pipeline module does not define any transition leaving from a state containing an empty `mid` queue. When we define a new rule or method, we define a new transition relation with a domain restricted to the constraints coming from the domain of the method transitions of the submodules. The partiality of the transition relations is what is named *guards* in

BSV: rules and methods are guarded, and all the methods of submodules used by a rule or a method need to be ready for it to be executable.

In practice, the transition relations are not represented extensionally (by listing the elements of the set). Instead they are built in one of the two following ways:

1. Manually writing predicates representing the relations, directly in Gallina (see the two upcoming examples). Such modules are named *primitive modules*.
2. The relations are derived automatically from pieces of Fjfj syntax, using submodules for which the relations are already defined (see subsection 3.6.3). Such modules are named *standard modules*.

First, let's discuss how to define primitive modules, in Gallina.

Remark (Embedding values in natural numbers)*.* While the internal state of a primitive module is arbitrary, it could for example contain lists, trees or functions, the arguments to methods and the values returned by value methods are necessarily of type $\mathbb{N}$, *i.e.* are bitvectors. This restriction intuitively corresponds to the fact that hardware interfaces are always bitvectors in the end, and while for verification purposes we might want to make non-synthesizable modules, non-synthesizable *interfaces* would be going one step further.

Register: The state of all modules is always represented as a pair of a field containing the type T of the state, and a field *state* containing the actual value (which is of that type T). In this example, we will describe a register holding a natural number, so $T = \mathbb{N}$.

A register has one value method (read), one action method (write), and no rules, and the module is written by the following propositional predicates.

```
Definition write arg st newst :=
  ∃ old_value,
  st = {| T := N; state := old_value |} ∧
  newst = {| T := N; state := arg |}.

Definition read _arg st ret :=
```

```
(* when an argument is not used, we prefix its name
   with an underscore, e.g. [_arg] *)
  ∃ old_value,
  st = {| T := N; state := old_value |} ∧ ret = old_value.

Global Instance mkReg : module _ :=
 primitive_module#(vmet [ read ] amet [ write ]).
```

The write method is an action method, and it constrains the new internal state of the register to be the argument fed to the write method. Note that the method is always ready (it is always possible to call it) as long as the state contained a value of type ℕ (which at some higher level is conceptually guaranteed by typing).

Let's elaborate on some generality of the internal state of Fjfj modules. Any module in Fjfj contains some values of certain types. For example, maybe a queue contains a list of elements, a register contains a single number, a cross product of two substate elements, etc. So the representation of those values, which can have various types, is a pair of a type (the field `T` of the struct) and a value of that type (the field `state`).

To get accustomed to this way of defining the semantics of a register, let's unfold the definition for write and confirm it corresponds to our intuition of what a write to a module does:

$$
\begin{aligned}
\mathbb{write} &= \{(st, arg, nst)| && \text{write arg st nst}\} \\
&= \{(st, arg, nst)| && \exists\, old, st = \{|T := N; state := old|\}\wedge \\
& && nst = \{|T := N; state := arg|\}\} \\
&\sim \{(old, arg, arg)\}
\end{aligned}
$$

In the last line of the equation, we omit the types to avoid the clutter, and see the expected transition function: *a register in an arbitrary state old will transition to state arg when we write the value arg in it.*

Custom primitive modules, which are not synthesizable, are also often useful for verification. A good example of such a primitive module is the following unbounded queue:

Unbounded queue: An unbounded queue has a list of values as internal state (it clearly is not synthesizable to hardware), a value method (first), and two action methods (enq and deq). It has no internal rules. It keeps its internal state as a list of values:

```
Definition first _arg st ret :=
        ∃ h t,
        st = {| T:= list N; state := cons h t |} ∧
        ret = h.
(*
    The predicate "first _ st ret" is true if and only if
    (st, ret) is in the relation for the method "first".

    In that case, it means that the "first" method returns value "ret"
    when called in state "st".
*)

Definition deq _arg st newst :=
    ∃ h l,
    st = {| T:= list N; state := cons h l |} ∧
    newst = {| T:= list N; state := l |} .
(*
    The predicate "deq _ st newst" is true if and only if
    (st, newst) is in the relation for the method "deq".

    In that case, it means that the "deq" method called in state st, changes the internal
    state to newst, that is, it removes the first element of the list in st.
*)

Definition enq arg st newst :=
    ∃ t, st = {| T := list N; state := t |} ∧
    newst = ({| T:= list N; state := app t [arg] |}).
```

```
Global Instance mkQueue : module _ :=
  primitive_module#(vmet [ first ] amet [ enq; deq ]).
```

Note that the predicate describing the relation for the `deq` method implies that the method cannot be called if the internal state is not of the form `cons head l`. This is an example of a partial relation and hence an example of a guard: one can only dequeue from this module if there is an element in it.

It is also interesting to remark that those relations are not necessarily functional relations. We commonly will capture nondeterminism in those predicates, as with the following two examples:

Non-deterministic even number: The following is a module that has a single value method that can return an arbitrary even number:

```
Definition get_even _arg _st ret :=
    ∃ n, ret = 2 * n .


Global Instance mkNonDetEven : module _ :=
  primitive_module#(vmet [ get_even ]).
```

In this case, there might be more than one possible return value. Similarly to this non-deterministic *value method*, a module could have a non-deterministic action method, where the new state is not functionally determined from the old state.

Note that the nondeterminism can be controlled by an arbitrary logical formula, for example, the following artificial example:

Flimsy register

```
Definition write arg st newst :=
   ∃ old_value,
   st = {| T := N; state := old_value |} ∧
   (newst = {| T := N; state := arg |} ∨ newst = {| T := N; state := old_value |}) .
```

```
Definition read _arg st ret :=
 (* when a method does not use its argument we name it _arg *)
   ∃ old_value,
   st = {| T := N; state := old_value |} ∧ ret = old_value.

Global Instance mkFlimsyReg : module _ :=
  primitive_module#(vmet [ read ] amet [ write ]).
```

This primitive module, only presented for its pedagogical value, is a weird register that is nondeterministically faulty: when one writes to a FlimsyReg, either the register is updated with the value written, or nothing happens and the old value is kept in the register.

To summarize, we describe the semantics of a primitive module by describing the transition relations as logical predicates, described within the fairly powerful logic of Coq (higher-order logic, where arbitrary nested exists and forall quantification is allowed). There are two uses for such descriptions: one is to axiomatize the behavior of an external IP block, or a basic element that we don't have access to; the other one is to introduce abstract mathematical intermediate models for verification purposes. In this second scenario, we will have to *prove* that our original design (for example a queue made up of registers) is indeed in a sense similar to our simplified axiomatized model. That will leads us to the formal notion of simulation (also named refinement), which is the key to formalize in which sense two modules are similar in Fjfj and will be introduced in chapter 4. But before that, let's describe formally the syntax and the semantics of Fjfj and how we can formally build the relation corresponding to the transition relations of a parent module when we have access to the transition relations of the submodules.

3.6 Formal semantics

3.6.1 Syntax

The grammar of Fjfj is given in Figure 3-1.

```
value_expr ::=
| v (* Variable *)
| c (* Constants *)
| (f value_expr value_expr)
   (* Pure combinational function, we present only the binary case *)
| (set v value_expr)  (* set variable *)
| (if value_expr value_expr value_expr)
| abort
| {value_method instance value_expr}
  (* Notice the difference with a combinational function call *)

action_expr ::=
  | (if value_expr action_expr action_expr)
  | value_expr (* Make an action_expr from a value_expr *)
  | (begin
      action_expr
      ...
      action_expr)
  | pass
  | {action_method instance value_expr}

base ::=
   (action_method (v1 ... vn) action_expr)
   (value_method (v1 ... vn) value_expr)
   (rule  action_expr)
```

Figure 3-1: Syntax of Fjfj

```
Local Instance submodules : instances :=
    #|
    Reg.mkReg a;
    Reg.mkReg b;
    QueuePkg.mkQueue f;
    |#.

Definition myr :=
    (rule
        (begin
            (set va {read a})
            (set va {read b})
            {write a (+ va ba)})
```

Figure 3-2: Example to illustrate the mechanics of the formal semantics

3.6.2 Semantics

The goal of the formal semantics is to specify what value gets returned by each value method (and when value methods can be called) and what are the state updates caused by the rules and action methods. That is, we want to define the transition relation of rules and methods, from pieces of Fjfj syntax.

For the rest of this section when illustrating different aspects of the semantics we will use the minimal example of Figure 3-2. It does not have any action or value methods; it simply consists of a rule.

Submodule semantics/Submodule state First, let's describe the hierarchical structure of the state. It reflects the hierarchical structure of the module instantiation, and it will be used by the semantics. In our example the state of the module is a dictionary indexed by keys that are the names of the instances of submodules (`a` and `b` here), and the value is the current value held by the submodule. For example we could have a state *st* with:

$$st = \{a : 1, b : 1, f : []\}$$

Note that if the module contained Fjfj submodules (nonprimitive ones) we could access those dictionaries hierarchically. We could for example, for a parent module that would contain two submodule instances of our example module, imagine a state $i = \{m1 : \{a : 1, b : 1, f : []\}, m2 : \{a : 2, b : 2, f : [3]\}\}$. We use standard notations to refer to the state corresponding to a submodule instance. For example, $st[a]$ is 1, and $st[f]$ is the empty list, while $i[m2][f]$ is [3].

The reader should not confuse a *module state* with a *module semantics*. One is data and gets transformed by the relations, the other one is more like code (the set of 4 relations we defined in section 3.5) and explains how those state values get transformed and observed.

Collecting action-method calls We will collect the action methods called by a current action, which must be a rule or an action method of the parent module. We build a map α that gives for each submodule which of its action methods are being requested by the parent action. Remember that at most one action method can be called for each submodule. The bookkeeping for α will ensure this constraint holds.

In our example of Figure 3-2, the α for rule `myr` when $st = \{a : 1, b : 1, f : []\}$ will be $\alpha = \{a : \{\text{method} : \text{write}; \text{arg} : 2\}\}$. Note that for each instance, we record a struct with two fields: one field `method` to keep track of the name of the method called, and one field `arg` to keep track of the argument that the action tried to call the submodule with. It is important to note that at this stage we do not perform any update for the substate of the submodule corresponding to the instance register a. We simply remember that the action wants to perform method write with argument arg on the submodule instance a.

In general, when there is an action method being called on a submodule (at most one such call can exist on every submodule), we will write $\alpha(\text{inst}).\text{method}$ to refer to which action method was called for submodule inst, and $\alpha(\text{inst}).\text{arg}$ to refer to the argument to the method. Here inst is short for *instance*.

On top of building α, the semantics also need to keep track of the local variables defined by the rules, in Γ, and keep track in β of the set of value methods that have been called, to make sure that no value method is called twice. In our example, the Γ and β derived for our rule r when $st = \{a : 1, b : 1, f : []\}$ will be $\Gamma = \{va : 1, vb : 1\}$, and $\beta = \{(\text{a}, \text{read}), (\text{b}, \text{read})\}$. This

indicates the set of *value methods* called during the derivation for the rule. If a subexpression were to try to reuse a value method, it would be caught in this set.

We write the judgment $(\alpha_1, \Gamma_1, \beta_1) \dashv\!\text{action_expr}\!\mapsto (\alpha_2, \Gamma_2, \beta_2)$ to indicate that P transforms an initial triple of environment, requested action calls and requested value calls into a new triplet. Note that action expressions do not return values, they only have side effects. Putting everything together for our example, if we started in a state $st = \{a : 1, b : 1, f : []\}$ we could derive:

$$(\emptyset, \emptyset, \emptyset) \dashv\!\text{myr}\!\mapsto (\{a : \{\text{method} : \text{write}; \text{arg} : 2\}\}, \{va : 1, vb : 1\}, \{(\text{a}, \text{read}), (\text{b}, \text{read})\})$$

And more generally if we start in an arbitrary *symbolic state* $st = \{a : x, b : y, f : l\}$ we can derive:

$$(\emptyset, \emptyset, \emptyset) \dashv\!\text{myr}\!\mapsto (\{a : \{\text{method} : \text{write}; \text{arg} : x + y\}\}, \{va : x, vb : y\}, \{(\text{a}, \text{read}), (\text{b}, \text{read})\})$$

We also define a similar judgment for value expressions, which instead of returning an updated α datastructure[1], returns a value. We write $(\Gamma_1, \beta_1) \dashv\!\text{value_expr}\!\mapsto_{\text{value}} (\Gamma_2, \beta_2, ret)$ to indicate that the value expression takes a pair of environments and a set of *value* method calls performed so far, relating them to new environments and new sets of value-method calls performed, plus a value returned by the value expression: *ret*. The rules for $* \dashv\!*\!\mapsto_{\text{value}} *$ and $* \dashv\!*\!\mapsto *$ are given in Figure 3-3 and Figure 3-4 respectively.

Remark (Using state in judgment rules). Both of those judgments can make reference to the state of the submodules (the instances), *e.g.* $st[a]$. However, we omit *st* in the notation to avoid clutter, as it is kept constant during the derivation. Note that *st* is only used in the two judgment rules for calling a submodule.

Explanations for Figure 3-3 Let's first explain one typical rule of Figure 3-3 (the `SetVar` rule) and then the interesting case (`Call`). If we look at the `SetVar` case, the rule from top to bottom reads as:

[1] which would not make sense as value expressions do not request actions on submodules

- If in a value expression e transforms an environment, of local variables Γ and of already-seen called value methods β, into a new environment of local variables Γ' and add extra value-method calls to β to transform it into β', while returning value r

- Then setting the value expression e in a variable x transforms an environment Γ and a set of observed value methods β into a new environment obtained by updating Γ' by binding x to value r, the set of observed value methods β' and the full `set` expression returns r.

This is simply the formal encoding of what one might expect from those traditional programming-language constructs. Note that technically the language allows nested sets: `(set x (set y 3))`, which would set both variable x and y to 3. In practice, we never use such a construction.

The main interesting case is the `Call` case. A call to a value method of a submodule is possible only if this precise value method was not already called so far (it is not in β'). Moreover, we need to look at the semantics of the corresponding value methods of the submodule *e.g.* $[\![\text{instance}]\!].\text{vmethods}[\text{vm}]$. By definition this object is a set of triples (sub_{st}, arg, ret), such that $(sub_{st}, arg, ret) \in [\![\text{instance}]\!].\text{vmethods}[\text{vm}]$ whenever calling the value method with argument arg when the submodule has state sub_{st} returns the value ret.

In practice, because this set is described by a predicate, the membership test is simply declaring that (sub_{st}, arg, ret) verifies some predicate that constrains the return value. Note, as was pointed out in section 3.5, that if a pair of a state and an argument is not in the domain of the relation, then the method cannot be called. We say the method's guard is false in Bluespec terminology.

Explanations for Figure 3-4 The details are similar to the previous case. It is worth noting that we don't directly perform the state update for all submodules. Instead, we record each action method that was called for each submodule. In the second phase of the semantics that we are to present in the next paragraph, we will actually build the new state.

Remark (Guards and partial relations). The judgments we just presented can be derived only if all the value methods *and* all the action methods of the submodules that are called accept to produce results, or accept to be be called for the action methods, *i.e.* if the guards

$$\frac{}{(\Gamma,\beta) \dashv\!\![\mathsf{v}]\!\!\mapsto_{\text{value}} (\Gamma,\beta,\Gamma[\mathsf{v}])}\textsc{Var}$$

$$\frac{(\Gamma,\beta) \dashv\!\![e]\!\!\mapsto_{\text{value}} (\Gamma',\beta',r)}{(\Gamma,\beta) \dashv\!\![(\text{set } \mathtt{x}\ e)]\!\!\mapsto_{\text{value}} (\Gamma'[\mathtt{x}\ \ \mathtt{:=}r],\beta',r)}\textsc{SetVar}$$

$$\frac{(\Gamma,\beta) \dashv\!\![e]\!\!\mapsto_{\text{value}} (\Gamma',\beta',1) \qquad (\Gamma',\beta') \dashv\!\![t]\!\!\mapsto_{\text{value}} (\Gamma'',\beta'',r)}{(\Gamma,\beta) \dashv\!\![(\text{if } e\ t\ f)]\!\!\mapsto_{\text{value}} (\Gamma'',\beta'',r)}\textsc{IfVT}$$

$$\frac{(\Gamma,\beta) \dashv\!\![e]\!\!\mapsto_{\text{value}} (\Gamma',\beta',0) \qquad (\Gamma',\beta') \dashv\!\![f]\!\!\mapsto_{\text{value}} (\Gamma'',\beta'',r)}{(\Gamma,\beta) \dashv\!\![(\text{if } e\ t\ f)]\!\!\mapsto_{\text{value}} (\Gamma'',\beta'',r)}\textsc{IfVF}$$

$$\frac{(\Gamma,\beta) \dashv\!\![e_1]\!\!\mapsto_{\text{value}} (\Gamma',\beta',r_1) \qquad (\Gamma',\beta') \dashv\!\![e_2]\!\!\mapsto_{\text{value}} (\Gamma'',\beta'',r_2)}{(\Gamma,\beta) \dashv\!\![(f\ e_1\ e_2)]\!\!\mapsto_{\text{value}} (\Gamma'',\beta'',f(r_1,r_2))}\textsc{PureF}$$

$$\frac{(\Gamma,\beta) \dashv\!\![e]\!\!\mapsto_{\text{value}} (\Gamma',\beta',v_{arg}) \qquad (\text{inst},\text{vm}) \notin \beta' \qquad (st[\text{inst}], v_{arg}, ret) \in [\![\text{inst}]\!].\text{vmethods}[\text{vm}]}{(\Gamma,\beta) \dashv\!\![\{\text{vm inst } e\}]\!\!\mapsto_{\text{value}} (\Gamma',\beta' \cup \{(\text{inst},\text{vm})\}, ret)}\textsc{Call}$$

Figure 3-3: Judgment rules for $* \dashv\!\![*]\!\!\mapsto_{\text{value}} *$. It tracks the effect of value expressions on the environment, on the data structure tracking the value methods used by the expression, and the returned value.

of the value methods and the action methods are ready.

We now have all the ingredients to define the derived semantics of rules, action methods, and value methods.

Derived semantics of rules A transition from state st to state new_{st} by rule R, noted $st \to_R new_{st}$, is defined as :

$$\begin{aligned}
&\exists \alpha, (\emptyset,\emptyset,\emptyset) \dashv\!\![R]\!\!\mapsto (\alpha,*,*) \wedge \\
&\qquad \forall \text{ inst} \in Dom(\alpha), (st.\text{inst}, \alpha(\text{inst}).arg, new_{st}.\text{inst}) \in [\![\text{inst}]\!].\text{amethods}[\alpha(\text{inst}).\text{method}] \\
&\qquad \forall \text{ inst} \in Dom(st),\ \text{inst} \notin Dom(\alpha), (st.\text{inst} = new_{st}.\text{inst})
\end{aligned} \tag{3.1}$$

This definition might look intimidating but it says the intuitive thing: in plain words, a new state is a valid transition when *every new submodule state either corresponds to a transition of an action method being called on the submodule if an action is called on the submodule, or the submodule is left unchanged if no action was called on the submodule.*

$$\frac{(\Gamma, \beta) \dashv\!\![e]\!\!\mapsto_{\text{value}} (\Gamma', \beta', _)}{(\alpha, \Gamma, \beta) \dashv\!\![e]\!\!\mapsto (\alpha, \Gamma', \beta')}\textsc{Expr}$$

$$\frac{}{(\alpha, \Gamma, \beta) \dashv\!\![\text{pass}]\!\!\mapsto (\alpha, \Gamma, \beta)}\textsc{pass}$$

$$\frac{(\Gamma, \beta) \dashv\!\![e]\!\!\mapsto_{\text{value}} (\Gamma', \beta', 1) \qquad (\alpha, \Gamma', \beta') \dashv\!\![t]\!\!\mapsto (\alpha', \Gamma'', \beta'')}{(\alpha, \Gamma, \beta) \dashv\!\![(\text{if } e\ t\ f)]\!\!\mapsto (\alpha', \Gamma'', \beta'')}\textsc{IfT}$$

$$\frac{(\Gamma, \beta) \dashv\!\![e]\!\!\mapsto_{\text{value}} (\Gamma', \beta', 0) \qquad (\alpha, \Gamma', \beta') \dashv\!\![f]\!\!\mapsto (\alpha', \Gamma'', \beta'')}{(\alpha, \Gamma, \beta) \dashv\!\![(\text{if } e\ t\ f)]\!\!\mapsto (\alpha', \Gamma'', \beta'')}\textsc{IfF}$$

$$\frac{(\Gamma, \beta) \dashv\!\![e]\!\!\mapsto_{\text{value}} (\Gamma', \beta', ret) \qquad \text{inst} \notin Dom(\alpha) \qquad (st[\text{inst}], ret, _) \in [\![\text{inst}]\!].\text{amethods}[\text{am}]}{(\alpha, \Gamma, \beta) \dashv\!\![\{\text{am inst } e\}]\!\!\mapsto (\alpha \cup \{\text{inst} \mapsto \{\text{method: am; arg: } ret\}\}, \Gamma', \beta')}\textsc{CallAct}$$

Figure 3-4: Judgment rules for $* \dashv\!\![*]\!\!\mapsto *$ tracking the effect of action expressions on the environment, the data structure tracking the value method used and the one tracking the action methods requested.

This allow us to define the semantics of a rule R as:

$$[\![R]\!] := \{(st, new_{st}) | st \rightarrow_R new_{st}\}$$

As an example, let's give the semantics of the rule of our example in Figure 3-2. Remember that we already previously derived the only possible derivation for `myr` starting in an arbitrary state of the right shape (containing the expected instances) $st = \{a : *, b : *, f : *\}$. It was the following:

$$(\emptyset, \emptyset, \emptyset) \dashv\!\![\text{myr}]\!\!\mapsto (\{a : \{\text{method : write; arg : } st.\text{a}{+}st.\text{b}\}\}, \{va : st.\text{a}, vb : st.\text{b}\}, \{(\text{a}, \text{read}), (\text{b}, \text{read})\})$$

We can plug that expression into Equation 3.1 to get the following reasoning and obtain the semantics of the rule `myr`:

$$\begin{aligned}
\text{myr} &= \{(st, new_{st}) | \quad st \rightarrow_{myr} new_{st}\} \\
&= \{(st, new_{st}) | \quad (st.\text{a}, st.\text{a} + st.\text{b}, new_{st}.\text{a}) \in [\![\text{a}]\!].\text{amethods}[\text{write}] \\
&\qquad\qquad\qquad\qquad \wedge\ new_{st}.\text{b} = st.\text{b} \wedge new_{st}.\text{l} = st.\text{l}\} \\
&= \{(st, new_{st}) | \quad new_{st}.\text{a} = st.\text{a} + st.\text{b} \wedge new_{st}.\text{b} = st.\text{b} \wedge new_{st}.\text{l} = st.\text{l}\}
\end{aligned}$$

The last line is obtained by remembering the semantics of a register write $[\![\mathrm{a}]\!]$.amethods[write] presented in section 3.5.

Derived semantics of action methods Very similarly to the semantics of a rule, we can use the same judgment rules to define the transitions for an action method, $st \rightarrow_{am,arg} new_{st}$ by:

$$\begin{aligned}
&\exists \alpha, (\emptyset, \{\mathrm{arg} : arg\}, \emptyset) \dashv\!\![am]\!\!\mapsto (\alpha, *, *) \wedge \\
&\quad \forall \ \mathrm{inst} \in Dom(\alpha), (st.\mathrm{inst}, \alpha(\mathrm{inst}).arg, new_{st}.\mathrm{inst}) \in [\![\mathrm{inst}]\!].\mathrm{amethods}[\alpha(\mathrm{inst}).\mathrm{method}] \\
&\quad \forall \ \mathrm{inst} \in Dom(st), \mathrm{inst} \notin Dom(\alpha), st.\mathrm{inst} = new_{st}.\mathrm{inst}
\end{aligned} \tag{3.2}$$

And as expected we pose the semantics of an action method as follows:

$$[\![\mathrm{am}]\!] := \{(st, arg, new_{st}) |\ st \rightarrow_{am,arg} new_{st}\}$$

Derived semantics of value methods Finally, the value methods (which only return a value and do not need to update any state) return values given by the already-explained judgment $* \dashv\!\![*]\!\!\mapsto_{\mathrm{value}} *$:

$$st \rightarrow_{vm,arg} ret ::= (\emptyset, \{\mathrm{arg} : arg\}, \emptyset) \dashv\!\![vm]\!\!\mapsto_{\mathrm{value}} (*, *, ret)$$

They are easier than action methods and rules as they do not update any state. We define the semantics of value methods as expected:

$$[\![\mathrm{vm}]\!] := \{(st, arg, ret) |\ st \rightarrow_{vm,arg} ret\}$$

Remark (Notation clarification). The notation $* \rightarrow_* *$ means something different if the piece of syntax is a value expression or if it is an action expression. In the first case, it returns a value (a bitvector), in the second it returns a new state.

Sometimes it is useful to state that there exists a sequence of rules that goes from one state to another. We use the notation $x \rightarrow_M^* x'$ to signify that there exists a sequence of

rules of M to go from state x to x'.

We now have given all the elements that define the semantics of an Fjfj module from the semantics of the submodules and the syntax of the rules and methods of the module.

3.6.3 General definition of a new Fjfj module

Given:

1. $S = \{\mathbb{m}_i | i \in [0..k]\}$, a family of submodule semantics (each composed of an internal state type and 3 sets of relations, see section 3.5)
2. $R = \{r_i | i \in [0..n_{rules}]\}$ action expressions (pieces of syntax)
3. $V = \{vm_i | i \in [0..n_{vmethods}]\}$ value expressions (pieces of syntax)
4. $A = \{am_i | i \in [0..n_{amethods}]\}$ action expressions (pieces of syntax).

We define the semantics of a non-primitive module (S, R, V, A) as follow:

$$\begin{gathered}
[\![(S, R, V, A)]\!] = \{ \\
\text{Internal type} := \mathbb{m}_0.T \times \ldots \mathbb{m}_k.T; \\
\text{Rule relations} := \{\mathbb{r}_i | \mathbb{r}_i = [\![r_i]\!]\}; \cup_l \text{lift}(\mathbb{m}_l.rules); \\
\text{Action-method relations} := \{\mathbb{am}_i | \mathbb{am}_i = [\![am_i]\!]\}; \\
\text{Value-method relations} := \{\mathbb{vm}_i | \mathbb{vm}_i = [\![vm_i]\!]\} \\
\}
\end{gathered}$$

Note that we do not directly use the *syntax* of the submodules (which might not even exist if the submodules are primitive modules) to define the semantics of the parent module. In that sense the semantics is modular.

3.7 Discussion

3.7.1 Trace semantics versus transition systems

In this chapter, we defined the semantics of modules: the relations defining the value and action-method transitions and the relations defining the rule transitions.

As we mentioned, the semantics of the object can be thought of as a labelled transition system (LTS), and from that transition system one can derive a set of traces, *i.e.* the set of sequences of *atomic interactions* that are valid interactions in the LTS.

One might wonder if instead of considering the semantics of a module as the collection of relations, one could not simply consider the set of atomic interactions as the semantics of a module.

This can look appealing as we can then say that a module is defined by the set of traces of interactions that it admits. It is indeed possible to define a language around the set of traces, but this introduces multiple difficulties. The composition operator (calling a submodule) becomes more complicated. Among other things it introduces causality and issues of monotonicities: we can easily define a module where in a situation, one of its methods might be allowed to be called only if something happens in the future.

The metatheory of such a language becomes more difficult, especially when it comes to comparing modules (see the discussion in chapter 4 on the refinement theorem), and one always needs to worry about nonintuitive and degenerate modules.

So while implicitly we will often think of the set of traces of atomic interactions acceptable to a module, those are not the *definition* of the semantics of a module.

3.7.2 Multiple calls to the same value methods

While we motivated why we need to prevent having a module call two action methods of one of its submodules, we have not yet motivated why we have a restriction on calling the *same* value method twice. Indeed the β structure makes sure to abort any rule or method that would perform a double call to a value method.

This comes from a concern about modular compilation: while semantically we could

allow to calling a value method several times per rule (or method) without causing trouble with atomicity, separate hardware compilation requires a module interface to have a fixed number of wires.

That is, we want to be able to compile a submodule to Verilog without even knowing the parent using the module, and each separate call by the parent rule would need a different set of wires to materialize the call, as we need to communicate the arguments of the call which might be different for the different call sites and receive different returned values.

To avoid that issue, we simply restrict to having no two dynamic calls to the same value method per rule or method. There might be syntactically more than one call site (due to conditionals), but only one is dynamically active each cycle, and we would simply mux the arguments and the returned value.

Hence, if one wants to use several times the same value method, the value method should be defined multiple times. For example, a register file with two read ports will necessarily have defined two read methods (which will have twice the same body), as we cannot call twice the same read method. For uniformity and because every one of our methods has exactly one argument, we enforce this constraint for every method, including value methods that have zero arguments, even though that case could technically be handled.

Chapter 4

Correctness and Refinement

In the previous chapter we presented Fjfj. Now we introduce verification in Fjfj. While standard sequential software verification is usually about establishing Hoare triples (pre/-postconditions of program fragments), verification in Fjfj is about constructing simulation relations between implementation systems and specification systems.

Our fundamental verification task will consist in writing two modules, one being our actual implementation design, the other one being the specification, and showing that every sequence of interactions with the implementation (action method calls and value method calls leading to observations) could also have happened the same way if we had instead interacted with the specification.

In such a case, we would say that *the specification simulates the implementation* or equivalently that *the implementation refines the specification.*

Often we will do this exercise hierarchically to propose modular specifications of systems.

In this chapter, we will define what those simulations precisely are and start by giving simple examples. We will then state and prove a key theorem of Fjfj: the refinement theorem. In a nutshell, it is a substitution principle for Fjfj.

4.1 Simulations as refinement

4.1.1 State simulation

Let M_I and M_S be two Fjfj modules which expose the same *interface*, *i.e.* , they have the same value methods and action methods. Let their corresponding semantics be $[\![M_I]\!], [\![M_S]\!]$, respectively.

Let i and s be states for the implementation and specification modules respectively. We say that the module M_I in state i can *simulate* the module M_S in state s, when:

1. For every value method *vm*,

$$\forall\ arg\ ret\ .\ (i, arg, ret) \in [\![M_I]\!].\text{vmethods}[vm] \Rightarrow (s, arg, ret) \in [\![M_S]\!].\text{vmethods}[vm]$$

   In other words, all value methods ready in the implementation are also ready in the specification, and all results that can be returned by the implementation (there might be several due to nondeterminism) are also possible results returned by the specification.

2. For every action method am,

$$\forall\ arg\ i'\ .\ (i, arg, i') \in [\![M_I]\!].\text{amethods}[am] \Rightarrow \exists s'.(s, arg, s') \in [\![M_S]\!].\text{amethods}[am]$$

   That is, all the action methods that are ready in the implementation are also ready in the specification.

In such a situation, we write $i\ {}_{M_I}\!\!\prec_{M_S} s$. $\prec$ is a relation between states of the implementation and states of the specification. Note that $\prec$ is *not* a symmetric relation. Intuitively it says that if a module M_I is in state i, nothing that it can do cannot be done by a module M_S in state s. We will often omit M_I and M_S in the notation and write $i \prec s$ when they are obvious from the context.

4.1.2 Module simulation

We say that M_I simulates M_S along the relation $\phi \subset State_{implementation} \times State_{specification}$, noted $M_I \sqsubseteq_\phi M_S$, when:

1. For every initial state i of the implementation, there exists an initial state s of the specification, such that $i \prec s \wedge \phi\ i\ s$

2. For every i_1 and s_1 such that $i_1 \prec s_1 \wedge \phi\ i_1\ s_1$, and for every r and i_2 such that $(i_1, i_2) \in [\![M_I]\!].\text{rules}[r]$ there exists a sequence of rules (and corresponding intermediate states) $\text{r}_1, \ldots, \text{r}_k,\ s_2 \ldots s_k$, s.t.

$$\forall j \in [1..k], (s_j, s_{j+1}) \in [\![M_S]\!].\text{rules}[r_j] \ \wedge$$

$$i_2 \prec s_k \wedge \phi\ i_2\ s_k$$

3. For every i_1 and s_1 such that $i_1 \prec s_1 \wedge \phi\ i_1\ s_1$, and for every am, arg and i_2 such that $(i_1, arg, i_2) \in [\![M_I]\!].\text{amethods}[am]$, there exists s_2 such that $(s_1, arg, s_2) \in [\![M_I]\!].\text{amethods}[am]$ and there exists a sequence of rules (and corresponding intermediate states) $\text{r}_2, \ldots, \text{r}_k,\ s_2 \ldots s_k$, such that,

$$\forall j \in [2..k], (s_j, s_{j+1}) \in [\![M_S]\!].\text{rules}[r_j] \ \wedge$$

$$i_2 \prec s_k \wedge \phi\ i_2\ s_k$$

Conditions 2. and 3. can be a mouthful to digest. We can more easily represent them with the commutative diagrams of Figure 4-1.

Remark (Refinement is a weak simulation)*.* This is a form of weak simulation (not a bisimulation as the relation is again asymetric).

We follow the traditional naming scheme to overload the word *simulates*, both for the states and for the whole module. In the rest of the dissertation, most reference to simulation will refer to the full modules. Finally we note $M_I \sqsubseteq M_S$ whenever $\exists \phi, M_I \sqsubseteq_\phi M_S$.

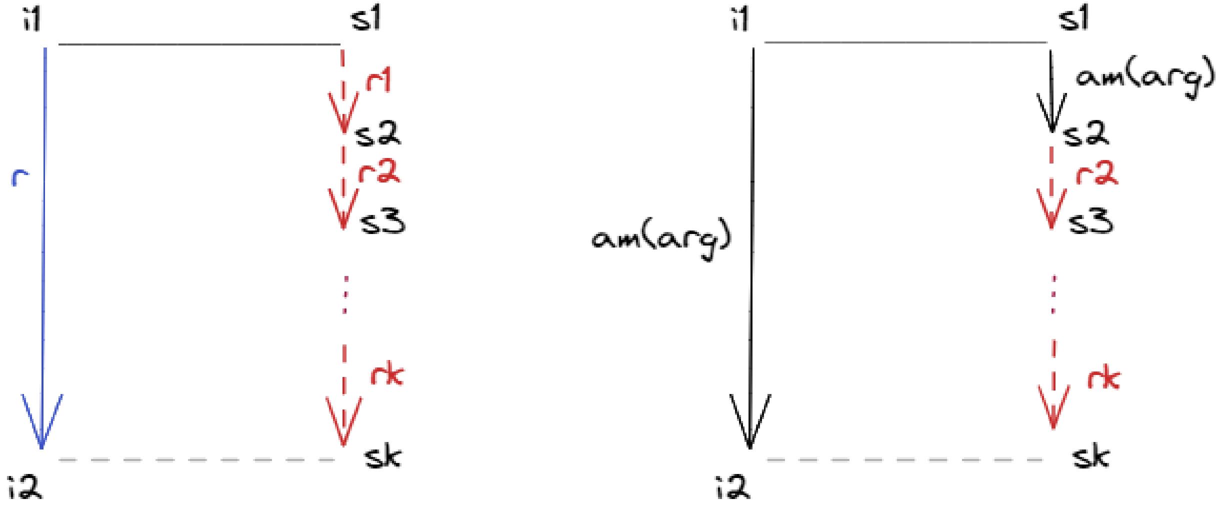


Figure 4-1: Key conditions of simulation relations. Full arrows and full lines represent preconditions that are universally quantified, while dashed lines designate existential obligations. Blue-colored transitions are rule transitions that only exist in the implementation machine, and red ones are rule transitions that only exist in the specification machine. Black transitions are the method transitions, which exist in the two machines. Gray lines indicate a relation between an implementation state and a specification state: the implementation simulates the specification, and they are related by the relation ϕ

4.1.3 Basic properties of simulations

Theorem 4.1 (Reflexivity of $\prec$). *For every module M and state i, $i\ {}_M{\prec}_M i$.*

Proof. Property 1 of subsection 4.1.2 is straightforward, and Property 2 follows by picking $s' = i'$.

Theorem 4.2 (Transitivity of $\prec$). *Let i_1, i_2, i_3 be three states of three modules M_1, M_2, M_2. If $i_1\ {}_{M_1}{\prec}_{M_2} i_2$ and $i_2\ {}_{M_2}{\prec}_{M_3} i_3$ then $i_1\ {}_{M_1}{\prec}_{M_3} i_3$.*

Proof. Follows directly from the definition.

Theorem 4.3 (Reflexivity of $\sqsubseteq$). $\forall M, M \sqsubseteq M$.

Proof. Take $\phi(x, y) := x = y$.

Theorem 4.4 (Transitivity of $\sqsubseteq$). *Let M_1, M_2, M_3 be three modules such that $M_1 \sqsubseteq M_2$ and $M_2 \sqsubseteq M_3$; then $M_1 \sqsubseteq M_3$.*

Proof. Let ϕ_{1-2} be a simulation witnessing $M_1 \sqsubseteq_{\phi_{1-2}} M_2$ and ϕ_{2-3} be a simulation witnessing $M_2 \sqsubseteq_{\phi_{2-3}} M_3$, then picking $\phi_{1-3}(x,z) := \exists y, \phi_{1-2}(x,y) \wedge \phi_{2-3}(y,z)$, we can verify that ϕ_{1-3} is a simulation of M_1 by M_3.

4.2 Simple simulations, step-by-step

For practice, let's give a few examples of pairs of modules (implementations and specifications), where the proof of refinement between implementation and specification is very straightforward. It will give us an opportunity to get accustomed to our framework and what it looks like in the proof assistant.

The reader will see more elaborate proofs of refinement that would have some interesting computer-architecture contents in chapter 7.

4.2.1 Register file

Here is a specification for a one-port register file.

```
Definition read_rf arg st ret :=
    (* The notation *( rf )* stands for the state
    of a primitive module that contains the value rf.
    The notation is not to be confused with comments in coq *)
    ∃ (rf: N → N), st = *( rf )* ∧
    ret = rf arg.

Definition write_rf arg st newst :=
    ∃ (rf: N → N), st = *( rf )* ∧
    dlet {idx val} := arg in
    newst = *( fun addr ⇒ if addr =? idx then val else rf addr )*.

Global Instance mkRfSpec : module _ :=
    primitive_module#(vmet [ read_rf ] amet [ write_rf ]).
```

Implementation Here is an implementation, of a 4-register register file.

```
Local Instance rf_submodules : instances :=
#|
    mkReg r0;
    mkReg r1;
    mkReg r2;
    mkReg r3
|#.

Definition write_rf :=
  (action_method (idx val)
    (begin
      (if (< idx 4)
        pass
        abort)
      (if (= idx 0)
        {write r0 val}
        pass)
      (if (= idx 1)
        {write r1 val}
        pass)
      (if (= idx 2)
        {write r2 val}
        pass)
      (if (= idx 3)
        {write r3 val}
        pass))).

Definition read_rf :=
  (value_method (idx)
```

```
  (begin
    (if (< idx 4)
      pass
      abort)
    (set res 0)
    (if (= idx 0)
      (set res {read r0})
      pass)
    (if (= idx 1)
      (set res {read r1})
      pass)
    (if (= idx 2)
      (set res {read r2})
      pass)
    (if (= idx 3)
      (set res {read r3})
      pass)
    res)).

Global Instance mkRf : module _ :=
  module#(vmet [ read_rf ] amet [ write_rf ]).
```

Remark (Comparing internal data-structures). In the specification, the data held by the register file is recorded in a function. Note that this specification covers potentially infinitely many registers. In contrast, the implementation is made of a finite number of registers (4 here), and the selection between those registers is basically a mux-tree.

Clearly, the specification is not synthesizable as the state space of the specification is infinite, and the benefit of the specification compared to the implementation is mainly the absence of a mux-tree in the specification. Hence, it is easier/cheaper to evaluate the specification symbolically than to evaluate the implementation symbolically. This is a common pattern when verifying hardware modules: a module will often benefit from being written

differently when written for synthesis or for verification. For synthesis, only synthesizable constructs can be used, and the critical path and area are the factors to optimize for performance. In contrast, when a module is written for verification, one does not care for critical path and area but typically one care mainly about ease of stating invariants and ease of symbolic evaluation. In the verification setup, one might make use of arbitrary non-synthesizable structures, like first-class functions.

Thanks to refinement, we can often write a specification module allowing us to use convenient and appropriate data structures (often non-synthesizable) when doing verification. This register file specification is such an example.

Theorem 4.5 (Register file refinement). $mkRf \sqsubseteq mkRFSpec$

1. Picking a simulation relation candidate ϕ In this case, let's pick

```
Definition ϕ (ist : SModule) (sst : SModule) :=
  ∃ (rfs: N → N) (r0 r1 r2 r3 : N),
    sst = *( rfs )* ∧
    ist = *[| r0; r1; r2; r3 |]* ∧
    rfs 0 = r0 ∧ rfs 1 = r1 ∧ rfs 2 = r2 ∧ rfs 3 = r3.
```

Let's remember that `*( something )*` stands for the state of a *primitive* module that would contain `something`. (That is to not be confused with `(* this is a comment *)`). Let's also introduce a related notation for the state of *nonprimitive* modules. The state of a *nonprimitive* is made of several substates (one per submodule), for example, we note `*[| a; b; c |]*` for a module state containing 3 submodules which are containing `a`, `b` and verb|c|. Sometimes we have a hierarchy of states so we end up writing `*[| a; [| e;f |]; c |]*` to say that the second submodule is itself a nonprimitive module that contains two primitive modules containing `e` and `f`.

This ϕ reads as: the specification state is of the shape of a primitive state that contains a function *rfs* from $\mathbb{N}$ to $\mathbb{N}$, the implementation state is of the shape of a nonprimitive module state composed of 4 primitive submodule states $r0, r1, r2$, and $r3$, such that the function *rfs*

is equal to r_i when its argument is i, i.e. the specification register file contains in its first 4 registers the same value as the 4 implementation registers.

2. Showing that $\phi\ i\ s$ implies $i \prec s$ The most difficult part of that proof is to understand the simple statement we are trying to prove. We have to prove that for any state of register files that are related by ϕ, reading in the implementation register file would return a value that would be the same as reading the specification register file. There is a small subtlety: the implementation register file has only 4 registers, while the specification register file has infinitely many of them. So how can that property be true? The trick is the *guard* on the implementation register file: one can never read the implementation register file at any index higher or equal to 4. Hence, if we assume that we successfully read the implementation register, necessarily the index was smaller than 4.

Now that the reader understands why the property is true and intuitively trivial, let's elaborate step-by-step on this easy proof to explain how we carry out the proof in the proof assistant.

Indeed, as the reader might already know, even elementary properties can sometimes require a disproportionate amount of effort to prove. To avoid this potentially demoralizing fact, one solution is to build a set of *tactics* (or proof-script automations) that are hopefully able to handle those easy facts. So we have to provide suitable base tactics to make the proof assistant able to discharge proof obligations automatically, hopefully as fast as a standard human can prove reasonably obvious facts.

```
Lemma ϕ_implies_state_simul:
  ∀ ist sst, ϕ ist sst → ist ≺ sst.
  (* We start the proof by using our tactic "prove_state_simulation"
  that will generate all the proof obligations corresponding to the
  definition of "ist state is simulating sst". *)
  prove_state_simulation.
  {
    (* The first goal corresponds to proving that if we
    assume that we successfully call the [read_rf] method at an index
```

```
[arg] from the implementation register file (hypothesis named HB)
then we can prove (statement below the line) that we can call the
[read] method of the specification register file at index [arg],
and we will obtain the same value.

One can check that what is below the line is exactly the definition
of the [read] value method of the specification register
file returning [ret_valu]e: it is exactly the predicate we defined
for the semantics of the [read] value method when we defined this spec. *)
```

```
HB: array: [*( rfs 0 )*;
            *( rfs 1 )*;
            *( rfs 2 )*;
            *( rfs 3 )*] -(
   (begin
            (set idx anonymous_met_arg)
            (if (< idx 4) pass Abort)
            (set res 0)
            (if (= idx 0) (set res {read r0}) pass)
            (if (= idx 1) (set res {read r1}) pass)
            (if (= idx 2) (set res {read r2}) pass)
            (if (= idx 3) (set res {read r3}) pass)
            res) )→ ret_value
------------------------------------------------------------
∃ rf : N → N,
  *( rfs )* = *( rf )* ∧ ret_value = rf arg
```

```
(* We can use our symbolic-evaluation tactic to tackle this goal.
Our tactic, from the hypothesis HB, deduces that there are 5
possible "program paths" in which the semantics judgment
saying that the [read_rf] value method was successfully called
can be derived.
```

```
4 cases are very similar (corresponding to [arg] = 0, 1, 2 or 3), and
the last case ([arg] ⩾ 4) is actually absurd, as we will see
in a minute.

Let's display one of the 4 first cases that we made Coq generate
for us just after the call to our symb_ex_split tactic: *)
symb_ex_split.
```

```
Heqb0: 3 ≠ 2     Heqb2: 3 ≠ 0     HA0: (3 <? 4) = true     Heqb1: 3 ≠ 1
-------------------------------------------------------------------------
∃ rf : N → N,
  *( rfs )* = *( rf )* ∧ rfs 3 = rf 3
```

```
(* We can see that the tactic records all the hypotheses it accumulated
when symbolically running this program.
In this case, the case shown is for [arg] = 3, so we have in
the context the facts that the semantics passed on the false branch
of [arg] not being equal to any of 0,1 or 2, and
the fact that 3 is smaller than 4 (the first branch).

Note that the tactic also simplified the goal, replacing [arg]
in the goal by 3 (as in this case [arg] = 3) and replaced [ret_value]
by [rfs 3], as expected from a little symbolic evaluator.

Looking at the goal, it is trivial, and indeed is proven
easily by Coq automation (and the same for the other similar
3 cases), by the following tactic: *)
all: eauto.
(*
  Only one case is left: the case [arg] ⩾ 4.
  This case is interesting because we can see a set of hypotheses
  that is contradictory: no number is simultaneously smaller
  than 4 and neither 0, 1, 2 or 3.
```

```
  We use a little tactic to solve this kind of arithmetic goals:
*)
HA0: (arg <? 4) = true    Heqb2: arg ≠ 0    Heqb0: arg ≠ 2    Heqb: arg ≠ 3
Heqb1: arg ≠ 1
--------------------------------------------------------------------------
∃ rf : N → N, *( rfs )* = *( rf )* ∧ 0 = rf arg
  arithmetic_hammer.
}
(* We are done with the first proof obligation. *)
{
  (* The second case is absolutely trivial; it requires us to prove that
  if we assume that we can successfully call the [write_rf] action method
  of the implementation register file, then we can necessarily call the
  [write_rf] action method of the specification register file:
  in Bluespec terms, the guard is ready.

  As the [write_rf] action method of the specification register file can
  always be called, this obligation is trivial, and we don't bother
  to show it.  *)
  eauto.
}
(* We don't have any subgoals left, and we can declare the proof finished: *)
Qed.
```

Now is a good time to go back on an important principle of our approach to verification (that we mentioned in the introduction). Our framework (and more generally Coq), in contrast to standard model-checking frameworks, is not designed to perform very large case analysis. In the world of SMT solvers and model checkers, it is common to have the computer doing bit-blasting on dozens of bits, effectively doing case analysis over millions of cases in seconds. This bulldozer technique works well enough for those approaches but does not for us.

Our case-analysis techniques are much more modest. Our rule of thumb is never to attempt a proof strategy that would require more than a hundred cases. In practice this works well, because as we explained in the introduction, our philosophy is to consider proofs that are mimicking what we think about in our brains when considering the correctness of a design, and those can physically never involve too many cases. Even for our just-presented example, this little proof of a register file could also have been carried out without these 5 case splits.

3. Showing that ϕ is a witness of a simulation of the implementation by the specification Because we proved the previous lemma that ϕ implied state simulation, the only obligation we have left to prove is showing that $\phi\ i\ s$ is preserved by both action-method transitions and rule transitions, see Figure 4-1.

Let's start with the statement of the theorem:

```
Theorem correct : refines mkRf4 mkRfSpec ϕ.
  (* We start by using the proof tactic [prove_refinement]. This tactic unfolds
  the definition of [refines] and creates one subgoal per action method and per
  rule of the implementation module. The content of those subgoals is to prove that [ϕ]
  is preserved by all transition relations and that state simulation is preserved.
  As we already showed a lemma that [ϕ] implies state simulation, the second proof
  obligation will be simply a consequence of our lemma. *)
  prove_refinement.
  {
    (* As the implementation module (our 4-register register file),
    does not have any rules and only has one action method, the [write_rf] method,
    there is a single subgoal. *)
```

```
related_i_s: ϕ *( st )* init_s     HA1: st -{
                                       (begin
                                              ( set (idx val) anonymous_met_arg)
                                              (if (< idx 4) pass Abort)
                                              (if (= idx 0) {write r0 val} pass)
                                              (if (= idx 1) {write r1 val} pass)
                                              (if (= idx 2) {write r2 val} pass)
                                              (if (= idx 3) {write r3 val} pass) )
                                       }→ nA
HB: update st with nA gives nxt_st
------------------------------------------------------------------------
∃ almost_mid_s mid_s : SModule,
  RfSpec.write {#idx newval} init_s almost_mid_s  ∧
  (almost_mid_s →* mid_s)  ∧
  ϕ *( nxt_st )* mid_s  ∧  ( *( nxt_st )* ≺ mid_s)
```

```
(* We have 3 interesting hypotheses:
- Hypotheses [HA1] and [HB] state that we can call the [read_rf] value method
starting in state [st], producing a log [nA], and that applying the update inside
[nA] to [st] will lead to a state [nxt_st].

- [related_i_s] is stating that [ϕ] relates the starting implementation
state to the specification state.

Now we can use our symbolic-evaluation tactic, which will invert the
hypotheses [HA1] and [related_i_s] to deduce a set of possible cases for these
assumptions to be true. *)
symb_ex_split.
{
  (* Similarly to the previous proof, using our symbolic-evaluation tactic,
  Coq generated 5 cases (in 5 different subgoals). The first four are
  similar, so let's only detail one of them.
```

The reader should notice that the goal here is exactly what was presented in Figure 3.1, we are requested to prove that we can call the [write_rf] method of the specification starting from our initial specification state. This will lead us to a new state [almost_mid_s], and then we have an opportunity to execute an arbitrary sequence of rules of the specification that should lead us to a state [mid_s] such that [ϕ] relates the end states of implementation and specification. *)

```
HA: (idx <? 4) = true     Heqb2: idx ≠ 0     Heqb0: idx ≠ 2     Heqb: idx = 3
Heqb1: idx ≠ 1     array_ext: aget nxt_st 3 = *( newval )*
array_ext0: *( rfs 2 )* = aget nxt_st 2     array_ext1: *( rfs 1 )* = aget nxt_st 1
array_ext2: *( rfs 0 )* = aget nxt_st 0
--------------------------------------------------------------------------------
∃ almost_mid_s mid_s : SModule,
  RfSpec.write {#idx newval} *( rfs )* almost_mid_s ∧
  (almost_mid_s →* mid_s) ∧
  ϕ *( nxt_st )* mid_s ∧ ( *( nxt_st )* ≺ mid_s)
```

(* In this case, we are not expecting to perform any rule on the specification side to catch up with the implementation, so we use our tactic [replay_method_with_no_rules], which takes as an argument the lemma that proved [phi] indeed implies state simulation. *)

```
HA: (idx <? 4) = true     Heqb2: idx ≠ 0     Heqb0: idx ≠ 2     Heqb: idx = 3
Heqb1: idx ≠ 1     array_ext: aget nxt_st 3 = *( newval )*
array_ext0: *( rfs 2 )* = aget nxt_st 2     array_ext1: *( rfs 1 )* = aget nxt_st 1
array_ext2: *( rfs 0 )* = aget nxt_st 0
--------------------------------------------------------------------------------
ϕ *( nxt_st )*
  *(
  fun x : N ⇒
  if N.eq_dec idx x then newval else rfs x )*
```

(* Hence we are left with proving that [ϕ] relates the new state of the implementation and the new state of the specification.

We can unfold those definitions and manipulate the expression a bit:*)

```
rm_existentials; print_arrays; split; eauto; merge_simpl; simpl.
```

```
HA: (idx <? 4) = true     Heqb2: idx ≠ 0     Heqb0: idx ≠ 2     Heqb: idx = 3
Heqb1: idx ≠ 1     array_ext: aget nxt_st 3 = *( newval )*
array_ext0: *( rfs 2 )* = aget nxt_st 2     array_ext1: *( rfs 1 )* = aget nxt_st 1
array_ext2: *( rfs 0 )* = aget nxt_st 0
______________________________________
*( nxt_st )* =
*( array: [*( ?r0 )*; *( ?r1 )*; *( ?r2 )*; *( ?r3 )*]
)* ∧
(if N.eq_dec idx 0 then newval else rfs 0) = ?r0 ∧
(if N.eq_dec idx 1 then newval else rfs 1) = ?r1 ∧
(if N.eq_dec idx 2 then newval else rfs 2) = ?r2 ∧
(if N.eq_dec idx 3 then newval else rfs 3) = ?r3
```

(* After this simple tactic to unfold and clean up the goal, we are requested to provide [?r0] [?r1] [?r2] [?r3] (existential variables), verifying a bunch of simple equations.

Remember that in our case, [idx] is actually a concrete value, so we can rewrite our goal by using this assumption and propagate the simplification to the constraints. *)

```
rewrite Heqb; simpl; split; eauto.
```

```
array_ext: aget nxt_st 3 = *( newval )*     array_ext0: *( rfs 2 )* = aget nxt_st 2
array_ext1: *( rfs 1 )* = aget nxt_st 1     array_ext2: *( rfs 0 )* = aget nxt_st 0
______________________________________
*( nxt_st )* =
*(
array: [*( rfs 0 )*; *( rfs 1 )*;
        *( rfs 2 )*; *( newval )*] )*
```

(* We are left with a goal of equality between two symbolic arrays, under some assumptions. To prove this kind of equality between arrays we have built a custom tactic: *)

```
    wrapped_arrays_equal 4%nat.
  }
  (* Similarly we handle the 3 other cases corresponding to [idx] = 0,1 and 2,
  which we hide in this literate-programming development. One can check out
  the source code to see them. *)
  {
    (* We are left with a single goal, as in the previous proof, which is
    contradictory, so we use the same [arithmetic_hammer] tactic.*)
```

```
HA: (idx <? 4) = true     Heqb2: idx ≠ 0     Heqb0: idx ≠ 2     Heqb: idx ≠ 3
Heqb1: idx ≠ 1
________________________________________
∃ almost_mid_s mid_s : SModule,
  RfSpec.write {#idx newval} *( rfs )* almost_mid_s ∧
  (almost_mid_s ↠* mid_s) ∧
  ϕ *( nxt_st )* mid_s ∧ ( *( nxt_st )* ≺ mid_s)
```

```
    arithmetic_hammer.
  }
}
Qed.
```

4.2.2 Queues

Let's recall the specification for a queue given in section 3.5:

```
Definition first _arg st ret :=
    ∃ head l,
    st = {| T:= list N; state := cons head l |} ∧
    ret = head.

Definition deq _arg st newst :=
    ∃ head l,
    st = {| T:= list N; state := cons head l |} ∧
```

```
    newst = {| T:= list N; state := l|} .

Definition enq arg st newst :=
    ∃ l, st = {| T := list N; state := l|} ∧
    newst = ({| T:= list N; state := app l [arg]|}).

Global Instance mkFifoSpec : module _ :=
    primitive_module#(vmet [ first ] amet [ enq; deq ]).
```

We now give a one-element queue implementation and give the ϕ relation that proves that the specification simulates the one-element queue. The full proof can be found in the Coq development.

Implementation of one-element queue

```
Local Instance fifo_submodules : instances :=
#| reg.mkReg valid; reg.mkReg data |#.

Definition enq :=
(action_method (e)
  (begin
    (if (= {read valid} 0)
      (begin
        {write data e}
        {write valid 1}
      abort))).

Definition deq :=
(action_method ()
  (begin
    (if (= {read valid} 1)
```

```
        {write valid 0}
      abort))).

Definition first :=
(value_method ()
    (if (= {read valid} 1)
        {read data}
        abort)).

Global Instance mkFifo1 : module _ :=
module#(vmet [ first ] amet [enq; deq]).
```

Theorem 4.6 (Queue refinement). $mkFifo1 \sqsubseteq mkFifoSpec.$

Proof. In this case we use the following natural ϕ:

$$\begin{aligned}
\phi\ i\ s := \exists\ (v\ d : N)(l_a : listN),& \\
&i = *[|v; d|]* \wedge \\
&s = *(l_a)* \wedge \\
&(\forall a, l_a = [a] \iff v = 1 \wedge d = a) \wedge \\
&(l_a = [] \iff v = 0).
\end{aligned}$$

□

Remark (The implementation queue is a strict refinement of the specification queue). The two queues are not *bisimilar* (or equivalent). Indeed, some behaviors exhibited by the specification queue will never occur in the 1-element implementation queue. An example is the behavior $[enq(1); enq(2)]$, which would require to store at least two elements.

The usefulness of the notion of refinement comes from the fact that it allows us to never have to use an implementation module (for example the one-element queue) in any of the verification work. Instead, we can always replace all the queues with their simpler-to-manipulate list-based specification.

The fact that once a refinement is proven such a substitution becomes valid is the *refinement theorem*. We now formally introduce and prove the theorem.

4.3 Refinement theorem

The justification that we can always replace an implementation module with its specification in any part of a design resides in the *refinement theorem*. We state and prove the theorem in this section.

Let $(\mathbb{m}_i | i \in [1..k])$ be a family of k modules, and let $\mathbb{m}_0$ and $\mathbb{m}'_0$ be two modules defining the same value methods and action methods and such that $\mathbb{m}_0 \sqsubseteq \mathbb{m}'_0$.

Let (S_I, R, A, V) be an Fjfj module, composed of submodules $S_I = (\mathbb{m}_0, \mathbb{m}_1, \ldots, \mathbb{m}_k)$, and pieces of syntax for rules $R = \{r_i | i \in [0..n_{rules}]\}$, action methods $A = \{am_i | i \in [0..n_{amethods}]\}$ and value methods $V = \{vm_i | i \in [0..n_{vmethods}]\}$. Let (S_S, R, A, V) be the same module when we substitute the first submodule by its specification: $S_S = (\mathbb{m}'_0, \mathbb{m}_1, \ldots, \mathbb{m}_k)$ The refinement theorem guarantees that:

$$[\![(S_I, R, A, V)]\!] \sqsubseteq [\![(S_S, R, A, V)]\!]$$

Note that here, without loss of generality, we refined the first submodule of the system. We could similarly have refined a module in any position (all the others left unchanged).

4.3.1 Proof of the refinement theorem

To prove the refinement theorem we need to find ϕ, a simulation relation between the two systems, given ϕ_0 a relation witnessing a simulation $\mathbb{m}_0 \sqsubseteq_{\phi_0} \mathbb{m}'_0$.

The goal of our proof will be to verify that the following relation is such a simulation:

$$\begin{gathered} \phi(i, s) := \\ (\forall instance.instance > 0 \Rightarrow i[instance] = s[instance]) \wedge \\ (\phi_0(i[0], s[0]) \wedge i[0] \;{}_{\mathbb{m}_0}\!\prec_{\mathbb{m}'_0} s[0]) \end{gathered}$$

The notation $s[idx]$ comes from the fact that the implementation and specification states

inhabits a cross product (see in subsection 3.6.3). This notation is simply accessing the idxth element of a product state.

Intuitively the simulation is stating that in both the implementation and the specification, the common submodules should stay equal, and the state corresponding to the changed submodule should simulate ($\prec$) the state of the specification, and be related through ϕ_0.

To verify that ϕ satisfies all the conditions of subsection 4.1.2, we need a couple of lemmas.

Lemma 4.7 (Lifting of expression semantics from implementation to specification). *Let i, s be such that $\phi(i, s)$. If a judgment $(\emptyset, \emptyset) \dashv\!\![P]\!\!\mapsto_{value} (\Gamma, \nu)$ can be derived from the state i of the implementation, the same judgment $(\emptyset, \emptyset) \dashv\!\![P]\!\!\mapsto_{value} (\Gamma, \nu)$ can be derived from the state of the implementation s.*

Proof. By induction on the derivation $(\emptyset, \emptyset) \dashv\!\![P]\!\!\mapsto_{\text{value}} (\Gamma, \nu)$. The only interesting case is the `CALL` case. The case for a value method of an $\mathbb{m}_i, i > 0$ is immediate. The case where the call is precisely a call to the replaced submodule follows because of the well-chosen ϕ: We have $\phi_0(i[0], s[0]) \wedge i[0] \,{}_{\mathbb{m}_0}\!\!\prec_{\mathbb{m}'_0} s[0]$, and hence by definition of $i[0] \,{}_{\mathbb{m}_0}\!\!\prec_{\mathbb{m}'_0} s[0]$ we have the precondition to apply the `CALL` rule for the submodule $\mathbb{m}'_0$. ☐

Lemma 4.8 (Lifting of action semantics from implementation to specification). *Let i, s be such that $\phi(i, s)$. If a judgment $(\emptyset, \emptyset, \emptyset) \dashv\!\![P]\!\!\mapsto (\alpha, \Gamma, \nu)$ can be derived for a state i of the implementation, the same judgment $(\emptyset, \emptyset, \emptyset) \dashv\!\![P]\!\!\mapsto (\alpha, \Gamma, \nu)$ can be derived from the state of the implementation s.*

Proof. Similarly to the previous proof, we proceed by a proof by induction on the derivation of $(\emptyset, \emptyset, \emptyset) \dashv\!\![P]\!\!\mapsto (\alpha, \Gamma, \nu)$. The only variation with the previous proof is that we need to apply Lemma 4.7 itself when an action expression contains a subexpression that is a value expression. The only interesting case is the `CALLACT` case, which goes through with a similar argument to the one given for Lemma 4.7. ☐

Lemma 4.9 (Lifting state simulation from submodules to modules). *A key lemma to lift a state simulation relation from the submodules to the parent modules. For every i, s, s'_0 such*

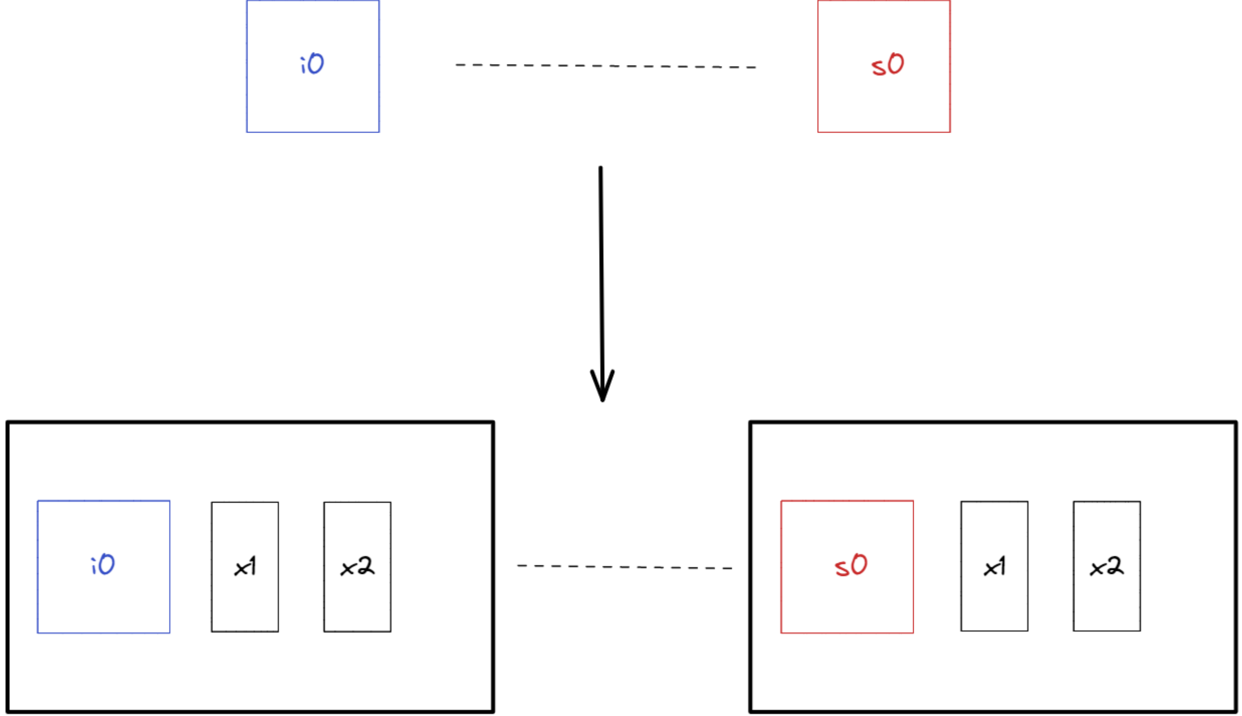


Figure 4-2: Lifting of state simulation from sub-module states to module states.

that $s[0] \rightarrow^* s'_0$ *and* $\phi_0(i[0], s'_0) \ \wedge i[0] \,{}_{\mathbb{m}_0}\!\!\prec_{\mathbb{m}'_0} s'_0$, *and for every* $j \neq 0, s[j] = i[j]$, *then*

$$\exists s', s \rightarrow_* s' \wedge \phi(i, s') \wedge i \,{}_{S_I}\!\!\prec_{S_S} s'$$

We give a visual representation of the special case where $s[0] = s'_0$ *in Figure 4-2.*

Proof. The proof is done by induction on the derivation of $s[0] \rightarrow^* s'_0$. The inductive step is straightforward. The idea is to replay the steps of $s[0] \rightarrow^* s'_0$ as substeps on the parent $s \rightarrow_* s'$ where all the steps are actually directly steps inside the first child.

The base case is the subtle case, as it requires the the just-introduced lemmas. In the base case we pick $s' := s$ (hence, $s[0] = s'_0$), and we need to prove $i \,{}_{S_I}\!\!\prec_{S_S} s$. The first submodule $i[0]$ of i simulates the first submodule s'_0 of s. To prove the state simulation we have two obligations: one for action methods, one for value methods. For value methods, we apply Lemma 4.7 that states that any observation of a value method in system S starting in state end_i is also an observation of a value method on the specification side, when the specification starts in system s. For action methods, we proceed similarly using Lemma 4.8.

□

Lemma 4.10 (Refinement theorem).

$$[\![(S_I, R, A, V)]\!] \sqsubseteq_\phi [\![(S_S, R, A, V)]\!]$$

Proof. We need to prove that the 3 conditions of subsection 4.1.2 hold. The reader interested in the details should look at the Coq development. Here we give a sketch of the first case. The two others are relatively similar. Let's take an arbitrary action method $am \in A$. By definition we have to prove that for all i_1 and s_1 such that $i_1 \prec s_1 \;\wedge\; \phi\; i_1\; s_1$, then $\forall\, arg\; i_2\; .\; (i_1, arg, i_2) \in [\![(S_I, R, A, V)]\!]$.amethods$[am]$ there exists s_2 such that $(s_1, arg, s_2) \in [\![(S_S, R, A, V)]\!]$.amethods$[am]$ and there exists a sequence of rules (and corresponding intermediate states) $r_2, \ldots, r_k,\; s_2 \ldots s_k$ such that $\forall j \in [2..k], (s_j, s_{j+1}) \in [\![(S_S, R, A, V)]\!]$.rules$[r_j]$ and $i_2 \prec s_k \wedge \phi\; i_2\; s_k$

First let's notice that if the action method call does not actually use the submodule that is being substituted, we get the result trivially, by picking zero extra rule to run, and applying the key lemma, Lemma 4.9, with $s'0 = s_1[0]$. The interesting case is when the method call actually uses the submodule $\mathtt{m}_0$ that is being substituted. In this case, the proof requires one more step. By hypothesis, the call to the action method am induces a call of exactly one action method sub_{am} to the submodule $\mathtt{m}_0$ (with argument sub_{arg}), and by the hypothesis of refinement ($\mathtt{m}_0 \sqsubseteq_{\phi_0} \mathtt{m}'_0$), the method $sub_{am}(sub_{arg})$ can be simulated in $\mathtt{m}'_0$ (potentially adding extra rules inside the submodule $\mathtt{m}'_0$). This simulation is giving us the preconditions to be able to apply Lemma 4.9 and get our conclusion.

□

It is easy to extend the refinement theorem to simultaneously substitute several submodules (for example, all of them) instead of just one. This way, we get a more general version of our refinement theorem:

Theorem 4.11 (Generalized Refinement theorem). *Let $(\mathtt{m}_i | i \in [1..k])$ be a family of modules, and let $(\mathtt{m}'_i | i \in [1..k])$ be another family of modules such that $\forall i \in [1..k] \mathtt{m}_i \sqsubseteq \mathtt{m}'_i$.*

Let (S_I, R, A, V) be an Fjfj module, composed of submodules $S_I = (\mathtt{m}_0, \mathtt{m}_1, \ldots, \mathtt{m}_k)$, and

pieces of syntax for rules $R = \{r_i | i \in [0..n_{rules}]\}$*, action methods* $A = \{am_i | i \in [0..n_{amethods}]\}$ *and value methods* $V = \{vm_i | i \in [0..n_{vmethods}]\}$*. Let* (S_S, R, A, V) *the same module when we substitute all the submodules by their specification:* $S_S = (\mathfrak{m}'_0, \mathfrak{m}'_1, \ldots, \mathfrak{m}'_k)$ *The generalized refinement theorem guarantees that:*

$$[\![(S_I, R, A, V)]\!] \sqsubseteq [\![(S_S, R, A, V)]\!]$$

Proof. It follows by induction on the number of submodules, using the transitivity of refinement (Theorem 4.3) and the just-proven simpler refinement theorem (Lemma 4.10). □

4.4 Other ways to compare modules

4.4.1 Comparison with previous work

Kami [19] had formalized a rule-based language in Coq. Similar to this work, the central notion of correctness was also a notion of refinement between specifications and implementations. Here we discuss the differences between our two notions of simulations.

Kami's notion of module does not have notions of value methods and action methods, it only has one global notion of method. As such, it cannot impose the restriction to one action method call per submodule. Hence, instead of characterizing the module one method at a time, the notion of refinement of a module in Kami requires considering the effect of calling simultaneously an arbitrary subset of methods: that's what is called a substep in Kami's terminology.

Hence, the rule-based language philosophy of studying systems *one thing at a time*, i.e. one rule at a time when there are no modules, becomes not possible to uphold when one adds a module boundary: we can't study the module one method at a time.

Note that practically, due to this notion of substeps, there are potentially exponentially more cases to consider in proving a Kami refinement. This does not arise for a two-method interface, like a minimal queue, but if a system has more than 3 or 4 methods it starts being significant. This aspect of Kami is actually close to modeling the behavior of standard non-modular BSV (BSV without synthesize boundary). The Kami paper shows a good

characteristic example of the subtlety of their notion of refinement: a server-like module that has an enqueue and a dequeue method. While one might want to characterize the module with only two atomic transition relations (nondeterministic relations for enqueue and dequeue), instead one needs more transition relations for Kami's kind of refinement. For example, we need to consider the transition caused by both enqueuing and dequeuing simultaneously (and enqueueing and performing a subrule of the server simultaneously). The paper talks about 4+7=11 such relations, instead of 2.

The choice made by Kami is necessary if one wants to guarantee an *inlining principle* in the language. Inlining is one of the key principles of reasoning in Kami [17], but it is not valid in Fjfj. This already informally existed in Bluespec: inlining does not hold when one is using modular Bluespec (Bluespec with so-called *synthesize boundaries*).

Finally, a side effect of Kami's notion of refinement is that some modules that we could hope to say refine one another do not satisfy the definition of refinement in Kami's sense: while in some cases we might never call two methods simultaneously in reality, Kami needs to worry that we could potentially, as long as those sequences do not cause double writes. Hence, sometimes refinement just does not hold in Kami's sense because of a situation where the specification would be forbidden to call several methods in a substep while the implementation would accidentally allow it.

4.4.2 Trace inclusion versus our notion of simulation

Independently of the way one defines the notion of step in the semantics of the language, we can wonder about the relative expressivity of defining refinement either as a simulation (like we did) or as a trace inclusion: a module refines another one if the set of traces admissible by the implementation module is a subset of the set of traces admissible by the specification module.

Within our rule-based setup, it was thought that those two definitions were equivalent. The example in Figure 4-3 shows that they are not equivalent definitions. Using the trace-based definition of refinement makes the refinement theorem significantly harder to prove (we did not succeed in writing such a mechanized proof when we originally explored this definition of refinement). As we have not yet found practical cases where this stronger

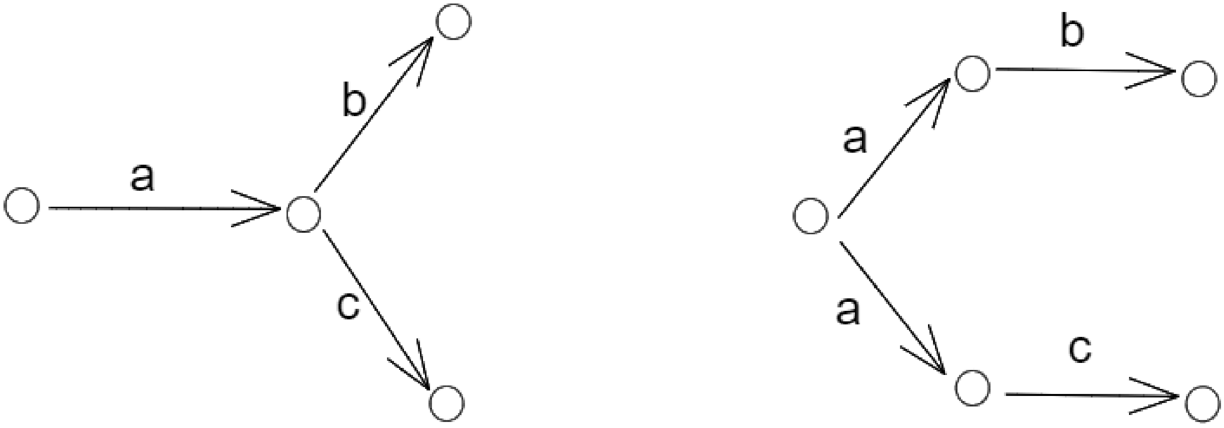


Figure 4-3: Classical example from process calculi originally adapted in the rule-based setup by Andrew Wright. This diagram shows two systems that are trace equivalent but one cannot simulate the other.

notion of refinement will be useful, we decided to stick to our simulation-based definition of refinement.

Chapter 5

Specifications

Now that we have the language framework to state and prove refinements, we are ready to write more specifications and implementations, beyond the simple examples we gave in the previous chapter.

It happens that quickly we run into the following question: *What makes a good specification for hardware design?* This chapter focuses on discussing and elaborating on several techniques to come up with good specifications and explaining what makes them good.

We will often derive chains of intermediate specifications, every design more specialized than the previous one, hoping to keep the reasoning nonmonolithic. Each link of these chains of specifications is composed of two sides: the implementation side and the specification side. As such, an intermediate design in these chains will, depending on the context, alternatively be considered a specification or an implementation. In other words: *One person's specification is another's implementation.*

5.1 Different kinds of specifications

5.1.1 Operational specifications

Operational GCD Let's illustrate operational specification with a simple strawman for a Greatest Common Divider (GCD) specification. The reader should keep this simple GCD specification in mind, as we will explain several of its limitations (and how to fix them)

throughout this chapter.

```
Module EuclideanGCD.
    Local Instance gcd_submodules : instances :=
      #|
        mkReg a;
        mkReg b;
        mkReg valid
      |#.

    Definition enq :=
        (action_method (va vb)
          (begin
            (if (= {read valid} 0)
              (begin
                {write a va}
                {write b vb}
                {write valid 1})
              abort))).

    Definition deq :=
        (action_method ()
          (begin
          (* the value in b is zero, so the euclidean algorithm finished *)
            (if (& (= {read valid} 1) (= {read b} 0))
                {write valid 0}
                abort))).

    Definition first :=
        (value_method ()
          (if (& (= {read valid} 1) (= {read b} 0))
```

```
          (begin
              {write valid 0}
              {read a})
            abort)).

  Definition swap :=
      (rule
        (begin
        (set va {read a})
        (set vb {read b})
        (if (< va vb)
          (begin
              {write a vb}
              {write b va})
            abort))).

  Definition compute :=
      (rule
        (begin
        (set va {read a})
        (set vb {read b})
        (if (= {read valid} 1) (> va vb)
              {write a (- va vb)}
            abort))).

 Global Instance mkGCDOperational1 : module _ :=
      module#(rules [swap; compute] vmet [ first ] amet [ enq; deq ]).

End EuclideanGCD.
```

Note that the specification is not only said to be operational because it is written as a Fjfj

module. One could also write a *primitive module* with a morally operational specification, *i.e.* the registers and the queues are such operational primitive modules.

Even for this GCD module, we could have written an alternative specification as a primitive module, where the Euclidean algorithm would not have been performed through rules but through a single call to a Coq function computing the GCD (the GCD function of the standard library, for example).

Alternatively, one can remark that we could have written the specification slightly differently. Instead of using a rule that repetitively subtract the values in the two registers, we could have used a rule that repetitively computes the remainder of one value by the other, using Euclidean division. This would give another specification module, which would be *bisimilar* to `mkGCDOperational`, but would take a smaller number of steps to produce its results. Because the two specifications would be bisimilar (or equivalent), it points out that it does not matter which algorithm is used: an implementation might use yet another algorithm to compute the GCD and still be proven to refine this specification.

The algorithm internally used within an operational specification does not prescribe an algorithm to be used in implementations.

Though, still, the necessity to choose one algorithm or the other for the specification might feel unsatisfying, and we can consider that it reduces interpretability (one needs to know Euclid's algorithm and its variations to know what this specification means). An alternative is to write the GCD specification in an *operation-agnostic* way, as described in the upcoming subsection 5.1.2.

5.1.2 Propositional specifications

In the literature there exists another kind of specification that is not directly operational but instead constrains the relations between inputs and outputs.

Broadly, this kind of programming/specification has been given various names in the literature, either *relational* [15], *declarative* [45], *equational*, or even *definitional* [28].

To keep up with the tradition of each generation of computer scientists renaming those, we decide to name them *propositional specifications*. This name is justified for us because those specifications are written by writing explicit *propositions* (terms in `Prop` in Coq), relating

input and output.

Let's give a first example of purely propositional specifications of a GCD module:

Relational GCD

```
Module RelationalGCD.
    Definition enq arg st new_st :=
        st = *((None  : option (N * N)))* ∧
        dlet {a b} := arg in new_st = *( Some (a,b) )*.

    Definition isGCD p n m :=
    (* Classic predicate for GCD:*)
        (* p divides both n and m *)
         (p | n) ∧ (p | m) ∧
         (* any other divisor of n and m necessarily divides p *)
        (∀ q : N, (q | n) → (q | m) → (q | p)).

    Definition first (arg :N) (st : SModule) (ret : N) :=
        ∃ (a b : N), st = *( Some(a,b) )* ∧
        (* Constraint the result returned to be
          the greatest common divisor to a and b *)
        isGCD ret a b.

    (* Consume the request, allowing to send a further request *)
    Definition deq (arg :N) (st : SModule) (new_st: SModule) :=
        ∃ (a b : N), st = *( Some(a,b) )* ∧
        new_st = *( None : option (N * N))*.

   Global Instance mkGCDRelational : module _ :=
        module#(vmet [ first ] amet [ enq; deq ]).
End RelationalGCD.
```

Contrary to the previous GCD specification, this one does not use Euclid's algorithm. It does not use any algorithm; the specification only states that the values returned by the `first` method are such that they are a greatest common divisors of input values.

It is not even completely obvious from this specification that there exists a single output that verifies those constraints (that the GCD is unique), though it is a well-known mathematical property that it is the case.

5.2 Representations and data structures for specification

We will now illustrate how one should often introduce intermediate specifications in our proof, to replace low-level, synthesizable modules by clean data structures on which it is easy to write invariants to connect with the specification. We will especially discuss key properties that make a significant practical difference in stating invariants.

Let's illustrate this concept with our simple pipeline design computing the composition g and f on some stream of input. We first introduced this example in section 3.4. The design is composed of 3 one-elements queues (input, mid, and output fifo) and 2 pipeline stages interleaved that are computing respectively f and g.

Let's propose the following simple specification for the module:

```
Module PipelineSpec.
    Local Instance submodules : instances :=
      #|
        Queue.mkQueueSpec inp
      |#.

    Definition enq :=
        (action_method (el)
            {enq inp (g (f el))}).
```

```
    Definition deq :=
        (action_method ()
            {deq inp}).

    Definition first :=
        (value_method ()
            {first inp}).

   Global Instance mkGofFSpec : module _ :=
        module#(vmet [ first ] amet [ enq; deq ]).
End PipelineSpec.
```

5.2.1 Avoiding low-level simulation relations

The implementation uses three one-element queues, each built using a valid register and a data register, while the specification uses a single specification queue that contains a list of elements.

Impedance mismatch leading to painful simulations We could manually try to write the simulation relation (in pseudocode), arriving at:

```
(impl.in.valid = 0 && impl.mid.valid=0 && impl.out.valid = 0)
        <-> spec.in = []) ∧
(impl.in.valid = 1 && impl.mid.valid=0 && impl.out.valid = 0)
        <-> spec.in = [g(f(impl.in.data))]) ∧
 ... // 5 more cases
(impl.in.valid = 1 && impl.mid.valid=1 && impl.out.valid = 1)
        <-> spec.in = [impl.out.data :: g(impl.out.data) :: g(f(impl.in.data))])
```

Interestingly, many proofs in previous systems used invariant relations of this form. Those are quite labor-intensive to write, and very hard to maintain. All those cases are caused by a mismatch of data structures between the queues in the implementation (that are not made

of lists) and the queues in the specification. Conceptually, we want an abstraction that constructs the elements of the implementation queue represented as a list (those are at most singleton lists), so we can combine those states with standard list combinators. A clean way to achieve this abstraction is explained in the next paragraph.

Easier invariant The right way to provide this abstraction and extract the elements contained in the implementation is to introduce an intermediate design, a generalization of our implementation that uses the refinement of a specification queue by the one-element queue:

```
Module IntermPipeline.
    Local Instance submodules : instances := #|
      mkSpecQueue in;
      mkSpecQueue mid;
      mkSpecQueue out
    |#.

    (* We reuse the rules and methods of our implementations,
    but now the queues are not using the submodules Queue1.mkQueue1.
    Instead they are using more general n-element queues. *)
    Global Instance mkPipeline : module _ :=
      module#(rule [pipeline.do_f; pipeline.do_g]
              vmet [pipeline.first]
              amet [pipeline.enq; pipeline.deq]).
End IntermPipeline.
```

So by application of the refinement theorem, we have $pipeline.mkPipeline \sqsubseteq IntermPipeline.mkPipeli$ and it only is left to prove that $IntermPipeline.mkPipeline \sqsubseteq PipelineSpec.mkGofFSpec$. But those last two modules are using lists everywhere, and we can easily state the following simulation relation between the implementation state and specification state (in pseudocode):

```
spec.in = impl.out ++ map g (impl.mid) ++ map (fun x => g (f (x))) (impl.in)
```

which we can easily check to verify the conditions characterizing a simulation relation defined at the beginning of chapter 4.

How figured out this last invariant The simulation relation we just exhibited for the proof is more elaborate than the previous simulations we showed. The reader might correctly remark that if the simulation relation is already nontrivial for such simple systems, simulation-crafting is itself a first-order difficulty when proving designs (even when the specification has been written following best practices). We will discuss extensively simulation-crafting in chapter 7, and we will come back to this specific example to see that it is an example of simulation that fits one of our general techniques for crafting simulations.

5.2.2 Canonicity of representation

Another aspect in which picking the right data structure for our specification matters a lot is related to what we call *canonicity*. Let's consider two implementations of a 4-element queue. One is built with 4 registers, an enqueue pointer, a dequeue pointer and a register indicating an empty queue. We can call this version the *circular-buffer version* The other version is simply using a list of length at most 4, which we can call the *list version*.

The list version has the property of *canonicity*: if we want to think about an empty queue, we only need to consider a single case: the empty list. In contrast, the circular buffer has 4 different states that represent the empty case (enqueue pointer and dequeue pointer are equal, and the empty register is set). The fact that many low-level states are *equivalent* but not actually equal means that proofs are more complex and need to work up to a custom equivalence relation. This equivalence relation is not always trivial and may needlessly complexify proofs. Those issues disappear in the list version: there are no multiple state representations of the same conceptual state. We say that the data structure is canonical.

We will always favor data structures that avoid requiring that we state invariants about symmetric cases; preferring to bake those equivalent states directly in a custom data structure will make proofs easier and smaller. Sometimes complete canonicity is hard to get, but we found that reducing equivalent states as much as possible is good practice.

Interestingly, we found this design principle to be even critical in our model-checking experimentation (which we explored in the solver-aided programming language Rosette [57], but that we do not describe in this dissertation). There it not only meant easing of the proofs and reduction in human effort, but it also allowed us to have SMT solvers successfully proving obligations that were too numerous/too difficult to prove on the original system (full of unexploited symmetries).

5.2.3 Generalization to handle concurrency and parallelism: GCD example

Looking back on our various forms of specification of GCD modules, they all presented a fatal flaw: any real implementation would likely be capable of working simultaneously on more than one pair of inputs, and none of them allowed more than one token at a time.

In this case, addressing the issue and getting a more general specification is easy: we can simply add an unbounded queue in front of our previous GCD modules to buffer inputs, or an unbounded queue in the back on the results. Note that those two propositions are equally satisfying, and they define equivalent modules. Let's name such a specification `mkGeneralizedGCD`.

Such a specification will be a correct specification for a large family of implementations, as long as the implementations return results in order. For example, a valid implementation could internally use an array of GCD processing elements, processing requests in parallel, and would simply need some bookkeeping logic to remember in which order responses should be returned. Even though internally this fancy module would have completely overlapped and changed the order of computations, from the perspective of the interface, the implementation could be logically thought of as the specification.

Generalization and new behaviors `mkGeneralizedGCD` is clearly more general than our previous versions, but we might wonder: are there cases where this specification is too general and should not be used?

There are two contradictory answers: from the perspective of good design practice, we propose that no; `mkGeneralizedGCD` is a good way to think about a GCD engine, and if

one were to need a GCD engine, they should probably think of writing the design assuming it were speaking to a `mkGeneralizedGCD` module. However, technically speaking, it is easy to build a design for which a one-element GCD works, while `mkGeneralizedGCD` introduces issues.

For example, let's consider we want to build a machine that computes both the GCD and the sum of the inputs. One could think of sketching the following:

```
Local Instance submodules : instances :=
 #| GCDMachine gcd;
    mkReg sum |#.

Definition enq :=
    (amethod (a b)
        (begin
            {enq gcd (# a b)}
            (* The syntax (# _  ... _ ), that we see here for the first time,
            is the syntax to call multiple-arguments functions or methods. *)
            {write sum (+ a b)})).

Definition deq :=
    (amethod () {deq gcd}).

Definition first :=
    (amethod ()
        (begin
            {first gcd}
            {read sum})).
```

While this implementation is correct if we use a one-element GCD module (it indeed computes the sum and the GCD), it becomes incorrect if we use the `mkGeneralizedGCD` module. Indeed, enqueuing in the module will overwrite the value in the sum register. Arguably,

that's because it is a bad design: we should not use a register but another queue to store the sums, though it still raises a valid point. Specifying a system is thinking about in which context the design will be used, and it is often important to make sure we cover pipelining/-multiple outstanding token behaviors. Generalization of the specification allows us to tackle this issue, but it regularly introduces conservatively large amounts of concurrency. In some cases, this extra concurrency might need extra logic from the surrounding design to be kept under control. At times, a designer might not have considered such a situation and might have actually thought in their head about the simpler design that is limited to never exploit any of that concurrency. In such a case, one might use a one-element GCD as a specification.

We now move on to an interesting case of generalized specification: the specification of a processor. The generalized specification is nontrivial, and we will have to do a sophisticated proof to show that the generalization is safe in the context where it is used. Before that, we motivate our special interest in modular specification.

5.3 Processor generalization for modular specification

One reason the proofs of correctness of complex digital systems are difficult and time-consuming is because the proofs are usually not modular; we often need global invariants of the system that mention all the different submodules of the design simultaneously and ensure constraints between their states. The invariants need to be significantly rewritten whenever the design changes. While proper proof-engineering discipline helps, the proofs previously written were practically rarely able to be ported to slightly different designs (for example, changing a one-element queue into a two-element queue).

We believe the primary reason for this lack of modularity is the lack of proper *modular specification* for a variety of modules in the design.

We will show in this section that the intuitive specification of the standard system (core + memory) must be *generalized* to encompass behaviors seen in various implementations.

Generalization, however, is sometimes dangerous because it admits behaviors that are not seen in the original intended specification, and thus it can make the whole design wrong. Therefore, the generalized specification of a submodule must be shown to be safe in its

context of use.

We will introduce new insights into the way we think about the correctness of a processor isolated from its memory. Those ideas allowed us to prove several smaller refinements, each targeting a different part of the full design. The approach simplified our proofs and increased the potential reusability of those smaller and separately proven modules. This section covers the architectural intuitions and motivations, and introduces the notion of generalization specification. In the next chapter, we will give a detailed example of the concepts we introduce here.

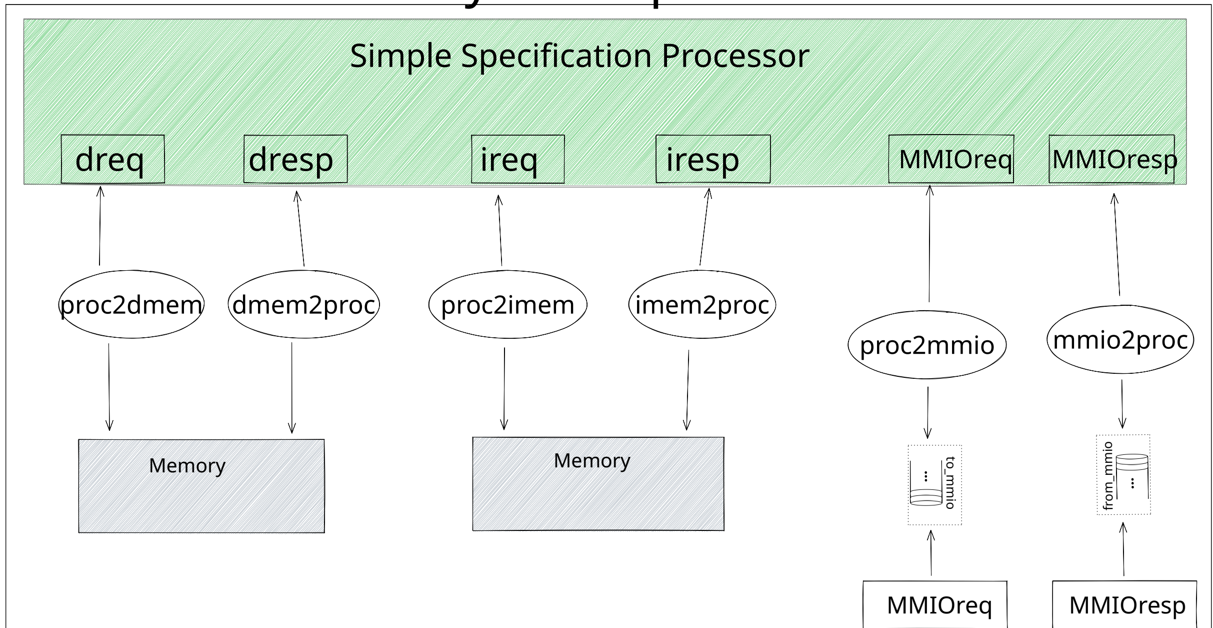
5.3.1 Processor, memory, and system specification

Before we dive in, remember that the words *refinement* and *correct* now have been given precise formal meanings. *Correctness* refers to the existence of a simulation relation (a refinement) between the specification design and the implementation design, which only talks about the behaviors of the two systems at their interface. See chapter 4 for more details.

Let us show the need for generalizing specifications using a machine composed of a processor and a memory. The machine interacts with the environment using memory-mapped IO (MMIO), and as such MMIO is the only interface to the system. One may implement such a machine using many microarchitectures such as unpipelined, in-order pipelined or out-of-order microarchitecture for the processor; and different cache hierarchies, replacement policies and reordering policies for the memory. Typically, a machine will have multiple simultaneously outstanding memory loads. In contrast, the MMIO requests and responses are typically handled sequentially and nonspeculatively, as both are interactions with the outside world that can have arbitrary side effects. We focus on the processor side of the system and assume that the memory module is kept constant as a first-in-first-out memory specification.

We show diagrams of the specification and the implementation systems respectively in Figure 5-1 and Figure 5-2.

While the interaction between the memory and the processor is not visible from the outside world, all implementations must show exactly the same I/O behavior for any given



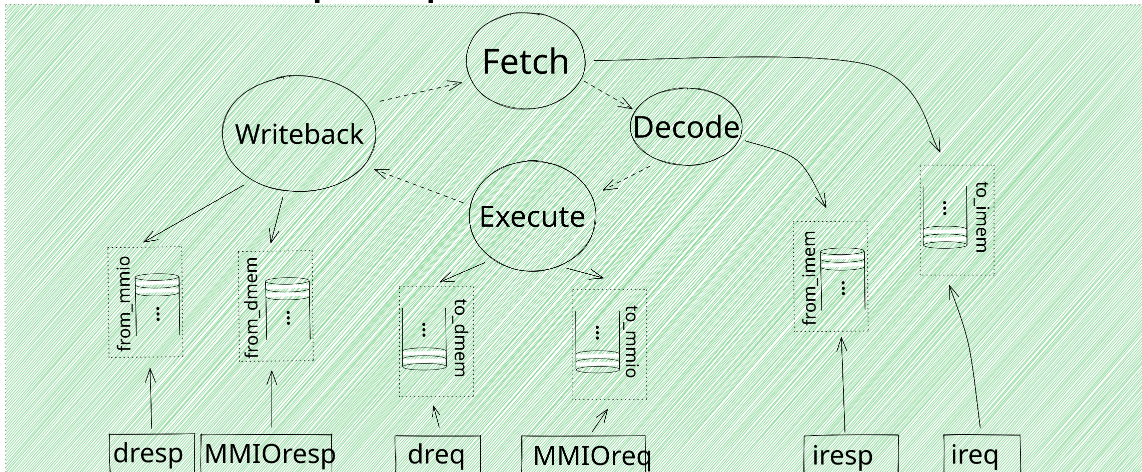
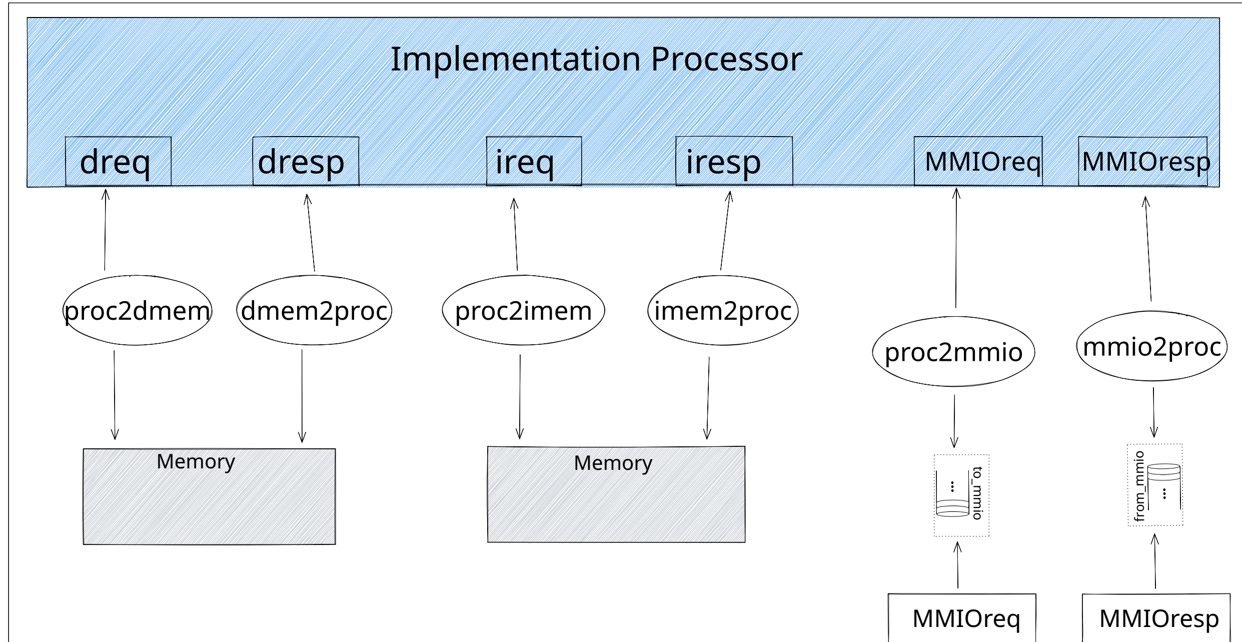


Figure 5-1: Architectural sketch of the full-system specification and the processor specification. Dotted rectangles delimit module boundaries, regular rectangles indicate the methods of a module (sometimes omitted), circles represent rules, and a full arrow always goes from a rule or a method to a method of a submodule indicating that the source object calls the method at the target (here mostly the enqueue and dequeue methods of queues). Dotted arrows informally represent that two rules are mutually exclusive and that the first one needs to occur before the second one can happen. The top-level module simply connects the processor to the memories (instruction and data) and connects the MMIO requests-and-responses queues to the public-facing interface. The processor specification is simply a very simple *multicycle* specification: there is exactly one instruction in-flight at any time in this machine.

Full System Implementation



Implementation Processor

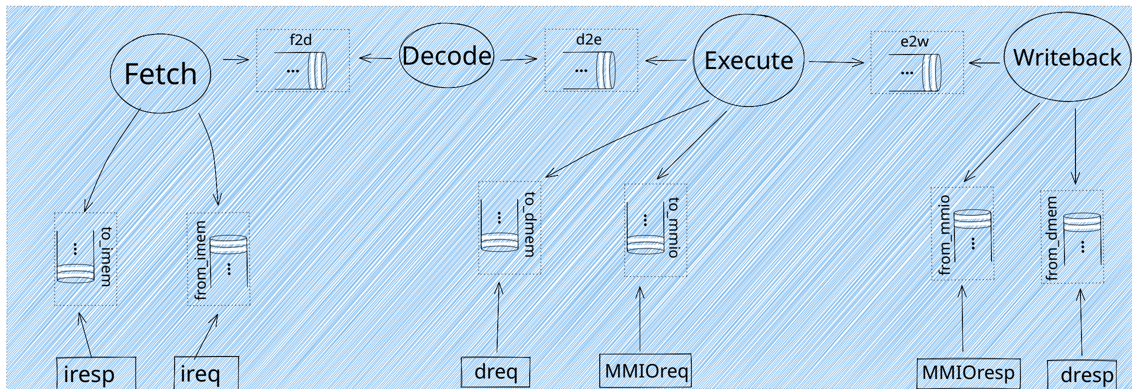


Figure 5-2: Architectural sketch of the full-system implementation and sketch of the processor implementation. The top-level module simply connects the processor to the memories (instruction and data) and connects the MMIO requests-and-responses queues to the public-facing interface. The only difference with Figure 5-1 is that in this design, the processor implementation is a pipelined machine.

program.

To prove a refinement between an implementation and specification, one would hope to reason modularly: first prove that the pipelined processor implements the one-instruction-at-a-time processor specification, prove that the cached memory implements the memory specification, and then use our refinement theorem presented in chapter 4 to combine the two proofs together.

However, this is an invalid approach because the lemmas about the subsystems in isolation are most of the time not true. For example, the pipelined processor is *not* a refinement of the one-instruction-at-a time processor. Indeed, the pipelined processor might emit spurious loads (speculation) and will likely reorder some instruction and data loads. It is easy to find traces of loads and stores that can be emitted by the pipelined processor that will never be emitted for the specification, so we can easily prove that $mkPipelinedProcessor \not\sqsubseteq mkAtomicSpecProcessor$. (The reader curious about the situation for the memory subsystem can also figure out that similarly, it is easy to find traces of real memory subsystems that are actually not traces of the standard way one specifies a simple atomic memory.)

Hence, we do not have a valid specification just for the processor. We only have a specification for the full system.

This statement is a bit counterintuitive because if we look at the full-system specification, it is made of two components: a processor component and a memory component. However, those two components are *not* valid specifications of the processor and the memory respectively. Only when we put those two components *together* do they behave the same way as the two implementation components together.

Loosely speaking, a system of memory plus processor will most of the time verify the following properties:

$$FullSystem(Memory, ProcessorImpl) \sqsubseteq FullSystem(Memory, SimpleProcessorSpec)$$

while at the same time:

$$ProcessorImpl \not\sqsubseteq SimpleProcessorSpec$$

Or more generally, when considering also the memory:

$$FullSystem(MemoryImpl, ProcessorImpl) \sqsubseteq FullSystem(MemorySpec, SimpleProcessorSpec)$$

while at the same time:

$$MemoryImpl \not\sqsubseteq MemorySpec$$
$$ProcessorImpl \not\sqsubseteq SimpleProcessorSpec$$

This is unfortunate as we were planning to tackle the problem modularly, and here we clearly have a deficit of modularity.

5.3.2 Generalizing specifications: recovering modularity

In this section, we design a system specification composed of processor and memory specifications that is actually compatible with modular refinement for a large family of processors and memory implementations. In subsection 5.3.3, we will point out the limitations of this specific generalized specification, hinting at the need for even further generalization in the future.

Here is the intuitive idea: consider the more general specification for our system, sketched in Figure 5-3.

This new specification introduces two *nondeterministic load machines*: the *Data-Load Buffer* and the *Instruction-Load Buffer*. Those two machines are simply machines that emit loads for arbitrary addresses, at arbitrary times, and store the results into a local load buffer.

An intuitive way to understand this specification is that an actual processor not only emits the memory operations that are generated by the program (both instruction and data loads), it also emits extra loads (both instruction and data) to imitate speculative execution. Instead of characterizing precisely the restrictions and shapes of loads that can be emitted, studying carefully control, data, and address dependencies, this specification indicates that we should *conservatively* think of a processor as a machine that can emit loads for a random address at any time.

At least the proof obligation on the core (and on the memory) is not obviously false

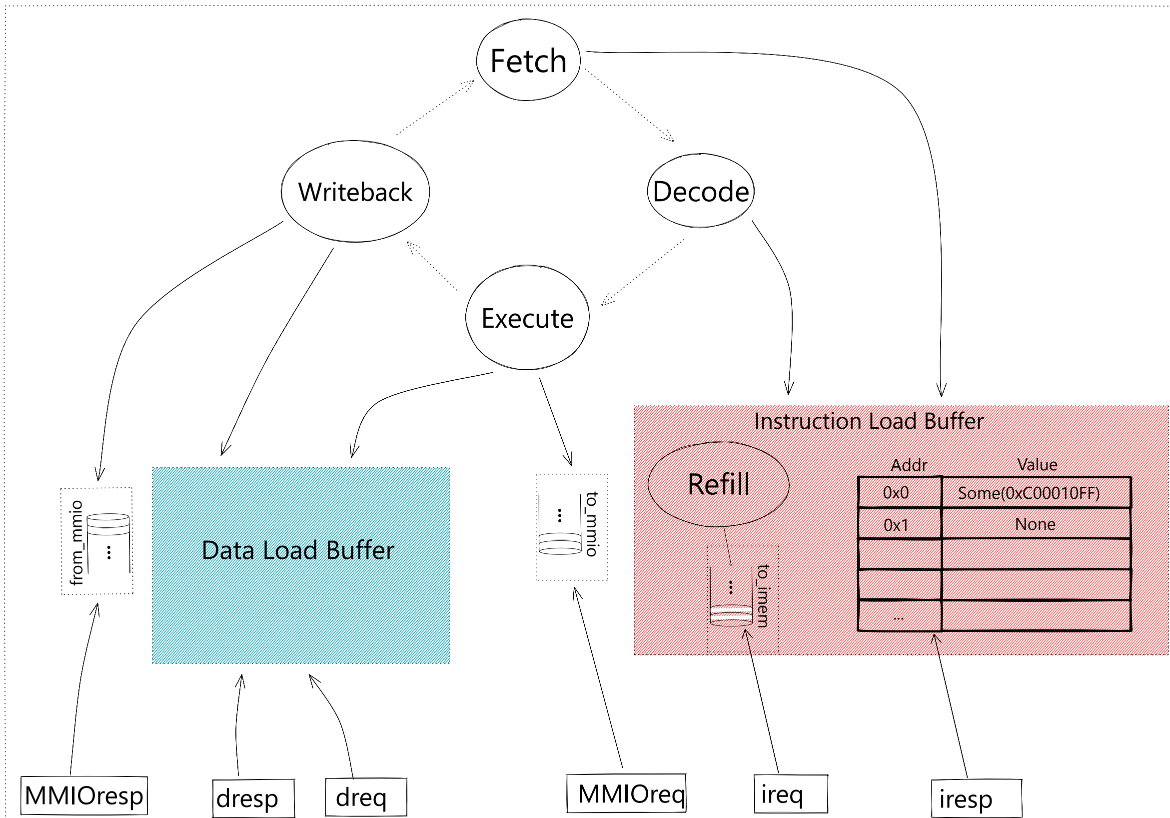


Figure 5-3: Architectural sketch of our generalized processor

anymore: the *Generalized Processor Specification* which has been augmented with the non-deterministic load machines now seems to be able to fake the speculative loads that were making it impossible to use the ISA processor specification alone as a specification for the processor.

However, we need to worry about the effect of those extra loads that are emitted due to our generalized processor specification.

Clearly, not every generalization is a reasonable or good generalization of a specification. For example, if we had augmented the specification to say that the generalized processor could emit arbitrary *stores*, it would have been an unreasonable/invalid generalized modular specification. So, there are both valid generalized specifications and invalid generalized specifications. Intuitively, we know that loads don't matter and that in a sense our generalized specification is not so different from the simple specification. In the next section, we see that deciding if a generalized specification is valid or invalid does not have to be a subjective question. We formalize a criterion which allows us to claim that a generalized specification is a valid generalized specification.

What is a valid generalized specification? Now that we have defined a generalized specification, we have two definitions of full-system specifications. The one that is a bit scary, using a generalized specification for the processor, and the one that was much simpler using the simple specification for the processor. The latter was appealing from the perspective of simplicity and readability, but was preventing us from tackling the processor and the memory in isolation. The former is less clear and further from the usual specification, but it allows us to reason modularly.

We say that a generalized specification is valid in its context if the full system using the generalized specification is a refinement of the full system using the simple specification. So in our case, a generalized processor specification is valid only if:

$$FullSystem(GeneralizedProcessorSpec, Memory) \sqsubseteq FullSystem(SimpleProcessorSpec, Memory)$$

This is a strong and nonobvious condition (it holds for the generalized specification that

we outlined). It is intuitively saying that all the extra memory loads cannot influence or modify the set of original MMIO events emitted by the simple full system. In other words, there are no new full-system MMIO behaviors, despite the obvious existence of new *internal* behaviors.

The full proof that this condition holds for our generalized processor specification will be sketched in chapter 6. The mechanized proof can be found in the Coq development.

Putting everything together: full-system decomposition To summarize, thanks to this first level of generalized specification, we introduce in Figure 5-4 the first level of the overall proof strategy we will follow in chapter 6.

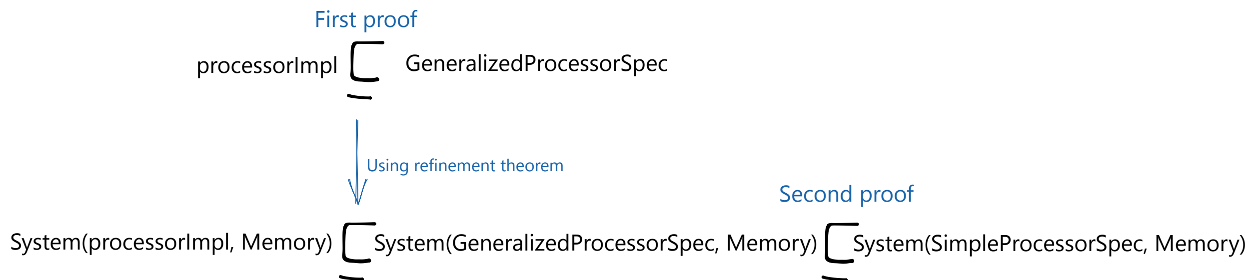


Figure 5-4: Simple proof-decomposition overview

5.3.3 Limitations of specification behaviors

We now describe two forms of limitations of the behaviors one can observe in our specifications. Those are especially interesting to understand better the architectural subtleties implied by our specifications.

We will first discuss constraints on MMIO behaviors imposed by all our specifications and then discuss constraints imposed by our specific generalized processor specification on memory accesses.

MMIO constraints implied by our specifications It is easy to realize that the top-level full-system specifications (both the generalized and the original one) guarantee the following

invariant on MMIO accesses: *The response to every load request needs to be received before we can send the next MMIO operation (load or store).*

In particular, there cannot be two outstanding MMIO loads in our specification systems. Indeed, the specification cannot generate a sequence of two outstanding MMIO load requests back-to-back as it needs to wait for the response to the first MMIO load to complete the writeback stage of that instruction before it can start fetching another instruction.

This remark points out a subtlety usually ignored by the standard architectural wisdom. Usually, MMIO loads and stores are handled with special care, as it is often believed that they can only be sent once we are certain that they are not speculative, so that they won't be squashed due to an older instruction that caused a misspeculation. A conservative way to implement this policy is to simply emit MMIO loads and stores only when those instructions are the oldest in the pipeline. Such an implementation could be proven to be safe with respect to our specification. But another implementation, often considered equivalent if slightly more aggressive, would be to allow sending of MMIO operations as soon as they are guaranteed not to be cancelled. For example, that version allows sending two MMIO loads back-to-back. This seemingly reasonable implementation choice actually violates our specifications: the implementation can now exhibit the behavior `MMIOLoadReq(addr1), MMIOLoadReq(addr2) ... [responses]`, while the specification will never have a trace with two requests not interleaved by a response.

It is important to note that those constraints on MMIO events are not a limitation of our framework in the sense that we *would not be able to prove the correctness of the design because it would be technically or practically too difficult.* No, we cannot prove such correctness because we just showed that such an implementation *is not a refinement of our full-system specification.*

The architects interested in allowing such aggressive issuing of MMIO loads would then need to look for a more expressive full-system specification: the standard full-system ISA specification is insufficient. However, such a specification would likely suddenly become much more complicated as intuitively it will require the specification to be able to run several outstanding instructions simultaneously already, entering the realm of computer architecture.

Program:
Ld r0 X; St 1 Y

Invalid traces:
[LdReq X; StReq 1 Y; LdResp X _]
[StReq 1 Y; LdReq X; LdResp X _]

Program:
St 1 Y; Ld r0 X

Valid traces:
[LdReq X; StReq 1 Y; LdResp X _]
[LdReq X; LdResp X _; StReq 1 Y]

Figure 5-5: Typical load/store reorderings that are both possible and impossible to observe in our generalized specification. The underscores in response events represent the irrelevant values returned by the loads.

Memory-ordering constraints implied by our specifications Similarly to the constraints on the traces of MMIO events, the generalized processor specification implies a similar – if slightly more sophisticated – invariant for standard memory operations: *Stores can be emitted only when all previous outstanding program loads have received their responses.* In Figure 5-5 we show two programs illustrating both a kind of load/store reordering that can occur in our generalized specification but also a kind of load/store reordering that cannot occur, giving an intuition of the constraints put on any processor implementing our generalized specification.

Contrary to the just-presented MMIO constraints, the memory constraints are not full-system constraints and *do not imply that a processor performing aggressive store-issuing cannot implement our full-system specification once hooked up to a memory.*

Those constraints only limit our current ability to prove the correctness of the sophisticated designs that would exhibit aggressive store-issuing *using our current modular generalized processor specification.*

It is our opinion that there exists an even more general processor specification that does not force those memory constraints and that would allow us to prove the correctness of those processors. We have not yet found this ideal processor specification.

Initially memory is initialized with zeroes

Message Passing (MP)		Load Buffer (LB)		Store Buffer (SB)	
St 1 X	Ld r0 Y	Ld r0 X	Ld r1 Y	St 1 Y	St 1 X
St 1 Y	Ld r1 X	St 1 Y	St 1 X	Ld r0 X	Ld r1 Y

Message Passing (MP)	Load Buffer (LB)	Store Buffer (SB)
Can we observe: r1 = 0, r0 = 1 ?	Can we observe: r0 = 1, r1 = 1 ?	Can we observe: r0 = 0, r1 = 0 ?
TSO: No	TSO: No	TSO:Yes
SC: No	SC: No	SC: No
Generalized Spec: Yes	Generalized Spec: No	Generalized Spec: yes

Figure 5-6: Simple proof-decomposition overview

Induced memory model The previous discussion introduced memory-ordering constraints in our generalized specification. Those are reminiscent of the question of *memory model*.

This might look unexpected, as traditionally memory models only appear in the context of multicore machines, and our generalized processor specification does not even mention memory, let alone other cores.

However, specifying the traces of loads and stores of the processors actually *induces* a memory model, once we connect several such processors to an atomic memory.

We find it very intriguing to observe in Figure 5-6 that, despite the simplicity of our generalized specification, the memory model induced is weaker than Total Store Order (TSO) [50] (and so is weaker than Sequential Consistency (SC)[39]). The weak behaviors for MP and SB are easy to obtain in our generalized specification:

- For the MP litmus test, the reader thread can simply prefetch address `X` at the beginning of the execution, then wait for the writer thread to be completed before executing the two loads. The first load to `Y` will get the new value directly from memory (so 1), while the second will get the old value (0) from the load buffer.
- For the SB litmus test, both threads can simply prefetch the values for `X` and `Y` in their

load buffers. Because the load that each thread emits is targeting an address to which it does not write, the load buffer still contains the prefetched value when the loads are emitted. Hence, we can observe the outcome `r0=0, r1=0`.

As our generalized specification does not define fences, atomics, or mixed-size accesses, we cannot say that we are compliant with RVWMO, the RISC-V memory model. However, we manually inspected the standard base litmus tests (message passing, store buffering, etc.) that don't use the capabilities that we did not specify, and so far, our generalized specification does not violate RVWMO. Our preliminary compliance with RVWMO was not intentional but is welcome, as we fear that our long-term plan to tackle multicore systems modularly using our generalized specification could be jeopardized if our processor specification would already violate the memory model.

Chapter 6

A Full Example: Pipelined Processor

In this chapter, we showcase our complete workflow from modular specification and implementation to modular proof. The architecture we design and prove correct is a family of pipelined processors.

We will begin by giving both the details of the processor specification and the details of the generalized processor specification that we introduced in the previous chapter.

Then we will apply the modular-decomposition approach to decompose the processor itself into a frontend and a backend specification module. That will lead us to give our first processor implementation, built by putting together an implementation for the frontend and an implementation for the backend.

From there, we will give the full overview of the different refinements to prove, and in the rest of the chapter we will sketch those proofs.

6.1 Original and generalized processor specifications

We first start with the simple processor specification, in Figure 6-1.

The three interfaces to the outside world (MMIO, instruction memory, and data memory) are request-response interfaces. It means that the responses are not instantaneous. Those interfaces materialize as 6 methods that directly access the corresponding queues (`from_imem`, `to_imem`, etc.). For brevity, we don't show the code of those methods.

The machine sequentially executes instructions one after the other. Each instruction goes

```
#|
  mkReg pc; mkRf rf;
  mkReg state_machine;
  mkReg decode_inst;
  mkReg src_val1; mkReg src_val2;
  mkReg dest_val;
  mkFifo1 to_imem;
  mkFifo1 from_imem;
  mkFifo1 to_dmem;
  mkFifo1 from_dmem;
  mkFifoSpec to_mmio;
  mkFifoSpec from_mmio
|#.

Definition do_fetch := (rule
 (if (= {read state_machine} `fetch)
    (begin
        {write state_machine
          `decode}
        (set pc_req {read pc})
        {enq to_imem
          (# (* is_write *) 0
             (* addr *) pc_req
             (* data*) 0)})
    abort)).

Definition do_decode := (rule
 (if (= {read state_machine} `decode)
    (begin
        (set resp {first from_imem})
        (set (addr_resp data_resp) resp)
        {write state_machine `execute}
        (set decoded (decode data_resp))
        (set src1 {read1 rf
                    (src1 decoded)})
        (set src2 {read2 rf
                    (src2 decoded)})
        {write src_val1 src1}
        {write src_val2 src2}
        {write decode_inst decoded}
        {deq from_imem})
    abort)).
```

```
Definition do_execute := (rule
 (if (= {read state_machine} `execute)
    (begin
        {write state_machine `writeback}
        (set decoded {read decode_inst})
        (set val1 {read src_val1})
        (set val2 {read src_val2})
        (set output_alu
               (alu (# decoded val1 val2)))
        (set addr_dest (memaddr (# decoded val1)))
        (if (ismemory decoded)
            (if (store decoded)
              {enq to_dmem (# 1 addr_dest val2)}
              {enq to_dmem (# 0 addr_dest 0)})
            (if (ismmio decoded)
              {enq to_mmio (# (mmiostore decoded)
                              val1 val2)}
              pass))
        {write dest_val output_alu})
    abort)).

Definition do_writeback := (rule
 (if (= {read state_machine} `writeback)
    (begin
        {write state_machine `fetch}
        (set decoded {read decode_inst})
        (set producedvalue {read dest_val})
        (if (& (ismemory decoded)
               (! (store decoded)))
          (begin
            (set (addr data) {first from_dmem})
            (set producedvalue data)
            {deq from_dmem})
          (if (& (ismmio decoded)
                 (! (mmiostore decoded)))
            (begin
              (set producedvalue {first from_mmio})
              {deq from_mmio})
            pass))
        (if (has_destination decoded)
            {write rf (# (destination decoded)
                         producedvalue)}
            pass)
        {write pc (nextpc (# {read pc}
                            {read src_val1}))})
    abort)).
```

Figure 6-1: Description of a simple processor specification that sequentially fetches, decodes, executes and writes-back instructions. The verbosity comes from the multiplicity of cases: memory instructions (loads and stores), arithmetic instructions, control instructions, MMIO instructions (loads and stores). The code for methods, simply interacting with the 6 queues, is straightforward and omitted.

through 4 standard steps. First, the instruction goes through the fetch step. The instruction is requested from instruction memory (through the `to_imem` queue), using the address stored in the PC register. Then the instruction received from instruction memory gets decoded, then it is executed, potentially producing a memory or an MMIO request (load or store). Finally, the effect of the instruction is committed in the writeback step, updating both the program counter and the register file. The following observations are crucial to keep in mind, as they highlight the architectural simplicity of this specification:

- In this specification, there is at most one instruction-load request in flight at any time.
- In this specification, there is at most one data-memory request (load or store) in-flight at any time.
- The machine is only working on executing a single instruction at any time. (No multiple instructions in flight.)
- The machine *directly* emits requests and receives responses using the 3 pairs of queues: `to_imem`, `from_imem`, `to_dmem`, `from_dmem`, `to_mmio`, `from_mmio`.

We now compare this specification to our generalized processor specification, presented in Figure 6-2.

We highlight in red the changes specific to the generalized specification. Note that a significant part of the original specification is left unchanged, with only the following changed:

- There are 2 new rules `emit_dndl` and `emit_indl`, which emit instruction and data loads to nondeterministic addresses by pushing requests into the appropriate queues.
- Upon reception of a memory response, we put the response in the corresponding load buffer instead of putting the response *directly* in the response queue.
- When fetching, instead of sending a request to instruction memory, the specification queries the local instruction-load buffer. The load buffer returns instantaneously with the last value it holds for that address (if it held any value). The value is then enqueued into the `from_imem` queue as if it had just received the response from the environment.

```
(* New extra submodules:
mkLB ild_buffer; mkLB dld_buffer;
mkND nondet *)

Definition enq_iresp :=
 (action_method (el) {loadResp ild_buffer el}).

Definition enq_dresp :=
 (action_method (el) {loadResp dld_buffer el}).

Definition emit_indl := (rule
    (set addr {choose nondet})
    {loadReq dld_buffer addr}
    {enq to_imem (# 0 addr 0)}).

Definition emit_dndl := (rule (begin
      (set addr {choose nondet})
      {loadReq dld_buffer addr}
      {enq to_dmem (# 0 addr 0)})).

Definition do_fetch := (rule
  (if (= {read state_machine} `fetch)
      (begin
          {write state_machine `decode}
          (set pc_req {read pc})
          (set buffer_result
            {lookup ild_buffer pc_req})
          (set (valid result) buffer_result)
          (if (= valid 1)
            {enq from_imem (# pc_req result)}
            abort))
      abort)).

Definition do_decode := (rule
  (if (= {read state_machine} `decode)
      (begin
          (set resp {first from_imem})
          (set (addr_resp data_resp) resp)
          {write state_machine `execute}
          (set decoded (decode data_resp))
          {write src_val1
            {read1 rf (src1 decoded)}}
          {write src_val2
            {read2 rf (src2 decoded)}}
          {write decode_inst decoded}
          {deq from_imem})
      abort)).
```

```
Definition do_execute := (rule
  (if (= {read state_machine} `execute)
    (begin {write state_machine `writeback}
      (set decoded {read decode_inst})
      (set val1 {read src_val1})
      (set val2 {read src_val2})
      (set output_alu (alu (# decoded val1 val2)))
      (if (ismemory decoded)
        (begin
          (set addr_dest (memaddr (# decoded val1)))
          (if (store decoded)
            (begin
              {storeReq dld_buffer addr_dest}
              {enq to_dmem (# 1 addr_dest val2)})
            (begin
              (set res {lookup dld_buffer addr_dest})
              (set (valid result) res)
              (if (= valid 1)
                {enq from_dmem (# addr_dest result)}
                abort))))
        (if (ismmio decoded)
          {enq to_mmio
            (# (mmiostore decoded) val1 val2)}
          pass))
      {write dest_val output_alu})
    abort)).

Definition do_writeback := (rule
 (if (= {read state_machine} `writeback)
   (begin {write state_machine `fetch}
     (set dinst {read decode_inst})
     (set nval {read dest_val})
     (if (& (ismemory dinst) (! (store dinst)))
       (begin
         (set (addr data) {first from_dmem})
         (set nval data)
         {deq from_dmem})
       (if (& (ismmio dinst) (! (mmiostore dinst)))
         (begin
           (set nval {first from_mmio})
           {deq from_mmio})
         pass))
     (if (has_destination dinst)
       {write rf (# (destination dinst) nval)}
       pass)
     {write pc
       (nextpc (# {read pc} {read src_val1}))})
   abort)).
```

Figure 6-2: Generalized processor specification. The differences with the original specification are highlighted in red.

- When executing a memory operation, there are two cases:

    1. It is a memory load; instead of sending the request to data memory, the specification queries the local load buffer. The load buffer returns instantaneously with the last value it holds for that address (if it held any value). The value obtained is then enqueued in the `from_dmem` queue, as if it had been a response received from data memory.

    2. It is a memory store; the load-buffer entry corresponding to the address is invalidated, and we send the store to the actual memory

Notice that our previous observations need to be updated, as they highlight the more sophisticated memory behaviors showcased by this generalized specification:

- There is an arbitrary number of outstanding instruction requests in-flight at any time.

- There is an arbitrary number of outstanding data-load requests in-flight at any time.

- Because there is no acknowledgment on stores, there can be several outstanding store requests in-flight at any time.

- The machine is still executing a single instruction at any time. (No multiple instructions in flight).

The next step is to describe precisely the inner workings of the load buffers.

6.1.1 Load buffers

In Figure 6-3, we show the specification of the load buffer. Note that load buffers are helper modules. We only use them for specification purpose;; as such, we never actually build an implementation that would be a synthesizable hardware load buffer.

The way the load buffer works is as follows: for each address, it keeps track of:

- A field *res* to keep track of an (ordered) list of response values that have not been invalidated by stores to that same address.

```
Record Entry := {
    res : list N;
    outV : nat;
    outI : nat
  }.

Definition invalidate' (e : Entry) : Entry :=
  {| res := tl (res e);
    outV := outV e ; outI := outI e|}.

Definition storeReq' (e : Entry) : Entry :=
  {| res := [];
     outV := 0 ; outI := outI e + outV e|}.

Definition loadReq' (e : Entry) : Entry :=
  {| res := res e;
     outV := outV e + 1; outI := outI e |}.

Definition loadResp' (e : Entry) (data : N) :=
  match outI e with
  | O ⇒ match outV e with
    | O ⇒
      (* Impossible case  *)
      {| res := res e;
         outV := O; outI := O |}
    | S n ⇒
         {| res := res e ++ [data];
            outV := n; outI := O |}
    end
  | S n ⇒
     {| res := res e;
     outV := outV e; outI := n |}
  end.

Definition lossy_map_state :=
  N → Entry.
```

```
Definition downgrade st new_st :=
   ∃ (map_st : lossy_map_state) (addr_killed : N),
     st = *( map_st )* ∧
     new_st = *( fun addr ⇒ if addr =? addr_killed
         then invalidate' (map_st addr)
         else map_st addr )*.

Definition lookup arg st ret :=
  ∃ (map_st : lossy_map_state),
  st = *( map_st )* ∧
  match res (map_st arg) with
  | h::_ ⇒ ret = {# 1 h}
  | _ ⇒ ret = {# 0 0}
  end.

Definition loadReq addr st ret :=
  ∃ (map_st : lossy_map_state),
  st = *( map_st )* ∧
  ret = *( fun new_addr ⇒  if new_addr =? addr
      then loadReq' (map_st addr)
      else map_st new_addr )*.

Definition storeReq addr st ret :=
  ∃ (map_st : lossy_map_state),
  st = *( map_st )* ∧
  ret = *( fun new_addr ⇒ if new_addr =? addr
              then storeReq' (map_st addr)
              else map_st new_addr )*.

Definition loadResp arg st ret :=
  ∃ (map_st : lossy_map_state),
  st = *( map_st )* ∧
  dlet {addr data} := arg in
  (outI (map_st addr) + outV (map_st addr)) ≠ O ∧
  ret = *( fun new_addr ⇒ if new_addr =? addr
                then loadResp' (map_st addr) data
                else map_st new_addr )*.

Global Instance mkLB : module _ :=
  primitive_module#(rules [downgrade]
    vmet [lookup] amet [loadReq; storeReq; loadResp]).
```

Figure 6-3: The load buffer stores one *Entry* per address. The entry contains two counters to characterize the number of outstanding load requests, plus a list of values that have been previously seen. On the left we define helpers to manipulate entries, and on the right we define the rule, the value methods and the action methods using those helpers.

- A field *outI* that counts the number of outstanding *invalid* load requests: requests that were emitted before a later store. Hence, we should drop the responses when they arrive.

- A field *outV* that counts the number of currently *valid* outstanding load requests. We will add the responses to the list when they arrive.

The bookkeeping machinery (*outI* and *outV*) is slightly subtle, so let us see an example scenario for why it is necessary and how it works.

Consider the following trace of events: `LoadReq addr;  LoadReq addr; LoadResp addr v0; StoreReq`

The sequence starts with two load requests to the same address, followed by a response for the first of the two loads and then a store. Finally, the response to the second load arrives, after the store was emitted.

A naive idea to specify the load buffer would be to append responses when we receive them and to clear the list of values observed when we emit a store. This would be incorrect, as the previous sequence would lead to an incorrect state of the load buffer. The last load response would fill the load buffer *after* the store has cleared it, making it possible to read an old stalled value from the load buffer (the value before the store was performed).

Hence, we need to handle the first load response differently from the second load response: the first one is a legitimate response, while the second one is a response that should *not* be added to the load buffer, as it is already out-of-date when it arrives. Hence, we need to be able to decide, for any incoming load response, if the response is a valid response or if it is a response that has already been shadowed by a store and hence should be discarded.

Our two-counters mechanism allows us to decide whenever to discard a load response. The scheme goes as follows. At any time, the sum of *outI* and *outV* counts the number of both invalid (outdated) and valid outstanding load requests to that address. If we perform a store, we update the counters to declare every current outstanding load to be invalid, as they all became shadowed by the store. If we perform a load, we simply bump the *outV* counter. When we receive a load response, if *outI* was not zero, it means that we expected an invalid response to that address, so we discard the response and decrement *outI*. Otherwise, we append the value of the response to the list of values for that address in the load buffer, and

we decrement $outV$.

The load buffer also has a self-invalidation rule: at any time, the load buffer can spontaneously discard the oldest value that was read for a given address.

Note that in the single-core case (that is the case we fully explored for this thesis), we can easily prove the following invariant: *there are never two different values at a given address of the load buffer*. One might then wonder why, in that single-core case, we decided to record a list of previously seen values instead of the last seen value, if all the elements of the lists are guaranteed to be the same. This is tying back into what we introduced in chapter 5: choosing the proper data structure for the specification can make a significant difference in stating invariants and performing proofs. In this case, using a list allows us to more easily state some of the invariants that relate the state of the load buffer with the state of the pipeline (what the processor internally believes are the memory requests it already sent).

6.2 Breaking up the generalized processor into smaller parts

In the previous section, we introduced the generalized processor specification that allows us to tackle the proof of correctness of the full design by splitting between the memory and the processor. We can play the same game, hierarchically, to split the processor itself into two smaller submodules. This next level of decomposition splits our generalized processor specification between a generalized frontend and a generalized backend.

We show the architectural diagram of our new decomposed processor in Figure 6-4.

Contrary to our generalized specification that was sequentially fetching, decoding, executing and writing back the effects of an instruction, this new specification speculatively and asynchronously requests the next instructions, while the backend *sequentially* decodes, executes and writes-back. As such, this new decomposed generalized processor specification works quite differently from our previous generalized processor specification: it introduces a form of instruction-fetch speculation and of basic pipelining between the frontend and the backend. The backend specification, however, is internally completely sequential.

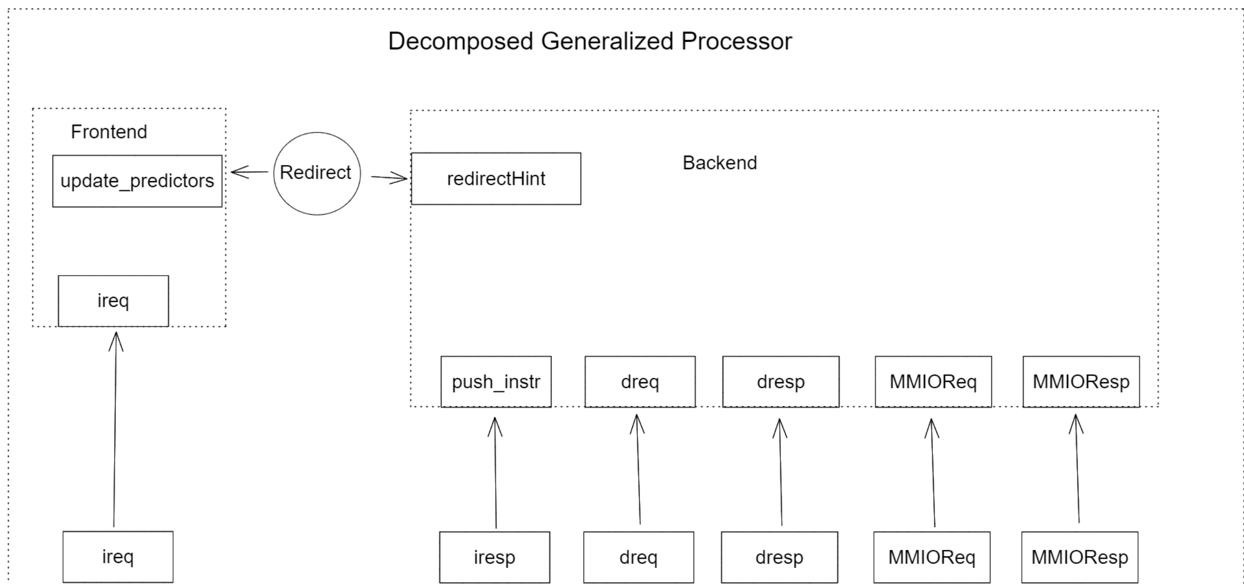


Figure 6-4: Architectural sketch of our decomposed processor specification: the frontend asynchronously generates requests for the external instruction memory, the external instruction memory responds (with a response tagged with the corresponding PC), and the response is pushed to the backend. The backend buffers those instructions and filters the ones that it is not expecting (the mispredicted instructions). We represent liberally the interface of the consumer interface of the queues with a single method, instead of the two methods (`deq` and `first`) of the real interface.

This modular decomposition is also the first decomposition that really starts blurring the boundary between specifications and implementations. We consider it to be a specification because the backend is not pipelined, and the frontend is not synthesizable. However, we clearly see a family of implementations starting to emerge from this decomposition. We will see that the gap between this specification and our final implementation is not so large and mainly resides within the backend.

We now elaborate on the precise interface of the frontend and the backend modules and their respective specifications. We also give our implementations for both modules. Note that our decomposition is implicitly limiting the family of processors that we can prove.

6.2.1 Frontend

Specification The interface of the frontend of this new decomposed processor is composed of the following methods:

- A `first_ireq` value method, which does not have arguments and returns the next instruction request that the frontend would like to send to the instruction memory.
- A `deq_ireq` action method, to dequeue the front request.
- An `update_predictors` action method to potentially update the different prediction structures of the frontend.

What might be more surprising is the actual specification of our frontend module, presented in Figure 6-5.

In this specification, the methods `update_predictors` and `deq_ireq` are always ready and do not do anything. That is because the frontend is very unconstrained: the frontend is simply generating an arbitrary stream of instructions. No extra constraints are actually necessary to guarantee the correctness of the full processor.

Remark (Focus on safety). If we cared about liveness, the story would not be the same, as we would need some guarantees that the next real instruction eventually appears in the stream.

```
Local Instance frontend_modules : instances :=
  #|
    Nondet.mkND nondet
  |#.

Definition first_ireq :=
    (value_method () (# (* is_write *) 0 (* addr *) {choose nondet} (* data*) 0)).

Definition deq_ireq := (action_method () pass).

Definition update_predictors := (action_method (el) pass).
```

Figure 6-5: Frontend specification

Implementation We now give a minimal pipelined implementation of the frontend:

```
Local Instance frontend_modules : instances :=
  #|
    reg.mkReg pc;
    mkFifo stream
  |#.

Definition first_ireq :=
  (value_method () {first stream}).

Definition fetch_next :=
  (rule
    (begin
      {enq stream (+ pc 4)}
      {write pc (+ pc 4)}).

Definition deq_ireq := (action_method () {deq stream}).

Definition update_predictors :=
    (* correcting a misprediction *)
```

```
(action_method (el)
    {write pc el}).
```

One could also choose to clear the `stream` queue on a redirection, to get a faster restart. One can also easily make this frontend more advanced, for example using a BTB predictor, instead of using `pc + 4`.

In the next subsection, we explain the backend of our decomposed specification. Critically, the backend needs to keep track of what is the next expected instruction, to filter the wrong-path instructions that could be generated by the frontend.

6.2.2 Backend

The backend specification is presented in Figure 6-6. The interface is as expected:

- The `push_instr` action method is the entry point of instructions in the backend: it is the method called whenever a new instruction response in coming back from instruction memory.
- The `first_dreq` value method returns the next request to data memory that the backend would like to emit.
- The `deq_dreq` dequeue the next request to data memory.
- The `enq_dresp` action method is called whenever we have a response available from data memory, and we want to communicate it to the backend.
- `mmio_req`, `enq_mmio` and `deq_mmio` are similar but for the request to MMIO.
- `redirect` and `redirectv` are the sources of the redirections for the frontend.

Notice that this backend specification is sharing most of its of code with the generalized processor specification of Figure 6-2. There are only a few differences:

- The backend specification does not fetch, it directly receives fetched data from its `push_instr` method.

```
#|
  mkReg expected_pc; mkRf rf;
  mkReg state_machine; mkReg decode_inst;
  mkReg src_val1; mkReg src_val2;
  mkReg dest_val;
  mkND nondet; mkDLB dld_buffer;
  mkFifoSpec to_dmem; mkFifo1 from_dmem;
  mkFifoSpec to_mmio; mkFifoSpec from_mmio;
  mkFifoSpec instr_stream
|#.

Definition enq_dresp :=
  (action_method (el)
    {loadResp dld_buffer el}).

Definition redirect :=
  (action_method () pass).

Definition redirectv :=
  (value_method () {choose nondet}).

Definition push_instr :=
  (action_method (resp_from_imem)
    {enq instr_stream resp_from_imem}).

Definition emit_dndl := (rule
  (begin
    (set addr {choose nondet})
    {loadReq dld_buffer addr}
    {enq to_dmem (# 0 addr 0)})).

Definition do_decode := (rule
  (if (= {read state_machine} `decode)
   (begin
     (set (addr_resp data_resp)
       {first instr_stream})
     (if (= addr_resp {read expected_pc})
      (begin
        {write state_machine `execute}
        (set decoded (decode data_resp))
        (set src1 {read1 rf (src1 decoded)})
        (set src2 {read2 rf (src2 decoded)})
        {write src_val1 src1}
        {write src_val2 src2}
        {write decode_inst decoded})
      {deq instr_stream} ))
   abort)).
```

```
Definition do_execute := (rule
 (if (= {read state_machine} `execute)
   (begin {write state_machine `writeback}
     (set decoded {read decode_inst})
     (set val1 {read src_val1})
     (set val2 {read src_val2})
     (set output_alu (alu (# decoded val1 val2)))
     (if (ismemory decoded)
       (begin
         (set addr_dest (memaddr (# decoded val1)))
         (if (store decoded)
           (begin
             {storeReq dld_buffer addr_dest}
             {enq to_dmem  (# 1 addr_dest val2 )})
           (begin
             (set buffer_result
               {lookup dld_buffer addr_dest})
             (set (valid result) buffer_result)
             (if (= valid 1)
                 {enq from_dmem (# addr_dest result )}
                 abort))))
       (if (ismmio decoded)
         {enq to_mmio (# (mmiostore decoded) val1 val2)}
         pass))
     {write dest_val output_alu})
   abort)).

Definition do_writeback := (rule
 (if (= {read state_machine} `writeback)
   (begin {write state_machine `decode}
     {deq instr_stream}
     (set decoded {read decode_inst})
     (set producedvalue {read dest_val})
     (if (& (ismemory decoded)  (not (store decoded)))
       (begin
         (set (addr data) {first from_dmem})
         (set producedvalue data)
         {deq from_dmem})
     (if (& (ismmio decoded)  (not (mmiostore decoded)))
       (begin
         (set producedvalue {first from_mmio})
         {deq from_mmio})
       pass))
     (if (has_destination decoded)
         {write rf
           (# (destination decoded) producedvalue)}
         pass)
     {write expected_pc
       (nextpc (# {read expected_pc} {read src_val1}))})
   abort)).
```

Figure 6-6: Backend specification

- The rest of the specification (decode, execute, and writeback) is left unchanged.

- There is redirection machinery, though the machinery allows generation of arbitrary redirection values: this is a situation similar to the frontend, as again the redirection machinery is only relevant for liveness but not for safety.

Implementation We now give an implementation of this backend specification in Figure 6-7 and Figure 6-8. Let us highlight the important parts of this implementation and the key differences with the specification:

- It is pipelined: it works simultaneously on several instructions.

- There is no rule that sends instruction or data loads to arbitrary addresses, and there are no more load buffers. Instead, there is a simple load queue (`ldQ_sent` and `ldQ_received`) that allows multiple outstanding loads, even to the same address.

- The machine stalls on MMIO and memory stores and executes those instructions sequentially. This is a similarity with the specification.

- The implementation filters wrong-path instructions at the time of execution instead of at decoding time.

- It uses a different kind of register file: the register file keeps track of the outstanding dependencies (Read-After-Write (RAW) and Write-After-Write (WAW) hazards).

Remark (Conservative stalling). Notice that stalling for memory store is a conservative implementation of a limitation of the specification that we already discussed in subsection 5.3.3: *Stores can be emitted only when all previous outstanding program loads have received their responses*. We implement this policy conservatively by making sure that there are no instructions ahead in the pipeline when we emit a store. We could have a more aggressive store-issuing policy that would allow the stores to be sent as long as there are no load instructions still lying ahead.

```
#|mkReg expected_pc; mkRf rf; mkFifoSpec d2e; mkFifoFSpec e2w;
  mkFifoFSpec ldQ_sent; mkFifoFSpec ldQ_received; mkReg processing_st_mmio;
  mkFifoSpec to_dmem; mkFifoSpec from_dmem; mkFifoSpec to_mmio; mkFifoSpec from_mmio |#.

Definition push_instr := (action_method (pc_received data_resp)
    (begin
      (set decoded (decode data_resp))
      (if (has_destination decoded)
        {declare_write rf (destination decoded)}
        pass)
      (set src1 {read1 rf (src1 decoded)})
      (set src2 {read2 rf (src2 decoded)})
      {enq d2e (# src1 src2 pc_received decoded)})).

Definition do_execute := (rule
  (begin
    (set (val1 val2 coming_pc decoded) {first d2e})
    {deq d2e}
    (set cur_pc {read expected_pc})
    (if {read processing_st_mmio} abort pass)
    (if (= cur_pc coming_pc)
      (begin
        (set output_alu (alu (# decoded val1 val2)))
        (set ppc (nextpc (# cur_pc val1)))
        {write expected_pc ppc}
        (if (ismemory decoded)
          (begin
            (set addr_dest (memaddr (# decoded val1)))
            (if (store decoded)
              (begin
                (set empty_e2w {assume_empty e2w})
                {write processing_st_mmio 1}
                {enq to_dmem (# 1 addr_dest val2)}
                {enq e2w (# 1 coming_pc ppc decoded addr_dest)})
              (begin
                {enq ldQ_sent addr_dest}
                {enq to_dmem (# 0 addr_dest 0)}
                {enq e2w (# 1 coming_pc ppc decoded addr_dest)})))
          (if (ismmio decoded)
            (begin
              {write processing_st_mmio 1}
              (set empty_e2w {assume_empty e2w})
              {enq to_mmio (# (mmiostore decoded) val1 val2)}
              {enq e2w (# 1 coming_pc ppc decoded output_alu)})
            {enq e2w (# 1 coming_pc ppc decoded output_alu)})))
      {enq e2w (# 0 coming_pc 0 decoded output_alu)}))).
```

Figure 6-7: Backend implementation 1/2

```
Definition do_writeback :=
  (rule
    (begin
      (set (valid coming_pc ppc decoded dest_val) {first e2w})
      {deq e2w}
      (set producedvalue dest_val)
      (if valid
        (begin
          (if (& (ismemory decoded) (! (store decoded)))
            (begin
              (set (addr data) {first from_dmem})
              {deq ldQ_received}
              (set producedvalue data)
              {deq from_dmem})
            (if {read processing_st_mmio}
              (begin
                (if (& (ismmio decoded) (! (mmiostore decoded)))
                  (begin
                   (set producedvalue {first from_mmio})
                    {deq from_mmio})
                pass)
                {write processing_st_mmio 0})
              pass))
          (if (has_destination decoded)
            {write rf (# (destination decoded) producedvalue)}
            pass))
        (if (has_destination decoded)
          {withdraw_write rf (destination decoded)}
          pass)))).

Definition enq_dresp :=
  (action_method (addr data)
    (begin
      (set addr_expected {first ldQ_sent})
      {deq ldQ_sent}
      (if (= addr_expected addr)
        (begin
          {enq ldQ_received addr}
          {enq from_dmem (# (*addr*) addr  (*data*) data )})
        abort))).
```

Figure 6-8: Backend implementation 2/2

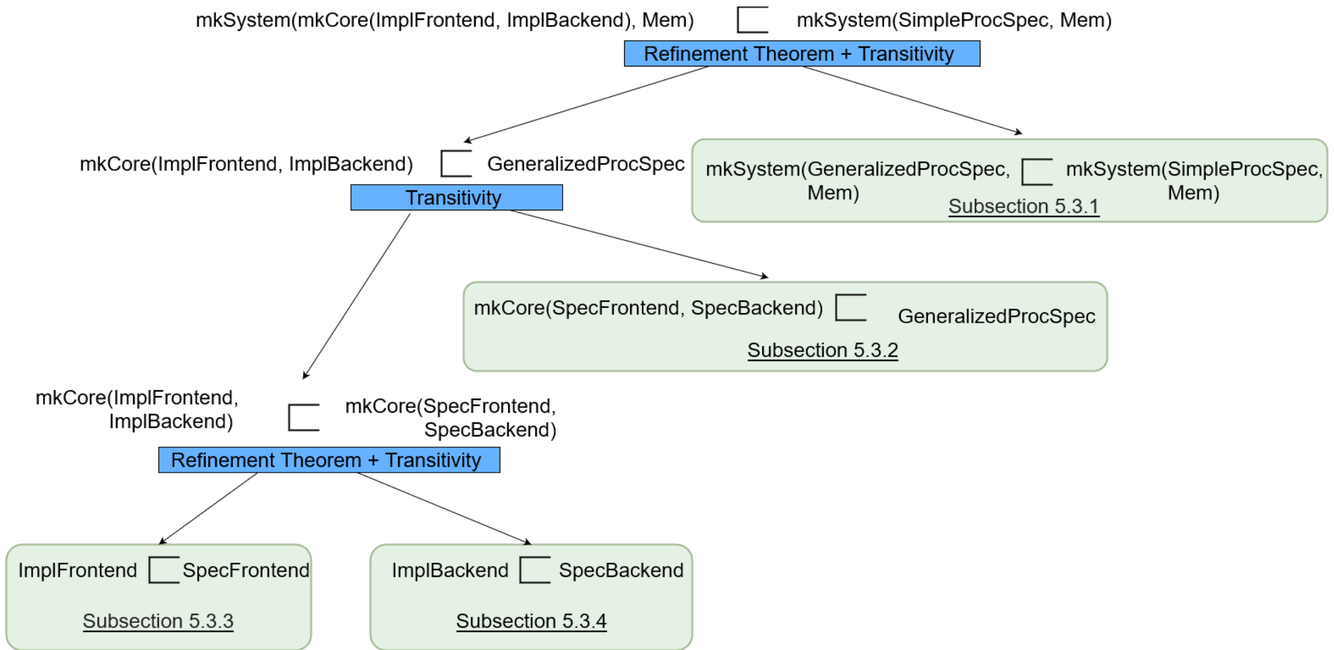


Figure 6-9: Arrangement of refinements.

6.3 Processor proofs

Now that we have presented the way we decompose our top-level specification into smaller specification modules and given the implementations of our leaf modules, we present the different proofs we completed.

In Figure 6-9 we give an overview of the arrangement of the different proof obligations that we present in the rest of this section. The figure showcases how the different refinements assemble to produce our top-level theorem statement. A key side benefit of this modular decomposition is that it gives us proof reusability. If we change the frontend or the backend of our implementation machine, we will often need to only alter the corresponding subproof.

6.3.1 The generalized processor specification is valid

We first start with a proof of the theorem stating that our generalized specification is a valid generalization of the processor specification, in the context of the full-system specification:

Theorem 6.1 (The generalized processor specification is a valid generalization).

$$mkSystem(mkGeneralizedProc, mkMemory) \sqsubseteq mkSystem(mkSimpleProc, mkMemory)$$

Proof. Let us first remember at a high level what this theorem is about. This proof is responsible for bridging a gap between the implementation (the left side, which is our generalized specification), which has potentially many nondeterministic loads in-flight to memory, with the specification that *never has more than one load in-flight to memory.*

The proof needs to show that those extra loads, which do drastically transform the way memory is used, *do not change the external (MMIO) behavior of the system.* The proof is about how the two processors interact with their memory, especially the effect of the load buffers and the two nondeterministic load rules.

The idea of the refinement relation is to keep the implementation and the specification machines in-sync, verifying the following key conditions:

1. The PC, register file, and all the other internal structures of the two processors are kept completely in-sync. Only structures related to the memory and the load buffers get related in a nontrivial way.

2. The specification is kept in a state where the load requests are resolved: no load request is outstanding, and we always push the requests all the way through the specification system, down to the response queue of the core.

3. The simulation guarantees that both the instruction response queue and the instruction load buffer of the implementation, when they contain a value, always reflect the content of the instruction memory:

```
∀ addr,
List.Forall (fun x ⇒ mem imem_i addr = x)
            (ild_buffer addr) ∧
List.Forall (fun x ⇒ dlet {addr data} := x in
            mem imem_i addr = data)
            (resps imem_i)
```

4. The most sophisticated part of the relation relates the two data-memory subsystems: we state that if we were to flush the outstanding requests of the implementation machine – that is push the requests through the memory system and update the load buffer and the memory accordingly – we would obtain a new load buffer and a new memory where the new memory would be precisely the memory of the specification system. Moreover, the new load buffer would record no outstanding (valid or invalid) loads, and all the values in the load buffer would be in agreement with the memory. We note this relation `mem_flushes mi ms reqs dlb`, where `mi`,`reqs` and `dlb` are the components of the data-memory subsystem of the implementation and `ms` is the corresponding part in the specification. (See the paragraph below on data-memory flushing for more explanation and detail on how we precisely state and manipulate the `mem_flushes` predicate.)

Remark (Flushing as a simulation relation). Every part of this relation can be seen as a statement declaring that *once we flush a part of the implementation machine, we obtain the corresponding part of the specification machine*. For example, the relation constraining the instruction memories is exactly stating that once we will have flushed the instruction-load requests, the state of the instruction-load buffer will agree with the state of the instruction memory of the specification. Because the machine never emits stores to instruction memory, the flushing-like invariant is easier to state for instruction memory than it is for the data memory. Another simpler (but degenerate) example is for the parts of state related to the processor. The implementation processor parts are required to be the same as the specification processor parts: this is a degenerate form of flushing.

Before we explain in more detail the relation that states the flushing of data memory, let's notice that we are close to being able to prove our desired refinement.

Indeed, thanks to (1.) and the fact that the two specifications almost run the same code (remember Figure 6-2 and Figure 6-1), the simulation relation is almost straightforward. More precisely:

Decode and writeback The simulation for both the `decode` and the `writeback` transitions requires close to little work, as those two transitions do not touch either of the load buffers.

Fetch The simulation for the `fetch` transition is slightly more sophisticated: the specification is reading the instruction load buffer and pushing the value seen there in the `from_imem` queue. To achieve the same effect, the implementation needs to perform three rules: (a) emit a load to the `to_imem` queue with the `fetch` transition of the *specification*, (b) pull that load and push it to the memory (using the parent rule connecting the processor and the instruction memory),and (c) then pull the response from the memory and push it to the `from_imem` queue of the specification. As the instruction-load buffer reflects what is in the memory, thanks to part (3.) of the simulation relation, it is easy to prove that this sequence of rules of the specification mimics the `fetch` transition of the implementation.

Execute The simulation for the `execute` transition is the trickiest and is the one requiring our custom `mem_flushes` predicate. Mimicking the effect of the `execute` transition of the implementation with transitions of the specification requires a different strategy depending on the kind of instruction that we are simulating.

- If the instruction is a nonmemory instruction, the situation is similar to the `decode` and `writeback` cases, as the implementation's transition does not touch the data-load buffer. The strategy is simply to run the `execute` transition of the specification.

- If the instruction is a store, the strategy is also simply to run the `execute` transition of the specification. However, there is a bit of administrative work to reestablish the `mem_flushes` predicate after the transition.

- If the instruction is a load, the strategy is similar to the `fetch` case: (a) emit a load to the `to_dmem` queue with the `execute` transition of the *specification*, (b) push that load through the data memory of the specification (using the parent rule connecting the processor and the instruction memory), (c) pull the response from the memory to push it into the `from_dmem` queue of the specification processor. For this strategy to work, we need to guarantee that the response produce by step (b) is the same as the one observed when the `execute` rule of the implementation queries its data-load buffer. In other words, the strategy will work if we can prove the following lemma:

```
Lemma flushes_read_dlb:
  ∀ mi ms reqs dlb, mem_flushes mi ms reqs dlb →
  ∀ addr el,
  In el (res (dlb addr)) → mem ms addr = el.
```

In the next paragraph, we both explain the precise definition of the `mem_flushes` predicate and its relation to the proof of the lemma.

Data-memory flushing As we have outlined in the previous paragraph, we need to formalize a proposition that expresses the following property: *If we were to flush the outstanding requests of the implementation machine, we would obtain a new load buffer and a new memory. The new memory would be precisely the memory of the specification system, the new load buffer would contain no outstanding loads, and all the values in the load buffer would be in agreement with the content of the memory.*

We elaborate the formal construction of this flushing predicate, as it is a technique that we found to be very productive in many proofs. We encode the flushing property as an inductive proposition with a few cases. First, the base case: when there are no outstanding requests or responses, a memory with any compatible load buffer flushes to itself:

```
∀ (m : N → N) (dld_buffer : N → Entry),
  (* Ensuring that [dld_buffer] is compatible with [m]: *)
  (∀ addr,
    outI (dld_buffer addr) = 0 ∧
    outV (dld_buffer addr) = 0 ∧
    (∀ observed, In observed (res (dld_buffer addr)) → m addr =  observed)) →
  (* Then implementation memory [m] with no requests ([]) and the load buffer
  [dlb_buffer] flushes to [m]. *)
  mem_flushes {| mem := m; resps := [] |} {| mem := m; resps := [] |} [] dld_buffer
```

And then we simply have 6 inductive cases, one case per potential transition of the system:

- Pushing a store in the request queue.

- Pushing a load in the request queue.
- Resolving a load from the head of the request queue.
- Performing a store from the head of the request queue.
- Dequeuing a load response from the memory-response queue and updating the load buffer.
- Invalidating an entry in the data-load buffer.

For each case, we define a case of the inductive proposition describing how flushability gets transformed by the corresponding transition. The 6 cases are similar, so we only show the first one:

```
∀ mi ms req reqs dlb,
dlet {iswrite addr data} := req in
iswrite = 1 →
mem_flushes mi ms reqs dlb  →
mem_flushes mi
    {| mem := (fun a ⇒ if addr =? a then data else (mem ms a));
      resps := resps ms |}
    (reqs ++ [req])
    (fun a ⇒ if addr =? a then storeReq' (dlb addr) else dlb a)
```

This expression reads as: if we already know that the implementation memory `mi` coupled with a load buffer `dlb` and a sequence of requests `reqs` flushes to an equivalent specification memory `ms`, then we know that `mi` coupled with an extended sequence of requests `reqs++[store(addr,data)]` and an updated load buffer `dlb` where we cleared the entry at address `addr` flushes to a slightly updated `ms`: `mem ms addr` is set to `data`.

Now, constructing this inductive proposition allows us to easily prove the lemma `flushes_read_dlb` by induction over the derivation of the flushing. To simplify, this proof is an induction over the length of the flushing sequence. Moreover, the inductive is following the structure of

the transitions, so in the refinement proof proving that the predicate `mem_flushes` stays true after each transition is quite straightforward.

This conclues the overview of the proof of validity of the generalized specification. □

6.3.2 Refinement between decomposed generalized specification and generalized specification

Theorem 6.2 (The decomposed processor specification is valid).

$$mkCore(SpecFrontend, SpecBackend) \sqsubseteq GeneralizedProcessorSpec$$

Proof. This second refinement is significantly easier to prove than the previous one. As we explained when we introduced the decomposed processor in section 6.2, contrary to the generalized specification, the decomposed processor specification decouples the fetching of instructions from the rest of the machine. So in this refinement, while the specification runs instructions completely sequentially, the implementation speculatively and asynchronously requests the next instructions, while the backend *sequentially* decodes, executes and writes-back.

So, the focus of this refinement is about a transformation of instruction fetches. The implementation uses a speculative frontend, generating instruction requests to arbitrary addresses and sending the responses directly to the backend. The implementation does not use an instruction-load buffer. In contrast, the specification uses an instruction-load buffer. While the implementation machine's backend alternates its state between `Decode`, `Execute` and `Writeback`, the specification machine also alternates with the `Fetch` state. The correspondence between those two machines is easy to figure out:

```
(i_state_machine ≠ decode → i_state_machine = s_state_machine) ∧
(i_state_machine = decode → s_state_machine = fetch)
```

where `i_state_machine` denotes the state of the implementation's backend, while `s_state_machine` denotes the state of the specification.

Moreover, we have an elementary form of flushing invariant for the load buffer of the specification: all the responses that are in the instruction-response queue of the implementation machine are compatible with the instruction-load buffer of the specification:

```
∀ addr,
  let load_to_addr := List.flat_map
          (fun resp ⇒ dlet {el_addr el_data} := resp in
                       if el_addr =? addr then [el_data] else []) fifo_from_imem in
  ild_buffer addr = load_to_addr
```

The rest of the state of the implementation and specification machines stay in perfect sync during the simulation proof. Once we set up this simulation relation, the proof of refinement goes through very directly.

The most interesting case in that proof occurs for the transition of the implementation machine corresponding to the decoding of a wrong-path instruction. Thanks to the proposition relating the states of the two machines, we know that the specification is in the `fetch` state. As expected, in that case, we do not take any step on the specification side, and we leave the specification in its `fetch` state. Indeed, as the instruction is a wrong-path instruction, the implementation machine is not actually doing anything to the architectural state. So, no work needs to be performed on the specification side.

□

6.3.3 Frontend refinement

Theorem 6.3 (Frontend refinement).

$$ImplFrontend \sqsubseteq SpecFrontend$$

Proof. This is by far the easiest refinement to prove (even easier than the register-file example we gave as our first proof of refinement). This is because, as mentioned, the specification does not constrain the implementation at all. □

6.3.4 Backend refinement

Theorem 6.4 (Backend refinement).

$$ImplBackend \sqsubseteq SpecBackend$$

Proof. This proof is the most sophisticated we achieved in our framework, as the gap between the two designs studied is the largest gap we have seen so far. Indeed, as the implementation is pipelined, it has to worry about data hazards. Moreover, it uses a simple load queue to control its interactions with memory. In contrast, the specification runs instructions sequentially and uses our abstract load buffer to remember its loads sent to memory. The two machines also do not filter out wrong-path instruction the same way: the specification machine filters wrong-path instruction at decode time, while the implementation machine has to wait for execute time. This large impedance mismatch between the two designs leads to a more complex simulation relation.

Our strategy is to leverage the following architectural wisdom. *While instructions in-flight in the implementation have all been partially executed at different levels, there exists a point in the pipeline when the implementation performs the step that will atomically* commit *the effect of the instruction.*

Our strategy will simply be to have our simulation keep the specification in sync with the last committed instruction of the implementation. However, this commit point is not completely straightforward to identify in our machine. While the `writeback` stage is the commit point of most instructions, it is not the commit point for all instructions. More precisely, both `store` instructions and `MMIO` instructions have an earlier commit point: they commit at execute time.

So, our simulation strategy will reflect the dynamically moving commit point. Let us present piece-by-piece the key parts of our simulation relation, through the different sub-modules it constrains.

Register file The easiest part is the relation related to the register files of the implementation and the specification, as the two registers files always agree on the value for each

register:

```
∀ idx,
let '(valid, idata) := irf idx in
let sdata := rf idx in
idata = sdata
```

Data memory The part of the relation explaining the mapping between the load buffer and the load queue and load-response queue is fairly straightforward: we simply state that the load buffer is guaranteed to contain the same responses as the one we have received in the load queue of the implementation. Moreover, the load queue also agrees with the load buffer on the number of outstanding valid loads. And as we never send a store when the previous loads did not return, we guarantee that there are never invalid outstanding loads to memory:

```
(∀ addr,
  (* The content of the load buffer for an arbitrary address *)
  res (dld addr) =
  (* is equal to the responses received from memory for that address *)
  map (fun y ⇒ dlet {iaddr idata} := y in idata)
      (filter (fun x ⇒ dlet {iaddr idata} := x in iaddr =? addr) from_dmem)) ∧
(∀ addr,
   outV (dld addr) =
   List.length (List.filter (fun x ⇒ x =? addr)%N ldQ_sent)) ∧
∀ addr, outI (dld addr) = 0
```

Pipeline state Another straightforward part of the simulation relation (but that is more of a mouthful to state) is the invariant stating that the list of instructions in-flight in the implementation (and the associated register values read for those instructions) are the same as the ones that are waiting in the input queue of the specification. Here is the corresponding, lightly edited, proposition:

```
Forall2
   (* We zip together the elements of [e2w ++ d2e] and the elements of [from_imem]
      (they are on the last line of this snippet). [from_imem] is the content of the
      queue of waiting instructions in the specification machine, while [e2w] and [d2e]
      are the two intermediate queues for decoded and executed instructions in
      the implementation machine.
      The Forall2 statement declares a relation for the pairs of instructions: *)
    (fun impl_ins spec_ins ⇒
       match impl_ins with
       | Decoded val1 val2 coming_pc decoded ⇒
         (* If the implementation instruction is an instruction that has been
         decoded but not executed, then the pc of this instruction is the same
         as the one in the specification queue, the values (the decoded
         instructions) are the same too, and the register values that were read
         at decode time and pushed into the [d2e] queue agree with the
         specification's register file.  We also state that if the instruction
         has a destination, then the scoreboard has an outstanding dependency. *)
         dlet {coming_pc' data} := spec_ins in
         coming_pc' = coming_pc  ∧
         (? decode data) = decoded  ∧
         val1 = rf (? src1 decoded)  ∧
         val2 = rf (? src2 decoded)  ∧
         ((? has_destination decoded) mod 2 = 1 →
            fst (irf (? destination decoded)) = false)
       | Executed valid coming_pc ppc decoded dest_val ⇒
        (* Same idea for this case, with an extra twist because Executed instructions
          might be poisoned instructions. *)
         [ ... ]
       end)
      (e2w ++ d2e) from_imem
```

Internal consistency Our simulation also contains what we call *reachability invariants*. Those properties are not directly relating the state of the implementation and the state of the specification machines. Instead, they are constraining the states of the implementation that are reachable (so the specification state is not mentioned in those properties).

The main invariant in this category simply states that the scoreboard should be doing its job of preventing RAW and WAW hazards, hence constraining the possible sequences of instructions in-flight.

We greatly benefit from being able to phrase our invariant in the very expressive logic of Coq. The invariant is commented inline.

```
∀ l e1 m e2 r,
let decoded_instructions_in_flight :=
  (* we first collect all the instructions in-flight in the implementation,
     poisoned or not *)
  (map (fun x ⇒
           dlet {valid coming_pc ppc decoded dest_val}:= x in
           decoded)
           e2w ++
     (map (fun x ⇒
           dlet {val1 val2 coming_pc decoded}:= x in
           decoded)
           d2e)) in
  (* For every ordered pair of instructions [e1] and [e2] in this
  list of in-flight instructions *)
  decoded_instructions_in_flight = l ++ [e1] ++ m ++ [e2] ++ r →
(* Then if the older instruction has a destination,
   this destination cannot be the source or the destination
   of the younger instruction *)
(? has_destination e1) mod 2 = 1 →
 (? destination e1) ≠ (? src1 e2) ∧
 (? destination e1) ≠ (? src2 e2) ∧
```

```
 ((? has_destination e2) mod 2 = 1 →
   (? destination e1) ≠ (? destination e2))).
```

We also have a similar proposition that states that `e2w` has the same loads as the one in the load queue (`ldQ_received` and `ldQ_sent`).

Finally, we need a proposition to guarantee that the sequence of PCs of the nonpoisoned instructions in-flight in the implementation form *a chain* anchored at the PC register of the implementation. That is, every nonpoisoned instruction's `nextPC` should be the `PC` of the next instruction, with the last nonpoisoned instruction's `nextPC` being equal to the value in the PC register. This kind of chaining structure is typical in processors. We found that stating this proposition was easiest using an inductive proposition (similar to the one we used for flushing the data memory in the proof of validity of our generalized specification).

Once all these pieces are put together (with a few more administrative propositions), we have our complete simulation relation. We then use the following simulation strategy (outlined earlier):

- For every nonmemory and non-mmio instruction, the commit time is `writeback`, so every transition in the implementation is simulated by no transition at all on the specification side, except for the `writeback` transition, where we make the specification catch up with the implementation by performing `decode`, `execute` and `writeback`.

- For memory load, the situation is almost similar to the previous case, except that we need to *simulate the emission of the load request* on the specification side. Indeed, the load request is an observable event, so the specification is *required* to emit a load request to the same address. Hence, the simulation of the different transitions stays the same, except for the `execute` transition which we mimic in the specification by a random load to the same address, using the `emit_dndl` transition.

- For stores and MMIO instructions (which stall at execute time, as we described when we introduced the design), the situation is different. The implementation runs executes and writes-back sequentially. Hence, in that case, the specification simulates the implementation simply by staying in-sync: `execute` of the implementation is simulated

by the sequence of `decode` and then `execute` of the specification, and `writeback` of the implementation is simulated by `writeback` of the specification.

Verifying that this strategy works with the simulation relation we outlined is systematic. □

6.4 Discussion

Kami [19] also explored processor proofs to evaluate the effectiveness of their verification framework. In this section, we elaborate on the differences between the two approaches. There are both differences in the designs being proven and differences in the way the proofs are done.

Differences in the processors designed There are a few differences between the Kami processor and the family of processors we proved. The main important difference is that the Kami processor introduced less concurrency when it came to its use of memory. It was limited to emitting at most one load or store. This is a significant difference, as having several outstanding loads is one of the design optimizations that forced us to introduce our notion of generalized specifications. In a sense, having more than one outstanding memory operation introduces a form of concurrency that makes modular verification more challenging.

There is another interesting technical difference. The Kami processor was written with very few conditional constructs. Instead of following the standard way of designing processors in Bluespec, where each rule covers multiple cases using conditionals, they favored writing disjoint rules, handling each kind of instruction in a different rule. This way, each rule is mostly branch-free. This is a valuable choice from the perspective of verification because it allows them to side-step known difficulties in the symbolic evaluation and case analysis of programs containing branches. However, from the design standpoint, it is not idiomatic. Instead, we used conditional as liberally as we would have if we had designed our processor directly in Bluespec.

Finally, the Kami processor strongly limited the maximum number of instructions in-flight in the pipeline, while our design can be instantiated with very large queues (and so have a very large number of instructions in-flight). Concretely, the bookkeeping queues in

the Kami design were of size 1, so the amount of concurrency, reordering, and dependencies within the pipeline could be fully expanded and explored manually. This difference leads us to detail the differences in our verification approaches.

Differences in proof strategies Because of the choice to keep the processor design fairly concrete with few outstanding instructions, the authors of Kami [19] could manually write a low-level simulation exhaustively relating their implementation and their specification. As the machine does not have many cases, this simulation (reminiscent of the example of we gave in subsection 5.2.1) is reasonably sized (a few hundred lines). However, a change in queue size would require rewriting the simulation relation completely, leading to a potentially exponentially longer relation, which could become a problem when the state space (number of instructions in-flight) of the machine grows. Such an approach would likely not have worked for our design, as we need higher-order invariants to cover arbitrarily many cases.

Finally, note that in Kami [19], the authors explored cache-coherency protocols (and so multicore machines) while we did not do so. We favored to study less sophisticated memory systems but more sophisticated processors.

Chapter 7

Coquetier: a simplification tactic for our Coq toolbox

The technique and the tool presented in this section are a collaboration with Samuel Gruetter.

One of the challenges in using Coq for verifying systems resides in the lack of robust automation to discharge (or simplify) reasonably easy goals.

There are many cases where a proof only requires a mix of simple symbolic evaluation and unification to be solved. In the few cases where some arithmetic or custom logic reasoning is required, the reasoning is also usually very elementary. Concretely, most proofs we completed require only middle-school (maybe high-school) mathematical knowledge. Sometimes, proving a refinement can feel like we are doing 10 not-so-interesting and repetitive math exercises. After the first or second case, it can be boring to the proof engineer.

In this chapter, we present our exploration to *simplify* goals in Coq using a proof-search technique based on E-graphs. E-graph-based techniques are commonly used in modern SMT solvers. There is some exploration of similar techniques in Lean [53], but there has been little published exploration in Coq. From a theoretical standpoint, we are mostly adapting and revisiting techniques pioneered by Greg Nelson in his thesis [47].

In this work, our goal was to explore this proof-search technique to simplify some of our easy proof goals automatically.

7.1 Simple examples

Let's consider the following very elementary goal.

```
Goal ∀ x y, x = y →
        [4;2] = y →
        ∃ t q, x = t::q.
Proof.
    intros.
    eexists; eexists.
```

```
x, y: list nat     H: x = y     H0: [4; 2] = y
------------------------------------------------
x = ?t :: ?q
```

We can try standard Coq tactics to solve this easy goal, but they fail:

```
    Fail congruence.
    Fail easy.
```

Note that if we are ready to instantiate the existential variables by-hand, then the standard `congruence` tactic can prove the desired equality.

```
    instantiate (1:= [2]).
    instantiate (1:= 4).
    congruence.
Qed.
```

This minimal example illustrate a common pattern. Standard Coq tactics are usually not designed to work in the presence of existential variables. This is a problem for us, as we are very often introducing and manipulating existential variables in our specifications. Note that if we embed this goal in any SMT solver, we will find that the solver can easily discharge it.

The goal of this chapter will be to introduce a proof-search strategy that can solve automatically this kind of goals without having to instantiate the existential variables.

Before diving into this technique, let us give two more examples. We start with an example where the theory necessary to prove the goal is slightly more sophisticated, and then a second example to motivate another central aspect of our approach.

```
Goal ∀ m, ∃ l,
  length l = (3 + (length m)) ∧ 1 :: l ++ [4] = [1;2;3;7] ++ m ++ [4].
Proof.
```

Manually inspecting this goal reveals that `l = [2;3;7] ++ 7` works.

However, this goal is still a bit more sophisticated than the previous one, as it involves the structure of lists: it uses the operators `++`, `::` and `length` and the equality axioms on those operators (which we name the theory of lists).

In other words, the goal basically requires unification modulo the theory of lists. We have defined a tactic to import the equational theory of lists in the context. We display a few of those equations:

```
  list_theory.
```

```
H: ∀ (A : Type) (l : list A), l ++ [] = l
H2: ∀ (A : Type) (l m n : list A),
   (l ++ m) ++ n = l ++ m ++ n
H9: ∀ (A : Type) (l : list A) (x : A),
   length (x :: l) = S (length l)
____________________________________
length ?l = 3 + length m ∧
1 :: ?l ++ [4] = [1; 2; 3; 7] ++ m ++ [4]
```

```
  (* Still the standard tactics fail to solve the goal. *)
  Fail easy.
  Fail congruence.
  (* As a side note, the [firstorder] tactic takes 36s to fail solving
    the goal as well.
    If we are ready to give the witness for [l],
    then the situation is slightly different: *)
  instantiate (1:= [2;3;7] ++ m).
  Fail congruence.
```

```
  (* Though firstorder succeeds instantaneously. *)
  firstorder.
Qed.
```

Our last example showcases a typical use case we have: sometimes we do not want to actually solve the goal in one step, but we would like to simplify the goal, using the contextual hypotheses.

This last example uses the `lset` function. `lset` is simply a function to set the nth element of a list.

```
Goal ∀ (l1 l2 : list nat) (x y : nat) (j : nat),
  j = (length l1 + length l1) →
  lset (l1 ++ (l1 ++ [x] ++ l2) ++ [1;2]) j y = l1 ++ l1 ++ [x] ++ l2 ++ [1;2].
Proof.
```

This new example states an equality for an expression that uses `lset`. To hope to prove this goal, we will need the reasoning principles to eliminate `lset`. This equation was not part of the theory of lists we had previously defined but can easily extend the theory. Here we display the key new lemma we added as `H14`.

```
  list_theory; list_getset.
```

```
H14: ∀ (A : Type) (l r : list A) (e ne : A),
   lset (l ++ [e] ++ r) (length l) ne = l ++ [ne] ++ r
______________________________________________________
lset (l1 ++ (l1 ++ [x] ++ l2) ++ [1; 2]) j y =
l1 ++ l1 ++ [x] ++ l2 ++ [1; 2]
```

We can try running `firstorder congruence`. This compound tactic should have a good chance of solving this simple goal, as there are no existential variables and the goal falls within a logic fragment that the tactic should be able to solve.

```
  Fail timeout 10 firstorder congruence.
```

However, the tactic starts running without succeeding (we tried running it for 250s before killing it).

There is a very good reason for the tactic not succeeding: this goal is false.

Indeed, the `lset` expression on the left is actually equal to `l1 ++ l1 ++ [y] ++ l2 ++ [1; 2]` instead of `l1 ++ l1 ++ [x] ++ l2 ++ [1; 2]`.

So the goal is only true if we assume an extra hypothesis: `x = y`.

```
Abort.
```

```
Goal ∀ (l1 l2 : list nat) (x y : nat) (j : nat),
  j = (length l1 + length l1) →
  (* The extra hypothesis: *)
  x = y →
  lset (l1 ++ (l1 ++ [x] ++ l2) ++ [1;2]) j y = l1 ++ l1 ++ [x] ++ l2 ++ [1;2].
Proof.
  list_theory; list_getset.
  firstorder congruence.
  (* Indeed, with this extra hypothesis, the standard compound tactic
  [firstorder congruence] succeeds.  *)
Qed.
```

Why did we just give this last example? We showcased a standard tactic failing in proving an incorrect goal: it looks more like a feature than a bug.

The issue is not that the tactic failed to prove the goal. The problem is that the user does not know the source of this failure. Maybe the tactic has a performance problem, or maybe the tactic is being tripped up by some expressions in the context that are not quite expected by the tactic, or maybe the goal we are trying to prove is simply false.

When the goal is complicated, manual inspection of the goal to identify the source of the problem can be difficult and time-consuming.

Verification tactics tend to assume that the common case is that one is always verifying a

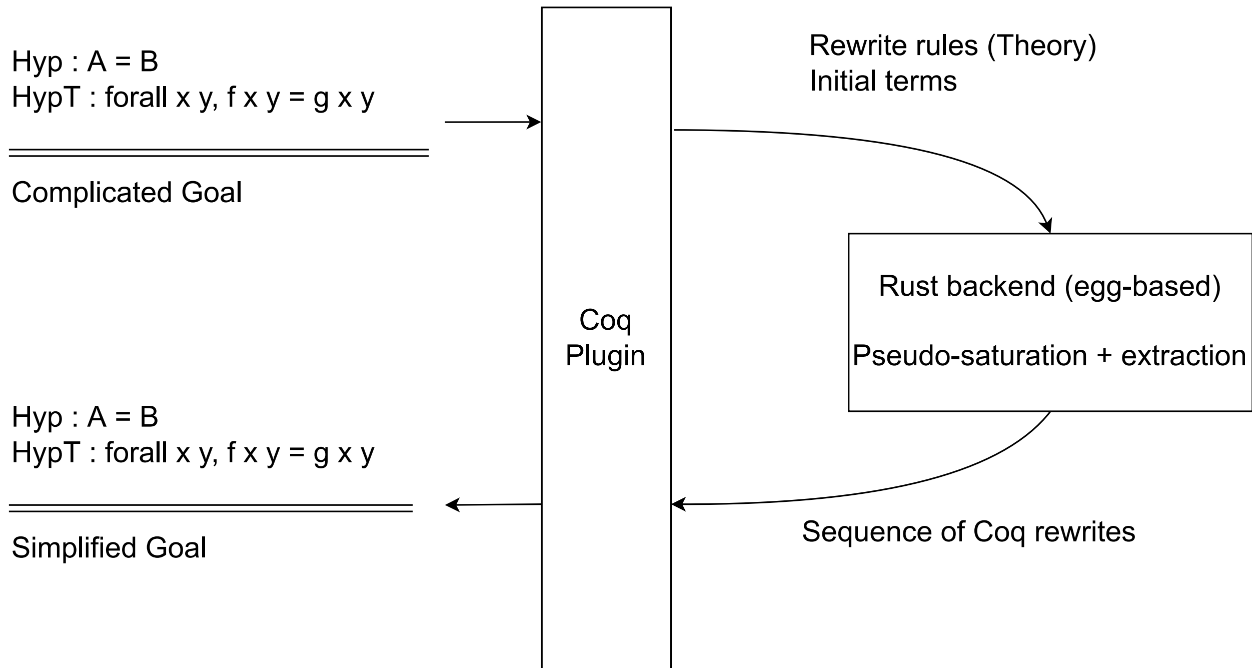


Figure 7-1: Software architecture of Coquetier, when it is used in goal-simplification mode.

correct design and is not making mistakes when going through the proof. We found ourselves commonly violating this assumption.

Instead of producing tactics aimed at solving the goal, we aim for a tactic that *simplifies* the goal to the best of its ability.

For example, in the case of the false goal, we would like the tactic to rewrite the goal to:

```
l1 ++ l1 ++ [y] ++ l2 ++ [1; 2] = l1 ++ l1 ++ [x] ++ l2 ++ [1; 2]
```

This expression is simpler than the original goal, and figuring out the missing hypothesis from this goal is significantly easier.

7.2 Overview of Coquetier

Coquetier is a Coq plugin that defines a few tactics that are useful to simplify (and solve) the kind of goals we showed in section 7.1. At a high level, Coquetier searches for the simplest term in the equivalence class of the goal modulo a user-defined constrained theory.

In Figure 7-1 we show the software architectural diagram of Coquetier. We try to mini-

mize the amount of work we do ourselves by reusing existing software. We use a rewriting, saturation, and pattern-matching engine in the form of a fast and increasingly popular Rust library: egg [61, 64]. This chapter will not go into the internals of the egg rewrite engine, which we have only seldom modified. Instead, the challenges we want to explain in the rest of this chapter (and our approach to solving them) are: (1) how to embed Coq terms and theorems in this kind of rewriting engine, (2) how to reinterpret back the output in Coq.

Note that the output generated by Coquetier is not trusted: it gives Coq a list of rewrites to apply, but Coq checks that those rewrites are legit and that they indeed transform the goal in the way Coquetier expected it to be change.

We finish this overview with a concrete example of how one can use Coquetier's tactics to solve the simple goals presented in of section 7.1.

Let us reconsider our initial very elementary goal but solve it without providing the witness manually, querying Coquetier instead.

```
Goal ∀ x y, x = y →
        [4;2] = y →
        ∃ t q, x = t::q.
Proof.
    intros.
    eexists; eexists.
```

```
x, y: list nat     H: x = y     H0: [4; 2] = y
--------------------------------------------------
x = ?t :: ?q
```

```
    coquetier_exists 4.
Qed.
```

Note that the tactic `coquetier_exists` takes an integer parameter. We will see later that it is a bound on the depth of the engine's search. Similarly, we can handle the other goals:

```
Goal ∀ m, ∃ l,
    length l = (3 + (length m)) ∧ 1 :: l ++ [4] = [1;2;3;7] ++ m ++ [4].
Proof.
```

```
  list_theory.
  coquetier_exists 4.
Qed.
```

Similarly, it works for our example that requires simplification only. The tactic gets stuck where we want it to get stuck. We do not display the hypothesis in the final goal.

```
Goal ∀ (l1 l2 : list nat) (x y : nat) (j : nat),
  j = (length l1 + length l1) →
  lset (l1 ++ (l1 ++ [x] ++ l2) ++ [1;2]) j y = l1 ++ l1 ++ [x] ++ l2 ++ [1;2].
Proof.
  list_theory; list_getset.
  coquetier_simpl 4.
```

```
(lset (l1 ++ (l1 ++ [x] ++ l2) ++ [1; 2]) j y =
 l1 ++ l1 ++ [x] ++ l2 ++ [1; 2])
```

```
________________________________________
(l1 ++ l1) ++ y :: l2 ++ [1; 2] =
l1 ++ (l1 ++ x :: l2) ++ [1; 2]
```

```
Abort.
```

7.3 Embedding Coq in egg

Before presenting our embedding of Coq in egg, let us give a quick and informal overview of the egg fundamentals we will need for Coquetier. We neither try to be exhaustive nor try to explain the internals of egg: actually, we even avoid talking about e-graphs, the underlying data structure used by egg. The curious reader should refer to [61, 64] and egg sources.

7.3.1 Egg fundamentals

First, we introduce two central objects in egg:

- *terms*, which are represented as s-expression. For example, `(plus 1 (times x 4))`.

- *patterns*, which are represented as s-expression with named holes. Named holes are prefixed with `?` and are to be distinguished from constants like `x` in the following example: `(plus ?h1 (plus ?h2 x))`. Note that every term is also a pattern: it is a pattern without holes.

At a high level, egg is a library to represent and manipulate efficiently a *set of sets of terms*. The inner sets are named *classes*. This name is especially justified as a class in egg represents a set of equal term. In other words, classes are *equivalence classes*.

So, egg represents a set of classes, and the operations supported on this set of classes are:

- `insert`, creating a new singleton class (a set of a single term), returning a handle to the class.
- `merge`, merging two classes forever.
- `query`, matching a pattern against the set of classes, returning all the classes that contain terms matching the pattern.
- `minimal_representative`, returning the smallest term in a class (for some user-defined metric).

All those operations are stateful, as they update the underlying object.

For example, we can imagine the following scenario in pseudocode:

```
e = empty
c0 = e.insert("(plus 1 (times x 4)") // c0 = {"(plus 1 (times x 4))"}
c1 = e.insert("(plus y 2)") // c1 = {"(plus y 2)"}
e.merge(c0, c1)
```

The last command merged the two classes we had just declared. After this command, we have `c0=c1={(plus y 2), (plus 1 (times x 4))}`.

Note that there are other implicit classes automatically created: every subterms of the terms we explicitly added are also represented, and they live in their own classes. For example, we can use the `query` method to find the class of the subterm `(times x 4)`. We can also use the same method to do more sophisticated queries (queries that have holes):

```
result = e.query("(times x 4)")
// result is a list of all the classes that satisfy this pattern
// as this pattern has no holes, there is a single way to match this pattern
// Hence, result is a list of a single class:  res =  [ `class of (times x 4)` ]

result = e.query("(plus ?a ?b)")
// result is all the possible matches of that pattern and how to fill
// the holes to obtain them.
// result = [(c0, ?a = `class of 1`, ?b = `class of (times x 4)`;
//           (c1, ?a = `class of y`, ?b = `class of 2`)]
// Note that one can also make nonlinear queries:
e.query("(plus ?a ?a)") // returns an empty list of matches
```

It is important to realize that for egg, everything is *syntaxic*. Even though our s-expressions seem to refer to arithmetic operations, the system does not know anything about arithmetic.

While it is always guaranteed that there will be a finite number of classes, the classes themselves can actually easily represent infinite sets of term. Consider the following example:

```
newc = empty.insert("(f 0 x)")
[ cx ] = e.query("x") // we get the class of x
e.merge(newc, cx) // We declare that (f 0 x) = x, which is a loop
```

After those commands, the class `newc` of `e` contains infinitely many terms. It contains `(f 0 x)`, `(f 0 (f 0 x))`, `(f 0 (f 0 .. x) ..)`, etc.

Now, observe that we can think of a simple rewrite rule simply as a pair of patterns. Typically a commutativity rewrite rule ($\forall\, x\ y.\ x{+}y = y{+}x$) can be framed as: `(add ?x ?y) => (add ?y ?x)`. Hence, in this setup it is easy to have a naive algorithm computing the closure modulo a set of rewrite rules (which potentially will fail to terminate): One simply iteratively picks a rewrite rule, queries the pattern on the left-hand side of the rule, and for all the matches, inserts the right-hand side of the rule and declares the two classes equal. If we reach saturation, we indeed computed the closure. However, even partial saturation is useful.

Using more than two patterns allows us to formalize a more general notion of rewrite

rules. We can define rewrite rules that also have preconditions:

$$\forall\ ?h_1 \ldots ?h_n.\ \mathit{hyp1}_{\mathit{lhs}} = \mathit{hyp1}_{\mathit{rhs}} \rightarrow \ldots \mathit{hypk}_{\mathit{lhs}} = \mathit{hypk}_{\mathit{rhs}} \rightarrow \mathit{clc}_{\mathit{lhs}} = \mathit{clc}_{\mathit{rhs}}$$

Let us look at a simple strategy for a rewrite rule having one precondition:

$$\forall\ ?h_1 \ldots ?h_n.\ \mathit{hyp}_{\mathit{lhs}} = \mathit{hyp}_{\mathit{rhs}} \rightarrow \mathit{clc}_{\mathit{lhs}} = \mathit{clc}_{\mathit{rhs}}$$

1. Query for the two patterns, $\mathit{hyp}_{\mathit{lhs}}$ and $\mathit{hyp}_{\mathit{rhs}}$, and keep only the matching results that produce the same classes (as the precondition states that the terms should be equal). Those are all the instantiations of the holes that verify the precondition.

2. Restrict again this list of matches, joining it with the result for pattern $\mathit{clc}_{\mathit{lhs}}$ producing a final list of matching results to rewrite.

3. For each matching result, insert $\mathit{clc}_{\mathit{rhs}}$ and declare the classes of the left-hand side and the right-hand side to be equal.

Egg-splanations We saw that all the terms in a class are equal modulo rewrite rules. A very useful feature of egg is that it can also output *why* two terms are in the same class. Every time a merge is triggered by a rewrite rule, a marker records which rule was used to perform that merge.

Then, using those markers, for any pair of terms in the same class, egg can produce a sequence of rewrite rules (with the instantiations for the holes) that explains how to transform one term into the other one.

This explanation (that we name an egg-splanation) is exactly what we replay on Coq's side.

7.3.2 Initial embedding

We now have the ingredients to propose a strawman embedding. We will explain the limitations of this initial embedding and fix it in subsection 7.3.4.

First, we don't aim at embedding the full language. We restrict ourselves to a very small fragment of Coq. If a hypothesis uses a Coq feature that we can't translate, we will simply ignore that hypothesis in the translation. If the goal uses a feature that is not supported, we won't be able to use our tactic at all.

The fragment we consider ensures:

- No anonymous functions.
- No `match` constructs. In particular, no `if` construct.
- No dependent types.
- All constants (and variables, for theorems) must be used with constant arity. Typically, if there exists a function f, every mention of f must be applied with the same number of arguments.

In particular, those restrictions guarantee that no new binders are introduced within terms. Any Coq term within that fragment can be represented by an s-expression of the following form: `term = (f term*)`, where `f` is a function of n arguments available in the Coq context (constants are functions of 0 arguments).

Let us give a quick example of translation. A term `Odd (f n) ∧ (g 0 = y)` gets translated to `(and (Odd (f (n))) (eq (nat) (g (0)) y))`. When a function has 0 arguments (it is a leaf), we will omit the last set of parenthesis: `(and (Odd (f n)) (eq nat (g 0) y))`. Observe the subterm `nat`, argument to the function `eq`, which might look unexpected. This is because Coq's `eq` construction is defined polymorphically, hence, when it is used it is being passed the type for which it is being used as an argument.

We have seen how to represent arbitrary terms and propositions in egg, and our representation is such that if we find two terms in the same class, we can generate a sequence of rewrite rules (which can be interpreted in Coq) to transform the first term in the second term. Intuitively, we have enough to prove equality between terms, potentially using theorems that state and manipulate other equalities between terms. But what if we have theorems that manipulate other predicates, what if we have more than just equalities? The next paragraph gives a solution to easily extend our little encoding and make egg able to

generate proofs of propositions (and have theorems that use propositions as preconditions) instead of just equalities.

Proposition extensionality The key insight is to use the so-called propositional extensionality axiom in Coq, which states that `forall (P : Prop), P <-> P = True`.

This common axiom guarantees that there is an equivalence between being able to prove a proposition and being able to rewrite the said proposition to the constant `True`.

This is convenient because if we look in the class of `True`, we can find all the propositions that egg proved.

Adding a few basic rewriting theorems about Boolean propositions allows us to get egg to do basic propositional reasoning for us. For example, we can add the following theory:

- `forall X, True ∧ X = X`
- `forall X, X ∧ True = X`
- `forall X Y Z, X ∧ (Y ∧ Z) = (X ∧ Y) ∧ Z`

Now, if we try to prove a goal `A ∧ (B ∧ C)`, we simply take the full Coq context (with all the universally quantified theorems of our theory and the initial facts we know), give it to egg using the encoding we outlined, saturate the rewrite rules corresponding to the theory and then ask egg to explore the class of `A ∧ (B ∧ C)`. If `True` is in that class, we have a proof. If `True` is not in the class, we can still simplify the goal as much as possible, simply by asking for a minimal term in the class of `A ∧ (B ∧ C)`. Hopefully, egg will return a smaller goal, conceptually simplifying all the parts of the goal that it did manage to prove.

Multiple coexisting notions of equality Let us finish this initial embedding by pointing out a nonobvious complexity. We have several coexisting notions of equality. The first is the Coq-level notion of equality: having a term `(eq T x y)` in our context Coq. The second is having a term `(eq T x y)` in the same class as the term `True` in egg: we say there is an explicit materialization of the equality in egg. The third is having two terms in the same class in egg: we say the equality is implicit in egg.

We are especially concerned with the last two (implicit equality and explicitly materialized equality), as they both live in the egg representation. At a high level, those notions are similar: they are simply different representations of the same facts. To avoid confusion, we can easily ensure that every explicitly materialized equality is also implicit by using the following lemma: `forall x y, (x=y) = True -> x = y`. In egg, this theorem becomes a rewrite rule that searches for any `(eq T ?x ?y)` term in the class of `True` and merges all resulting classes `?x` and `?y`. It is exactly turning an explicit equality term into an implicit egg equality. Observe that this theorem does not add new terms, it only merges existing classes. It is reasonably safe from a performance standpoint.

Let us discuss the other way around: materializing explicitly the implicit equalities known by egg. We could theoretically use a lemma like `forall x, True = (x=x)`, which would add an equality term in the class of `True` for all terms. This strategy leads to an explicit materialization of all implicit equalities (modulo some small restrictions) for nonobvious reasons. Indeed, one might expect to need to add an equality for every pair of equal terms: it is a consequence of the internal way egg uses e-graphs to represent classes of terms.

In any case, this is a heavy-handed approach which adds potentially many extra terms (typically, one new term per existing class), and most of those are not needed. We prefer materializing equality more lazily by crafting custom triggers. We will elaborate on related issues in our limitation section (section 7.4).

7.3.3 Metrics to find simpler terms

In the previous subsection, we discussed returning a minimal term in the class of the proposition `A ∧ (B ∧ C)` to produce a smaller equivalent goal.

This raises the question of which cost function do we pick to evaluate the cost of different terms in the class: what does small mean? There are several default cost functions provided in egg, typically the size or the depth of the term. It is also possible to define custom cost functions, which should respect some constraints. In a nutshell, the cost function should recurse on the structure of the term, and the cost of a node should be an increasing function of the cost of its children. Our default cost function is a size cost that we slightly skew by giving the constant term `True` a cost lower than any other constant. However, sometimes

this cost function is not good enough: a conceptually simpler intermediate goal has a higher number of nodes. Typically, this is caused by a function call encompassing a lot of conceptual complexity with a very low syntax-tree footprint.

To help to solve this issue, Coquetier gives the user the possibility to mark a special identifier as having a very high cost. Using this user-skewed cost function, one can for example search a class for a term that does not use a given function.

7.3.4 Representing types

The embedding we introduced in subsection 7.3.2 does not represent typed terms, only untyped terms. One might notice that there were types present in some of the terms we showed, typically `(Some T x)`, or `(eq T x y)`. This observation is not incompatible with the fact that we represent untyped terms: those types are only arguments used to specialize a polymorphic Coq definition. However, with our current encoding, egg is unable to know what is the type of an arbitrary term `(f x y)` within a class. There is an informal metatheorem that guarantees that all the terms in a class have the same type, but this type is unknown to egg.

This limitation of our encoding prevents us from using certain rewrites. Typically, the lemma we mentioned to explicitly materialize equality is impossible to use with our previous encoding (`forall {T : Type} (x : T), True = @eq T x x`.) Indeed, `T` does not appear in the left-hand side of the search pattern and so is invisible to egg.

To fix this limitation, we decided to represent typed terms instead of untyped terms. That is, instead of representing a `(f x y)`, we represent a judgment `(with_type T (f x y))`. The function `with_type` can be defined in Coq and is simply the identity function (returning the value of its argument). During translation to egg, we edit the term we try to translate, to add layers of `with_type`. When using this representation of typed terms, it is interesting to observe that types are treated similarly to any other term by egg: they are represented together in the big database of classes of equal terms. This is both a feature and a bug. It is a feature as it simplifies implementation: typed terms are handled almost for free, and we do not need to write a typechecker for our fragment of Coq in egg. However, it also introduces new performance issues that we will discuss in section 7.4.

7.3.5 Proving absurd cases

The cases we simplified so far implicitly assumed the hypotheses were not inconsistent. In practice, it is common that a goal is true because there is a contradiction within the hypotheses.

With what we presented so far, Coquetier would not help much in those cases. To handle those cases, we add a *consistency check* to Coquetier. After pseudosaturating the set of known facts modulo theories, and before trying to minimize the goal, Coquetier tries to find an inconsistency within the learned facts. If it finds an inconsistency (for example, `0=1`), then it directly tells Coq how to derive the inconsistency, which is then used to solve the goal completely .

The consistency check is based on the following simple criterion: we search for two equal (nonpropositional) terms that don't have the same head constructor. This criterion covers the typical absurd cases: `[] = a :: b`, `0 = 1`, `Some x = None`, any equality between two distinct integer constants, etc.

Computing this criterion requires us to identify constructors: when we generate the egg representation, we mark the constructors as being distinguished functions. Then the consistency check simply walks through all the classes, searching for two values in the same class that would start by distinct constructors. Thanks to the data structure used to represent classes in egg, checking this criterion is a cheap computation.

7.3.6 Solving existentials

In section 7.1, most examples we gave involved existential variables, as it was the original motivation for this work. However, we have not yet described how to find witnesses for existential variables (evars) using Coquetier.

Our strategy is fairly similar to the strategy we use for goal simplification:

1. We discard any hypothesis that mentions an evar.

2. We translate the context to egg (without also sending the goal to egg). The goal cannot be translated to an egg term as it contains evars.

3. We ask egg to apply the rewrites of the theory repeatedly to pseudosaturate the system.

4. We transform the Coq goal (which contains evars) into a pattern. Evars become egg variables.

5. We search for this pattern. We might find multiple matches, we display them, and we typically pick the smallest choice.

Note that a more sophisticated version could shelve the hypotheses discarded in step (1.) to integrate them in the pattern built in step (4.). Indeed, with our current strategy, we lose the possibility to generate a witness that would solve the goal by contradicting one of the discarded hypotheses on evars. Note also that because we only gather the constraints on the evars coming from the current goal, Coquetier can propose invalid witnesses when there are constraints on evars that are coming from other goals. In practice, most of our cases do not actually have any constraints on evars outside of the goal.

7.4 Limitations

In this section, we present the main limitations of Coquetier.

Computational blowup: The most fundamental limitation of Coquetier relates to the size of the set of terms that are congruent modulo theories. Even an elementary rewriting system - like the theory of addition with commutativity, associativity, and cancellation for 0 - cannot be saturated. In practice, we cannot reach saturation for most of our practical examples, and the systems are also typically too costly to pseudo-saturate for depth higher than 6.

The main reason why this limitation does not kill the approach entirely is that in numerous instances, constructing all the terms that are 3 to 5 rewrites away from our initial goal yields a new term, slightly smaller than our initial goal. We then restart a new simplification from this slightly simplified term. This way, step-by-step we can simplify the goal away, each step only searching locally around the current goal. Typically, performing

multiple local searches at depth 4 is computationally very cheap, while a single search at depth 7 is impractical.

Whenever any simplification of the goal requires a larger number of rewrites (typically when the goal needs to grow before it can be simplified), Coquetier's generic saturation strategy usually does not work very well. In those cases, we start having to worry about presenting the theory differently to add explicit triggers (markers to constrain the instantiations of universal variables) to guide the saturation. While one can encode a lot of control with triggers, for now, the triggers need to be figured out and added manually.

Types are not free: While our embedding of typed terms allows us to easily support important rewrites, our encoding creates performance issues. Typically, with our encoding egg will represent pairs of terms for every typed Coq term: `(with_type T term)` and the subterm `term`. The issue is that all the equalities, terms and Coq theorems are transparently modified by the plugin to operate on type-annotated egg terms. As such, all our Coq equations state equalities between egg terms of the form `(with_type T term)`. Hence, egg will never merge classes of untyped terms. This is not a correctness issue; egg is simply never aware that it could sometimes collapse classes of equal untyped terms. This leads to uncompressed internal representations, potentially slowing down egg's internal machinery. We explored teaching egg to merge classes of equal untyped terms by adding a rewrite that states the injectivity of `with_type`. This rewrite requires special care due to the unique status of `with_type`, but it can be added. However, it does not come for free. Most critically, it generates actual rewrites in egg that need to be performed. Moreover, those rewrites could potentially appear in the proof sequence generated by egg for Coq. In such a case, we could end up increasing the size of the proof term generated.

Chapter 8

Cycle-Accurate Semantics

This chapter is an adaptation of the Kôika paper [7], coauthored with Clément Pit-Claudel. So far, this dissertation exclusively focused on constructing a language setup in which hardware verification can be done modularly, proving the correctness of designs one method and one rule at a time, module by module. We put a special emphasis on introducing intermediate verification designs to make each of our verification tasks smaller and easier to tackle, also increasing potential reusability.

Note that in Fjfj, we never mentioned clock cycles or time. Even more interesting, thanks to well-chosen language restrictions, we side-stepped difficulties related to simultaneity and concurrency that are usually a significant challenge in hardware verification. Let's give an intuition of what those challenges typically are and then present Kôika, our effort to formally explore and study the mapping of the rule-based programming model to clock cycles. Kôika can be thought of as the nonmodular subset of Fjfj for which we would formally study scheduling and mapping to clock cycles. Kôika was introduced before Fjfj. The goal of this work was to gather evidence that using the rule-based design methodology – which traditionally purposefully ignored performance and time in its formal semantics – would not prevent us from reasoning about performance, precise time and liveness, once we would add a layer of scheduling.

8.1 Synchronous circuits and motivation for Kôika

A synchronous digital circuit is a state-transition system that specifies how the present state, held in registers, is transformed into the next state at every clock cycle. Popular hardware-description languages like Verilog expose this view fairly directly. However, for design purposes, it is not easy to think of the functionality of a complex digital circuit in terms of a global state-transition system.

Verilog does allow division of a design into separate concurrent blocks, each of which computes a subset of register updates every cycle. The natural concurrency of computing many state updates at once provides significant optimization opportunities, but, as in concurrent software, it introduces opportunities for bugs because of shared state. The well-developed alternative, that we used throughout this dissertation, is guarded atomic actions, as implemented in the hardware-description language Bluespec SystemVerilog (BSV) [49]. In BSV, the design specifies all the state elements, i.e., registers, and describes the behavior using a set of atomic rules. Each rule specifies a deterministic state transformation. It is guaranteed that *rules appear to execute atomically*, one-at-a-time, much like the established software concept of transactions. However, literal one-rule-at-a-time (ORAAT) execution in a hardware circuit would bring unacceptably poor performance. We still do need rules to execute concurrently, though in a controlled way that preserves the illusion of atomic execution. The BSV compiler does static analysis to construct a per-design *scheduler* circuit automatically, which chooses among the set of (enabled) rules in each clock cycle.

To appreciate the considerations that go into choosing a schedule, it is important to start from the quantitative metrics that matter for circuits. The most commonly cited are power, performance, and area. We think of circuits as directed graphs whose nodes are registers and gates, where no cycles are permitted on paths that only traverse gates. To a first approximation, area and power follow from register and gate counts, which we want to keep down. A major determinant of performance is the clock-cycle time, which is proportional to the *critical-path length*, i.e., the length of the longest path between any registers, even from the same register to itself. Path length here refers to the propagation delay of all the gates on a path between two registers.

We might be tempted to aim for a free lunch by removing gates to shorten the cycle time, but then we have postponed work to happen on later cycles, not necessarily shortening the total compute time. For example, consider a decomposition of function f into $f_2 \circ f_1$ (section 8.5) to allow pipelining. Let us assume that a circuit implementing f in one go requires a single cycle of length 10 seconds to execute, while circuits for f_1 and f_2 require 4 and 6 seconds, respectively. The unpipelined system processes one token per 10 seconds (the time to run f), while a system that repeatedly runs one of f_1 or f_2 has cycle time equal to the *maximum* of those cycle times, 6. Thus, in the two cycles it takes to run through the full pipeline, we take up $2 \cdot 6 = 12$ seconds, and our "optimization" actually made things worse. However, if both stages f_1 and f_2 execute concurrently each cycle, i.e. in a pipelined manner, we can process one token every 6 seconds.

The challenge in describing designs of this kind is that we often want rules to execute concurrently *even when they access some of the same state elements*. In the previous example, pipeline stages f_1 and f_2 would need to share some kind of queue, which f_1 enqueues into and f_2 dequeues out of simultaneously each cycle, f_1 enqueueing in cycle n the data consumed by f_2 in cycle $n + 1$.

The commercial BSV compiler relies on a static analysis to do ORAAT-preserving concurrent scheduling of rules. Its static analysis, combined with user-provided annotations (e.g. descending urgency and execution order), generally creates excellent circuits. This approach, however, is not satisfactory for two reasons. (1) The static analysis should be an abstraction of the dynamic semantics of a program. BSV's dynamic semantics applies only to one-rule-at-a-time executions, and the cycle-level semantics necessarily depends on the static analysis of rules. (2) BSV programmers often think deliberately about static-analysis details and even change their code to nudge the compiler in the right direction to achieve the desired degree of concurrency. We take a different approach in this chapter, providing a new core calculus Kôika maintaining the essence of BSV, preserving all its desirable properties and yet allowing direct control over the atomic actions executed each clock cycle, without relying on static analysis. Kôika programmers still need to think about the real rule conflicts but not about the compiler's abstraction thereof. Our calculus includes a *deterministic, cycle-accurate* operational semantics, enabling formal reasoning about performance, without

removing the ability to prove invariants by induction on sequential executions: the effect of the set of rules completed each cycle is proven to be always explainable in terms of ORAAT semantics.

Often the rules we want to run concurrently require controlled communication amongst themselves. BSV's ephemeral history registers (EHRs) provide a mechanism to enhance concurrency in rule scheduling. EHRs essentially enrich rule-based designs with what are known as *bypasses* in hardware design. EHR semantics guarantees that the observed behaviors can be reproduced with serial execution of rules. However, pure ORAAT semantics are unable to capture the performance implications of EHRs because in ORAAT semantics, EHRs are indistinguishable from ordinary registers. Kôika's semantics, on the other hand, capture both the functional and performance aspects of EHRs.

The commercial BSV compiler can be viewed as producing one *schedule* (concurrency strategy) automatically, and our calculus supports most of these schedules and others that allow more concurrency for performance[1].

This chapter presents the following:

1. Kôika, a core calculus for a BSV-like rule-based language to support formal reasoning about both functional and performance properties.

2. A key metatheorem that Kôika's operational semantics only produces executions that can be mimicked with one-rule-at-a-time execution.

3. A cycle-accurate operational semantics that does not depend upon any static analysis.

In the rest of this chapter, we start with an introduction to Kôika (section 8.2), next defining its formal semantics (section 8.3), which allows the execution of multiple rules in one cycle. We prove that the ORAAT property emerges from this semantics (section 8.4). Then we use Kôika's semantics to characterize the behavior of a pipeline (section 8.5).

[1]BSV and Kôika schedules can be difficult to compare because BSV allows multiple rules to write into the same register but picks which write prevails.

8.2 Informal introduction to scheduling in Kôika

8.2.1 Rules

Like in Fjfj, Kôika programs are composed of *rules* (roughly: atomic units of execution). Kôika rules only manipulate values stored in *registers*. Taken together, these rules define what happens in each *clock cycle*.

Very similarly to Fjfj, the following rule increments the value in register `r`:

```
rule increment =
  let v = r.rd in r.wr(inc(v))
```

First, this rule reads the value stored in register `r` into a variable `v`, then applies the combinational (i.e., pure) function `inc` to `v`, and finally writes the result in register `r`. All of these actions are performed as one atomic unit.

We introduce the example that we will use to present the notion of *scheduling*.

Collatz function: The example below computes terms of the Collatz sequence, defined by the two equations $u_{n+1} = u_n/2$ if u_n is a multiple of two, and $u_{n+1} = 3 \cdot u_n + 1$ otherwise.

```
rule divide =
  let v = r.rd in
  if iseven(v) then
    r.wr(v >> 1)
```

```
rule multiply =
  let v = r.rd in
  if isodd(v) then
     r.wr(3 * v + 1)
```

Until now, both Fjfj and Kôika have defined the semantics of rules in isolation. This so-called one-rule-at-a-time (ORAAT) semantics of a collection of rules was to pick a rule nondeterministically, execute it, and commit its results. (In case of an abort, the state does not change.) The process is repeated endlessly, as if exactly one rule executed in each clock cycle: if one rule writes to a register, the next rule observes the newly written value.

8.2.2 Scheduling

ORAAT is a conceptual model. In designing efficient hardware, however, we strive to execute as many compatible rules as possible in parallel in each clock cycle, without violating

the illusion of running rules one-at-a-time. In order to introduce concurrency, we define a *schedule*, which specifies the order in which we expect rule effects to become observable.

It is straightforward to see that rules operating on disjoint register sets can be run in parallel without affecting the final outcome. Regardless of scheduling order, their effects commute. In the following example, however, opportunities for parallel execution depend on scheduling choices:

```
rule write_r = r.wr(0b10)
rule read_r = s.wr(inc(r.rd))
```

Both of these rules access register `r`, and they may be sequenced in two ways: attempt to run `write_r` then `read_r`, or attempt to run `read_r` then `write_r`.

If we start with `write_r` then, according to ORAAT, `read_r` must observe the new value of `r`; hence, `read_r` cannot happen within the same cycle. If we start with `read_r`, on the other hand, it is safe to run both rules in the same cycle: the effect will be just the same as if we had executed `read_r`, waited until the next cycle, and executed `write_r`. In that case, we say that the two rules "fired" (ran) *concurrently*, or *simultaneously*.

For this program, a scheduler that runs `read_r` before `write_r` allows parallelism. A semantic characterization of the system restricted to ORAAT would not specify how these two rules should be sequenced, and it would therefore be insufficient because such distinctions are crucial in hardware design. Accordingly, unlike plain ORAAT semantics, our definition of a program includes a *scheduler specification*, which describes unambiguously the order in which rules will appear to have run in each cycle. With this specification, each program describes a *unique* sequential machine up to Boolean equivalence, and it becomes possible to reason cycle-accurately about performance.

Here is how we write the two schedules above:

```
schedule blocked = [write_r; read_r]
schedule parallel = [read_r; write_r]
```

The key point of making schedules explicit is to allow fine-grained control over concurrency, by enabling several rules to execute in one clock cycle as long as their effects are

compatible with the linear order specified by the scheduler. Our semantics ensure that concurrently executed rules produce results compatible with ORAAT semantics. A rule whose execution would cause a violation is aborted dynamically.

It is important to realize that a schedule specifies which rules execute within one cycle. It says nothing about inter-cycle scheduling; the same schedule is used every cycle. For example in the *blocked* schedule, `read_r` will never actually be performed because it will be preempted by `write_r` each cycle.

8.2.3 Ephemeral history registers (EHRs)

The language that we have outlined up to this point respects ORAAT but is overly restrictive. Indeed, without adding extra constructs in the language, there is no way to have data flowing between rules within a single cycle: rules are fully isolated from each other.

To relax this restriction while preserving ORAAT, we introduce two new operations on registers, `rd1` and `wr1`. (From now on we treat `rd` and `wr` as synonyms for `rd0` and `wr0`, respectively.) These new primitives allow programmers to control data forwarding between rules: data written by `wr0` in a register is readable by `rd1` on the same register, and data written by `wr1` becomes readable by `rd0` in the next cycle[2]. This mechanism coming from BSV is associated with the unwieldy name *ephemeral history register (EHR)* [52, 51].

```
rule inc_r =
  let v = r.rd0 in
  if v < 0b101 then
    r.wr0(inc(v))
```

```
rule check_r =
  let v = r.rd1 in
  if even(v) then
    s.wr0(v)
```

```
schedule fwd = [inc_r; check_r]
```

The scheduler in this program specifies that `inc_r` should execute first. Thus, if `check_r` attempted to read register `r` by `rd0`, it would abort; but it may read `r` using `rd1`. Doing so, it observes the value written by `inc_r`, if any, or the initial value of `r` otherwise. We say that the write to `r` was *forwarded* to `check_r`. Similarly, `check_r` would abort if it attempted a

[2] It is natural to consider generalizing this mechanism to a register with arbitrarily many "ports" (0, 1, 2, …), but it turns out that the two-port version is expressive enough to encode any number of ports, so we restrict our attention to this simpler case.

`wr0` into `r`, but a `wr1` would succeed and take precedence over any value previously written by `wr0`.

`rd0` and `wr0` can be performed in any order within a rule. However, if a `wr0` is performed by a rule then, to preserve ORAAT semantics, no later rule can perform a `rd0` to the same register in the same cycle. More generally, Kôika places dynamic restrictions on these new operations. No reads or writes can follow a `wr1` in a subsequent rule (as an example, a `rd0` following a `wr1` would observe a stale value if it ran the same cycle as a preceding `wr1`), and a `wr0` cannot follow a `rd1` in the same rule or across rules. We also add the restriction that `wr0` cannot follow `wr1` in the same rule. This restriction is not directly required but simplifies the semantics, and we cannot think of an interesting program that would benefit from relaxing the restriction. Roughly, we have the following restrictions across rules: `rd0` < `wr0`, `wr1`; `rd1` < `wr1`; `wr0` < `rd1` — but note that, in the absence of a `wr0`, `rd0` and `rd1` may be interleaved freely. We will formalize these restrictions in section 8.3. Finally, notice that the `rd`s are *per-register*: it is completely valid, and in fact often useful (particularly in building pipelines), to use `wr0` to store in a given register the result of a computation involving a `rd1` of another register. In other words, the numbers should *not* be read as timestamps describing a global order within the clock cycle.

Allowing for data forwarding between rules increases flexibility and enables additional concurrency, but it potentially lengthens the critical path of the generated circuit. Careful designers typically use forwarding sparingly, when it unlocks additional parallelism without increasing the critical path in a destructive way.

The Collatz example revisited: After revealing additional primitives beyond just `rd` and `wr`, we can revisit our Collatz example. It can be written as follows, using EHRs:

```
rule divide =
  let v = r.rd0 in
  if iseven(v) then
    r.wr0(v >> 1)
```

```
rule multiply =
  let v = r.rd1 in
  if isodd(v) then
    r.wr1(3 * v + 1)
```

```
schedule collatz = [divide; multiply]
```

Note that `multiply` performs a `rd1`, allowing both rules to run in the same cycle in certain

cases. More precisely, the circuit behaves in the following way:

- If the value in `r` is even but not a multiple of 4, both rules fire: the circuit writes $3 \cdot (\texttt{r}/2) + 1$ in `r`.

- If the value in `r` is a multiple of 4, only the first rule fires: the circuit writes $\texttt{r}/2$ in `r`.

- If the value in `r` is odd, only the second rule fires: the circuit writes $3 \cdot \texttt{r} + 1$ in `r`.

This example shows that the concurrent execution of rules can be enhanced substantially by using EHRs.

8.3 Formal description of Kôika

Combinational functions (pure mathematical functions that do not read or write registers) play no role in our semantics. Therefore, we avoid describing them by assuming a set of named combinational functions.

8.3.1 Syntax

A program is described by a set of rules and a scheduler:

$$\begin{aligned} \text{Program} \quad P \quad ::= \quad & [\texttt{rule} \;\; \textit{rule_name} = a]^* \\ & \texttt{schedule} \;\; \textit{schedule_name} = s \end{aligned}$$

$$\text{Schedule} \quad s \quad ::= \quad \texttt{done} \mid \texttt{cons} \; \textit{rule_name} \; s$$

As an abbreviation for `cons r1 (cons r2 ...)` we write `[r1, r2, ...]`, which represents the sequencing of rules.

Each *rule_name* in a schedule refers to a rule, which is an action that returns `tt` (the

unit value).

$$\begin{array}{rcll} \text{Actions} & a & ::= & b \mid x \mid \texttt{skip} \mid r.\texttt{rd}p \mid r.\texttt{wr}p(a) \\ & & & \mid \texttt{let}\ x = a\ \texttt{in}\ a \mid f(a, \ldots, a) \\ & & & \mid \texttt{if}\ a\ \texttt{then}\ a\ \texttt{else}\ a \mid \texttt{abort} \\ \text{Ports} & p & ::= & 0 \mid 1 \\ \text{Bitstrings} & b & ::= & \texttt{tt} \mid \texttt{0b(0|1)+} \\ \text{Registers} & r & & \text{Variables}\ x \qquad \text{Externals}\ f \end{array}$$

`skip` is the unit value for actions, standing for no action, which returns `tt` when executed. As a shorthand, we write a_1`;` a_2 for `let` $x = a_1$ `in` a_2 with unused x. Similarly, we write `if` b `then` a for `if` b `then` a `else skip`, as we have already used in the introduction to the language. We will consider only well-formed programs in this chapter. For example, a rule that attempts to write a 12-bit value into a one-bit register, or calls an external combinational function with arguments of inappropriate bit-widths or inappropriate number of arguments, or refers to a nonexistent register, etc., will not be considered. (Our implementation applies a very standard type system to rule out these failures.)

8.3.2 Semantics

A rule in our language is an action that returns `tt`. It is characterized by the log ℓ of reads and writes it performs. The semantics of executing a single rule in isolation can be thought of as defining a function from the register values (notated $\mathcal{R}$) to generate a log ℓ. This log ℓ is built inductively along with the local environment of binders Γ. The effect of executing this rule in isolation would be to use the generated log to update the registers: $\mathcal{R}_{\text{next_cycle}} = \mathsf{update}\,(\mathcal{R}, \ell)$, with:

$$\begin{cases} \mathsf{update}\,(\mathcal{R}, []) = \mathcal{R} \\ \mathsf{update}\,(\mathcal{R}, \ell +\!\!+ [(\mathsf{rd}*, r)]) = \mathsf{update}\,(\mathcal{R}, \ell) \\ \mathsf{update}\,(\mathcal{R}, \ell +\!\!+ [(\mathsf{wr}*, r, v)]) = \mathsf{update}\,(\mathcal{R}, \ell)[r \mapsto v] \end{cases}$$

Indeed there are at most two writes per register in the log, `wr0` and `wr1`, and we will come shortly to how it is guaranteed that `wr0` never precedes `wr1` for any register.

We want to give the semantics of executing multiple rules, as specified by the schedule, every clock cycle. Under such circumstances, a rule can see the side effects of rules scheduled earlier. Therefore, we accumulate the effects of all preceding rules in a global log L. Thus the semantics of a rule whose body is a are as follows:

$$[\![a]\!](\mathcal{R}, L) = \begin{cases} \mathsf{Log}\ \ell & \text{if } a \text{ succeeds and produces the log } \ell \\ \mathsf{Fail} & \text{if } a \text{ fails} \end{cases}$$

which we can use to define the effect of executing multiple rules according to a scheduler.

$$\frac{}{(L, \mathsf{done}) \Downarrow L}\ \textsc{Done}$$

$$\frac{[\![a]\!](\mathcal{R}, L) = \mathsf{Log}\ \ell \qquad (L +\!\!+ \ell, s_{\text{next}}) \Downarrow L'}{(L, \mathsf{cons}\ a\ s_{\text{next}}) \Downarrow L'}\ \textsc{SeqLog}$$

$$\frac{[\![a]\!](\mathcal{R}, L) = \mathsf{Fail} \qquad (L, s_{\text{next}}) \Downarrow L'}{(L, \mathsf{cons}\ a\ s_{\text{next}}) \Downarrow L'}\ \textsc{SeqFail}$$

Figure 8-1: Scheduler semantics, with `rule` $rl = a$

Note that there is no difference between a rule that fails and an empty rule.

Semantics of actions Now we can describe in detail the way log generation by rules is defined inductively:

- $\mathcal{R}$ records the state of all available registers at the beginning of a clock cycle. Hence $\mathcal{R}$ remains invariant throughout the execution of a rule, and in fact throughout the execution of all the rules in a schedule.

- Γ tracks pairs of names and values created by `let` constructs. It starts out empty.

- ℓ accumulates the reads and writes of the rule.
- L accumulates a trace of all the reads and writes performed by rules executed earlier in the same clock cycle (i.e., as part of the same schedule). L remains invariant through the execution of the rule but affects the validity of reads and writes by this rule.

The semantics of actions are defined by structural induction in Figure 8-2. We write $\Gamma \vdash (\ell, a) \downarrow_{(L,\mathcal{R})} (\ell', v)$, to indicate that *in environment* Γ*, with log* L *and registers* $\mathcal{R}$*, executing an action* a *transforms* ℓ *into* ℓ' *and returns value* v. When there is no ambiguity, we omit L and $\mathcal{R}$ and write $\Gamma \vdash (\ell, a) \downarrow (\ell', v)$ instead.

Careful inspection of our semantic judgments reveals that all premises are deterministic and computable. That means we can define a computable evaluation function unambiguously, returning either the result of executing the action or Fail if at any point in the execution the conditions of the relevant rules are not met and the rule execution cannot proceed. Thus, where we previously wrote "$[\![a]\!](\mathcal{R}, L) = \mathsf{Log}\,\ell$ if a succeeds and produces the log ℓ," we can now be precise:

$$[\![a]\!](\mathcal{R}, L) = \begin{cases} \mathsf{Log}\,\ell & \text{if } \emptyset \vdash ([], a) \downarrow_{(L,\mathcal{R})} (\ell, \mathsf{tt}) \\ \mathsf{Fail} & \text{otherwise} \end{cases}$$

Finally, at the end of each cycle, we update the values of all registers based on the reads and writes accumulated in log L. The same considerations of determinism and computability apply to the execution of schedulers, so we can define our final state-transition function δ_s, capturing all updates done to registers in a cycle when following scheduler s:

$$\delta_s(\mathcal{R}) = \mathsf{update}\,(\mathcal{R}, L) \;\text{ if } ([], s) \Downarrow L.$$

8.4 The One-Rule-at-a-Time Theorem

Our semantics builds a log accumulating the updates performed by all rules that the dynamic scheduler allows to run. It guarantees that performing a single update of the registers at the end of the cycle, after running multiple rules, yields the same state as performing updates after running each rule (as if a single rule had run in each cycle).

$$\frac{\Gamma[x] = v}{\Gamma \vdash (\ell, x) \downarrow (\ell, v)}\textsc{Var} \qquad \frac{}{\Gamma \vdash (\ell, \mathtt{skip}) \downarrow (\ell, \mathsf{tt})}\textsc{Skip}$$

$$\frac{}{\Gamma \vdash (\ell, b) \downarrow (\ell, b)}\textsc{Const}$$

$$\frac{\forall\, 1 \le i \le n.\, \Gamma \vdash (\ell_{i-1}, a_i) \downarrow (\ell_i, v_i)}{\Gamma \vdash (\ell_0, f(a_1, \ldots, a_n)) \downarrow (\ell_n, f\ v_1 \ldots v_n)}\textsc{Call}$$

$$\frac{\Gamma \vdash (\ell, a_c) \downarrow (\ell', \mathtt{0b1}) \qquad \Gamma \vdash (\ell', a_t) \downarrow (\ell'', v)}{\Gamma \vdash (\ell, \mathtt{if}\ a_c\ \mathtt{then}\ a_t\ \mathtt{else}\ a_f) \downarrow (\ell'', v)}\textsc{IfT}$$

$$\frac{\Gamma \vdash (\ell, a_c) \downarrow (\ell', \mathtt{0b0}) \qquad \Gamma \vdash (\ell', a_f) \downarrow (\ell'', v)}{\Gamma \vdash (\ell, \mathtt{if}\ a_c\ \mathtt{then}\ a_t\ \mathtt{else}\ a_f) \downarrow (\ell'', v)}\textsc{IfF}$$

$$\frac{\Gamma \vdash (\ell, a_1) \downarrow (\ell', v) \qquad \Gamma[x \mapsto v] \vdash (\ell', a_2) \downarrow (\ell'', v')}{\Gamma \vdash (\ell, \mathtt{let}\ x = a_1\ \mathtt{in}\ a_2) \downarrow (\ell'', v')}\textsc{Bind}$$

$$\frac{(\mathsf{wr1}, r, *) \notin L \qquad (\mathsf{wr0}, r, *) \notin L}{\Gamma \vdash (\ell, r.\mathtt{rd0}) \downarrow (\ell +\!\!+ [(\mathsf{rd0}, r)], \mathcal{R}[r])}\textsc{Read0}$$

$$\frac{(\mathsf{wr1}, r, *) \notin L \qquad v = \begin{cases} \mathcal{R}[r] & \text{if } (\mathsf{wr0}, r, *) \notin L +\!\!+ \ell \\ v_0 & \text{if } (\mathsf{wr0}, r, v_0) \in L +\!\!+ \ell \end{cases}}{\Gamma \vdash (\ell, r.\mathtt{rd1}) \downarrow (\ell +\!\!+ [(\mathsf{rd1}, r)], v)}\textsc{Read1}$$

$$\frac{\begin{array}{c}\Gamma \vdash (\ell, a) \downarrow (\ell', v) \qquad (\mathsf{wr0}, r, *) \notin L +\!\!+ \ell' \\ (\mathsf{wr1}, r, *) \notin L +\!\!+ \ell' \qquad (\mathsf{rd1}, r) \notin L +\!\!+ \ell'\end{array}}{\Gamma \vdash (\ell, r.\mathtt{wr0}(a)) \downarrow (\ell' +\!\!+ [(\mathsf{wr0}, r, v)], \mathsf{tt})}\textsc{Write0}$$

$$\frac{\Gamma \vdash (\ell, a) \downarrow (\ell', v) \qquad (\mathsf{wr1}, r, *) \notin L +\!\!+ \ell'}{\Gamma \vdash (\ell, r.\mathtt{wr1}(a)) \downarrow (\ell' +\!\!+ [(\mathsf{wr1}, r, v)], \mathsf{tt})}\textsc{Write1}$$

Figure 8-2: Rule semantics (assuming well-formed programs)

Let us illustrate that statement with the following example:

```
rule incr = x.wr0(x.rd0 + 1)
rule copy = y.wr0(x.rd1)
rule decr = x.wr1(x.rd1 - 1)
schedule _ = [incr; copy; decr]
```

The choice of scheduler and the port annotations (using `rd1` in `copy` and `wr1` in `decr`) ensure that all rules can run in each cycle. Overall, this program assigns $(x+1)-1$ to register `x` and $x+1$ to register `y`. This result (obtained by running multiple rules in a single cycle) respects one-rule-at-a-time semantics, because the same result can be obtained by running one rule per cycle, in the order specified by the scheduler: $x \leftarrow x+1$, then $y \leftarrow x$, and finally $x \leftarrow x-1$.

Most of the checks that appear in the premises of Figure 8-2 are there to preserve ORAAT semantics. For example, allowing a `rd0` to follow a `wr0` performed by an earlier rule in the same cycle would not respect ORAAT: the `rd0` would observe the old value of the register if it ran in the same cycle, vs. the new value if it ran in the next cycle.

We define the action of a single rule on the registers by applying the semantics of actions directly:

$$\frac{[\![a]\!](\mathcal{R}, \emptyset) = \mathsf{Log}\ \ell \qquad \mathcal{R}' = \mathsf{update}\,(\mathcal{R}, \ell)}{\mathcal{R} \rightarrow_a \mathcal{R}'}\ \text{RULE}$$

And we define a relation indicating that a state is reachable in several rules: $\mathcal{R} \rightarrow^*_{[]} \mathcal{R}$ and $\mathcal{R} \rightarrow^*_{\mathrm{h::t}} \mathcal{R}''$ when $\exists \mathcal{R}'.\ \mathcal{R} \rightarrow_h \mathcal{R}' \wedge \mathcal{R}' \rightarrow^*_{\mathrm{t}} \mathcal{R}''$.

We can now give the proper statement of a key property:

Theorem 8.1 (One-rule-at-a-time theorem). *If $\delta_s(\mathcal{R}) = \mathcal{R}_1$ then there exists a sequence of rules of the program that one-at-a-time reach the same state: $\exists\ rls \in \mathsf{traces}(s).\ \mathcal{R} \rightarrow^*_{rls} \mathcal{R}_1$, where $\mathsf{traces}(s)$ refers to all sequences of rules named in the scheduler s.*

This theorem can be found as `OneRuleAtATime` in the file `OneRuleAtATime.v`. In this paper, we detail only the following key lemma.

Lemma 8.2 (Committing the effect of previous rules). $\forall a, \Gamma, \ell, L_{new}, L_{old}, \ell', v, \mathcal{R}.$
$\Gamma \vdash (\ell, a) \downarrow_{L_{old} +\!\!+ L_{new}, \mathcal{R}} (\ell', v) \Rightarrow$
$\Gamma \vdash (\ell, a) \downarrow_{L_{new}, \mathsf{update}(\mathcal{R}, L_{old})} (\ell', v)$

Proof. This lemma corresponds to `interp_rule_commit` in `OneRuleAtATime.v`. The proof works by induction on a. We outline the most interesting cases here. One key invariant is that there is at most one `wr0` and `wr1` per register, as the semantics prevent double `wr0` or `wr1`.

r.rd1. Assume $\Gamma \vdash (\ell, r.\texttt{rd1}) \downarrow_{L_{\text{old}} +\!\!+ L_{\text{new}}, \mathcal{R}} (\ell', v)$. There are two cases depending on where the `rd1` read the value from.

Case 1: *Read from register.* Necessarily we have $\ell' = \ell +\!\!+ [(\mathsf{rd1}, r)]$, $\mathcal{R}[r] = v$, $(\mathsf{wr1}, r, *) \notin L_{\text{old}} +\!\!+ L_{\text{new}}$ and $(\mathsf{wr0}, r, *) \notin \ell +\!\!+ L_{\text{old}} +\!\!+ L_{\text{new}}$, which implies that $(\mathsf{wr1}, r, *) \notin L_{\text{new}}$ and $(\mathsf{wr0}, r, *) \notin L_{\text{new}} +\!\!+ \ell$. We also get $(\mathsf{wr0}, r, *) \notin L_{\text{old}}$ and $(\mathsf{wr1}, r, *) \notin L_{\text{old}}$, so:

$$\mathsf{update}\,(\mathcal{R}, L_{\text{old}})[r] = \mathcal{R}[r]$$

Hence we can use the READ1 rule reading from the register, with $L ::= L_{\text{new}}$ and $\mathcal{R} ::= \mathsf{update}\,(\mathcal{R}, L_{\text{old}})$ and so $\Gamma \vdash (\ell, r.\texttt{rd1}) \downarrow_{L_{\text{new}}, \mathsf{update}(\mathcal{R}, L_{\text{old}})} (\ell', v)$.

Case 2: *Read from log.* Necessarily we have $\ell' = \ell +\!\!+ [(\mathsf{rd1}, r)]$, $(\mathsf{wr0}, r, v) \in L_{\text{old}} +\!\!+ L_{\text{new}} +\!\!+ \ell$, and $(\mathsf{wr1}, r, *) \notin L_{\text{old}} +\!\!+ L_{\text{new}}$.

There are three subcases depending on the source of the unique `wr0` we read from. The write is in ℓ, in L_{new}, or in L_{old}. The only interesting case is when the write is in L_{old}. We have $(\mathsf{wr0}, r, v) \in L_{\text{old}}$ and $(\mathsf{wr1}, r, *) \notin L_{\text{old}}$, so $\mathsf{update}\,(\mathcal{R}, L_{\text{old}})[r] = v$. Moreover, $(\mathsf{wr0}, r, *) \notin L_{\text{new}} +\!\!+ \ell$ and $(\mathsf{wr1}, r, *) \notin L_{\text{new}}$.

So we have all the premises to apply READ1, reading directly from the register. This case is the most interesting because, in serializing a trace, we converted a log read into a register read. We get $\Gamma \vdash (\ell, r.\texttt{rd1}) \downarrow_{L_{\text{new}}, \mathsf{update}(\mathcal{R}, L_{\text{old}})} (\ell', v)$. □

Formalization The complete proof is carried out in full detail in `OneRuleAtATime.v`. A high degree of automation ensures that the proof of the key invariants is short (about 40 lines) and robust, which enabled us to iterate quickly when designing Kôika's semantics (in all cases, we were able to make changes to the semantics and confirm with few to no proof edits that the new semantics still respected ORAAT).

8.5 Case study: a cycle-accurate characterization of a small pipelined system

We now have all the pieces in place to demonstrate how one might use our semantics to prove interesting characteristics of a circuit beyond functional correctness. As a concrete example, we study the pipeline of two combinational functions f_1 and f_2 that our introduction alluded to. Recall that, if f_1 and f_2 both have critical-path length l, then a naïve implementation of their composition $f_2 \circ f_1$ in a single rule `do_f12` would have length $2 \cdot l$. On the other hand, if we decompose the system into two independent rules `do_f1` and `do_f2` connected through a one-element queue, we can reduce the critical-path length to just l.

But this path-length reduction *only matters if we can guarantee that f_1 and f_2 run concurrently in each cycle*, that is, the system actually runs in a pipelined manner. A traditional ORAAT semantics is enough to prove that the "pipelined" system is a correct refinement of the monolithic one (in the sense that it computes the same values) but is not sufficient to prove the two-rules-per-cycle property. In fact, it would be hard even to state such a property because "cycle" is not a meaningful concept in a typical ORAAT formalization.

In the following we present the implementation of a simple pipelined system and sketch its proof. We start with the implementation, in which two rules `do_f1` and `do_f2` are connected through a one-element queue composed of a data-holding register `r` and a flag `empty` indicating whether the queue is empty. Initially, `empty` is set to `true`.

```
rule feed_pipeline =
 clock.wr0(clock.rd0 + 1)
 input.wr0(input_stream(clock.rd0))
```

```
rule do_f2 =
 if empty.rd0 then
  abort
 else
  // dequeue
  out.wr0(f2(r.rd0));
  empty.wr0(true)
```

```
rule do_f1 =
 if empty.rd1 then
  // enqueue
  empty.wr1(false);
  r.wr0(f1(input.rd1))
 else
  abort
```

```
schedule pipeline =
  [feed_pipeline; do_f2; do_f1]
```

One-rule-at-a-time reasoning is sufficient to prove that our pipeline is functionally correct (it computes the composition of f_1 and f_2). The methodology we presented in this dissertation (or even its predecessor in [19]) applies directly.

More interestingly, we can also prove that the system processes one value per cycle and hence deserves to be called a *pipeline*. From the second cycle on, the circuit simultaneously performs `do_f1` and `do_f2` on each cycle (on the first cycle, only `do_f1` can fire, since there is no value in the pipeline for `do_f2` to dequeue and process).

The proof is in two steps. First, by applying our semantics to the program, we derive a sufficient (and, in fact, necessary) criterion for both `do_f1` and `do_f2` to fire simultaneously: both rules will fire in a cycle if `empty` contains `false` at the beginning of that cycle (i.e. the pipeline is not empty).

This property is not an invariant in the one-rule-at-a-time sense, since `do_f2` breaks the invariant by emptying the pipeline, and `do_f1` reestablishes it, but it is a *cycle invariant*: from our semantics, it is straightforward to show that (1) if the pipeline is empty, `do_f1` will fill it, (2) if the pipeline is nonempty, `do_f1` and `do_f2` will both fire in the same cycle, and (3) running `do_f2` and then `do_f1` in a nonempty pipeline maintains a nonempty pipeline (a one-rule-at-a-time argument is enough for this last part).

Hence, the pipeline fills and, once full, stays full: from the second cycle on, the pipeline runs both rules on every cycle and processes one element per cycle.

This concludes the proof that the pipelined design is indeed *pipelined*. It is a typical example of how Kôika can be used to reason about performance properties, in contrast with

the functional correctness guarantees that we tackled in the rest of this dissertation.

Chapter 9

Conclusion

In this dissertation, we explained our approach to proving the functional correctness of rule-based hardware designs in an interactive theorem prover.

We presented Fjfj, a restricted modular rule-based programming language embedded in Coq. Thanks to language restrictions, the semantics of Fjfj modules can be given by a few mathematical relations, each of which characterizes the transition relation of a method or a rule in isolation.

This formalization allows us to define Fjfj modules either from rule-based programs, or by describing the transitions as logical predicates directly in Coq. We showed the usefulness of the second style both to define nonoperational intermediate specifications and to axiomatize modules for which we do not have implementations. We also formalized a notion of refinement that could be kept simple thanks to our language restrictions, and we explained how refinement gives us a substitution principle.

After specifying and proving simple modules in our framework, we showcased fundamental issues when trying to give *modular specifications* for complex sequential machines like processors. In the case of processors, we proposed a solution by introducing our *generalized processor specification*. We leveraged this new intermediate specification to prove the functional correctness of a family of pipelined processors modularly. Thanks to modularity, local processor modifications that stay within the boundaries of our modular hierarchical decomposition can be proven correct in isolation of the rest of the design. This can save both engineering and computing time, as the parts of the proof that are unchanged do not

need to modified or machine-checked again.

Bibliography

[1] Abhinav Agarwal, Man Cheuk Ng, and Arvind. A comparative evaluation of high-level hardware synthesis using Reed–Solomon decoder. *Embedded Systems Letters*, 2(3):72–76, 2010.

[2] Markus Aronsson and Mary Sheeran. Hardware software co-design in Haskell. In *Proceedings of the 10th ACM SIGPLAN International Symposium on Haskell, Oxford, United Kingdom, September 7-8, 2017*, pages 162–173, 2017.

[3] Arvind and Xiaowei Shen. Using term rewriting systems to design and verify processors. *IEEE Micro*, 19(3):36–46, 1999.

[4] Jonathan Bachrach, Huy Vo, Brian C. Richards, Yunsup Lee, Andrew Waterman, Rimas Avizienis, John Wawrzynek, and Krste Asanovic. Chisel: constructing hardware in a Scala embedded language. In *The 49th Annual Design Automation Conference 2012, DAC '12, San Francisco, CA, USA, June 3-7, 2012*, pages 1216–1225, 2012.

[5] Sergey Berezin, Armin Biere, Edmund Clarke, and Yunshan Zhu. Combining symbolic model checking with uninterpreted functions for out-of-order processor verification. In Ganesh Gopalakrishnan and Phillip Windley, editors, *Formal Methods in Computer-Aided Design*, pages 369–386, Berlin, Heidelberg, 1998. Springer Berlin Heidelberg.

[6] Per Bjesse, Koen Claessen, Mary Sheeran, and Satnam Singh. Lava: Hardware design in Haskell. In *Proceedings of the third ACM SIGPLAN International Conference on Functional Programming (ICFP '98), Baltimore, Maryland, USA, September 27-29, 1998.*, pages 174–184, 1998.

[7] Thomas Bourgeat, Clément Pit-Claudel, Adam Chlipala, and Arvind. The essence of Bluespec: A core language for rule-based hardware design. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2020, page 243–257, New York, NY, USA, 2020. Association for Computing Machinery.

[8] B. A. Brady, R. E. Bryant, and S. A. Seshia. Learning conditional abstractions. In *2011 Formal Methods in Computer-Aided Design (FMCAD)*, pages 116–124, Oct 2011.

[9] B. A. Brady, R. E. Bryant, S. A. Seshia, and J. W. O'Leary. ATLAS: Automatic term-level abstraction of RTL designs. In *Eighth ACM/IEEE International Conference on Formal Methods and Models for Codesign (MEMOCODE 2010)*, pages 31–40, July 2010.

[10] C. Brock and W.A. Hunt. Formally specifying and mechanically verifying programs for the Motorola complex arithmetic processor DSP. In *Proceedings International Conference on Computer Design VLSI in Computers and Processors*, pages 31–36, 1997.

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

[12] Randal E. Bryant. Symbolic Boolean manipulation with ordered binary-decision diagrams. *ACM Comput. Surv.*, 24(3):293–318, sep 1992.

[13] Randal E Bryant. Formal verification of pipelined Y86-64 microprocessors with UCLID5. Technical report, Technical Report CMU-CS-18-122, 2018.

[14] Jerry R. Burch and David L. Dill. Automatic verification of pipelined microprocessor control. In David L. Dill, editor, *Computer Aided Verification*, pages 68–80, Berlin, Heidelberg, 1994. Springer Berlin Heidelberg.

[15] William E. Byrd, Michael Ballantyne, Gregory Rosenblatt, and Matthew Might. A unified approach to solving seven programming problems (functional pearl). *Proc. ACM Program. Lang.*, 1(ICFP), aug 2017.

[16] Andrew Canis, Jongsok Choi, Mark Aldham, Victor Zhang, Ahmed Kammoona, Jason Helge Anderson, Stephen Dean Brown, and Tomasz S. Czajkowski. Legup: high-level synthesis for FPGA-based processor/accelerator systems. In *Proceedings of the ACM/SIGDA 19th International Symposium on Field Programmable Gate Arrays, FPGA 2011, Monterey, California, USA, February 27, March 1, 2011*, pages 33–36, 2011.

[17] Joonwon Choi. An inlining approach to formal hardware semantics. Master's thesis, Massachusetts Institute of Technology, USA, 2016.

[18] Joonwon Choi, Adam Chlipala, and Arvind. Hemiola: A DSL and verification tools to guide design and proof of hierarchical cache-coherence protocols. In Sharon Shoham and Yakir Vizel, editors, *Computer Aided Verification - 34th International Conference, CAV 2022, Haifa, Israel, August 7-10, 2022, Proceedings, Part II*, volume 13372 of *Lecture Notes in Computer Science*, pages 317–339. Springer, 2022.

[19] Joonwon Choi, Muralidaran Vijayaraghavan, Benjamin Sherman, Adam Chlipala, and Arvind. Kami: A platform for high-level parametric hardware specification and its modular verification. *Proc. ACM Program. Lang.*, 1(ICFP), August 2017.

[20] Jason Cong, Bin Liu, Stephen Neuendorffer, Juanjo Noguera, Kees A. Vissers, and Zhiru Zhang. High-level synthesis for FPGAs: From prototyping to deployment. *IEEE Trans. on CAD of Integrated Circuits and Systems*, 30(4):473–491, 2011.

[21] Nirav Dave. *A Unified Model for Hardware/Software Codesign.* PhD thesis, USA, 2011. AAI0828101.

[22] Nirav Dave, Michael Katelman, Myron King, Arvind, and José Meseguer. Verification of microarchitectural refinements in rule-based systems. In Satnam Singh, Barbara Jobstmann, Michael Kishinevsky, and Jens Brandt, editors, *9th IEEE/ACM International Conference on Formal Methods and Models for Codesign, MEMOCODE 2011, Cambridge, UK, 11-13 July, 2011*, pages 61–71. IEEE, 2011.

[23] Nirav Dave, Man Cheuk Ng, Michael Pellauer, and Arvind. A design flow based on modular refinement. In *8th ACM/IEEE International Conference on Formal Methods and Models for Codesign (MEMOCODE 2010), Grenoble, France, 26-28 July 2010*, pages 11–20. IEEE Computer Society, 2010.

[24] Conal Elliott. Compiling to categories. *PACMPL*, 1(ICFP):27:1–27:27, 2017.

[25] Daniel D. Gajski. Specc design environment. *System Design*, page 217–235, 2001.

[26] David J. Greaves. Further sub-cycle and multi-cycle schedulling support for Bluespec Verilog. In Partha S. Roop, Naijun Zhan, Sicun Gao, and Pierluigi Nuzzo, editors, *Proceedings of the 17th ACM-IEEE International Conference on Formal Methods and Models for System Design, MEMOCODE 2019, La Jolla, CA, USA, October 9-11, 2019*, pages 2:1–2:11. ACM, 2019.

[27] Sumit Gupta, Nikil D. Dutt, Rajesh Gupta, and Alexandru Nicolau. Loop shifting and compaction for the high-level synthesis of designs with complex control flow. In *2004 Design, Automation and Test in Europe Conference and Exposition (DATE 2004), 16-20 February 2004, Paris, France*, pages 114–121, 2004.

[28] John V. Guttag, James J. Horning, Stephen J. Garland, Kevin D. Jones, A. Modet, and Jeannette M. Wing. *Larch: Languages and Tools for Formal Specification.* Texts and Monographs in Computer Science. Springer, 1993.

[29] Yann Herklotz, James D. Pollard, Nadesh Ramanathan, and John Wickerson. Formal verification of high-level synthesis. *Proc. ACM Program. Lang.*, 5(OOPSLA), oct 2021.

[30] James C Hoe. *Operation-centric hardware description and synthesis.* PhD thesis, Massachusetts Institute of Technology, 2000.

[31] James C. Hoe. *Operation-centric hardware description and synthesis.* PhD thesis, Massachusetts Institute of Technology, Cambridge, MA, USA, 2000.

[32] James C. Hoe and Arvind. Synthesis of operation-centric hardware descriptions. In *Proceedings of the 2000 IEEE/ACM International Conference on Computer-Aided Design, 2000, San Jose, California, USA, November 5-9, 2000*, pages 511–518, 2000.

[33] Bo-Yuan Huang, Hongce Zhang, Pramod Subramanyan, Yakir Vizel, Aarti Gupta, and Sharad Malik. Instruction-level abstraction (ILA) a uniform specification for system-on-chip (SoC) verification. *ACM Transactions on Design Automation of Electronic Systems (TODAES)*, 24(1):1–24, 2018.

[34] Lana Josipović. *High-Level Synthesis of Dynamically Scheduled Circuits*. PhD thesis, EPFL, Switzerland, 2021.

[35] Lana Josipović, Radhika Ghosal, and Paolo Ienne. Dynamically scheduled high-level synthesis. In *Proceedings of the 2018 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, FPGA '18, page 127–136, New York, NY, USA, 2018. Association for Computing Machinery.

[36] Warren A. Hunt Jr. Microprocessor design verification. *J. Autom. Reason.*, 5(4):429–460, 1989.

[37] Michal Karczmarek and Arvind. Synthesis from multi-cycle atomic actions as a solution to the timing closure problem. In *2008 International Conference on Computer-Aided Design, ICCAD 2008, San Jose, CA, USA, November 10-13, 2008*, pages 24–31, 2008.

[38] Michal Karczmarek, Arvind, and Muralidaran Vijayaraghavan. A new synthesis procedure for atomic rules containing multi-cycle function blocks. In *Twelfth ACM/IEEE International Conference on Formal Methods and Models for Codesign, MEMOCODE 2014, Lausanne, Switzerland, October 19-21, 2014*, pages 22–31, 2014.

[39] Leslie Lamport. How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE Transactions on Computers*, C-28(9):690–691, 1979.

[40] C. Y. Lee. Representation of switching circuits by binary-decision programs. *The Bell System Technical Journal*, 38(4):985–999, 1959.

[41] Kenneth L. McMillan. *The SMV System*, pages 61–85. Springer US, Boston, MA, 1993.

[42] Kenneth L McMillan. Verification of an implementation of Tomasulo's algorithm by compositional model checking. In *International Conference on Computer Aided Verification*, pages 110–121. Springer, 1998.

[43] Kenneth L. McMillan. A methodology for hardware verification using compositional model checking. *Science of Computer Programming*, 37(1-3):279–309, 2000.

[44] Mentor. Modelsim. https://www.mentor.com/products/fpga/verification-simulation/modelsim/.

[45] Bartosz Milewski. Category theory for programmers. https://bartoszmilewski.com/2014/10/28/category-theory-for-programmers-the-preface/, 2017.

[46] Matthew W. Moskewicz, Conor F. Madigan, Ying Zhao, Lintao Zhang, and Sharad Malik. Chaff: Engineering an efficient SAT solver. In *Proceedings of the 38th Design Automation Conference, DAC 2001, Las Vegas, NV, USA, June 18-22, 2001*, pages 530–535. ACM, 2001.

[47] Charles Gregory Nelson. *Techniques for Program Verification*. PhD thesis, Stanford, CA, USA, 1980. AAI8011683.

[48] Rishiyur S. Nikhil. Formal semantics of bsv as a Haskell program. https://github.com/rsnikhil/Bluespec_BSV_Formal_Semantics.

[49] Rishiyur S. Nikhil. Bluespec System Verilog: Efficient, correct RTL from high level specifications. In *Proceedings of the Second ACM/IEEE International Conference on Formal Methods and Models for Co-Design*, MEMOCODE '04, pages 69–70, Washington, DC, USA, 2004. IEEE Computer Society.

[50] Scott Owens, Susmit Sarkar, and Peter Sewell. A better x86 memory model: X86-tso. In *Proceedings of the 22nd International Conference on Theorem Proving in Higher Order Logics*, TPHOLs '09, page 391–407, Berlin, Heidelberg, 2009. Springer-Verlag.

[51] Daniel L. Rosenband. Hardware synthesis from guarded atomic actions with performance specifications. In *2005 International Conference on Computer-Aided Design, ICCAD 2005, San Jose, CA, USA, November 6-10, 2005*, pages 784–791, 2005.

[52] Daniel L. Rosenband and Arvind. Modular scheduling of guarded atomic actions. In *Proceedings of the 41th Design Automation Conference, DAC 2004, San Diego, CA, USA, June 7-11, 2004*, pages 55–60, 2004.

[53] Daniel Selsam and Leonardo de Moura. Congruence closure in intensional type theory. In Nicola Olivetti and Ashish Tiwari, editors, *Automated Reasoning - 8th International Joint Conference, IJCAR 2016, Coimbra, Portugal, June 27 - July 2, 2016, Proceedings*, volume 9706 of *Lecture Notes in Computer Science*, pages 99–115. Springer, 2016.

[54] Sanjit A. Seshia and Pramod Subramanyan. Uclid5: Integrating modeling, verification, synthesis and learning. In *2018 16th ACM/IEEE International Conference on Formal Methods and Models for System Design (MEMOCODE)*, pages 1–10, 2018.

[55] Xiaowei Shen. *Modular Verification of Hardware Systems*. Massachusetts Institute of Technology, 2000.

[56] Jeffrey X Su, David L Dill, and Clark W Barrett. Automatic generation of invariants in processor verification. In *International Conference on Formal Methods in Computer-Aided Design*, pages 377–388. Springer, 1996.

[57] Emina Torlak and Rastislav Bodík. A lightweight symbolic virtual machine for solver-aided host languages. In Michael F. P. O'Boyle and Keshav Pingali, editors, *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '14, Edinburgh, United Kingdom - June 09 - 11, 2014*, pages 530–541. ACM, 2014.

[58] Muralidaran Vijayaraghavan. *Modular Verification of Hardware Systems*. PhD thesis, Massachusetts Institute of Technology, 2016.

[59] Muralidaran Vijayaraghavan, Adam Chlipala, Arvind, and Nirav Dave. Modular deductive verification of multiprocessor hardware designs. In Daniel Kroening and Corina S. Pasareanu, editors, *Computer Aided Verification - 27th International Conference, CAV 2015, San Francisco, CA, USA, July 18-24, 2015, Proceedings, Part II*, volume 9207 of *Lecture Notes in Computer Science*, pages 109–127. Springer, 2015.

[60] Muralidaran Vijayaraghavan, Adam Chlipala, Arvind, and Nirav Dave. Modular deductive verification of multiprocessor hardware designs. In Daniel Kroening and Corina S. Pasareanu, editors, *Computer Aided Verification - 27th International Conference, CAV 2015, San Francisco, CA, USA, July 18-24, 2015, Proceedings, Part II*, volume 9207 of *Lecture Notes in Computer Science*, pages 109–127. Springer, 2015.

[61] Max Willsey, Chandrakana Nandi, Yisu Remy Wang, Oliver Flatt, Zachary Tatlock, and Pavel Panchekha. Egg: Fast and extensible equality saturation. *Proc. ACM Program. Lang.*, 5(POPL), jan 2021.

[62] Andrew Wright. *Modular SMT-Based Verification of Rule-Based Hardware Designs*. PhD thesis, Massachusetts Institute of Technology, USA, 2021.

[63] Xilinx. Vivado hls. https://www.xilinx.com/products/design-tools/vivado/integration/esl-design.html.

[64] Yihong Zhang, Yisu Remy Wang, Max Willsey, and Zachary Tatlock. Relational e-matching. *Proc. ACM Program. Lang.*, 6(POPL), jan 2022.