Modular SMT-Based Verification of Rule-Based Hardware Designs

by

Andrew C. Wright

B.S., University of Florida (2011)
S.M., Massachusetts Institute of Technology (2013)

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

June 2021

Author .....
Department of Electrical Engineering and Computer Science
May 20, 2021

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

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

Modular SMT-Based Verification of Rule-Based Hardware Designs

by

Andrew C. Wright

Submitted to the Department of Electrical Engineering and Computer Science
on May 20, 2021, in partial fulfillment of the
requirements for the degree of
Doctor of Philosophy

Abstract

The highly-concurrent nature of hardware logic designs makes their design and verification difficult. Bluespec SystemVerilog (BSV) simplifies this problem by introducing the rule-level abstraction. Modules are expressed in terms of guarded atomic actions that appear to fire in a sequential order, even when multiple rules fire concurrently per clock cycle. This allows designers to think about concurrent systems one step at a time.

This thesis aims to make hardware verification easier by leveraging the rule-level abstraction for verification using SMT-based verification, *e.g.* bounded and unbounded model checking. The main aspect of the rule-level abstraction taken advantage of is the modularity. Rule-level modules can only be interacted with through their interface methods, so if a module looks the same as its specification with respect to the legal sequences of method calls, then they can be used interchangeably without affecting the outer module. A modular verification technique that is based off of this idea can be used to replace complex submodules with simpler versions that reduce the complexity and the number of steps required for unbounded model checking. This modular verification methodology can take many problems that are infeasible for unbounded model checking and makes them feasible. Other aspects of rule-level hardware design languages (HDLs) are taken advantage of during verification such as the use of uninterpreted functions and abstract types.

In this thesis, I present Spec 'n' Check, an HDL inspired by BSV that is designed for powerful modularity and easy-to-write specifications. To fully support SMT-based verification, we present formal semantics for Spec 'n' Check along with a symbolic representation of the semantics that can be used in SMT solvers. We also present what it means for a module to implement a specification and the related metatheorems that describe how the implements relation can be used to verify larger modules. We show that with this work, it is possible to formally verify a RISC-V pipelined processor implemented in the synthesizable subset of Spec 'n' Check against an ISA specification in a matter of minutes of SMT solver run time. This is only possible thanks to module refinement and abstraction of the control and status register file (CSRF) and the memory system, and the use of uninterpreted functions.

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

Acknowledgments

First I would like to thank my advisor Prof. Arvind for his guidance throughout my time at MIT. While working with Arvind, I got involved in many exciting research and teaching opportunities, and I learned a lot from him. Without Arvind and his support and encouragement, this thesis would not have been possible.

I want to thank my committee members Profs. Adam Chlipala, Daniel Sanchez, and Armando Solar-Lezama. Their feedback and questions helped improved this thesis significantly.

I want to especially thank Prof. Adam Chlipala for sharing his expertise in everything related to formal verification of hardware. He has been interested in my work and provided encouragement since I worked with him on the DARPA SSITH project. When I started writing my thesis, he was available to give me essential feedback at multiple stages during the process. I also want to thank him for agreeing to be a late addition to my thesis committee.

I also would like to thank my labmate and pre-pandemic officemate Thomas Bourgeat. We have worked together on various projects since he was just a visiting student at MIT, and since then we have built multiple processors together. I enjoyed our many whiteboard talks where we shared our newest ideas for research and tried to figure out if they would work. Many concepts in this thesis were first presented to Thomas on a whiteboard, and his enthusiastic response to my work helped me feel confident with my thesis.

While at MIT I got the chance to fabricate some of my processor designs in ASICs as part of larger research projects. I want to thank Gage Hills and Prof. Max Shulaker for the opportunity to work with them and their colleagues to make the first realistic microcontroller out of carbon nanotube transistors. This work also gave me an opportunity to dive further into model checking, and as a result, it inspired some of the work in this thesis. I also want to thank Chiraag Juvekar, Utsav Banerjee, and Prof. Anantha Chandrakasan for the opportunity to make a RISC-V processor to go along with their DTLS accelerator.

I also want to thank all of my labmates past and present including Sizhuo Zhang, Sang Woo Jun, Murali Vijayaraghavan, Shuotao Xu, Chanwoo Chung, Joonwon Choi, and Tianhao Huang.

I want to especially thank my family back in Florida including my parents, my brother Bob, his wife Cynthia, and my nephews Max and Lucas. They have always been supportive of me and given me opportunities to take much needed breaks from my work. I want to thank them for their patience as I finally finish.

Finally I would like to thank my love, Leslie. This past year, the pandemic made us office mates since we were working from home, and I do not know how I would have made it through working on my thesis without always having her around. From making amazing 48-hour cookies to helping me set realistic time tables for my writing, she has done just about everything to help. This last year has been hard, but spending it with Leslie has made it all better.

Contents

List of Figures

List of Theorems

Chapter 1

Introduction

Hardware verification is a hard problem that takes a significant proportion of the total effort required to make an ASIC. ASIC verification is done at many abstraction levels ranging from high-level protocol verification [38, 63] down to back-end verification such as transistor-level layout-versus-schematic (LVS) verification [14], but logic-level verification takes the most effort of these. Even with significant amounts of pre-silicon testing, logic bugs are still found in post-silicon designs [45, 71].

Formal verification is an attractive solution for verifying logic because of its strong guarantees, but its applications are limited. Proof-based formal verification requires a significant amount of effort and expertise not typically available in design or verification teams [33, 82]. Automated formal verification using SAT or SMT-based model checking can be useful for finding bugs or verifying smaller designs [39], but its usefulness is often limited by exponential increases in complexity that put hard limits on the size of verification that can be completed in a reasonable time, similar to the state-space explosion problem for traditional model checking [39].

In general model checking, the state-space explosion problem states that when the number of states in a design increases, the state space grows exponentially. For SMT-based model checking, designs are unrolled over a number of *steps*, *e.g.* clock cycles for RTL, and as the number of steps increases, the state space of the verification problem grows exponentially as well. In addition to the state-space size, the complexity of expressions also significantly impacts the performance of SMT-based model checking. For example, the inclusion of a bit-

vector multiplier in a processor pipeline can significantly slow down bounded model checking.

In this thesis, we simplify the process of logic verification by using a high-level rule-based hardware description language (HDL) called Spec 'n' Check (heavily inspired by Bluespec SystemVerilog) along with SMT-based model checking. This work leverages the high-level abstraction and modularity of a rule-based HDL to introduce module refinement and abstraction to SMT-based model checking. This allows for rule-level safety properties to be verified more efficiently through model checking thanks to a style of modular verification that is specific to rule-based designs.

In rule-based model checking where rule firings are used as the steps, submodules can be replaced by similarly behaving modules in a way that doesn't change the property being verified but reduces the number of steps required for the verification. This is because a module's behavior can only be observed through its interface methods; no state or internal rule firings are visible outside the module. Therefore substitutable modules in our framework can require a different number of internal rule firings to compute results and therefore require a different number of model-checking steps for verifying properties about the outer module. Similar modular refinement in RTL model checking cannot decrease the number of steps required for verification because the refined module must have the same cycle-by-cycle behavior as the original module.

This work also leverages term-level abstractions including uninterpreted functions and abstract types. These term abstractions are commonly found directly incorporated into high-level modeling languages to simplify modeling and verification [73], but they are less commonly used in RTL since they are not included as part of the language; instead they either require designer effort to integrate these abstractions into the model-checking problem or they require automatic inference by tools that have limited scope [30, 29]. These abstractions have natural analogs in the synthesizable subset of Spec 'n' Check, *i.e.* user-defined functions and types such as structs, so they are easier to incorporate both manually and automatically in verification. Furthermore these abstractions are available directly in the non-synthesizable subset of Spec 'n' Check to produce high-level abstractions of modules.

With this work, it is possible to formally verify a pipelined processor implemented in the synthesizable subset of Spec 'n' Check against an ISA specification in a matter of minutes

of SMT-solver run time. This is thanks to module refinement and abstraction of the control and status register file (CSRF) and the memory system, and using uninterpreted functions in place of the main functions used in the pipeline, *e.g.* `decode`, `alu`, etc.

1.1 Modular Rule-Based Verification Overview

When verifying designs at the rule level, we only consider rule-level safety properties about modules. This includes verifying that a module implements a specification and verifying assertions within a module but does not include RTL-level concerns that depend on how the rule-level design is mapped to RTL.

We say a module implements a specification if the module can be weakly simulated [49] by the implementation. This is similar to (but slightly stronger than) saying all sequences of legal method calls of the module are legal sequences of method calls for the specification.

Modules can have assertions to check rule-level properties. Rule-level properties are properties that can be checked by adding registers, rules, and logic to a design. This allows modules to assert properties about the rule-level execution of a module, but it does not allow modules to assert properties that rely on the RTL implementation, including anything about clock cycles or exactly when rules fire. It also does not allow modules to assert when submodule methods are ready to fire.

One way we make these verification tasks easier is using modular verification. The idea behind modular verification is, if a module contains a submodule, the verification of the module can be performed by replacing the submodule with its specification.

Figure 1-1 shows an outer module with a hole in place for either a submodule or its specification. If verifying the assertion in the outer module or verifying the outer module implements some specification, then the specification can be used in place of the implementation for the verification, and any positive verification results will hold for the outer module using the implementation submodule.

Verifying assertions inside the submodule is trickier, especially when the correctness of the assertions depends on the outer module, but it is sometimes still possible to leverage modular verification. One way of doing this is to add an assertion to the specification that is

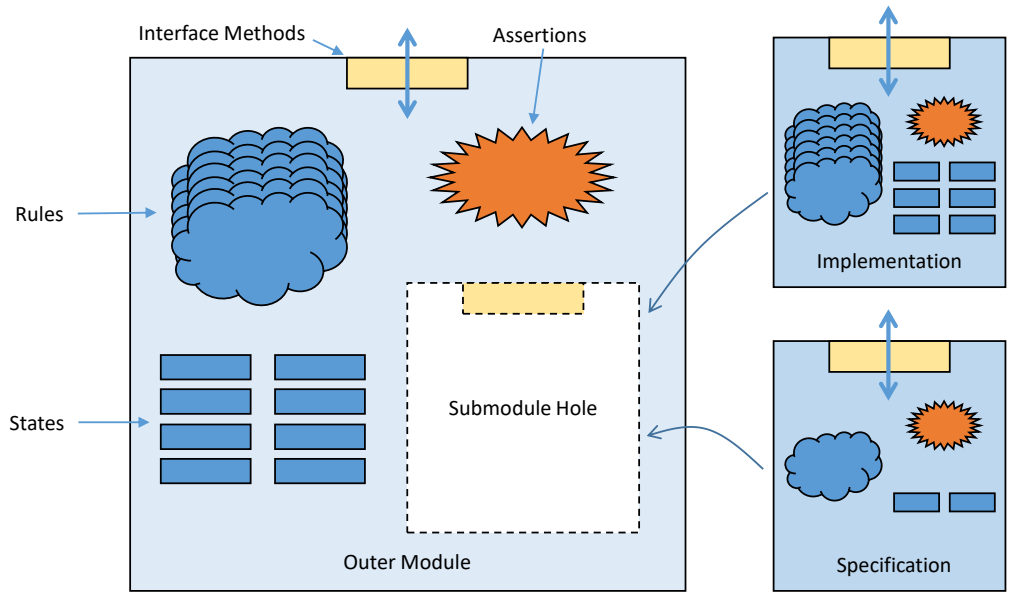

Figure 1-1: Outer module with hole for two potential submodules

related to the assertion in the implementation, and then verifying the specification's assertion holds in the outer module. Then the simulation relation between the implementation and the specification can be used to show that the implementation's assertion would also hold in the outer module without having to perform the complete verification.

1.1.1 Benefits of Modular Verification

To understand why modular verification is important, we look at what contributes to the complexity of SMT-based verification. In general, the complexity of an SMT-based model-checking problem depends on the number of states in the design, the complexity of the logic in the design, and the unrolling depth (in our case, how many rule-firings are considered during verification).

It is common for specifications to have fewer states, simpler logic, and fewer internal rules than implementations (many times specifications have no internal rules). For example, consider a multicycle multiplier module that uses Booth's algorithm for multiplication [24]. Booth's algorithm computes the product of two numbers through multiple additions, subtractions, and bit-shifts. In a rule-based language, the Booth multiplier would have a request method, a response method, and an internal rule. The request method would take in inputs,

the response method would return the result (once it is ready), and the rule implements a single step of the algorithm. For an n-bit multiplication, the rule must fire n times after the request method is called in order for the response method to be ready to call.

In a specification of a multicycle multiplier module, the multiplication would be done directly in the request method using the bit-level $*$ operator and then stored in a register so it can be returned by the response method later. This specification has significantly simpler logic than the implementation using Booth's algorithm since it uses a single bit-level function that represents multiplication instead of a sequence of bit-level functions that are equivalent to multiplication. Also, this specification requires no rules to compute the multiplication, while the implementation requires multiple rule firings.

Now consider the verification of a multicycle module that computes the binomial $ax^2 + bx + c$ where the three required multiplications are computed by an instance of the Booth multiplier. Since the binomial requires 3 multiplications, then computing the binomial requires the internal rule of the Booth multiplier to fire $3n$ times where n is the bit-width of the multiplication. If the verification of the binomial computation needs to be unrolled enough to see a computation from beginning to end (which is often true for modules like this), then the unrolling must be at least $3n$ steps just to cover the execution of the multiplier module. If this computation is using 32-bit numbers, then that means the unrolling must be at least 96 steps long. With this depth and complexity, it is not feasible to verify the binomial module as-is.

The binomial module can still be verified by leveraging modular verification. If we have verified the Booth multiplier implements its specification, then we can use the specification in place of the Booth multiplier when verifying the binomial module. That cuts out 96 steps of unrolling for 32-bit values, and it significantly simplifies the expressions used in the verification. Instead of the multiplier returning an expression made up of many additions, subtractions, and shifts, the multiplier specification just returns an expression using the $*$ operator. For the binomial module, modular verification takes a problem that is infeasable to verify with SMT-based model checking and reduces it to a trivial problem that can be verified in under a second.

Modular verification exists for many other domains including RTL, but RTL-based mod-

ular verification does not support reducing the number of clock-cycles required for model checking. For example, consider an RTL implementation of the binomial module that uses a multiplier that takes 32 clock cycles. This multiplier can be replaced by any equivalent 32-cycle multiplier without changes to the verification, but if the multiplier is replaced by a single-cycle multiplier, then the binomial module may break because it may expect the multiplier to take exactly 32 clock cycles. On the other hand, if the binomial module is correct with the single-cycle multiplier, there is no guarantee the module is still correct with the 32-cycle multiplier.

1.2 Contributions

In this thesis, we show how SMT-based verification of a rule-based hardware design language can efficiently verify designs in a modular manner by leveraging multiple opportunities for refinement and abstraction that simplify the model-checking problem. This includes substituting modules with their specifications, abstracting modules either fully or partially, abstracting functions and types that appear in the source language (*i.e.* term abstraction [30]), and abstracting function and type parameters to verify modules for all possible parameter values.

The primary contributions of this work are:

- Modular refinement and abstraction for SMT-based model checking that enables modular verification which can reducing the depth required for unbounded model checking.

- The unambiguous one-method-at-a-time semantics that makes modular refinement and abstraction useful for modular model checking, and makes specifications easier to write

- Symbolic representation of a rule-based HDL that is consistent with the dynamic semantics

The benefits of the contributions are shown in various examples of verification that are either harder or impossible using traditional SMT-based model checking techniques on RTL.

1.3 Outline

The outline of the dissertation is as follows.

Chapter 2 presents the background information necessary for the entire dissertation. This includes background information on SMT-based verification on RTL, rule-based HDLs, and verification of rule-based HDLs.

Chapter 3 introduces examples of the verification we want to be able to do using simple functional units as representative modules. This includes the notion of verifying a module for all executions symbolically, verifying a module against a specification, and performing modular refinement.

Chapter 4 introduces examples of verification on processors. The complexity of programmable machines makes the verification much more involved than the verification of functional units. This chapter introduces some of the modular verification techniques we want to be able to perform to simplify the verification into a more manageable problem.

Chapter 5 introduces our rule-based HDL Spec 'n' Check, including how it differs from BSV and why these differences are important. This chapter describes both the syntax and semantics of the language to provide a formal foundation for later modular verification and SMT-based model-checking work. The main differences in Spec 'n' Check from other rule-based HDLs are its unambiguous one-method-at-a-time semantics, unguarded expressions, and abstract state elements.

Chapter 6 presents our approach to modular verification. Since modules can only be accessed through their interface methods, and Spec 'n' Check has unambiguous one-method-at-a-time semantics, if all sequences of possible method calls for two modules match, then any module using one of the two modules as a submodule cannot tell it apart from the other module.

Our approach to modular verification takes advantage of this fact and introduces an implements relation between modules that says a module implements a specification if there exists a simulation relation between states of the module and states of the specification such that all method calls in the module can be simulated by a number of rules and the same method call in the specification.

This chapter also introduces a modular verification theorem that allows specifications to be used in place of implementations when verifying certain assertions in a module.

Chapter 7 introduces the language of assertions added to Spec 'n' Check. The language of assertions includes differentiation between simple assertions and extended assertions depending on what the assertions can check, and differentiation between state and step assertions depending on when the assertions are checked.

The assertions are intended to only check rule-level properties. That means properties that depend on how the rule-level design is implemented in RTL are not expressible in the assertions. For example, we can assert that certain states are not reachable, because that is a rule-level property, but we cannot assert that rules will fire in consecutive clock cycles, because that depends on how the rules are mapped to RTL[1].

The main benefit of the assertion language is that if an assertion does not depend on a specific submodule's internal implementation or method guards, then the assertion can be verified using modular verification where the submodule is replaced with its specification.

Chapter 8 introduces the symbolic representations for modules implemented in Spec 'n' Check; these symbolic representations are referred to as the symbolic semantics of Spec 'n' Check. These symbolic semantics are constructed in such a way that they are equivalent to the semantics in Chapter 5, and they can be used in SMT solvers.

Chapter 9 introduces the SMT-based model-checking formulations used in Spec 'n' Check to verify assertions within a module or verify that a module implements a specification. This is done by constructing a symbolic representation of a sequence of steps of the module to verify and then running the SMT solver to find executions that satisfy given constraints. This chapter gives the constraints necessary for various verification routines including both bounded and unbounded model checking.

Chapter 10 presents our prototype implementation of our rule-based design language, ProtoRMC, and the necessary tools for constructing the symbolic semantics for the module and performing model checking. Our prototype implementation supports everything presented in the prior chapters of the thesis and more, including performance optimizations

[1]It would be useful to be able to express these properties in the rule-based language, but these assertions would be checked only in the RTL.

and SMT-based visualization techniques.

A diagram of the proposed flow is shown in Figure 1-2. ProtoRMC can be used on the module implementations and specifications to perform bounded model checking, k-induction (unbounded model checking), and visualization. Once the design has been verified, implementations must be chosen for the abstract state elements in the design (`mkReg` or `mkEHR`/`mkCReg`), and then it can be fed to the Bluespec compiler to produce synthesizable Verilog.

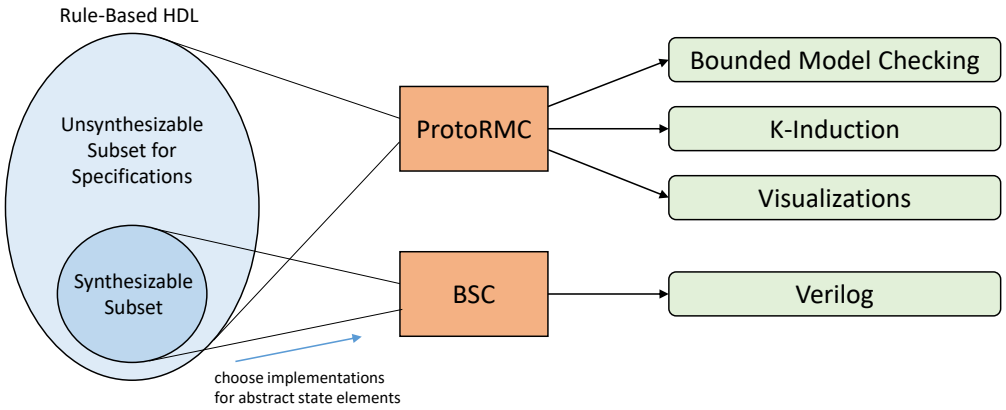

Figure 1-2: Overview of verification flow

Chapter 2

Background and Related Work

Since this work involves modular SMT-based verification of rule-based hardware designs, it is important to introduce existing SMT-based verification of hardware designs, existing verification of rule-based hardware designs, and modularity of rule-based HDLs.

2.1 SMT-Based Verification of Hardware Designs

Satisfiability Modulo Theories (SMT) solvers and satisfiability (SAT) solvers have been used in multiple ways to verify hardware designs, including equivalence checking, bounded model checking, and k-induction.

2.1.1 Equivalence Checking

During ASIC development, there are many processes that make changes to the RTL that should not change the overall behavior of the module. For example, register retiming effectively moves logic from one side of registers to the other [61]. It is essential to make sure these processes do not change the external behavior of the module.

Formal equivalence checking is the process of formally verifying two representations of a design have the same behavior [84], *e.g.* verifying a module after register retiming is equivalent to the same module before the retiming. Equivalence checking can be done in multiple ways, including using SMT and SAT solvers [48, 57]

2.1.2 Bounded Model Checking

Bounded model checking (BMC) [21, 36, 74], when applied to hardware, is a method of exhaustively searching for an assertion violation in all possible executions that start at reset (or some other known state) and cover fewer clock cycles than some given bound n. BMC was originally formulated using SAT. Currently, one of the most efficient ways is to formulate it at the term-level as an SMT problem and then use a solver to find if a violated assertion is reachable [22, 39].

BMC is implemented by constructing a symbolic next-state function for the circuit that depends on the current state and the inputs. An execution of depth n can be constructed by chaining n applications of the next-state function with fresh variables for the inputs in each clock cycle. Since this process exposes each state in the depth-n execution, it is possible to construct a symbolic expression corresponding to the assertions holding for each step.

A depiction of an unrolled circuit for bounded model checking can be seen in Figure 2-1. For each state S_i there is an input I_i to compute the next state S_{i+1}. There is also a predicate P_i for each state S_i, which is the predicate that all the assertions hold for the corresponding state.

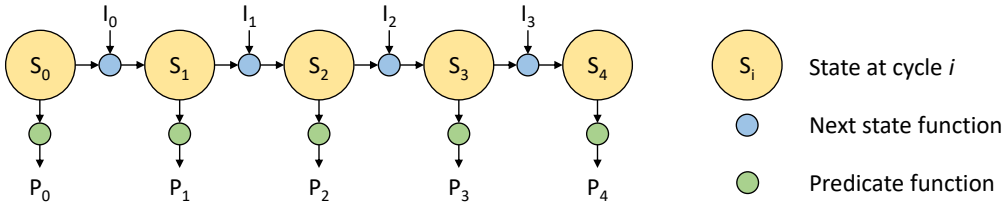

Figure 2-1: Depiction of RTL bounded model checking

The verification is run by feeding the SMT solver the constraints that the initial state is a legal reset state and that one of the states violate an assertion. If the SMT solver returns `sat`, then it found an assignment for the initial state and the inputs that causes the assertion to be violated. If the SMT solver returns `unsat`, then there is no possible way to violate the assertion in the first n clock cycles of running the module.

Performing bounded model checking on a design requires preparing the design to be checked for all possible inputs, not just legal inputs. This can be done in two ways: either write assertions that specify expected behavior for all possible inputs or write assertions that

should hold if the module is used legally and add assume statements that declare the inputs should be legal according to some specified condition.

For example, consider performing BMC on a two-element FIFO to make sure that each time an element is enqueued into the FIFO, the number of elements in the FIFO increases by one. This assertion makes sense, but model checking this will reveal a three-clock-cycle counterexample; if the FIFO is enqueued into three times in a row, the number of elements will not increase from two to three for the last enqueue because the FIFO is already full after two enqueues. This can be handled by either expanding the assertion to include what should happen when a full FIFO is enqueued into or adding an assumption that the FIFO is only enqueued into if the FIFO is not full.

When assume statements are used in BMC, they are treated as constraints when verifying the module in isolation, but when the module is used as a submodule in an outer module, the assumptions are treated as assertions to make sure the outer module is using the FIFO legally. This process is known as compositional reasoning [39].

2.1.3 K-Induction

SMT-based model checking can be performed for all reachable states by using k-induction [39, 44]. K-induction is the addition of an induction step to bounded model checking that tries to answer the question "if the assertion holds for k consecutive steps, will it hold for the next step as well?" If the base case (bounded model checking) and the induction case both hold, then the assertion holds for all reachable states.

The induction step uses the same unrolling as shown in Figure 2-1, but state S_0 is not assumed to be a reset state. The only constraint sent to the SMT solver is that the assertions hold for the first k steps but the assertion does not hold for step $k+1$.

In practice, the induction step is very likely to find false counterexamples when trying to prove an assertion. This is due to having non-reachable states that do not violate any assertions but lead to a violation in k or fewer cycles. When this happens, it is necessary to either increase the value of k or strengthen the assertions. Increasing the value of k can be very useful when it is small, but when k is large enough to capture most interesting behaviors in the module being verified, it is likely necessary to strengthen the assertion.

The assertion is strengthened by adding invariants that hold for all reachable states but do not hold for at least one of the states in the k-induction counterexample. This process of assertion strengthening is repeated until the base case fails or the k-induction is successful.

One of the biggest limitations for k-induction on RTL is that increasing k does not improve the performance of k-induction if there exist inputs that cause the state of the module to not change. If such inputs exist, then they can be use to increase k-induction counterexamples to $(k+1)$-induction counterexamples. This means k-induction must be combined with path compression [44] or something similar to avoid such sequences and get the most out of k-induction for RTL.

2.1.4 Abstraction

RTL model checking is severely limited by the state space explosion problem and by computationally intensive circuits. One way to combat against these problem is to use abstraction. Datapath abstraction limits the number of states in the datapath of a circuit by reducing or simplifying the type of data signals [54]. Function abstraction simplifies computationally intensive circuits by replacing them with uninterpreted functions [30, 29]. When first introduced, these abstractions were done manually, but there is research to perform automatic datapath abstraction [54] and automatic function abstraction [30, 29]. There is also work to use the syntax of the source language to determine good candidates for function abstraction [47].

2.2 Rule-Based Hardware Description Languages

Bluespec SystemVerilog (BSV) [69, 2] is the most popular rule-based hardware description language. There are also two BSV-inspired languages designed as part of research projects about proof-based formal verification of rule-based hardware designs: Kami [33] and Kôika [27]. We will present BSV as the canonical rule-based hardware description language. Kami and Kôika are very similar to BSV but have differences that are essential to their respective applications. The pertinent differences between these languages and BSV will be discussed later.

2.2.1 Bluespec SystemVerilog

Bluespec SystemVerilog [69, 2] is a high-level hardware design language (HDL) that is designed to make hardware design easier through its use of interfaces with guarded methods, its guarded atomic actions, or *rules*, and its rich expression language including parameterization and types.

Modules in BSV can only be interacted with through interface methods. In this sense, a module in BSV is similar to an object in an object-oriented software programming language like C++. Each method of a module contains a guard, *i.e.* an expression that says when the method can be called. When the guard of a method evaluates to true, we say that method is *ready* to fire, and if it evaluates to false, we say that method is *not ready*. For example, the dequeue method of a FIFO has a guard that says it is not ready when the FIFO is empty; likewise the enqueue method is not ready when the FIFO is full. Since the guards are explicitly declared by the programmer, the Bluespec compiler is able to ensure the circuits it produces never try to call methods that are not ready. On the other hand, in Verilog, designers typically have conventions about when signals can be enabled, and these conventions either are written in a natural-language specification, are included as assume statements within the Verilog code itself, or are not documented anywhere. In any case, it is necessary for users of the module to know the conventions expected by the module.

Rules are guarded atomic actions that describe state transitions that can happen within a module. Rules may call multiple methods, and if any of the methods are not ready, the entire rule is not ready and will not fire. Multiple rules can be fired within the same clock cycle as long as they all appear to happen in a serialized manner; this is known as *one-rule-at-a-time semantics*. The BSV language describes a non-deterministic execution model where any ready rule can fire at any time, but when the Bluespec compiler constructs hardware, it chooses a deterministic schedule that, when given which rules are ready, determines which rules fire.

BSV provides a lot of features over Verilog that improve the experience of writing the logic within each rule and method. Let's look at these features through an example of designing a processor. As a first step, custom types can be defined to make it easier to work with common

values and groups of values within the processor. For example, custom types can represent fetched instructions, decoded instructions, and executed instructions. Next functions can be defined to represent large chunks of combinational logic such as a decoder function that takes in a fetched instruction and returns a decoded instruction. Finally the processor can take in parameters, including type, function, and value parameters, so the single processor design in BSV can represent a range of implementations in RTL. As an extreme case of parameterization, the processor's data types and functions can be parameterized so the processor can implement a wide range of ISAs, and the specific ISA implemented determines the types and functions used as parameter values when instantiating the processor.

BSV also provides advantages in verification thanks to its high-level rule-based abstraction. Expressing a design using rules prevents certain low-level logic bugs from existing in the design, and it presents the verification problem at a higher abstraction level instead. To understand how this works, consider software as an example. When writing a function in C, register management is handled automatically by the compiler, so verification of a C function can ignore low-level aspects of saving and restoring registers during a function call. If the code was written in an assembly language instead, then part of the verification would include making sure registers are saved and restored properly according to the ISA's ABI.

Bluespec Compiler

The Bluespec compiler (BSC) [2] is a recently open-sourced compiler for BSV that takes in a BSV design and produces either a C++ simulation executable or a tree-like hierarchy of Verilog modules. The Verilog is created from the rule-based description of hardware by constructing a scheduler circuit that says which rules are fired when given a set of ready rules. It is important that this scheduler ensures that each time rules are fired together, the rules can be serialized so that one-rule-at-a-time semantics are preserved. BSC constructs the scheduler circuit by using the Esposito scheduling algorithm [42].

When a BSV module is compiled to Verilog, it must be free from nearly all types of parameterization. The only parameters that can be included in compiled Verilog are simple bit-value parameters that cannot affect the process of constructing submodules, *e.g.* the type or number of instantiated submodules cannot depend on the parameter.

2.3 Verification of Rule-Based Hardware Designs

There are many existing ways to verify rule-based hardware designs. The most common method of verification remains compilation to Verilog and then performing RTL verification; this includes simulation, random testing, RTL model checking [52], and more. There are a few more sophisticated techniques that take advantage of aspects of BSV that are not part of traditional RTL methods that we look at below.

2.3.1 Proof-Based Verification

Kami and Kôika are rule-based HDLs designed for proof-based formal verification of rule-based designs.

Kami [33, 82, 83] is a Coq library designed for proof-based verification of hardware expressed in a rule-based HDL like Bluespec. Kami uses labeled transition systems to represent modules for verification, and lists of labeled transitions are used to represent the behavior of modules. Modules are compared against their specifications based on the legal lists of labeled transitions.

In Kami, modules are not arranged in a hierarchy, so a module can call methods of any other module. As a result, modules in Kami have to worry about a rule in one module calling a method in the same module indirectly through a method in another module. As a result, Kami's refinement theorem has to consider the possibility of a module calling its own rules, even if it does not make sense semantically.

Kami has been used to prove the correctness of multiple designs including a modular cache-coherency hierarchy [83, 82] and a simple pipelined RISC-V processor [33].

Kôika [27] is a rule-based HDL embedded in Coq that is designed to give dynamic (*i.e.* concurrent rule execution) semantics of a core of the Bluespec language. Kôika introduces a novel way to write explicit schedules, and the generated hardware checks scheduling conflicts dynamically so many patterns that cause false conflicts in BSV can fire concurrently without issue in Kôika. Kôika can be used to prove theorems about modules similarly to in Kami, but Kôika can also prove timing behaviors of modules such as that the time taken to

execute a module does not depend on the data sent to the module.

There are two other proof-based approaches to verifying BSV. There has been work to write proofs about BSV by translating BSV into PVS [72, 67], but its use seems to be limited to small modules. Also there are multiple examples of hand-written proofs to verify hardware represented as a term rewriting systems, a predecessor to BSV [13, 75].

2.3.2 Random Testing

When a module has assertions that are expected to hold for all possible inputs, then it is possible to perform randomized testing to try to find inputs to violate assertions [50]. BlueCheck [68, 1] is a synthesizable BSV library inspired by Haskell's QuickCheck [35] to perform random testing. BlueCheck constructs random number generators for each argument of each called method, and it constructs random number generators to control when methods are enabled. When BlueCheck finds a bug, BlueCheck can repeatedly run variations on the trace that triggers the bug to try to find a shorter trace for easier debugging. BlueCheck can also be configured to detect deadlock and minimize traces that result in deadlock for easier debugging.

2.3.3 Model Cheking

There have been two known attempts of translating BSV to a language for model checking [76, 77]. These provide the ability of checking designs at the rule level, but they require translating to a different language, so the results are only as good as the translation. The work [77] presents a notion of refinement, but it is defined using subsequences of states, so it is limited to refinements that include the same state variables.

2.3.4 Rule Refinement Verification

There is also research to formally verify rule refinements in BSV using an SMT solver [41]. This paper has formalizations for Bluespec programs and their equality, but their formalism does not include methods, so its usefulness is limited to closed systems.

They detect a refinement is legal by building up traces of rules in the refinement until the trace is equivalent to a trace in the original system. They show their methodology using a 4-stage pipelined processor with a merged fetch/decode stage, and their target refinement is splitting that stage into separate fetch and decode stages. They claim their methodology can be used to prove a single-cycle processor can be refined into a 5-stage pipelined processor.

2.4 Modularity of Rule-Based Hardware Designs

In order to perform modular verification, it is important to have modular semantics that behave well with modular verification. Below we look at the modularity of BSV and Kami (note that Kôika does not currently have modularity).

2.4.1 Modularity of BSV

The modularity provided by BSV has been shown to be a very powerful development tool when designing large complex digital systems like out-of-order-execution processors [86, 87]. Unfortunately there are aspects of the modularity of BSV that are not ideal for modular verification. The two main issues are the potential for non-atomic method calls and ambiguous method orderings

Non-Atomic Method Calls

Non-atomic method calls are possible when modules are not compiled separately. In BSV, the compiler attribute `(* synthesize *)` is added before modules to mark that the module should be separately compiled; this is known as a *synthesize boundary*. A BSV module with a synthesize boundary is compiled into a separate Verilog module during compilation, but modules without synthesize boundaries are effectively inlined into their parents during compilation. As a result, the semantics of a module with a synthesize boundary differs from the semantics of a module without a synthesis boundary.

As an example, consider the following module:

```
1 module mkRegSwapper(RegSwapper);
2     Reg#(Bit#(1)) x <- mkReg(0);
```

```
    Reg#(Bit#(1)) y <- mkReg(1);
    method Action x_gets_y();
        x <= y;
    endmethod
    method Action y_gets_x();
        y <= x;
    endmethod
endmodule
```

If the methods `x_gets_y` and `y_gets_x` are called from *separate* rules, those rules cannot fire in the same clock cycle because the parallel composition of the two methods does not match either sequential ordering of the methods (the parallel composition of the two methods looks like the registers `x` and `y` swap values). But on the other hand, if `x_gets_y` and `y_gets_x` are called from the *same* rule and `mkRegSwapper` does not have a synthesize boundary, the rule is allowed to fire, and the effect of calling the two methods is the non-atomic parallel composition of the two methods, *i.e.* registers `x` and `y` swap values.

Therefore without synthesize boundaries, methods do not appear to fire atomically when multiple methods of the same submodule are called from the same rule.

On the other hand, if `mkRegSwapper` has a synthesize boundary, then the process of separate compilation marks `x_gets_y` and `y_gets_x` as conflicting methods because their parallel composition does not match either sequential ordering of the methods. As a result, any rule that tries to call both methods is illegal.

Ambiguous Method Ordering

When multiple methods of the same module are called from the same rule, and they appear to fire atomically in some sequential order, the sequential order is ambiguous and depends on the implementation of the submodule. For example, consider the module below:

```
module mkFIFOUser(FIFOUser);
    FIFO#(Bit#(1)) fifo <- mkFIFO;
    // omitted state
    rule useFIFO;
        // omitted code
        fifo.enq(1);
        if (p) begin
            fifo.clear();
```

```
9           end
10      end rule
11      // omitted rules and methods
12  endmodule
```

This module has a `useFIFO` rule that, when it fires, always enqueues into `fifo` and, depending on the value of `p`, may clear `fifo` as well. In a typical sequential programming language, the ordering of the calls to `enq` and `clear` determines the ordering in which they appear to execute. BSV is not a sequential language; instead all the statements in a rule are viewed as firing in parallel. Assuming atomic method calls (*e.g.* `mkFIFO` has a synthesize boundary), after `useFIFO` fires with `p` being true, the final state of `fifo` is either an empty FIFO (`enq` happens first) or a FIFO containing 1 as its only value (`clear` happens first).

That means the syntax does not impose which apparent ordering the `enq` and `clear` methods should have. The apparent ordering between `enq` and `clear` depends entirely on the implementation of `mkFIFO`. Therefore a one-method-at-a-time description of the behavior of `mkFIFO` is not sufficient for determining what `mkFIFOUser` does; it is also necessary to have information about how multiple methods behave when fired concurrently.

Implication on Specifications

As a result of non-atomic method calls and ambiguous method orderings, in order to specify the behavior of a module in BSV, it is necessary to describe how methods interact with each other when called together. This results in specifications that are significantly more complicated than specifications that just say what each method does individually.

2.4.2 Modularity of Kami

Kami has a module system that is similar to BSV without synthesize boundaries, but the semantics of the language present the modularity in a very different manner. Kami's modularity is set up so it is easy to perform modular refinement and composition; these are very important properties for proof-based verification.

Modules in Kami are all at the same hierarchy level semantically. Modules can call any other module's methods, and in turn, a module can have its own methods called by methods

it calls. This introduces many opportunities for multiple methods to be called on the same module at the same time.

Kami's method-call semantics are effectively inlining just like BSV modules without synthesize boundaries. Therefore concurrent method calls can result in non-atomic behaviors. This presents the same complications for specifications as mentioned for BSV.

In order to show a module implements a specification in Kami, it is necessary to show that all possible sequences of method calls on the module are legal sequences of method calls on the specification. That includes all possible concurrent method calls as well. Therefore, in order for a specification to be valid for an implementation, it must have the same concurrent behavior for all method calls as the implementation has.

2.5 Additional Related Work

Most of the related work for this thesis is integrated with the background presented previously in this chapter. There are a few topics of additional related work that are do not fit as background and are presented here.

2.5.1 High-Level Modeling Languages

There are a number of modeling languages that support high-level abstractions and formal methods including SMT-based verification. These languages are sometimes used for verifying models of hardware, but these models are hand-translated and typically omit some details of the design. While such models do not give formal guarantees of the original design, they give the designer confidence that their approach to designing the module can be correct.

There are multiple examples of using uninterpreted functions in a custom modeling language or representation to model and verify processors. Burch [32] used uninterpreted functions with equality to prove the correctness of an in-order pipelined processor. The use of uninterpreted functions makes ISA-level details parameterized, including the number of registers and the datapath width. The logic used is the quantifier-free logic of uninterpreted functions and predicates with equality and propositional connectives. The verified processor is a pipelined implementation of a subset of DLX (simplified MIPS-like ISA).

Berezin [19] combined symbolic model checking with uninterpreted functions to verify an out-of-order processor using Tomasulo's algorithm. The verified processor is a very simple implementation of Tomasulo's algorithm without branches or memory instructions. Instruction fetch is not PC-based; instead instructions are fed to the processor from the outside world. The verification methodology takes an in-order specification and feeds instructions to the in-order and out-of-order processors at the same time. The equivalence between the two processors is checked by flushing the out-of-order processor's pipeline.

The UCLID5 language [73] is presented as a high-level modeling and verification language for heterogeneous systems with hardware and software, so it supports bot a program-like sequential execution model and an RTL-like concurrent execution model. UCLID5 is based on the CLU logic [11] which includes unbounded counters, lambda expressions, and uninterpreted functions. UCLID5 supports modular modeling an verification, but module composition is along the lines of composing finite state machines. UCLID5 can be used to model processors [31], but does not have a way to produce Verilog/RTL from such processor models. UCLID5 could be used to model rule-level hardware, but it would require a designer to conform to the abstraction manually, or it would require a tool to convert rule-level designs to the abstraction of UCLID5.

Other formal modeling and verification languages include Ivy [70, 66, 4], TLA$^+$ [59] and Alloy [56]. These languages have different targets, but have the same limitation of UCLID5 when it comes to verifying rule-level hardware. These languages either do not have support for the execution model or the interfaces of rule-level hardware, and they do not support generation of Verilog/RTL. Therefore using these languages for rule-level hardware verification would require manually conforming to the abstraction or a significant amount of additional tooling.

One interesting thing about IVy is the focus on interactive verification rather than fully-automatic verification like the other tools. The authors of Ivy recognize that most efforts towards verifying real systems use little proof automation (using [51] as an example), so they rely on users to strengthen invariants when proofs fail. Ivy has been used to prove hardware protocols such as TileLink [63].

2.5.2 Alternate Approaches to Unbounded Model Checking

There are multiple approaches to unbounded model checking. We use an approach that relies on bounded model checking and k-induction, and if the model checking results in inconclusive results, we depend on the user to strengthen the invariants manually when provided with a k-induction counterexample, similar to Ivy.

There are also approaches for unbounded model checking that strengthen invariants automatically. One of the most recent major advancements in this area is the development of IC3 [28], also known as Property Directed Reachability (PDR) [46]. This algorithm incrementally attempts to produce an inductive invariant. IC3/PDR is a bit-level algorithm, but more recent work extends this to word level [53, 23, 60, 34]. Another related algorithm is kAvy which combines PDR with k-induction to get the benefits of each [58].

2.5.3 Modular Verification in Model Checking

A generic method for modular verification in model checking is compositional reasoning, *i.e.* assume-guarantee reasoning [65, 39]. This approach allows two modules to be checked independently of each other by using assumptions in place of the other module definition. If each module satisfies the assumptions of the other module, then properties checked in isolation hold for the complete module. Compositional model checking has been shown to be effective for proving properties about processors [62, 64].

A different approach to modular verification of processors is Instruction-Level Abstraction (ILA) [55, 79, 80]. Instead of dividing the design into smaller pieces, the verification is divided into smaller pieces by instruction.

Another way of doing modular verification with model checking is seen in model checking of process algebra [39]. Process algebra supports modular refinement during verification provided processes are equivalent or the processes are related by the refinement relation. Modular refinement can be used during model checking a large system to replace a process by another one with fewer states in order to simplify the verification. This approach is the most similar to the modular verification we do in this thesis.

Chapter 3

Verifying Functional Units

This chapter explores how a simple module such as one that computes $f(f(f(x)))$ can be implemented in different ways, and how those implementations can and cannot be substituted for one another.

3.1 Functional Unit Interface

The `FuncUnit#(inT,outT)` interface, as shown in Figure 3-1, is a common interface for hardware modules that compute a function. In a typical implementation, the `start` method is called with a given argument to start the computation for that value. Eventually the `getResult` method becomes ready, and it can be called to get the result of the computation. This interface is parameterized by its input type `inT` and output type `outT`.

```
1 // interface definition
2 interface FuncUnit#(type inT, type outT);
3     method Action start(inT v);
4     method ActionValue#(outT) getResult();
5 endinterface
```

Figure 3-1: Functional unit interface

This interface can represent a wide range of modules in terms of function and complexity. Thanks to the parameterized input and output types, this interface can represent functions that take in and return multiple values collected together in a tuple, structure, or vector.

Therefore this interface could represent anything from a module that increments an 8-bit value to a pipelined FFT function that takes in and returns a vector of complex floating-point numbers.

In this chapter, we are going to present multiple modules with this interface that implement the function $f(f(f(x)))$, how these implementations can be verified, and which implementations can be used as substitutes for other implementations. We start with a simple implementation, implement different microarchitectural refinements, and explore how they affect performance and verification.

3.2 Baseline $f(f(f(x)))$ Implementation

Figure 3-2 shows a simple implementation of $f(f(f(x)))$ for an arbitrary pure function f passed in as a parameter. Note that despite the use of BSV-like syntax, this is a slight variation that introduces some concepts of Spec 'n' Check that are formalized later including abstract state elements (`mkState`) and the reset predicate.

This module uses the `FuncUnit` interface where both the input and output types are the type parameter `t`. The first line of the module declares the name of the module to be `mkThreeF`, declares a parameter function of type `t` $\rightarrow$ `t` called `f`, and declares the interface to be `FuncUnit#(t,t)`.

The body of the module definition starts with state-variable declarations. This module has two internal state variables: `x` and `count`. The `x` variable holds the input, intermediate computations, and the output, while the counter variable `count` keeps track of the state of the computation. These are abstract state variables, so when this module is compiled to RTL, `x` and `count` can be implemented either as registers or as ephemeral history registers (EHRs), depending on the schedule of the rules and methods in this module. Also, if the module is not being compiled to RTL, *e.g.* it only exists for verification purposes, the type `t` can be an infinite, non-synthesizable type such as unbounded integers.

Below the state variables is the reset predicate, *i.e.* the predicate that determines what is a legal reset state. In this module, at reset the `count` variable must be set to zero. There are no reset constraints for the `x` variable because when `count` is zero, the value of `x` does

```
// module definition
module mkThreeF#(Function#(t,t) f)(FuncUnit#(t,t));
    // state variables
    State#(t)       x     <- mkState;
    State#(Bit#(3)) count <- mkState;
    // reset predicate
    reset(count == 0);
    // rules
    rule applyF(count > 0 && count < 4);
        x <= f(x);
        count <= count + 1;
    endrule
    // methods
    method Action start(t v) if (count == 0);
        x <= v;
        count <= 1;
    endmethod
    method ActionValue#(t) getResult() if (count == 4);
        count <= 0;
        return x;
    endmethod
endmodule
```

Figure 3-2: `mkThreeF` module that computes $f(f(f(x)))$

not matter.

Next is the `applyF` rule. Once `x` has an input value written to it, this rule fires three times to compute `f(f(f(x)))`. This rule can only fire when `count` is greater than zero and less than four.

The method definitions are at the bottom of the module's body. The `start` method is used to provide input to `mkThreeF` and can only be called when the guard is true, *i.e.* `count` is zero. This method writes the input to `x` and sets `count` to one so the `applyF` rule can fire. The `getResult` method is used to get the output of the module by returning the value of `x`. This method also sets `count` to zero so that `start` can be called and the module can compute another result.

3.2.1 Testbench for `mkThreeF`

We want to verify the correctness of `mkThreeF`. If we simulate a set of test vectors to verify `mkThreeF`, the module is only shown to be correct for the specific function f and the specific sequence of test vectors used during simulation. Instead we want to verify this module for all functions f and all sequences of inputs. The first step towards doing this is constructing the dynamic testbench shown in Figure 3-3.

```
module mkThreeFTestbench#(Function#(t,t) f)(FuncUnit#(t,t));
    // device under test
    FuncUnit#(t,t) dut <- mkThreeF(f);
    // verification state
    State#(t) expected_result <- mkState;
    // methods
    method Action start(t v);
        dut.start(v);
        expected_result <= f(f(f(v)));
    endmethod
    method ActionValue#(t) getResult();
        let result <- dut.getResult();
        assert(result == expected_result);
        return result;
    endmethod
endmodule
```

Figure 3-3: Testbench for `mkThreeF`

This testbench takes in a function parameter `f` and instantiates an instance of `mkThreeF` with `f` as its function parameter as the device under test, or `dut`. The `expected_result` state holds the expected result for each use of the `dut` module. The value of `expected_result` is computed when `start` is called by computing `f(f(f(v)))` where `v` is the argument of the `start` method. The value of `expected_result` is compared against the result of the `dut`'s `getResult` method with an assertion.

This testbench exposes the same interface as `mkThreeF`, and the guards of the testbench's methods match the guards of `mkThreeF`'s methods as well. That means this testbench allows `mkThreeF` to be randomly tested by randomly calling the `start` and `getResult` methods of the testbench (with random arguments for `start`) when they are ready. If the assertion in `getResult` is never violated during a random execution, then `mkThreeF` produced the correct results for the random method calls.

3.2.2 Checking Testbench Assertions Statically for `mkThreeF`

Instead of verifying assertions dynamically in simulation, we want to verify assertions statically to ensure they hold for all possible parameters and for all possible sequences of method calls and rule firings. One way to do this is using symbolic analysis, *i.e.* executing the module using symbolic variables instead of concrete values. The general idea is that if the function `f` is treated as a symbolic function and if `start` is called with the symbolic variable `a`, then if `getResult` returns the symbolic expression `f(f(f(a)))`, `mkThreeF` produces the correct results for all `f` and `a`.

To start the verification, we will assume `mkThreeFTestbench` is in a reset state and `start` is called with symbolic value `a`. After `start` executes, the values of `x` and `count` within the `dut` are `a` and `1` respectively, and the value of `expected_result` in the testbench is `f(f(f(a)))`. Since `count` is `1`, the only ready rule or method is `applyF`. After executing `applyF`, `x` and `count` are `f(a)` and `2` respectively. Repeating this process two more times gives `x` and `count` the values `f(f(f(a)))` and `4`. At this point, the only method ready to call is `getResult`. Calling `getResult` returns `f(f(f(a)))` which matches the value in `expected_result`, so the assertion in `getResult` holds.

This verifies that when `mkThreeFTestbench` starts from reset, the assertion in `getResult`

always holds for the first computation. The next step is to show that this assertion always holds for all reachable states.

In more sophisticated examples, we would use a form of induction to show that if the assertion holds initially it will hold forever, but in this case, after executing `getResult`, `count` is equal to zero so therefore `mkTreeFTestbench` is back in a reset state. Since the assertion always holds for the first call to `getResult` after reset, and `getResult` always returns to a reset state, then the assertion in `mkThreeFTestbench` always holds. If the reset in the example was specified to be a unique state instead of a reset predicate, this analysis would not work.

3.3 `mkThreeF` With Fixpoint Optimization

Now that we have presented `mkThreeF` and statically verified it using symbolic analysis of `mkThreeFTestbench`, let us consider a new version of `mkThreeF` that contains an optimization where the module returns early if it reaches a fixpoint of f. This new module, `mkThreeF_opt` is shown in Figure 3-4

This module is identical to `mkThreeF` except for its rule `applyF_opt`. This new rule checks to see if `x` is a fixpoint of `f`, and if it is, then the rule sets `count` to `4` so that `getResult` can be called immediately. With this new rule, `mkThreeF` computes `f(f(f(x)))` by firing `applyF_opt` either once, twice, or three times instead of always firing `applyF` three times.

This new rule complicates the verification because now there is more than one possible sequence of rules to get to the assertion in the testbench's `getResult` method. Consider the symbolic analysis performed in Section 3.2.2. After `start` was fired with argument `a`, the symbolic values of `x` and `count` were `a` and `1` respectively. Now with `applyF_opt`, the next state depends on the condition `f(a) == a`. Since nothing is known about the implementation of `f` and the value of `a`, we have to consider both cases: `f(a) == a` and `f(a) != a`. After applying `applyF_opt` the value of `x` is `f(a) == a ? a : f(a)` and the value of count is `f(a) == a ? 4 : 2`. At this point, the guards of the rules and methods are not constant, meaning the next rule or method to fire is not known. Instead, the next rule or method depends on the value of `f(a) == a`; if true, then `getResult` is ready to fire,

```
module mkThreeF_opt#(Function#(t,t) f)(FuncUnit#(t,t));
    // state variables
    State#(t)       x     <- mkState;
    State#(Bit#(3)) count <- mkState;
    // reset predicate
    reset(count == 0);
    // rules
    rule applyF_opt(count > 0 && count < 4);
        let next_x = f(x);
        if (next_x == x) begin
            // x is a fixpoint of f
            count <= 4;
        end else begin
            x <= next_x;
            count <= count + 1;
        end
    endrule
    // methods
    method Action start(t v) if (count == 0);
        x <= v;
        count <= 1;
    endmethod
    method ActionValue#(t) getResult() if (count == 4);
        count <= 0;
        return x;
    endmethod
endmodule
```

Figure 3-4: `mkThreeF` variant containing fixpoint optimization

otherwise `applyF_opt` is ready to fire.

If we continue this symbolic analysis by firing `applyF_opt` as many times as possible until `getResult` is ready, we get the following symbolic state:

```
x = f(a) == a ? a :
    (f(f(a)) == f(a) ? f(a) :
    (f(f(f(a))) == f(f(a)) ? f(f(a)) : f(f(f(a))) ))
count = 4
```

It is possible to prove this `x` satisfies the assertion `x == f(f(f(a)))`, but it takes more effort than before since `x` does not naively simplify to `f(f(f(a)))`. Fortunately, tools like SMT solvers can easily verify assertions like this one.

3.3.1 Using `mkThreeF_opt` in Place of `mkThreeF`

When can we use `mkThreeF_opt` in place of `mkThreeF`? Both modules produce the same results (according to verification against `mkThreeFTestbench`), but `mkThreeF_opt` computes its result differently than `mkThreeF` does. Specifically, `mkThreeF_opt` can take a different number of rule firings to compute its result. Since internal rules are not observable to the outside world, modules using `mkThreeF` cannot directly detect if an instance of `mkThreeF` was swapped with an instance of `mkThreeF_opt` or vice-versa. Since an outer module cannot detect the difference between `mkThreeF` and `mkThreeF_opt`, it is impossible for an assertion in the outer module to be always true for one module but violated for the other.

On the other hand, if these two modules are compiled to RTL FSMs, then they may take different numbers of clock cycles to produce the result. Therefore, the two FSMs will not be equivalent, and the two modules cannot be substituted for each other without the risk of breaking the outer module.

3.4 Specification for `mkThreeF` and `mkThreeF_opt`

Up to this point, we have checked modules by making sure they do not violate an assertion in a testbench. In order to simplify the process of figuring out which modules can be substituted for other modules, we introduce the notion of a specification. A specification is a module

that describes the legal behaviors of other modules through its implementation. Informally, we say a module implements a specification if all the legal behaviors of the module's interface are legal behaviors of the specification's interface.

Specifications are not intended to be synthesized to RTL, so they can use non-synthesizable types, and they do not need to worry about critical paths or timing. Specifications are more useful when their implementations are as simple as possible, and one way to keep a specification simple is to not use any internal rules when writing the specification.

Specifications are useful for module substitution because if a module is verified with a specification in place of a specific submodule implementation, then the verification holds for all submodules which implement the specification[1].

As an example, Figure 3-5 shows a specification for modules with the `FuncUnit` interface that implements a pure function f. This specification is parameterized by the function it implements. If the function `f3(x)` is defined to be `f(f(f(x)))`, then `mkFuncUnitSpec(f3)` is a specification for `mkThreeF(f)` and `mkThreeF_opt(f)`. Therefore, if a module works using `mkFuncUnitSpec(f3)` as a submodule for some function `f`, then the module would work using `mkThreeF(f)` or `mkThreeF_opt(f)` in place of `mkFuncUnitSpec(f3)`.

3.5 Buffered `mkThreeF`

The module `mkThreeF` requires `getResult` to be called before `start` can be called to start computing the next value. We can add a buffer to `mkThreeF` to hold computed results so `start` can be called after the previous computation finished but before `getResult` was called. An implementation of this module using a two-element FIFO buffer can be seen as module `mkThreeFBuffered` in Figure 3-6. The implementation of the two-element FIFO buffer `mkFIFO2` is in Figure 3-7.

The two-element FIFO `mkFIFO2` has a `canonicalize` rule to match one way this module can be implemented in BSV in order to have conflict-free `enq` and `deq` methods. If this module were implemented without the canonicalize rule, then `enq` would either enqueue into position 1 or position 2 in the FIFO depending if there were already an element in position

[1] This is presented formally in Chapters 6 and 7.

```
module mkFuncUnitSpec#(Function#(inT,outT) f)
        (FuncUnit#(inT,outT));
    // state variables
    State#(outT) result <- mkState;
    State#(Bool) result_ready <- mkState;
    // reset predicate
    reset(!result_ready);
    // methods
    method Action start(inT v) if (!result_ready);
        result <= f(v);
        result_ready <= True;
    endmethod
    method ActionValue#(outT) getResult() if (result_ready);
        result_ready <= False;
        return result;
    endmethod
endmodule
```

Figure 3-5: Specification for pure functional units

1. That means both `enq` and `deq` could change the state of `v1`, introducing a scheduling constraint between the two methods in BSV.

The presence of the `canonicalize` rule allows `enq` to only touch position 2 and `deq` to only touch position 1, clearly making `enq` and `deq` conflict free. If `enq` is called and the FIFO is empty, then `canonicalize` moves the newly enqueued element from position 2 to the head of the FIFO at position 1.

In BSV, this FIFO would be implemented using EHRs so that canonicalize can always fire after the `enq` and `deq` methods. Since we are using abstract state elements, we can just rely on the one-rule-at-a-time semantics, but the same bypassing behavior can be implemented in the RTL provided the necessary abstract states are implemented as EHRs in RTL.

3.5.1 `mkThreeFBuffered` Verification

Unlike the modules `mkThreeF` and `mkThreeF_opt`, this module does not match the specification `mkFuncUnitSpec(f3)` because this module can handle more than one request at a time. Specifically, this module can have `start` called, then fire `applyF` three times, and then have `start` called again. This behavior is not legal for `mkFuncUnitSpec` because the guard

```
module mkThreeFBuffered#(Function#(inT,outT) f)
        (FuncUnit#(inT,outT));
    // state variables
    State#(t)        x     <- mkState;
    State#(Bit#(3)) count <- mkState;
    // reset predicate
    reset(count == 0);
    // rules
    rule applyF(count >= 1);
        let next_x = f(x);
        if (count == 3) begin
            out_fifo.enq(next_x);
            count <= 0;
        end else begin
            x <= next_x;
            count <= count + 1
        end
    endrule
    // methods
    method Action start(t v) if (count == 0);
        x <= v;
        count <= 1;
    endmethod
    method ActionValue#(t) getResult() if;
        out_fifo.deq();
        return out_fifo.first();
    endmethod
endmodule
```

Figure 3-6: Implementation of `mkThreeF` with a buffer

of `start` remains false until `getResult` is called. As a result, `mkThreeFBuffered` cannot be used in place of `mkFuncUnitSpec(f3)` or modules like `mkThreeF` and `mkThreeF_opt` without possibly introducing new buggy behaviors in the outer module. This is because the extra capacity in the buffer can break assertions in modules that depend on `start` not being ready if `getResult` has not been called yet for the previous start method. As an example, consider the module `mkThreeFTestbench`. In this testbench, calling `start` the second time overwrites the `expected_result` for the previous request, causing the assertion in `getResult` to fail.

Ideally, we would like a specification for `mkThreeFBuffered` that also counts as a specification for `mkThreeF` and `mkThreeF_opt`. The specification in Figure 3-8 can be used to

```
module mkFIFO2(FIFO#(t));
    // state variables
    State#(t)    d1 <- mkState;
    State#(t)    d2 <- mkState;
    State#(Bool) v1 <- mkState;
    State#(Bool) v2 <- mkState;
    // reset predicate
    reset(!v1 && !v2);
    // rules
    rule canonicalize(v2 && !v1);
        v1 <= v2;
        d1 <= d2;
        v2 <= False;
    endrule
    // methods
    method Action enq(t x) if (!v2);
        v2 <= True;
        d2 <= x;
    endmethod
    method Action deq() if (v1);
        v1 <= False;
    endmethod
    method t first() if (v1);
        return d1;
    endmethod
endmodule
```

Figure 3-7: Implementation of 2-element FIFO buffer

specify all those implementations.

The specification `mkFuncUnitConcurrent3Spec` specifies a `FuncUnit` interface that computes the pure function f but can have three overlapping requests at a time. In the `start` method, this specification computes the return value and enqueues it into the three-element FIFO `result_fifo` so that `start` can be called two more times before `getResult` would have to be called. That means this specification can support three outstanding requests at a time.

When the function passed into the specification is `f(f(f(x)))`, the module `mkFuncUnitConcurrent3Spec` is a specification for `mkThreeFBuffered`, `mkThreeF`, and `mkThreeF_opt`. Even though `mkThreeF` and `mkThreeF_opt` cannot support three overlap-

```
module mkFuncUnitConcurrent3Spec#(Function#(inT,outT) f)
        (FuncUnit#(inT,outT));
    // three-element FIFO
    FIFO#(outT) result_fifo <- mkFIFO3;
    // methods
    method Action start(inT v);
        result_fifo.enq(f(v));
    endmethod
    method ActionValue#(outT) getResult();
        let result = result_fifo.first();
        result_fifo.deq();
        return result;
    endmethod
endmodule
```

Figure 3-8: Spec for pure functional unit that accepts three overlapping requests

ping requests, `mkFuncUnitConcurrent3Spec` always returns the same values as `mkThreeF` and `mkThreeF_opt` when used in the same way, *i.e.* always alternating `start` and `getResult`. As a result, `mkThreeF` and `mkThreeF_opt` implement `mkFuncUnitConcurrent3Spec`, and any module that does not violate assertions when using `mkFuncUnitConcurrent3Spec` would also not violate assertions with `mkThreeF` or `mkThreeF_opt`.

3.6 General Functional Unit Specification with Concurrent Requests

The module `mkFuncUnitConcurrent3Spec` is the specification for a functional unit that can handle at most 3 concurrent requests at a time. But what if we want a specification that can handle all functional units that support any number of overlapping requests? That is equivalent to producing a buffer that can hold any number of elements in it and using that buffer in place of `mkFIFO3` in `mkFuncUnitConcurrent3Spec`. Let's call such a buffer `mkFIFOSpec`, since it would be a general specification for all possible FIFO buffers.

We can construct `mkFIFOSpec` using infinite-sized types. Figure 3-9 shows the implementation of `mkFIFOSpec`. Note that the reset predicate for this module is more sophisticated than previous reset predicate examples. Instead of just specifying a value for a subset of the

states, this reset predicate is a relation on the state saying any state where `enqP` and `deqP` are equal is a legal reset state.

We then make `mkFuncUnitConcurrentSpec` similarly to `mkFuncUnitConcurrent3Spec` except `mkFIFOSpec` is used as the FIFO buffer.

```
module mkFIFOSpec(FIFO#(t));
    State#(Integer) enqP <- mkState;
    State#(Integer) deqP <- mkState;
    State#(Array#(Integer, t)) data <- mkState;

    reset(enqP == deqP); // any empty state is a reset state

    method Action enq(t x);
        data[enqP] <= x;
        enqP <= enqP + 1;
    endmethod
    method Action deq() if (deqP < enqP);
        deqP <= deqP + 1;
    endmethod
    method t first() if (deqP < enqP);
        return data[deqP];
    endmethod
endmodule
```

Figure 3-9: Implementation of general FIFO specification

3.7 Pipelining `mkThreeF`

Instead of just adding a buffer to `mkThreeF`, we can make a pipelined implementation of $f(f(f(x)))$ that still uses the `FuncUnit` interface but can compute results with a higher throughput. Figure 3-10 shows a pipelined implementation of `mkThreeF`.

This module applies `f` in the `start` method and the `stage2` and `stage3` rules. The pipeline registers `s1`, `s2`, and `s3` keep track of the data between the stages, and the "valid" registers `s1_valid`, `s2_valid`, and `s3_valid` keep track of whether the value in the corresponding data register is valid or not.

Just like `mkThreeFBuffered`, `mkThreeFPipelined` matches the specification `mkThreeFConcurrent3Spec` and the general specification `mkThreeFConcurrentSpec`.

```
module mkThreeFPipelined#(Function#(t,t) f)(FuncUnit#(t,t));
    // state variables
    State#(t) s1 <- mkState;
    State#(t) s2 <- mkState;
    State#(t) s3 <- mkState;
    State#(Bool) s1_valid <- mkState;
    State#(Bool) s2_valid <- mkState;
    State#(Bool) s3_valid <- mkState;
    reset(!s1_valid && !s2_valid && !s3_valid);

    rule stage2(s1_valid && !s2_valid);
        s2 <= f(s1);
        s1_valid <= False;
        s2_valid <= True;
    endrule
    rule stage3(s2_valid && !s3_valid);
        s3 <= f(s2);
        s2_valid <= False;
        s3_valid <= True;
    endrule

    method Action start(t v) if (!s1_valid);
        // implements stage one of the pipeline too
        s1 <= f(v);
        s1_valid <= True;
    endmethod
    method ActionValue#(t) getResult() if (s3_valid);
        s3_valid <= False;
        return s3;
    endmethod
endmodule
```

Figure 3-10: Pipelined implementation of `mkThreeF`

```
module mkMultiFuncUnit
        // module definition parameter
        #(Module#(FuncUnit#(inT,outT)) mkFuncUnit)
        // interface
        (FuncUnit#(t,t));
    // construct a vector containing two mkFuncUnits
    Vector#(2, FuncUnit#(inT,outT)) units
                            <- replicateM(mkFuncUnit(f));
    // pointers for the unit to use for input and output
    State#(Bit#(1)) in_unit <- mkState();
    State#(Bit#(1)) out_unit <- mkState();
    // reset predicate
    reset(in_unit == out_unit);

    method Action start(inT x);
        units[in_unit].start(x);
        in_unit <= in_unit + 1;
    endmethod
    method ActionValue#(outT) getResult();
        let result <- units[out_unit].getResult();
        out_unit <= out_unit + 1;
        return result;
    endmethod
endmodule
```

Figure 3-11: `mkMultiFuncUnit` Module

3.8 Duplicating Functional Units for More Throughput

As the final module in this chapter, consider the module `mkMultiFuncUnit` shown in Figure 3-11. This module takes in a module constructor with interface `FuncUnit` and instantiates two implementations of that module to produce a single `FuncUnit` interface with (up to) double the throughput of a single module. This module alternates between the two submodules each time `start` and `getResult` are called so that the two submodules are used to compute alternating results, and the input ordering is preserved for the results.

This module can present a tradeoff between `mkThreeF` and `mkThreeFPipelined` when instantiated with `mkThreeF`. `mkThreeF` uses one instance of `f`, but it can only compute one result at a time. `mkThreeFPipelined` uses three instances of `f`, but it can compute

three results at a time. `mkMultiFuncUnit(mkThreeF)` uses two instances of `f` (one for each submodule), and it can compute two results at a time.

3.8.1 `mkMultiFuncUnit` Verification

Its not hard to show that when `mkMultiFuncUnit` is instantiated with `mkFuncUnitSpec(f)`, that it implements the specification `mkFuncUnitConcurrent2Spec(f)`[2]. As a result of this, any module `mkFoo` that implements `mkFuncUnitSpec(foo)` can be used to produce `mkMultiFuncUnit(mkFoo)` that implements `mkFuncUnitConcurrent2Spec(foo)`.

The module `mkMultiFuncUnit` does not implement `mkFuncUnitConcurrent2Spec` for all possible module parameters. For example, consider a module that does not implement a pure function such as a module that has an internal accumulator, and each time start is called, the argument is added to the accumulator, and then the new value of the accumulator is returned by the `getResult` method. Assuming the accumulator starts at zero, the first two calls to `mkMultiFuncUnit` will return the value passed in to `start`. On the other hand, the second call to `start` for `mkFuncUnitConcurrent2Spec` will return the sum of the first two arguments passed to start.

`mkMultiFuncUnit` also does not implement `mkFuncUnitConcurrent2Spec` if the module parameter of `mkMultiFuncUnit` supports concurrent requests like `mkThreeFBuffered` or `mkThreeFPipelined`. In this case, `mkMultiFuncUnit` implements the more general specification `mkFuncUnitConcurrentSpec`.

3.9 Module Summary

The relationship between the modules in this chapter can be represented in the hierarchy shown in Figure 3-12. In this figure, each level has a number of implementations on the left, and a specification on the right. Within each level, the modules produce equivalent behaviors when viewed from the outside world. Across levels, modules in the upper levels produce a subset of the behaviors of the modules in the lower levels. Arrows in this figure point from

[2]This module is the same as `mkFuncUnitConcurrent3Spec` but with a two-element FIFO instead of a three-element FIFO

implementations to specifications; double-sided arrows mean the modules are related in both directions, *i.e.* either module implements the other. Also, this figure assumes the function $g(x)$ is equal to $f(f(f(x)))$.

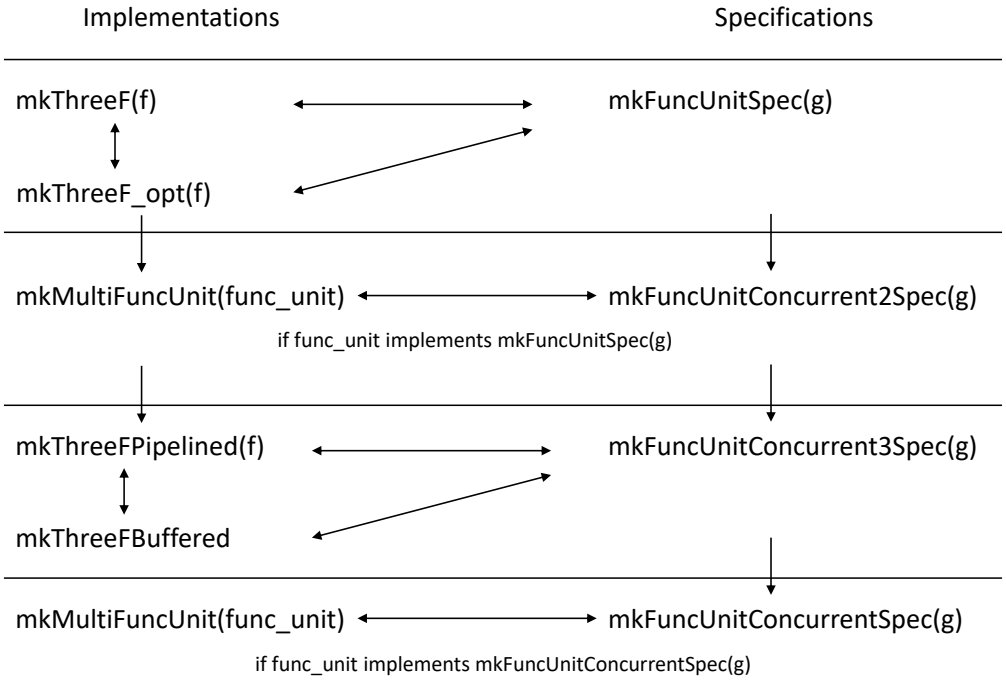

Figure 3-12: Hierarchy of `FuncUnit` implementations and specifications

Therefore an outer module that uses any module in this list as a submodule will be able to produce the same set of results if the submodule is substituted for another one in the same level. If it instead uses a submodule on the higher level, the new module will produce a subset of the behaviors of the original module. On the other hand if it uses a submodule on a lower level, the new module will produce a superset of the behaviors of the original module.

When verifying a module that uses one of these as a submodule, we can substitute the submodule with any submodule on the same level or below, and any positive verification results using the new submodule will apply for the original design as well. On the other hand, if we have already verified a module, we can substitute submodules at the same level or above since doing so introduces no new, potentially incorrect behaviors.

3.10 Verification Insight

Through these example modules and specifications, we have seen a taste of the verification we want to be able to do. From this, we have some insight into how and why module substitution works and what we want to add to our language for verification.

3.10.1 Benefits of Uninterpreted Functions

In these examples of verification, uninterpreted functions were used to represent a function parameter so that verification could be performed for all possible functions. Since uninterpreted functions abstract away function implementation details, verification using uninterpreted functions instead of fully defined functions often runs much faster. Therefore it is common to replace defined functions with uninterpreted functions when possible, *i.e.* when the function abstraction does not introduce bugs.

3.10.2 Module Substitution

Typically substituting a module with another module in a software language requires the new module to be able to do everything the old module can do in order not to violate an assertion in the software.

In our verification approach, we can replace a specification used as a submodule with any implementation of that specification without violating any previously verified assertions in the outside module, even though the new submodule may exhibit only a subset of the behaviors of the specification. This is only possible because of the invisible power of guards, that is, guards prevent rules from firing methods that are not ready, in a way that is invisible to the module that is calling the method.

Let us consider two alternative implementations for guards to understand their importance in verification. In each case we will consider substituting `mkThreeF` in place of `mkThreeFPipelined` (note that this substitution is normally legal because `mkThreeF` exhibits a subset of the behaviors of `mkThreeFPipelined`)

First, if guards only act as an if statement for the body of the method, then firing a

method that is not ready would not prevent a rule from firing, but it would also not change the state of the submodule. In this case, if `mkThreeF` was used in place of `mkThreeFPipelined` in a module, if the module ever called the `start` method a second time before calling `getResult`, then `mkThreeF` would return a different value than `mkThreeFPipelined` once `getResult` was called to get the result of the second call to `start`.

As the second case, consider if the values of guards are observable. In this case, a module could count how many times a method could be called before it becomes not ready. This would produce different results for `mkThreeF` in place of `mkThreeFPipelined`, so it would be possible to construct a module that satisfies all assertions for `mkThreeFPipelined` but violates an assertion for `mkThreeF`.

Therefore it is important to keep guards as they are to keep the level of flexibility we have when substituting a module for another module.

3.10.3 Similar Verification of Verilog

This approach to verification is not easily repeated in Verilog. There are ways to replicate some of the verification tasks, but they all require ad-hoc manual solutions.

The first shortcoming for replicating the verification in Verilog is the absence of function parameters in Verilog, so similarly parameterized designs cannot be expressed without ad-hoc solutions. If one can construct a representation for a function parameter in the design, then it can be manually substituted with an uninterpreted function in the SMT representation before model checking.

The second shortcoming is that the modular verification technique presented in this chapter does not work in Verilog due to the rigid timing requirements of RTL. Replacing a module with a simpler module that computes the result in fewer clock cycles could introduce bugs that were not present in the original design.

In rule-based designs, modular verification only works because internal rules are not visible to the outside world.

3.10.4 Language Additions for Verification

The examples presented in this chapter also reveal part of what is needed in order to do this style of verification.

Module to Symbolic Expression Generator. Statically verifying assertions requires producing symbolic expressions for modules. We need a way to be able to automatically generate these symbolic expressions from our source language. When a module takes in a function parameter, we need it represented as an uninterpreted function so it can represent all possible functions. Likewise we want to be able to easily replace a function used in a module with an uninterpreted function so the verification can be simplified.

Reset Predicates. To improve the verification experience, we want to add reset predicates to our language. Currently, BSV defines reset states by requiring each register to either have a specified reset value or accept all values as a legal reset value. This is sufficient for constructing a working module, but in verification it is often beneficial to be more general.

We introduce the notion of a reset predicate in our language to describe the set of reset states with a Boolean expression that is only true for reset states. This allows us to describe all reset states that are semantically equivalent to some canonical reset state. For example, in `mkMultiFuncUnit` a canonical reset state might set `in_unit` and `out_unit` to 0, but with our reset predicate, we also allow setting `in_unit` and `out_unit` as a reset state.

Simplified Modular Semantics. To simplify specifications, we want to put restrictions on method calls beyond the current restrictions of other rule-based HDLs. Chapter 2 introduced the ambiguity that can arise from calling multiple methods of the same submodule in the same rule.

If we allowed rules to call multiple action methods, then when verifying modules against specifications, we would need to ensure all possible combinations of action methods in the module match the corresponding combination of action methods in the specification. Some of these combinations may not make sense or ever be used together, but if the behaviors of the module and the specification disagree on one of these combinations, then the module does not fully implement the specification.

In our rule-based hardware design language we restrict rules from calling two action meth-

ods of the same module. This simplifies specifications and verification against specifications. If a module needs to call two action methods of the same submodule, then the submodule can expose a new interface method that is the combination of the two methods. When the module needs to call the two action methods of the submodule, it can call the new combined method instead.

3.11 Conclusion

This chapter introduced small examples of the type of verification we want to be able to do by looking at multiple variants of the same module with different microarchitectural implementations.

In the next chapter, we go through a similar exercise of verifying refinements of modules, but the modules are processors. Processors are larger and more complex than the simple functional units presented in this chapter, so more abstractions will be introduced in order to perform the desired verification.

Chapter 4

Verifying Processors

Verifying processors presents many unique challenges that are not present when verifying functional units. When verifying functional units, it is easy to specify modules through their interfaces, as shown in Chapter 3. On the other hand, it is hard to specify processors through their interfaces; instead their operation is best described in terms of executing instructions to update architectural state.

Furthermore, processors are larger systems with multiple submodules and lots of computational complexity. Processors are a great examples of designs that approach or pass the limit of computational complexity supported by naive SMT-based model checking. Therefore, abstractions are essential to processor verification using SMT-based model checking.

In this chapter, we want to verify processors that implement a simplified 32-bit RISC-V ISA[1]. We start with a simple single-cycle processor implementation, and we verify a multicycle processor and a pipelined processor against it by checking their states match when an instruction is executed on each processor. This verification takes advantage of function and data abstraction to hide the instruction-execution details from the SMT solver since the functions used in the single-cycle processor are the same functions used in the multicycle and pipelined processor implementations.

After that we introduce two aspects of all processors that are often overlooked when verifying processor models [19, 32]: exceptions and self-modifying code. For these examples,

[1]Simplifications include only allowing 32-bit load and store instructions, allowing writes to register x0 so it is not always 0, truncating byte addresses to word addresses before accessing memory, etc. These simplifications are for presentation purposes only.

we show how module abstractions can be leveraged to handle the complexities introduced by these cases.

As the final step in the verification, we show how the single-cycle processor can be verified against an ISA specification. This shows that our verification of module refinements can be linked to real-world specifications that do not use the same functions for instruction execution.

4.1 Processor Design Experience

We have designed many different RISC-V processors, including a tiny microcontroller with a reduced datapath implemented with carbon nanotube FETs [52], a 32-bit in-order pipelined processor with a DTLS accelerator [15, 16], and a 64-bit out-of-order processor [86, 87] which we added security enhancements to [26]. From these designs we learned the importance of formal verification over just running programs.

As an example, one of the most common RISC-V bugs found with formal verification against a RISC-V spec (according to the creator of `riscv-formal` [85]) is failing to zero out the least significant bit in the destination of the JALR instruction. It is possible to run many programs, including booting Linux, without triggering this bug since most programs do not rely on this behavior, but once the processor runs a program that does rely on this behavior (hopefully before fabricating an ASIC), it will likely result in a misaligned instruction exception instead of the processor jumping to the corrected target.

4.2 Single-Cycle Processor

Consider the simple 32-bit processor with split instruction and data memories shown in Figure 4-1. This is our reference implementation we want to verify our other processors against.

This processor only contains architecturally visible state, and it executes each instruction in a single rule firing. The details of instruction decoding and execution are primarily contained within the functions `decode`, `alu`, and `control`. The implementations of these

```
module mkSimpleProc(Empty);
    State#(Bit#(32)) pc <- mkState;
    Vector#(32, State#(Bit#(32))) rf <- mkState;
    State#(Array#(Bit#(30),Bit#(32))) imem <- mkState;
    State#(Array#(Bit#(30),Bit#(32))) dmem <- mkState;
    reset(pc == 0);
    rule step;
        let inst = imem[pc[31:2]];
        let dinst = decode(inst);
        let rs1_val = rf[getRS1(dinst)];
        let rs2_val = rf[getRS2(dinst)];
        let imm_val = getImm(dinst);
        let alu_out = alu(dinst, rs1_val, rs2_val, imm_val,
                          pc);
        let next_pc = control(dinst, rs1_val, rs2_val,
                          imm_val, pc);
        let addr = addrCalc(rs1_val, imm_val);
        // update memory
        if (isStoreInst(dinst)) begin
            dmem[addr[31:2]] <= rs2_val;
        end
        // update registers
        if (usesRD(dinst)) begin
            if (isLoadInst(dinst)) begin
                rf[getRD(dinst)] <= dmem[addr[31:2]];
            end else begin
                rf[getRD(dinst)] <= alu_out;
            end
        end
        // update pc
        pc <= next_pc;
    endrule
endmodule
```

Figure 4-1: Simple 32-bit reference RISC processor

functions are not important for now, as the same functions will be used in the processors we are verifying against this reference processor. Therefore, these functions are able to be treated as uninterpreted functions for now.

We say a processor matches this reference processor if, when the two processors execute an instruction from the same architectural state, they end up in the same architectural state. This can be described as a simulation relation between the processor and this reference.

4.3 Multicycle Processor

The reference processor `mkSimpleProc` has a major issue that keeps it from being implemented efficiently in hardware: it assumes two memories that are both accessed combinationally within the same clock cycle. In a realistic processor, these memory accesses would not be done combinationally; they would be done in split requests and responses. In fact, in all but the smallest processors, the memory would be accessed through a cache that has a request/response interface with a variable access latency.

If we want to make a simple realistic processor, then we need to split memory accesses into separate requests and responses. To avoid all control and data hazards, we can implement a processor that only executes a single instruction at a time across multiple clock cycles. As our first processor variant, consider the multicycle processor presented across Figures 4-2 and 4-3.

Instead of directly accessing the memory arrays, this processor accesses memory through a pair of request/response memory interfaces shown at the top of Figure 4-2.

This processor executes instructions across four rules: fetch, decode, execute, and writeback. The fetch rule sends a request to the instruction memory to fetch the instruction. The decode rule receives the instruction from the instruction memory, decodes it, and reads the registers from the register file. The execute rule constructs the immediate value, computes the output of the ALU, computes the next PC, and sends a data load or store request to the data memory for load and store instructions. The writeback rule gets the result from the data memory in case of load instructions, writes to the register file, and updates the PC. After writeback finishes, the fetch rule is fired to start the next instruction.

```
interface InstMem;
    method Action inst_req(Bit#(30) word_addr);
    method ActionValue#(Bit#(32)) inst_resp();
endinterface
interface DataMem;
    method Action data_req(Bool write, Bit#(30) word_addr,
                           Bit#(32) data);
    method ActionValue#(Bit#(32)) data_resp();
endinterface

// Type declaration
typedef enum {Fetch, Decode, Execute, Writeback} ProcState;

module mkMulticycleProc(Empty);
    // Architectural state
    State#(Bit#(32)) pc <- mkState;
    State#(Array#(Bit#(5),Bit#(32))) rf <- mkState;
    InstMem imem <- mkInstMem;
    DataMem dmem <- mkDataMem;

    // Microarchitectural state
    State#(ProcState) state <- mkState;
    State#(DecodedInst) dinst <- mkState;
    State#(Bit#(32)) rs1_val <- mkState;
    State#(Bit#(32)) rs2_val <- mkState;
    State#(Bit#(32)) alu_out <- mkState;
    State#(Bit#(32)) nextPc <- mkState;

    reset(pc == 0 && state == Fetch);
    ...
```

Figure 4-2: State declarations of a multicycle 32-bit RISC processor

```
    ...
    rule fetch(state == Fetch);
        imem.inst_req(pc[31:2]);
        state <= Decode;
    endrule
    rule decode(state == Decode);
        let inst <- imem.inst_resp();
        let dinst_var = decode(inst);
        rs1_val <= rf[getRS1(dinst_var)];
        rs2_val <= rf[getRS2(dinst_var)];
        dinst <= dinst_var;
        state <= Execute;
    endrule
    rule execute(state == Execute);
        let imm_val = getImm(dinst);
        alu_out <= alu(dinst, rs1_val, rs2_val, imm_val, pc);
        next_pc <= control(dinst, rs1_val, rs2_val, imm_val,
                           pc);
        let addr = addrCalc(rs1_val, imm_val);
        if (isLoadInst(dinst)) begin
            dmem.data_req(False, addr[31:2], ?);
        end else if (isStoreInst(dinst)) begin
            dmem.data_req(True, addr[31:2], rs2_val);
        end
        state <= Writeback;
    endrule
    rule writeback(state == Writeback);
        let rf_write_data = alu_out;
        if (isLoadInst(dinst)) begin
            rf_write_data <- dmem.data_resp();
        end else if (isStoreInst(dinst)) begin
            dmem.data_resp();
        end
        // update registers
        if (usesRD(dinst)) begin
            rf[getRD(dinst)] <= rf_write_data;
        end
        // update pc
        pc <= next_pc;
        state <= Fetch;
    endrule
endmodule
```

Figure 4-3: Rules of a multicycle 32-bit RISC processor

4.3.1 `mkMulticycleProc` Verification

To verify this processor, we want to show the state of the multicycle processor matches the corresponding state of the simple processor each time both processors execute an instruction. Therefore we need a custom testbench that is implemented outside the bounds of our hardware design language to force the simple single-cycle processor's `step` rule to fire each time the multicycle processor's `writeback` rule fires.

Once the testbench is in place, the actual verification of the multicycle processor against the simple processor is straightforward. This is because the guards of the multicycle processor's rules are mutually exclusive, and thanks to the way the state is updated, the rules can only execute in the logical order fetch, decode, execute, and writeback. We can simulate the processors starting from the same random architectural states as a first pass for finding bugs, but we want to verify the processor for all possible states.

We want to statically verify that when the processors start from the same architectural state, they always step to the same next architectural state. To do this static verification, we assume we have a way to execute each processor with symbolic expressions to get the symbolic representation of executing an instruction, and we assume we have an SMT solver that can check properties of these symbolic expressions (the details of these are introduced in later chapters).

To improve the performance of the SMT solver, all functions are abstracted away and therefore treated as uninterpreted functions. This significantly simplifies the complexity of each step in the unrolling used in the verification, and thus the SMT solver is able to verify in under a second that executing an instruction in `mkMulticycleProc` always matches the execution of the corresponding instruction from the same architectural state in `mkSimpleProc`.

4.4 Pipelined Processor

Now let us consider the pipelined processor shown across Figures 4-4 and 4-5.

This processor divides the execution of an instruction into the same four steps as the multicycle processor, but the pipelined processor allows for different steps for different in-

```
module mkPipelinedProc(Empty);
    State#(Bit#(32)) pc <- mkState
    State#(Array#(Bit#(5),Bit#(32))) rf <- mkState;
    InstMem imem <- mkInstMem;
    DataMem dmem <- mkDataMem;

    State#(F2D) f2d <- mkState;
    State#(D2E) d2e <- mkState;
    State#(E2W) e2w <- mkState;
    State#(Bool) f2d_valid <- mkState;
    State#(Bool) d2e_valid <- mkState;
    State#(Bool) e2w_valid <- mkState;
    State#(Bool) f2d_poisoned <- mkState;

    reset(pc == 0 && !f2d_valid && !d2e_valid
            && !e2w_valid);

    rule fetch(!f2d_valid)
        imem.inst_req(pc[31:2]);
        pc <= pc + 4; // predicted pc
        f2d <= F2D{pc: pc, ppc: pc + 4};
        f2d_valid <= True;
        f2d_poisoned <= False
    endrule
    rule decode(f2d_valid && !d2e_valid);
        let inst <- imem.inst_resp();
        if (!f2d_poisoned) begin
            dinst = decode(inst);
            // block this rule from firing if one of the
            // source registers is in use
            if (e2w_valid) begin
                guard(getRS1(dinst) != getRD(e2w.dinst));
                guard(getRS2(dinst) != getRD(e2w.dinst));
            end
            let rs1_val = rf[getRS1(dinst)];
            let rs2_val = rf[getRS2(dinst)];
            d2e <= D2E{pc: f2d.pc, ppc: f2d.ppc, dinst: dinst
                    rs1_val: rs1_val, rs2_val: rs2_val};
            d2e_valid <= True;
        end
        f2d_valid <= False;
    endrule
    ...
```

Figure 4-4: Fetch and decode rules of a pipelined 32-bit RISC processor

```
    ...
    rule execute(d2e_valid && !e2w_valid);
        let imm_val = getImm(d2e.dinst);
        let alu_out = alu(d2e.dinst, d2e.rs1_val,
                          d2e.rs2_val, imm_val, d2e.pc);
        next_pc <= control(d2e.dinst, d2e.rs1_val,
                           d2e.rs2_val, imm_val, pc);
        addr = addrCalc(d2e.rs1_val, imm_val);
        if (isLoadInst(d2e.dinst)) begin
            dmem.data_req(False, addr[31:2], ?);
        end else if (isStoreInst(dinst)) begin
            dmem.data_req(True, addr[31:2], d2e.rs2_val);
        end
        if (d2e.ppc != next_pc) begin
            pc <= next_pc;
            f2d_poisoned <= True;
        end
        e2w <= E2w{pc: d2e.pc, dinst: d2e.dinst
                alu_out: alu_out};
        e2w_valid <= True;
        d2e_valid <= False;
    endrule
    rule writeback(e2w_valid);
        let rf_write_data = e2w.alu_out;
        if (isLoadInst(e2w.dinst)) begin
            rf_write_data <- dmem.data_resp();
        end else if (isStoreInst(e2w.dinst)) begin
            dmem.data_resp();
        end
        // update registers
        if (usesRD(e2w.dinst)) begin
            rf[getRD(e2w.dinst)] <= rf_write_data;
        end
        e2w_valid <= False;
    endrule
endmodule
```

Figure 4-5: Execute and writeback rules of a pipelined 32-bit RISC processor

structions to be executed at the same time.

The correctness of this processor relies on correctly handling pipeline hazards. The first hazard is a control hazard that arises from fetching instructions before the previous instruction finishes executing, The fetch stage assumes the next PC to fetch is always PC+4, but in the case of branches and jump instructions, the next PC may be different from that, and it will not be known until the instruction is executed. Therefore, when an instruction changes the PC to something other than PC + 4, it must kill previous instructions and refetch the next instruction with the correct PC. Since an instruction in the `f2d` register has corresponding bookkeeping in the instruction memory (*i.e.* it has a response ready to return), it must be poisoned instead of just killed so that the decode stage can remove the bookkeeping in the instruction memory and then kill the instruction.

The second hazard is the data hazard that comes from reading registers before the previous instruction writes its result to the register file. We handle this hazard by stalling the reading instruction if the writing instruction is writing to a register that is going to be read.

4.4.1 `mkPipelinedProc` Verification

The most important key to verifying this processor is recognizing the true architectural PC for the pipelined processor may either be in the `pc` state variable or it may be within the pipeline in `f2d`, `d2e`, or `e2w` depending on where the oldest instruction in the pipeline is.

The verification of the pipelined processor is very similar to the verification of the multicycle processor. Just like in the multicycle processor verification, this verification requires a custom testbench that ensures the `step` rule of the processor specification is fired each time the `writeback` rule of the pipeline is fired. The testbench also requires assertions that the two processors have matching architectural PCs, register files, and memories. The verification also uses abstracted versions of all the functions to simplify the complexity of the verification. When this testbench is verified, the verification takes less than a second to run and only needs to consider sequences of executions up to 13 steps long[2].

[2]This verification is done using rule-based k-induction as introduced in Chapter 9.

4.5 Processor with Exceptions

The previous processors do not include exceptions in them. Exceptions add complexity to the control hazards in the pipeline because when an exception happens, the next instruction needs to see the updated exception state of the processor. For example, exceptions typically redirect the PC to an exception handler in a higher privilege mode, and the location of the exception handler typically requires a higher privilege mode to access the memory locations. Therefore if an exception happens and the exception handler is fetched before the privilege mode has been updated, the fetch will incorrectly raise a second exception.

To support exceptions and privilege modes, we add a control and status register file (CSRF) that contains the privilege level and other registers specified by the RISC-V ISA to manage taking exceptions and returning from them.

The registers in the CSRF are accessed in multiple places in the pipeline. Some of the CSRF registers are sent to the instruction and data memory for memory accesses; this is to check if the accessed addresses are legal to access or not depending on the control registers. Other CSRF registers are used during decode to determine which instructions are currently legal; for example, the MRET instruction that returns from an interrupt is not legal in lower-privilege modes. The registers in the CSRF can either be updated directly by the CSRRx family of instructions, or they can be updated indirectly by raised exceptions.

The interface for our CSRF is shown in Figure 4-6. In the pipelined processor, the `getMemoryInfo` method is called from the fetch stage, the `getDecodeInfo` method is called from the decode stage, and the rest of the methods are called from the writeback stage (at most one is called per instruction). The `executeCSRRx` instruction is called to implement the CSRRx instructions.

4.5.1 CSRF Abstractions

The CSRF has a lot of complexity inside of it. We would be better off for verification if we could treat it as an abstract module.

First consider a fully abstracted CSRF with no rules or guards. A fully abstract module

```
interface CSRF;
    method MemoryCSRF getMemoryInfo();
    method DecodeCSRF getDecodeInfo();
    method ActionValue#(Bit#(32)) raiseException(
                                Bit#(4) cause, Bit#(32) pc);
    method ActionValue#(Bit#(32)) returnFromException();
    method ActionValue#(Bit#(32)) executeCSRRx(Bit#(2) op,
                                Bit#(12) csr, Bit#(32) data);
endinterface
```

Figure 4-6: CSRF interface

without rules or guards is a module where the state has been replaced by a single state variable of an abstract type, and each method is replaced by an uninterpreted function. Each uninterpreted function takes in as arguments the state of the module and the corresponding method's arguments. An uninterpreted function either returns a return value, a next state value, or both, depending on the type of method being abstracted.

With a fully abstract CSRF, the values from the methods `getMemoryInfo` and `getDecodeInfo` can change each time `executeCSRRx` is called, even if `executeCSRRx` is not changing any of the CSRs that are read by `getMemoryInfo` and `getDecodeInfo`. This puts additional unnecessary constraints on the processor, because in order for a pipelined processor with exceptions to match a single-cycle processor with exceptions, the pipelined processor has to see the same value for `getMemoryInfo` and `getDecodeInfo` as the single-cycle processor for each instruction.

Instead of a fully abstract CSRF, we use the partially abstracted CSRF shown in Figure 4-7. This is only partially abstract because the abstraction knows about two classes of CSRs: normal CSRs and special CSRs. Normal CSRs are only accessed through CSRRx instructions, but special CSRs are visible throughout the pipeline through the `getMemoryCSRs` and `getDecodeCSRs` methods.

This abstract implementation of the CSRF hides the implementation details, but it reveals details about dependencies between the methods. For example, if `executeCSRRx` is called and given a normal CSR, then the values for `special_csrs`, and therefore `getMemoryCSRs` and `getDecodeCSRs`, will not change. Therefore when executing a CSRRx

```
module mkCSRFAbstraction(CSRF);
    State#(NormalCSRs) normal_csrs <- mkState;
    State#(SpecialCSRs) special_csrs <- mkState;

    method MemoryCSRs getMemoryCSRs();
        return get_memory_csrs(special_csrs);
    endmethod
    method DecodeCSRs getDecodeCSRs();
        return get_decode_csrs(special_csrs);
    endmethod
    method Action#(Bit#(32)) raiseException(
                                Bit#(4) cause, Bit#(32) pc);
        special_csrs <= raise_exception(special_csrs,
                            normal_csrs, cause);
        return get_exception_pc(special_csrs, normal_csrs);
    endmethod
    method ActionValue#(Bit#(32)) returnFromException();
        special_csrs <= return_from_exception(special_csrs,
                            normal_csrs);
        return get_return_pc(special_csrs, normal_csrs);
    endmethod
    method ActionValue#(Bit#(32)) executeCSRRx(Bit#(2) op,
                                Bit#(12) csr, Bit#(32) data);
        if (is_special_csr(csr)) begin
            let (next_special_csrs, val) =
                execute_csrrx_special(special_csrs, op, csr,
                                      data);
            special_csrs <= next_special_csrs;
        end else begin
            let (next_normal_csrs, val) =
                execute_csrrx_normal(normal_csrs, op, csr,
                                     data);
            normal_csrs <= next_normal_csrs;
        end
        return val;
    endmethod
endinterface
```

Figure 4-7: CSRF abstraction

instruction that updates a normal CSR, then the next instruction does not need to be refetched since it saw the correct values for `getMemoryCSRs` and `getDecodeCSRs`. On the other hand, if executing a CSRRx instruction that updates a special CSR, the values of `getMemoryCSRs` and `getDecodeCSRs` may change, so it is necessary to fetch the next instruction again.

Using this specification, we can verify our implementation of exceptions on the pipelined processor matches exceptions implemented on our single-cycle processor. Thanks to the abstract CSRF, we are able to verify that the writeback stage only needs to refetch the next instruction when there is an exception, a return from an exception, or a "special" CSR is written; it does not need to refetch the instruction when writing a "normal" CSR.

4.6 Unified Instruction and Data Memory

The previous processors all assume separate instruction and data memories, but most modern processors have a unified address space where stored data can later be fetched as instructions and executed. If we modify our processors to have a unified memory without making any other changes, our pipelined processor would start seeing different results than the other processors, even if the instruction and data request/response interfaces are coherent. This is because instructions in pipelines can be fetched before the previous instruction completes its execution, so if the previous instruction wrote to the value in memory containing the next instruction, then the pipeline will execute the old instruction while the single-cycle and multicycle processors will execute the new instruction.

ISAs must specify what the legal behaviors are when instruction fetches and data stores touch the same locations in memory. The typical solution of modern ISAs is to allow instruction fetches to fetch stale data unless a specific fence instruction is executed that forces instruction fetch to see recently stored data.

RISC-V deals with unified instruction and data memory by allowing instruction fetches to see stale data and including the FENCE.I instruction to synchronize instruction fetches with data stores. The RISC-V specification describes the behavior of the FENCE.I instruction as follows [6]:

> A FENCE.I instruction ensures that a subsequent instruction fetch on a RISC-V hart will see any previous data stores already visible to the same RISC-V hart.

Note that in RISC-V, a *hart* is a hardware thread.

We want to modify the memory modules in our processors to have a unified instruction and data memory, and we want to adopt the RISC-V FENCE.I instruction to synchronize instruction fetches with store instructions. This presents two new verification problems in the processor: the first is ensuring the memory interface is returning legal stale values when self-modifying code does not have a necessary FENCE.I instruction, and the second is ensuring FENCE.I is implemented correctly.

From the processor's point of view, the problem of which stale data is returned by the memory interface does not matter; that should be able to be specified and verified while only looking at the cache. On the other hand, the FENCE.I instruction requires the processor and the memory system to cooperate to correctly implement the specification. The unified memory interface needs a new `fence_i` interface method to perform the necessary operation to make instruction fetches see the results of all previous data stores. This implementation can range in complexity from doing nothing, to flushing a store buffer, all the way to flushing both the instruction and data caches. The pipeline needs to make sure it calls the `fence_i` method when a FENCE.I instruction is executed, and if the next instruction has already been fetched, it needs to be killed and fetched again.

4.6.1 Unified Memory Abstraction

We can view the correctness criteria of the processor axiomatically. When a FENCE.I instruction is executed, the next instruction to execute must be fetched after the FENCE.I instruction is executed. We could verify this property if we have a unified memory abstraction where the instruction fetch results change when FENCE.I is called, so it is easy to tell if an instruction was fetched before or after a fence instruction. Also, the way in which FENCE.I changes the instruction fetch results can depend on previous stores, so we know that the FENCE.I wasn't called too early either.

We construct a unified memory abstraction to track these axioms of the FENCE.I in-

struction. Our abstract unified memory is shown in Figure 4-8. `InstState` and `DataState` are abstract types, while `inst_req_uf`, `data_req_write_uf`, `data_req_uf`, and `fence_i_uf` are uninterpreted functions.

Looking at the methods in this module reveals that the ordering of `inst_req` and `data_req` does not matter, since neither writes to a state that is read by the other. All the ordering constraints are induced by `fence_i`. Since `fence_i` reads `data_state`, the result of this function changes depending on the relative ordering of the `fence_i` method call and other calls to `data_req` that write to the memory. Since `fence_i` writes to `inst_state`, the result of `inst_req` depends on the most recent call to `fence_i`.

This module is set up so that if a processor using this unified memory abstraction produces the same results as a single-cycle or multicycle processor using this same abstraction, then the processor correctly implements the ordering constraints necessary for the FENCE.I instruction.

4.6.2 Verification of Modified Pipelined Processor

To implement the FENCE.I instruction in a pipelined processor, we need a mechanism to kill or poison all younger instructions in the pipeline, call the `fence_i` method on the memory system, and then fetch the next instruction again. This can be done with minor modifications to the processor in Figures 4-4 and 4-5.

To verify this modified processor, we make the equivalent modifications to the single-cycle processor and replace the references to `imem` and `dmem` with references to the abstract states used by the memory abstraction. Then when we rerun the verification, the verification completes in under a second and confirms that the pipeline implements the memory access orderings necessary for FENCE.I.

Note that the abstraction used for the unified memory is not a specification for a real unified memory system. That is because a real memory system sometimes allows stores to be visible through the instruction fetch method without a FENCE.I instruction. This could be the case if the memory system has separate instruction and data caches, and a line was evicted from the data cache before it was fetched from the instruction cache. This behavior is orthogonal to the implementation of FENCE.I, so it is not necessary to model it as part

```
interface UnifiedMem;
    method Action inst_req(Bit#(30) word_addr);
    method ActionValue#(Bit#(32)) inst_resp();
    method Action data_req(Bool write, Bit#(30) word_addr,
                           Bit#(32) data);
    method ActionValue#(Bit#(32)) data_resp();
    method Action fence_i();
endinterface

module mkUnifiedMemAbstraction(UnifiedMem);
    State#(InstState) inst_state <- mkState;
    State#(DataState) data_state <- mkState;

    State#(Bit#(32)) ireq_resp <- mkState;
    State#(Bool) ireq_valid <- mkState;
    State#(Bit#(32)) dreq_resp <- mkState;
    State#(Bool) dreq_valid <- mkState;

    method Action inst_req(Bit#(30) word_addr) if (
                                                !ireq_valid);
        ireq_resp <= inst_req_uf(inst_state, word_addr);
        ireq_valid <= True;
    endmethod
    method ActionValue#(Bit#(32)) inst_resp() if (ireq_valid);
        ireq_valid <= False;
        return ireq_resp;
    endmethod
    method Action data_req(Bool write, Bit#(30) word_addr,
                           Bit#(32) data);
        if (write) begin
            data_sate <= data_req_write_uf(data_state,
                          word_addr, data)
        end
        dreq_resp <= data_req_uf(data_state, word_addr);
        dreq_valid <= True;
    endmethod
    method ActionValue#(Bit#(32)) data_resp();
        dreq_valid <= False;
        return dreq_resp;
    endmethod
    method Action fence_i();
        inst_state <= fence_i_uf(inst_state, data_state);
    endmethod
endmodule
```

Figure 4-8: Abstracted unified memory module

of the memory abstraction.

4.7 Verification Against an ISA Specification

In the previous sections in this chapter, we verify processors against a simpler processor, but in practice, we want to be able to verify against an ISA specification. For example, RISC-V has an official formal specification implemented in the SAIL ISA specification language [8]. SAIL has also been used to specify ARMv8 [12]. These specifications are instruction-by-instruction specifications of the expected behavior of the ISAs.

For our ISA specification we will consider the example specification written in BSV shown in Figure 4-9. This models the structure seen in instruction-by-instruction ISA specifications. While specifications are not typically written in HDLs (with few minor exceptions such as riscv-formal [7]), a representation like this one can be obtained by automatically translating a specification (like SAIL or RISCV-PLV [9, 25]) into an HDL. Note that this specification and our processor implementations are written using word-level operations like addition. We assume all word-level operations are implemented correctly when synthesized to bit-level gates, and the verification of these operations is done at a lower abstraction level.

Verifying the processors `mkMulticycleProc` and `mkPipelinedProc` directly against this specification results in too much complexity for the SMT solver to reasonably handle. Instead, we can verify the single-cycle processor against the ISA specification by just comparing the symbolic expression of the next-state function for each of their `step` rules. This verification completes in a short amount of time, so `mkMulticycleProc` and `mkPipelinedProc` match the ISA specification as well.

4.8 Conclusion

In conclusion, we verified a few variants of processors against an ISA specification by leveraging a single-cycle processor spec that contained the same functions found in the multicycle and pipelined implementations as an intermediate step. By using the same functions as found in the multicycle and pipelined implementations, the functions can be treated as un-

```
module mkISASpec(Empty);
    State#(Array#(Bit#(30), Bit#(32))) imem <- mkState;
    State#(Array#(Bit#(30), Bit#(32))) dmem <- mkState;
    State#(Array#(Bit#(5), Bit#(32))) regs <- mkState;
    State#(Bit#(32)) pc <- mkState;
    reset(pc[1:0] == 0); // pc must be aligned

    rule step;
        let inst = imem[pc[31:2]];
        let opcode = inst[6:0];
        let funct3 = inst[14:12];
        if (opcode == 'b0110111) begin // LUI
            let immU = {inst[31:12], 12'b0};
            let rd = inst[11:7];
            regs[rd] <= immU;
            pc <= pc + 4;
        end else if (opcode == 'b0010111) begin // AUIPC
            let immU = {inst[31:12], 12'b0};
            let rd = inst[11:7];
            regs[rd] <= immU + pc;
            pc <= pc + 4;
        end else if (opcode == 'b0110011 && funct3 == 'b000
                     && inst[30] == 0) begin // ADD
            let val1 = regs[inst[19:15]];
            let val2 = regs[inst[24:20]];
            let rd = inst[11:7];
            regs[rd] <= val1 + val2;
            pc <= pc + 4;
        end else if (opcode == 'b0110011 && funct3 == 'b000
                     && inst[30] == 1) begin // SUB
            let val1 = regs[inst[19:15]];
            let val2 = regs[inst[24:20]];
            let rd = inst[11:7];
            regs[rd] <= val1 - val2;
            pc <= pc + 4;
        end else ... // other instructions omitted
    endrule
endmodule
```

Figure 4-9: Example RISC-V ISA specification

interpreted functions to significantly improve the run-time performance of the verification task. By representing the processor using just architectural states and a single rule, the single-cycle processor spec can be verified against the ISA spec by just comparing the next-state function of the single rule which also significantly improves the run-time performance of the verification task. Therefore, combining the two results, our multicycle and pipelined processors can be verified against the ISA spec efficiently.

We also showed how module abstractions can be used to verify exceptions and unified instruction and data memory.

In the next chapters we present the formalisms behind this verification approach starting with the HDL Spec 'n' Check and its semantics.

Chapter 5

Rule-Based Hardware Design Language Semantics

To understand how the verification technique works, we present a core of our rule-based hardware design language, Spec 'n' Check, and its semantics. We assume programs written in Spec 'n' Check are already typechecked and there exists a legal initial reset state.

Spec 'n' Check can be synthesized to RTL if the module stays within a synthesizable subset of the language. The synthesizable subset is determined entirely by the types and functions used in the design; if all the types and functions are synthesizable, then the module is synthesizable. Otherwise, if a type like unbounded integers is used, then the module is not synthesizable.

This chapter presents Spec 'n' Check's syntax and semantics along with an example. After the presentation of Spec 'n' Check, there is a comparison between Spec 'n' Check and other rule-based hardware languages.

5.1 Grammar

The grammar of the core language is presented in Figure 5-1. Modules are made up of state variables, submodule instances, a reset predicate, and rules and methods. The bodies of rules and methods are made up of actions (a) and expressions (e). Actions are statements that can change the state of the module through state writes (s `<=` e) and action method

calls ($m.f_a(e)$), and expressions are pure expressions that can only read state variables.

e	:=	s	(state-rd)
	|	c	(const)
	|	t	(var)
	|	$f(e_1, \ldots, e_n)$	(func)
	|	`let` t `=` e_1 `in` e_2	(e-let)
a	:=	s `<=` e	(state-wr)
	|	`if` e `then` a_1 `else` a_2	(if)
	|	`guard(`e`)`	(guard)
	|	a_1 `;` a_2	(par)
	|	`let` t `=` a_1 `in` a_2	(a-let)
	|	e	(expr)
	|	$m.f_a(e)$	(action-method-call)
	|	$m.f_v(e)$	(value-method-call)
rule	:=	`rule` r `=` a	(rule)
amethod	:=	`amethod` f_a `=` $\lambda t.a$	(action-method)
vmethod	:=	`vmethod` f_v `=` $\lambda t.a$	(value-method)
statedecl	:=	`state` s	(state-decl)
submodinst	:=	`instance` m M	(submod-inst)
reset	:=	`reset` e	(reset)
module	:=	`module` M `=` $statedecl^*$ $submodinst^*$ $reset$ $rule^*$ $amethod^*$ $vmethod^*$	(module)

Terminals:

s	state name	f_a	action method name
c	constant value	f_v	value method name
t	variable name	m	submodule instance name
f	function name (defined externally)	M	module definition name
r	rule name		

Figure 5-1: Spec 'n' Check grammar

Functions are assumed to be defined externally and referenced by their names f. We assume each function f has a corresponding definition $\underline{f}$, so the semantics of $f(e)$ can be obtained by calling $\underline{f}$ on the value corresponding to expression e.

There are two types of methods in the core language: action methods and value methods. The body of an action method can be any well-formed action. On the other hand, value methods cannot make any state changes, so the body of a value method cannot include state writes or calls to action methods but can call other value methods. In the semantics, when symbols are used to denote method names, subscripts are used to help distinguish between

action methods and value methods; f_a denotes an action method, and f_v denotes a value method.

There is a further restriction on module instances: a module's instances must be tree-shapes. Specifically, this means for all modules M, no submodule instance of M can include M as a submodule (either directly or indirectly).

In addition to these constraints, all expressions and actions must be well-formed and type-checked. Well-formed actions and expressions imply all state reads and writes correspond to states of the module, and all variables are bound by a "let". Two important consequences of having type-checked actions and expressions are that functions are used with the right arities and argument types and all guard expressions and "if" predicate expressions are Booleans.

As an example of the core language, Figure 5-2 shows `mkFIFO2` and `mkThreeFBuffered` (from Section 3.5) implemented in Spec 'n' Check.

5.2 Semantics Basics

The semantics of a module describe its possible state transitions in terms of rule firings and top-level method calls. In order to do this, we need to first introduce some basics for the semantics.

5.2.1 Module Hierarchy

The *module* syntax in Figure 5-1 describes a module definition. Modules can be instantiated as submodule instances within other module definitions using the *submodinst* syntax "`instance` m n" where m is the instance name and n is the name of the module definition.

The collection of module instances used in a single module definition can be viewed as a static acyclic tree-like hierarchy where each submodule instance is a child of the module it is contained within. At each level of the hierarchy, methods of a module instance can only be called by its unique parent module, and each module can only call methods of its direct child submodule instances.

Furthermore, submodule instance state can only be accessed through the submodule instance's interface methods. Therefore it is possible to construct modules with different

```
module mkFIFO2 =
    state d1
    state d2
    state v1
    state v2

    reset !v1 && !v2

    rule canonicalize =
        guard(v2 && !v1) ; v1 <= v2 ; d1 <= d2

    amethod enq = λ x. guard(!v2) ; v2 <= True ; d2 <= x
    amethod deq = λ t. guard(v1) ; v1 <= False
    vmethod first = λ t. guard(v1) ; d1

module mkThreeFBuffered =
    state val
    state count
    instance out_fifo mkFIFO2

    reset count == 0

    rule step = guard(count >= 1) ;
        let next_val = f(val) in
        if count == 3 then
            out_fifo.enq(next_val) ; count <= 0
        else
            val <= next_val ; count <= count + 1

    amethod start = λ x. guard(count == 0) ;
        val <= x ; count <= 1
    amethod getResult = λ t. out_fifo.deq() ; out_fifo.first()
```

Figure 5-2: Two-element FIFO and pipelined $f(f(f(x)))$ implemented in the core language

internal state representation that are indistinguishable from parent modules provided they appear the same from their method calls.

Other rule-based hardware design languages such as BSV and Kami allow for sibling modules to call each other's methods. Furthermore, BSV also allows for multiple modules to call the methods of the same submodule. Spec 'n' Check is more restrictive in its module hierarchy than existing rule-based hardware design languages in order to support our method

of modular verification.

5.2.2 Module Executions

A module execution is made up of an initial state and a sequence of rules and methods that are executed to get to some final state. There is nondeterminism in the sequence of rules and methods because method calls depend on the outside world calling the methods of the module, and rules can fire arbitrarily between nonconcurrent method calls in any order provided the rules are ready.

The semantics defines the behavior of modules by describing which sequences of rules and method calls are legal.

Sequences of executed rules and top-level methods are known as *execution sequences*. Execution sequences are made up of transition labels corresponding to rule firings and method calls. There are three types of primitive labels: rule labels, action method labels, and value method labels. These labels are written as follows:

- r - Label for rule r (written $m.r$ if rule r is within submodule instance m)
- $f_a(x) \rightarrow_a y$ - Label for action method f_a called with argument x and returning value y
- $f_v(x) \rightarrow_v y$ - Label for value method f_v called with argument x and returning value y

In addition to these labels, we also use the empty label ε to denote an empty transition.

As an example of transition labels, in the module `mkThreeFBuffered`, if the `getResult` method is called and returns 4, this corresponds to the label $\texttt{getResult}(\varepsilon) \rightarrow_a 4$ where ε is used to represent an empty value.

The notion of labeled transitions is not new to rule-based HDLs. In Dave's work on verification of refinements [41], he uses transitions labeled by rule name as part of their formalism. In Kami [33], method labels are essential to their modularity system. They have two types of labels for methods (executing a method versus calling a method), but they do not have named transitions for rules.

5.2.3 Concurrent Method Calls

Spec 'n' Check allows a module to call two methods of the same module from the same rule or method, so we have to describe the semantics of such concurrent method calls. We choose to force method atomicity to simplify module specifications, so the semantics of concurrent method calls are defined in terms of the semantics of calling a single method. Furthermore, we force all methods of the same submodule called from the same rule or method to see the same state of the submodule before the method is called. These two design choices allow multiple value methods to be called together with at most one action method from the same rule or method, and the value methods appear to happen first.

We allow the construction of compound labels through the operator $\oplus$. The labels ℓ_1 and ℓ_2 can only be combined if they contain at most one action method between the two of them. The $\oplus$ operator is commutative and associative, so it does not matter in which order labels are combined to produce a compound label; if two compound labels are made up of the same method-call labels, then the two compound labels are equivalent.

Continuing the example from above, `getResult` calls `first` and `deq` of `mkFIFO2`; if `mkFIFO2` were treated as the top module, then this would correspond to the compound label $\texttt{first}(\varepsilon) \rightarrow_v 4 \oplus \texttt{deq}(\varepsilon) \rightarrow_a \varepsilon$.

5.2.4 Hierarchical State Values

The state of a module is an associative array, or a map, that maps state variable names and submodule instance names to their values, typically denoted by S in the semantics. The the value corresponding to the state of submodule instances is another associative array, so the state of a module is a nested data structure when it contains submodule instances. The notations used for associative arrays are shown in Figure 5-3.

As an example, Figure 5-4 shows a diagram of the data structure for state values of `mkThreeFBuffered`.

$[\,]$	Empty map
$[x \mapsto v]$	Singleton map where x maps to value v
$S[x \mapsto v]$	Updating key x in map S to value v
$S[U]$	Updating map S with the key-value pairs in map U
$S(x)$	Looking up the value corresponding to the key x in map S
$S_1 \uplus S_2$	Disjoint union of maps S_1 and S_2

Figure 5-3: Associative array notations

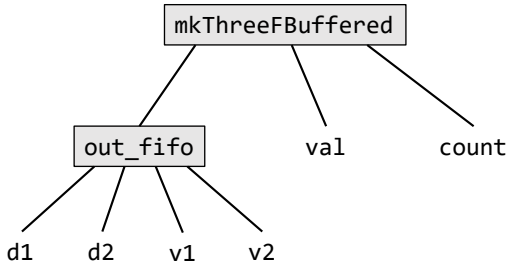

Figure 5-4: State layout of `mkThreeFBuffered`

5.2.5 Accessing Parts of a Module

In the semantics, some helper functions are used to extract bits of syntax from module definitions. These functions are defined below:

- $\mathsf{Rules}(M)$ - Returns the set of the rules of M
- $\mathsf{AMethods}(M)$ - Returns the set of the action methods of M
- $\mathsf{VMethods}(M)$ - Returns the set of the value methods of M
- $\mathsf{Reset}(M)$ - Returns the reset statement of M
- $\mathsf{SubmodInsts}(M)$ - Returns the set of submodule instances of M

5.3 Expression Semantics

The semantics for expressions explain how expressions can be evaluated to produce values. Expression semantics require a way to get the current value of a state element, and they require a way to get values for bound variables. As a result, the expression semantics are

written as a function that takes in the expression syntax, the values of state variables, and the values of bound variables.

The function for the expression semantics is written as

$$[\![e]\!](S, B) = v$$

where v is the resulting value of the expression e with state-value map S and bound-variable map B. Assuming well-formed module syntax, we do not need to worry about looking up values for state variables that do not exist in S or bound variables that do not exist in B.

The full definition for the expression-semantics function is shown in Figure 5-5.

state-rd	$[\![s]\!](S, B) = S(s)$
const	$[\![c]\!](S, B) = \underline{c}$
var	$[\![t]\!](S, B) = B(t)$
func	$[\![f(e_1, \ldots, e_n)]\!](S, B)$: $\forall i,\ v_i = [\![e_i]\!](S, B)$ return $\underline{f}(v_1, \ldots, v_n)$
e-let	$[\![\texttt{let } t \texttt{ = } e_1 \texttt{ in } e_2]\!](S, B)$: $v_1 = [\![e_1]\!](S, B)$ return $[\![e_2]\!](S, B[t \mapsto v_1])$

Figure 5-5: Expression semantics

5.4 Action Semantics

Actions in Spec 'n' Check are the portion of the syntax used to conditionally execute code, call methods, update states, check guards, and more. Each action corresponds to a state update and an optional return value. If the action does not yield a meaningful return value (such as guards), then the nil value ε is used in place of the return value.

A state update is a map that includes new values for a subset of the state values and submodules of the current module. The state update only contains entries for states whose value is updated as part of the action.

In Spec 'n' Check, it is illegal for a rule or method to update the same state or submodule twice. Therefore state updates are combined using the disjoint union operator $\uplus$ to ensure state updates never contain multiple values for the same state or submodule.

The action semantics are presented as a judgement that specifies an update map U and return value v for each action a. The action judgement is written as

$$M \vdash (a, S, B) \downarrow (U, v)$$

where M is the module, S is the state value map, and B is the bound variables.

The deduction rules that define the action semantics are shown in Figure 5-6. The angle brackets $\langle$ and $\rangle$ are used to signify syntax when used in the semantics. Some aspects of the semantics are explained in further detail below.

5.4.1 Stuck-At Semantics

These deduction rules can be used to get the semantics of actions, but some combinations of actions, state variables, and bound variables do not have corresponding action semantics. For example, there is no action semantics corresponding to the action `guard(false)`. This is because the semantics employ stuck-at behavior for guards that are not ready.

A violated guard will not yield action semantics, so any action whose semantics depend on an action with a violated guard will also not yield action semantics. Note that the if-true deduction rule does not depend on the judgement for the false action, so an unsatisfied guard in the non-taken branch does not affect the semantics of the if action.

5.4.2 Double-Write Error

A rule or method cannot update the same state or module twice; this is known as a *double-write error*. In practice, a module that can result in a double-write error is considered an illegal module. For example, the Bluespec compiler checks for double-write errors conservatively by checking if there exist values for registers that result in a double write, and if there is a possibility of a double write, the compiler returns an error.

state-wr
$$\frac{[\![e]\!](S,B) = v}{M \vdash (\langle s \texttt{ <= } e\rangle, S, B) \downarrow ([s \mapsto v], \varepsilon)}$$

if-true
$$\frac{[\![e]\!](S,B) = True \qquad M \vdash (a_1, S, B) \downarrow (U, v)}{M \vdash (\langle \texttt{if } e \texttt{ then } a_1 \texttt{ else } a_2\rangle, S, B) \downarrow (U, v)}$$

if-false
$$\frac{[\![e]\!](S,B) = False \qquad M \vdash (a_2, S, B) \downarrow (U, v)}{M \vdash (\langle \texttt{if } e \texttt{ then } a_1 \texttt{ else } a_2\rangle, S, B) \downarrow (U, v)}$$

guard
$$\frac{[\![e]\!](S,B) = True}{M \vdash (\langle \texttt{guard(}e\texttt{)}\rangle, S, B) \downarrow ([\,], \varepsilon)}$$

par
$$\frac{M \vdash (a_1, S, B) \downarrow (U_1, v_1) \qquad M \vdash (a_2, S, B) \downarrow (U_2, v_2)}{M \vdash (\langle a_1 \texttt{ ; } a_2\rangle, S, B) \downarrow (U_1 \uplus U_2, v_2)}$$

a-let
$$\frac{M \vdash (a_1, S, B) \downarrow (U_1, v_1) \qquad M \vdash (a_2, S, B[t \mapsto v_1]) \downarrow (U_2, v_2)}{M \vdash (\langle \texttt{let } t \texttt{ = } a_1 \texttt{ in } a_2\rangle, S, B) \downarrow (U_1 \uplus U_2, v_2)}$$

expr
$$\frac{[\![e]\!](S,B) = v}{M \vdash (\langle e\rangle, S, B) \downarrow ([\,], v)}$$

action-method-call
$$\frac{\begin{array}{c}\langle \texttt{instance } m \ M_{\texttt{sub}}\rangle \in \mathsf{SubmodInsts}(M) \\ \langle \texttt{amethod } f_a \texttt{ = } \lambda t.a\rangle \in \mathsf{Methods}(M_{\text{sub}}) \\ [\![e]\!](S,B) = x \qquad M_{\text{sub}} \vdash (a, S(m), [t \mapsto x]) \downarrow (U_{\text{sub}}, y) \\ \Omega = S(m)[U_{\text{sub}}]\end{array}}{M \vdash (\langle m.f_a(e)\rangle, S, B) \downarrow ([m \mapsto \Omega], y)}$$

value-method-call
$$\frac{\begin{array}{c}\langle \texttt{instance } m \ M_{\texttt{sub}}\rangle \in \mathsf{SubmodInsts}(M) \\ \langle \texttt{vmethod } f_v \texttt{ = } \lambda t.a\rangle \in \mathsf{Methods}(M_{\text{sub}}) \\ [\![e]\!](S,B) = x \qquad M_{\text{sub}} \vdash (a, S(m), [t \mapsto x]) \downarrow ([\,], y)\end{array}}{M \vdash (\langle m.f_v(e)\rangle, S, B) \downarrow ([\,], y)}$$

Figure 5-6: Action semantics

In the semantics, double-write errors are prevented through the disjoint-union operator $\uplus$, so an action that attempts to combine two updates for the same state will not have any semantics. This is the same stuck-at behavior used by guards but is not intended to be used in real designs.

5.4.3 Method Call Semantics

The method call semantics here are inlined semantics since they depend on the semantics of the action within the body of the method being called, but the updates of the methods'

actions are effectively "lifted" into updates for the entire module's state to make it so that multiple action methods of the same submodule instance cannot be called from the same action. This lifting process is described in detail below.

In the semantics for an action method call, the submodule update U_{sub} obtained by the action semantics of the method's body is an update for a subset of the states in the module m. This is used to compute an updated value for the entire submodule m by combining it with the current state of m. This is denoted by the variable Ω and is computed as $S(m)[U_{\text{sub}}]$. Therefore the update for the module calling the method is $U = [m \mapsto \Omega]$. Since action method calls update the entire state of submodules, if two action method of m are called from the same rule or method, they will conflict when their updates are joined with $\uplus$.

An alternative, more modular representation of method-call semantics is presented later in section 5.6.

5.5 Step Semantics

The next piece of the semantics is the semantics for method and rule firings, or *steps*. Steps are presented as labeled transitions and are are just simple applications of action semantics. A step-semantics judgement is of the form

$$M \vdash S \xrightarrow{\ell} S'$$

where M is the module definition, S is a state value of M, ℓ is a label corresponding to the transition, and S' is the resulting state value if ℓ is fired from state S. This judgement can be read as module M admits the state transition from S to S' with label ℓ. The label ℓ is either a simple label such as a rule r, an action method $f_a(x) \rightarrow_a y$, or a value method $f_v(x) \rightarrow_v y$, or a compound label produced by combining method labels with $\oplus$.

If ℓ is the rule label r, the judgement $M \vdash S \xrightarrow{r} S'$ means rule r is ready to fire from state S, and if it fires, it produces state S'. If ℓ is the action method label $f_a(x) \rightarrow_a y$, the judgement $M \vdash S \xrightarrow{f_a(x) \rightarrow_a y} S'$ means action method f_a is ready to fire from state S, and if it is fired with argument x, it results in state S' with return value y. If ℓ is the value method

label $f_v(x) \to_v y$, the judgement $M \vdash S \xrightarrow{f_v(x)\to_v y} S$ means value method f_v can be called with argument x from state S and returns value y without changing the state.

The deduction rules for producing steps are shown in Figure 5-7.

$$\text{step-rule}\quad \frac{\langle \texttt{rule}\ r\ \texttt{=}\ a\rangle \in \mathsf{Rules}(M) \qquad M \vdash (a, S, [\,]) \downarrow (U, v)}{M \vdash S \xrightarrow{r} S[U]}$$

$$\text{step-submod-rule}\quad \frac{\langle \texttt{instance}\ m\ M_{\text{sub}}\rangle \in \mathsf{SubmodInsts}(M) \quad M_{\text{sub}} \vdash S(m) \xrightarrow{r} \Omega}{M \vdash S \xrightarrow{m.r} S[m \mapsto \Omega]}$$

$$\text{step-action-method}\quad \frac{\langle \texttt{amethod}\ f_a\ \texttt{=}\ \lambda t.a\rangle \in \mathsf{AMethods}(M) \quad M \vdash (a, S, [t \mapsto x]) \downarrow (U, y)}{M \vdash S \xrightarrow{f_a(x)\to_a y} S[U]}$$

$$\text{step-value-method}\quad \frac{\langle \texttt{vmethod}\ f_v\ \texttt{=}\ \lambda t.a\rangle \in \mathsf{VMethods}(M) \quad M \vdash (a, S, [t \mapsto x]) \downarrow ([\,], y)}{M \vdash S \xrightarrow{f_v(x)\to_v y} S}$$

$$\text{step-concurrent-methods}\quad \frac{M \vdash S \xrightarrow{\ell_1} S \qquad \ell_1 \text{is a value method} \quad M \vdash S \xrightarrow{\ell_2} S' \qquad \ell_2 \text{is one or more methods}}{M \vdash S \xrightarrow{\ell_1 \oplus \ell_2} S'}$$

Figure 5-7: Step semantics

5.6 Modular Method-Call Action Semantics

In Figure 5-6, the semantics of method-call actions were defined by looking at the action within the body of the method. We want to be able to treat the semantics modularly so we can use the semantics to support verification using modular refinement, but requiring the semantics of a method call to look into the syntax of the method definition is not modular.

Instead we can define the method-call semantics using the semantic judgements of a method from Figure 5-7. This yields the modular method-call action semantics shown in Figure 5-8.

The modular action method-call semantics say that when considering $m.f_a(e)$ as an action, if the current state of m is $S(m)$, e evaluates to x, and the action method f_a yields

$$\text{action-method-call}\quad \frac{\langle \texttt{instance}\ \ m\ \ M_{\text{sub}}\rangle \in \mathsf{SubmodInsts}(M) \qquad [\![e]\!](S,B) = x \qquad M_{\text{sub}} \vdash S(m) \xrightarrow{f_a(x)\to_a y} \Omega}{M \vdash (\langle m.f_a(e)\rangle, S, B) \downarrow ([m \mapsto \Omega], y)}$$

$$\text{value-method-call}\quad \frac{\langle \texttt{instance}\ \ m\ \ M_{\text{sub}}\rangle \in \mathsf{SubmodInsts}(M) \qquad [\![e]\!](S,B) = x \qquad M_{\text{sub}} \vdash S(m) \xrightarrow{f_v(x)\to_v y} S(m)}{M \vdash (\langle m.f_v(e)\rangle, S, B) \downarrow ([\,], y)}$$

Figure 5-8: Method call action semantics

the M_{sub} step semantics $M_{\text{sub}} \vdash S(m) \xrightarrow{f_a(x)\to_a y} \Omega$ for some return value y and module state Ω, then the action semantics of $m.f_a(e)$ are the update $[m \mapsto \Omega]$ with the return value y.

These modular method-call semantics are equivalent to their inlined versions in Figure 5-6. This is formalized by the following theorem.

Theorem 5.1 (Modular Action Method Call Semantics)**.** *The set of action and step judgements produced by the semantics in Figures 5-6 and 5-7 is equal to the set of judgements produced by the same semantics with the method-call action semantics in Figure 5-6 replaced by the method-call action semantics in Figure 5-8.*

This theorem is proved using structural induction over action syntax and module structure. It has been formalized and proven in the Coq theorem prover.

5.7 Execution Semantics

The final piece missing in the semantics of a module is the execution semantics. The execution judgement is shown below:

$$M \vdash S \xrightarrow{\vec{ex}} S'$$

This judgement is read module M admits the execution from initial state S to the final state S' through the execution sequence of the rules and methods in $\vec{ex}$.

We say $\vec{ex}$ is a *legal execution sequence* for module M if there exists an initial state S (which satisfies the reset predicate) and final state S' such that $M \vdash S \xrightarrow{\vec{ex}} S'$.

The rules for executions are shown in Figure 5-9.

$$\text{reset}\quad \frac{\langle \texttt{reset}\ \ e\rangle = \mathsf{Reset}(M) \qquad \mathcal{E}[\![e]\!](S,[\,]) = True \qquad \forall \langle \texttt{instance}\ \ m'\ \ M'\rangle \in \mathsf{SubmodInsts}(M),\ M' \vdash S(m') \bullet\!\!\xrightarrow{\varepsilon}\!\!\!\!\twoheadrightarrow S(m')}{M \vdash S \bullet\!\!\xrightarrow{\varepsilon}\!\!\!\!\twoheadrightarrow S}$$

$$\text{append-step}\quad \frac{M \vdash S \bullet\!\!\xrightarrow{\vec{ex}}\!\!\!\!\twoheadrightarrow S' \qquad M \vdash S' \xrightarrow{\ell} S''}{M \vdash S \bullet\!\!\xrightarrow{\vec{ex}\,\ell}\!\!\!\!\twoheadrightarrow S''}$$

Figure 5-9: Execution semantics

5.8 Example

As an example of the semantics, consider `mkThreeFBuffered` from Figure 5-2. Let `f` be the function $\lambda x.\, x + 1$. This example will consider the semantics for using `mkThreeFBuffered` to compute $f(f(f(7)))$ assuming the potential initial state

$$\begin{aligned} S_0 \quad = \quad & [\texttt{val} \mapsto 0, \texttt{count} \mapsto 0, \\ & \quad \texttt{out\_fifo} \mapsto [\texttt{d1} \mapsto 0, \texttt{d2} \mapsto 0, \texttt{v1} \mapsto false, \texttt{v2} \mapsto false]]. \end{aligned}$$

Note that if this initial state does not satisfy the reset predicate, any execution starting from this state is not legal.

Reset Predicate on Initial State. First we check the reset predicate on the chosen initial state S_0. $S_0(\texttt{out\_fifo})$ is a legal reset state for `mkFIFO2` since `v1` and `v2` are false. Since $[\![\texttt{count == 0}]\!](S_0, [\,])$ is true, S_0 is a legal reset state for `mkThreeFBuffered`. This gives us the execution judgement

$$\texttt{mkThreeFBuffered} \vdash S_0 \bullet\!\!\xrightarrow{\varepsilon}\!\!\!\!\twoheadrightarrow S_0.$$

Start Method. From our selected initial state S_0 (and any state that satisfies the reset predicate), the only legal step transitions are applications of action method `start` since all other transitions result in guards that are false. Let a_{start} be the action of the `start` method. The action semantics of a_{start} if `start` is called with the value 7 is

$$\texttt{mkThreeFBuffered} \vdash (a_{\text{start}}, S_0, [\texttt{val} \mapsto 7]) \downarrow (U_0, \varepsilon)$$

where $U_0 = [\texttt{val} \mapsto 7, \texttt{count} \mapsto 1]$. Let S_1 be S_0 with the updates of U_0 applied to it (*i.e.* $S_0[U_0]$), therefore

$$\begin{aligned} S_1 &= [\texttt{val} \mapsto 7, \texttt{count} \mapsto 1, \\ &\qquad \texttt{out\_fifo} \mapsto [\texttt{d1} \mapsto 0, \texttt{d2} \mapsto 0, \texttt{v1} \mapsto \mathit{false}, \texttt{v2} \mapsto \mathit{false}]]. \end{aligned}$$

As a result, the following step judgement holds:

$$\texttt{mkThreeFBuffered} \vdash S_0 \xrightarrow{\texttt{start}(7) \rightarrow_a \varepsilon} S_1.$$

This further yields the execution semantics

$$\texttt{mkThreeFBuffered} \vdash S_0 \bullet\!\!\xrightarrow{\overrightarrow{ex_1}}\!\!\gg S_1.$$

where $\overrightarrow{ex_1} = [\texttt{start}(7) \rightarrow_a \varepsilon]$.

Step Rule. From the state S_1, the `step` rule can be fired three times in a row. Firing the `step` rule twice after state S_1 results in the execution semantics

$$\texttt{mkThreeFBuffered} \vdash S_0 \bullet\!\!\xrightarrow{\overrightarrow{ex_3}}\!\!\gg S_3$$

where

$$\begin{aligned} \overrightarrow{ex_3} &= [\texttt{start}(7) \rightarrow_a \varepsilon, \texttt{step}, \texttt{step}] \\ S_3 &= [\texttt{val} \mapsto 9, \texttt{count} \mapsto 3, \\ &\qquad \texttt{out\_fifo} \mapsto [\texttt{d1} \mapsto 0, \texttt{d2} \mapsto 0, \texttt{v1} \mapsto \mathit{false}, \texttt{v2} \mapsto \mathit{false}]]. \end{aligned}$$

Firing the `step` rule a third time causes the rule to behave differently. This is because when `count` is equal to 3 (as it is in state S_3), the predicate of the "if" action evaluates to true, causing the rule to call the `enq` method of `out_fifo` with 10 as the argument. The action of this method call depends on a step transition of `mkFIFO2` with the state of the FIFO equal to $S_3(\texttt{out\_fifo})$. Looking at the step semantics of `mkFIFO2` gives the judgement

$$\texttt{mkFIFO2} \vdash S_3(\texttt{out\_fifo}) \xrightarrow{\texttt{enq}(10) \rightarrow_a \varepsilon} \Omega_4$$

where $\Omega_4 = [\texttt{d1} \mapsto 0, \texttt{d2} \mapsto 10, \texttt{v1} \mapsto \mathit{false}, \texttt{v2} \mapsto \mathit{true}]$.

Now the value Ω_4 from the step semantics above can be used to get the semantics of the action `out_fifo.enq(next_val)` in the `step` rule of `mkThreeFBuffered`:

$$\texttt{mkThreeFBuffered} \vdash (\langle \texttt{out\_fifo.enq}(\texttt{next\_val}) \rangle, S_3, [\texttt{next\_val} \mapsto 10]) \downarrow ([\texttt{out\_fifo} \mapsto \Omega_4], \varepsilon)$$

Putting this together with the rest of the action of the `step` rule, a_{step}, gives the step semantics

$$\texttt{mkThreeFBuffered} \vdash S_3 \xrightarrow{\texttt{step}} S_4$$

and the execution semantics

$$\texttt{mkThreeFBuffered} \vdash S_0 \bullet\!\!\xrightarrow{\overrightarrow{ex_4}}\!\!\!\!\rightarrow S_4$$

where

$$\begin{aligned} S_4 &= [\texttt{val} \mapsto 9, \texttt{count} \mapsto 0, \\ &\qquad \texttt{out\_fifo} \mapsto [\texttt{d1} \mapsto 0, \texttt{d2} \mapsto 10, \texttt{v1} \mapsto \mathit{false}, \texttt{v2} \mapsto \mathit{true}]] \\ \overrightarrow{ex_4} &= [\texttt{start}(7) \rightarrow_a \varepsilon, \texttt{step}, \texttt{step}, \texttt{step}]. \end{aligned}$$

Canonicalize Rule. At this point, there are two different state transitions that are ready. Either the `start` method can be called again, or the `canonicalize` rule within `out_fifo` can be fired. We choose to consider firing the `canonicalize` rule in this example to get the result of $f(f(f(x)))$ in fewer steps.

The step semantics of firing `out_fifo`'s `canonicalize` rule within `mkThreeFBuffered` depends on the step semantics of firing the `canonicalize` rule where `mkFIFO2` is the top module. Therefore the judgement

$$\texttt{mkFIFO2} \vdash S_4(\texttt{out\_fifo}) \xrightarrow{\texttt{canonicalize}} \Omega_5$$

is used to produce the step judgement

$$\texttt{mkThreeFBuffered} \vdash S_4 \xrightarrow{\texttt{out\_fifo.canonicalize}} S_5$$

and thus the execution judgement

$$\texttt{mkThreeFBuffered} \vdash S_0 \bullet\!\!\xrightarrow{\overrightarrow{ex_5}}\!\!\gg S_5$$

where

$$\begin{aligned}
\Omega_5 &= [\texttt{d1} \mapsto 10, \texttt{d2} \mapsto 10, \texttt{v1} \mapsto true, \texttt{v2} \mapsto false] \\
S_5 &= [\texttt{val} \mapsto 9, \texttt{count} \mapsto 0, \\
&\qquad \texttt{out_fifo} \mapsto [\texttt{d1} \mapsto 10, \texttt{d2} \mapsto 10, \texttt{v1} \mapsto true, \texttt{v2} \mapsto false]] \\
\overrightarrow{ex_5} &= [\texttt{start}(7) \rightarrow_a \varepsilon, \texttt{step}, \texttt{step}, \texttt{step}, \texttt{out_fifo.canonicalize}].
\end{aligned}$$

GetResult Method. The final step in computing $f(f(f(x)))$ with `mkThreeFBuffered` is calling the `getResult` method. The action of the `getResult` method, referred to as $a_{\text{getResult}}$ here, calls the `first` and `deq` methods of `out_fifo`, so the semantics depend on the step semantics of `mkFIFO2`. The step semantics of `mkFIFO2` for `first` and `deq` from state $S_5(\texttt{out\_fifo})$ are shown below:

$$\begin{aligned}
&\texttt{mkFIFO2} \vdash S_5(\texttt{out_fifo}) \xrightarrow{\texttt{first}(\varepsilon)\rightarrow_v 10} S_5(\texttt{out_fifo}) \\
&\texttt{mkFIFO2} \vdash S_5(\texttt{out_fifo}) \xrightarrow{\texttt{deq}(\varepsilon)\rightarrow_a \varepsilon} \Omega_6
\end{aligned}$$

where $\Omega_6 = [\texttt{d1} \mapsto 10, \texttt{d2} \mapsto 10, \texttt{v1} \mapsto false, \texttt{v2} \mapsto false]$. These can be used in the action semantics for $a_{\text{getResult}}$ to get the judgement

$$\texttt{mkThreeFBuffered} \vdash (a_{\text{getResult}}, S_5, [t \mapsto \varepsilon]) \downarrow (U_5, 10)$$

where $U_5 = [\texttt{out\_fifo} \mapsto [\texttt{d1} \mapsto 10, \texttt{d2} \mapsto 10, \texttt{v1} \mapsto false, \texttt{v2} \mapsto false]]$. This yields the step semantics

$$\texttt{mkThreeFBuffered} \vdash S_5 \xrightarrow{\texttt{getResult}(\varepsilon)\rightarrow_a 10} S_6$$

and the final execution semantics for this example

$$\texttt{mkThreeFBuffered} \vdash S_0 \bullet\!\!\xrightarrow{\overrightarrow{ex_6}}\!\!\!\!\rightarrow S_6$$

where

$$\begin{aligned} S_6 &= [\texttt{val} \mapsto 9, \texttt{count} \mapsto 0, \\ &\qquad \texttt{out\_fifo} \mapsto [\texttt{d1} \mapsto 10, \texttt{d2} \mapsto 10, \texttt{v1} \mapsto \mathit{false}, \texttt{v2} \mapsto \mathit{false}]] \\ \overrightarrow{ex_6} &= [\texttt{start}(7) \rightarrow_a \varepsilon, \texttt{step}, \texttt{step}, \texttt{step}, \texttt{out\_fifo.canonicalize}, \texttt{getResult}(\varepsilon) \rightarrow_a 10]. \end{aligned}$$

Therefore, from looking at $\overrightarrow{ex_6}$, we see that calling `start(7)` results in `getResult()` returning 10 once it is ready.

5.9 Comparison to Other Rule-Based Languages

A primary goal of rule-based hardware design languages is to make hardware design easier using the rule abstraction.

Spec 'n' Check is designed as a variation on other rule-based HDLs to make verification easier and more efficient. There are four main changes made to Spec 'n' Check when compared to other rule-based HDLs: unambiguous one-method-at-a-time semantics, unguarded expressions, abstract state elements, and a tree-like module hierarchy.

5.9.1 Unambiguous One-Method-at-a-Time Semantics

Spec 'n' Check goes beyond one-rule-at-a-time semantics by adopting the notion of *unambiguous one-method-at-a-time semantics*, meaning that when multiple methods of the same submodule are called from the same rule, they appear to happen in an unambiguous sequential order that is independent of the submodule implementation. Spec 'n' Check does this by limiting rules to firing at most one action method per submodule and forcing value methods to always appear to happen before the action method.

In Chapter 2 (specifically Section 2.4), we showed that other HDLs do not follow one-method-at-a-time semantics and demonstrated issues that arise as a result.

5.9.2 Unguarded Expressions

Spec 'n' Check adopts an expression syntax whose semantics do not have a notion of a guard. Value methods cannot be used in arbitrary expression syntax without first being bound to variables because value methods have guards. If the value method is not ready, then its value cannot be bound to a variable, and therefore the outer rule or method is not ready. In Spec 'n' Check, the only way an expression can affect the guard of an action is through `guard(`e`)` and `if` e `then` a_1 `else` a_2 actions.

In BSV, value method calls are part of the expression syntax, and calls to value methods that are not ready return a sentinel *not-ready* value NR [42]. The presence of an NR value does not immediately cause a rule to be not ready. Instead, a rule only becomes not ready as a result of an NR value if it is written to a register, used as an argument in a method call, or used as the condition in an if statement. Furthermore, if a function gets NR as one of its arguments, it does not imply the function will return NR. This is the case for the ternary operator and the short-circuiting Boolean operators.

This approach to value method calls complicates function abstraction. As an example, consider the following BSV rule:

```
1 rule track_source;
2   let src = select_source(sel, fifo1.first(), fifo2.first());
3   next_src <= src;
4 endrule
```

This rule is only not ready if the return value of `select_source` is NR. If we treat `select_source` as an uninterpreted function, we have to decide what to do with NR values from the calls to `fifo1.first()` and `fifo2.first()`. If we assume `select_source` always returns NR when one of the arguments is NR, then `select_source` is not a correct abstraction for the ternary operator implementation of `select_source`, `sel ? fifo1.first() : fifo2.first()`. With the ternary operator, if `sel` is true, then the guard of `fifo2.first()` does not matter.

On the other hand, if we allow uninterpreted functions to return NR as an option whenever one of the arguments is NR, then the uninterpreted function may introduce more behaviors than desired. For example, if `select_source` is implemented with the ternary operator,

at least one of the `first` methods has to be ready in order for the rule to be fired, but an uninterpreted function can return an arbitrary value depending on only `sel` when both `first` methods are not ready, thus allowing the rule to fire without any of the methods being ready.

Having expressions without guards also makes it easier to have elidable assertions; this is discussed in more detail in Chapter 7.

5.9.3 Abstract State Elements

Spec 'n' Check adopts an abstract notion of state elements that make no assumption about their RTL implementation and allow for non-synthesizable modules to use infinite-sized types (normally for specifications). This is in line with Kami's notion of state, but differs from BSV's.

In BSV, state elements are declared by their RTL implementation, and a state's RTL implementation (`mkReg` vs `mkEHR`) determines how rules can be mapped to clock cycles. If more concurrency is desired, then state declaration and use need to be changed to permit the desired sequence of concurrent rules.

Adopting the abstract notion of state in Spec 'n' Check allows for the verification of a single rule-based description to cover a wide range of RTL implementations, and the presence of state elements with infinite-sized types makes it easier to write specifications of modules. For example, a FIFO can be specified as an infinite-sized buffer with unbounded integers as enqueue and dequeue pointers. This is easier to write as a specification because there is no need for logic to determine what happens when the enqueue and dequeue pointers overflow.

5.10 Conclusion

This chapter presented the grammar and semantics for the Spec 'n' Check rule-based hardware design language. This language differs from other rule-based languages in order to simplify the process of verification and to improve the usefulness of modular refinement during verification.

Chapter 6

Modular Verification

One of the most important aspects of SMT-based model checking of rule-based hardware designs is the ability to substitute modules for simpler versions of themselves during verification to make the verification process easier while still producing verification results that apply to the original modules. We refer to this process as modular verification.

When substituting a module with a new "simpler" module for verification, this new module is a specification that the original module is said to implement. This specification is simpler in terms of verification complexity (typically it has no internal rules), but it can do everything the original module can do in terms of its interface methods. For example, a module that computes an arithmetic function in a pipelined manner may be replaced during verification by a specification that computes the same arithmetic function within a single action and buffers the results using a FIFO.

There are three components of modular verification:

1. Verifying a module implements a specification

2. Verifying a property in a module holds by replacing a submodule with its specification

3. Verifying a property in a module holds when used in a specific context by relating it to its specification used in the same context

This chapter explains how modular verification works by introducing the core concepts and theorems behind it. The two biggest pieces introduced in this chapter are the implements

relation and the modular verification theorem. The implements relation says what it means for a module to implement a specification. The modular verification theorem says if an outer module can get to a certain state or exhibit a certain behavior using a submodule, then it can get to the same state or exhibit the same behavior using a specification for the submodule in its place.

6.1 Informal Approach

Before formally defining the implements relation and stating the related theorems, we first present informally the ideas behind modular verification. This section builds off the idea that the only way a module can interact with a submodule is through the interface methods. Specifically, the only way the outer module can tell a submodule's rule fired or a state changed is through side effects visible through the interface methods.

Therefore interface methods are sufficient for characterizing methods, and we do not need to track submodule rules for modular verification. Thus we can treat rules as silent transitions without labels.

6.1.1 Sequences of Steps with Silent Rules

Let $\Rightarrow$ be the reflexive transitive closure of $\rightarrow$ where all rule labels are dropped from the execution sequences, resulting in just method sequences.

For example, the judgement

$$M \vdash S \stackrel{\varepsilon}{\Rightarrow} S'$$

represents the execution of 0 or more rules to get from state S to state S'.

As another example, consider the judgement

$$M \vdash S \stackrel{\ell_m}{\Rightarrow} S'$$

where ℓ_m represents a label for a single method call. This represents the execution of 0 or more rules starting at state S, then executing the method specified by ℓ_m, and then executing 0 or more rules, finishing at state S'.

This can be extended to multiple method calls. Consider the judgement

$$M \vdash S \stackrel{\vec{m}}{\Rightarrow} S'$$

where $\vec{m}$ is a sequence of method calls. This represents the execution of an arbitrary execution sequence $\vec{ex}$ of rules and methods to get from S to S' where $\mathsf{Methods}(\vec{ex}) = \vec{m}$.

6.1.2 Executions with Silent Rules

We can take this a step further and look at executions of modules ignoring rules. The following is the judgement for an execution of module M starting from reset state S and ending in state S' that executes the methods in $\vec{m}$.

$$M \vdash S \stackrel{\vec{m}}{\bullet\!\!\Rightarrow} S'$$

This judgement is legal if there exists a sequence of rules and methods $\vec{ex}$ such that

$$M \vdash S \stackrel{\vec{ex}}{\bullet\!\!\twoheadrightarrow} S'$$

and $\mathsf{Methods}(\vec{ex}) = \vec{m}$.

6.1.3 Characterizing Modules by Their Behaviors

Again, since the only thing visible to the outside world is its interface methods, we can describe the behaviors of a module as the set of all $\vec{m}$ such that there exist states S and S' such that

$$M \vdash S \stackrel{\vec{m}}{\bullet\!\!\Rightarrow} S'.$$

Now consider an outer module M_{outer} with a submodule M_{sub}. Consider an arbitrary execution of module M_{outer}:

$$M_{\text{outer}} \vdash S_{\text{outer}} \stackrel{\overrightarrow{m_{\text{outer}}}}{\bullet\!\!\Rightarrow} S'_{\text{outer}}$$

This execution implies an execution of its submodule M_{sub}:

$$M_{\text{sub}} \vdash S_{\text{sub}} \overset{\overrightarrow{m_{\text{sub}}}}{\bullet\!\!\Longrightarrow} S'_{\text{sub}}$$

Now consider another submodule M_{sub2} whose behaviors are a subset of the behaviors of M_{sub}. If that is true, then there exists an execution of M_{sub2} with the same sequence of methods $\overrightarrow{m_{\text{sub}}}$, but different states:

$$M_{\text{sub2}} \vdash S_{\text{sub2}} \overset{\overrightarrow{m_{\text{sub}}}}{\bullet\!\!\Longrightarrow} S'_{\text{sub2}}$$

This execution should be able to be used in place of M_{sub}'s execution in the execution of M_{outer} to produce an execution of a new module M_{outer2} but with the same behavior of the old module M_{outer}, specifically:

$$M_{\text{outer2}} \vdash S_{\text{outer2}} \overset{\overrightarrow{m_{\text{outer}}}}{\bullet\!\!\Longrightarrow} S'_{\text{outer2}}$$

Following on from this reasoning, if we take an outer module and replace one of its submodules with a module whose legal behaviors are a subset of the original module's legal behaviors, then the new outer module's behaviors are a subset of the original outer module's behaviors. This is closely related to the modular refinement theorem introduced later in this chapter.

6.1.4 Relating States along with Method Calls

Determining if a module implements a specification by looking at legal sequences of method calls is sufficient for the first two components of modular verification mentioned in the introduction of this chapter (verifying a module implements a specification and verifying a property in a module holds by replacing a submodule with its specification), but it is not sufficient for the third component: verifying a property in a module holds when used in a specific context by relating it to its specification used in the same context. The third component of modular verification requires relating states of related modules.

Relating the states of implementations and specifications is a natural process when per-

forming SMT-based verification. This is because verifying the behaviors of an implementation are a subset of the behaviors of a specification using SMT is done by induction, and a side effect of the induction is a relation between the states of the implementation and the specification. This relation can then be used to perform the third component of modular verification.

Since it is natural to use induction to verify the behaviors of one module are a subset of the behaviors of another module, we choose to define the implements relation as the strongest relation implied by the induction performed using SMT, as presented in the next section.

6.2 Implements Relation

Modular verification requires a notion of a module implementing a specification so the specification can be used in place of the implementation during verification to break it into smaller, more manageable pieces. The *implements relation* is the relation that relates module implementations to specifications. The implements relation is defined using a simulation relation between the states of the implementation and the states of the specification. The possible states of a module M are denoted by the set $\mathsf{States}(M)$.

Definition 6.1 (Implements Relation)**.** Module M_I implements module M_S via simulation relation $R \subseteq \mathsf{States}(M_I) \times \mathsf{States}(M_S)$ if for all $(S_I, S_S) \in R$,

- For all S'_I and method labels ℓ_m such that $M_I \vdash S_I \xrightarrow{\ell_m} S'_I$, there exists S'_S and S''_S such that $M_S \vdash S_S \stackrel{\varepsilon}{\Rightarrow} S''_S$, $M_S \vdash S''_S \xrightarrow{\ell_m} S'_S$, and $(S'_I, S'_S) \in R$.
- For all S'_I and rule labels ℓ_r such that $M_I \vdash S_I \xrightarrow{\ell_r} S'_I$, there exists S'_S such that $M_S \vdash S_S \stackrel{\varepsilon}{\Rightarrow} S'_S$ and $(S'_I, S'_S) \in R$.
- For all M_I reset states $S_{I,0}$, there exists a M_S reset state $S_{S,0}$ such that $(S_{I,0}, S_{S,0}) \in R$.

M_I implements M_S is written $M_I \sqsubseteq M_S$.

An important result of the implements relation is the following theorem:

Theorem 6.1 (Implements Relation Theorem)**.** *If $M_I \sqsubseteq M_S$ via simulation relation R, then for all executions $M_I \vdash S_I \bullet\!\!\xrightarrow{\overrightarrow{ex_I}}\!\!\!\!\rightarrow S'_I$, there exist S_S, S'_S, and $\overrightarrow{ex_S}$ such that $M_S \vdash S_S \bullet\!\!\xrightarrow{\overrightarrow{ex_S}}\!\!\!\!\rightarrow S'_S$,*

$(S_I, S_S) \in R$, $(S'_I, S'_S) \in R$, *and the sequence of method calls in* $\overrightarrow{ex_I}$ *matches the sequence of method calls in* $\overrightarrow{ex_S}$.

Proof. This is proven by induction on $\overrightarrow{ex_I}$.

- First consider the case where $\overrightarrow{ex_I}$ is the empty sequence of labels ε. In this case $M_I \vdash S_I \overset{\varepsilon}{\bullet\!\!\twoheadrightarrow} S_I$, so therefore S_I must be an initial state of M_I. The definition of $M_I \sqsubseteq M_S$ implies that there exists an initial state S_S of M_S such that $(S_I, S_S) \in R$. Since S_S is an initial state of M_S, $M_S \vdash S_S \overset{\varepsilon}{\bullet\!\!\twoheadrightarrow} S_S$ holds, and trivially the methods in the two execution sequences (both ε) match.

- Now consider the induction step where the theorem holds for $\overrightarrow{ex_I}$ and we are proving it holds for $\overrightarrow{ex_I}\,\ell$ for some label ℓ. Assume $M_I \vdash S_I \overset{\overrightarrow{ex_I}}{\bullet\!\!\twoheadrightarrow} S'_I$ and $M_S \vdash S_S \overset{\overrightarrow{ex_S}}{\bullet\!\!\twoheadrightarrow} S'_S$ where $(S_I, S_S) \in R$, $(S'_I, S'_S) \in R$, and the methods of $\overrightarrow{ex_I}$ match the methods of $\overrightarrow{ex_S}$. Let $M_I \vdash S'_I \xrightarrow{\ell} S''_I$.

  - If ℓ is a method label, then $M_I \sqsubseteq M_S$ implies that there exists S''_S and S'''_S such that $M_S \vdash S'_S \overset{\varepsilon}{\Rightarrow} S'''_S$ and $M_S \vdash S'''_S \xrightarrow{\ell} S''_S$ with $(S''_I, S''_S) \in R$. That gives new execution sequences $M_I \vdash S_I \overset{\overrightarrow{ex_I}\,\ell}{\bullet\!\!\twoheadrightarrow} S''_I$ and $M_S \vdash S_S \overset{\overrightarrow{ex_S}\,\overrightarrow{ex'_S}\,\ell}{\bullet\!\!\twoheadrightarrow} S''_S$ where $\overrightarrow{ex'_S}$ is the rules corresponding to the step $M_S \vdash S'_S \overset{\varepsilon}{\Rightarrow} S'''_S$. The method sequences of these two executions are equal to the method sequences of $\overrightarrow{ex_I}$ and $\overrightarrow{ex_S}$ with ℓ appended at the end, so they have equal method calls in the execution sequences.
  - If ℓ is a rule label, then $M_I \sqsubseteq M_S$ implies that there exists S''_S such that $M_S \vdash S'_S \overset{\varepsilon}{\Rightarrow} S''_S$ with $(S''_I, S''_S) \in R$. That gives new execution sequences $M_I \vdash S_I \overset{\overrightarrow{ex_I}\,\ell}{\bullet\!\!\twoheadrightarrow} S''_I$ and $M_S \vdash S_S \overset{\overrightarrow{ex_S}\,\overrightarrow{ex'_S}}{\bullet\!\!\twoheadrightarrow} S''_S$ where $\overrightarrow{ex'_S}$ is the rules corresponding to the step $M_S \vdash S'_S \overset{\varepsilon}{\Rightarrow} S''_S$. The method sequences of these two executions are equal to the method sequences of $\overrightarrow{ex_I}$ and $\overrightarrow{ex_S}$, so they have equal method calls in the execution sequences.

  Therefore the theorem holds for $\overrightarrow{ex_I}\,\ell$, and combining with the induction base case, the theorem holds for all $\overrightarrow{ex_I}$.

□

A useful applications of this theorem is that if $M_I \sqsubseteq M_S$ via simulation relation R, then for each reachable state S_I of M_I, there exists a reachable state S_S of M_S such that $(S_I, S_S) \in R$, and the sequence of method calls used to get to S_I is equivalent to the sequence of method calls used to get to S_S.

6.3 Module Context

In order to describe how a module that implements a specification can be substituted by its specification, we need a way to be able to talk about modules with replaceable submodules and executions of such modules. We do this by introducing the notion of a *module context*, *i.e.* a module with a hole in place of a submodule, and then consider the behavior of the context when the implementation and the specification are plugged into the hole.

A module context has the same syntax as a normal module, except it has exactly one submodule instance statement where the module definition is a hole. This is written as

$$\texttt{instance}\ m\ \square$$

where m is the instance corresponding to the hole. The actions within the rules and methods of the module context can call methods of m just like a normal module, but since there is no module definition associated with the hole, the result of the method call cannot be known for sure until a module definition is plugged into the hole.

We write a module context as $M[\,]$ where the empty brackets denote a hole that a module definition can be plugged into. If the submodule M_H is used to fill the hole in $M[\,]$, the resulting module is written $M[M_H]$, and the $\square$ in the hole instance definition is replaced by M_H so the module no longer has a hole in it. The semantics of $M[M_H]$ are the same as any module as shown in Chapter 5.

When working with the semantics of $M[M_H]$, we can split state values of $M[M_H]$ into separate context state and hole state using square bracket notation that mimics $M[M_H]$. When we write that $S[S_H]$ is a state of $M[M_H]$, it is assumed that S represents the context state (the state of $M[\,]$) and S_H represents the hole state (the state of M_H). Therefore the

step-semantics judgement

$$M[M_H] \vdash S[S_H] \bullet\!\!\xrightarrow{\vec{ex}}\!\!\!\!\rightarrow S'[S'_H]$$

means that $M[M_H]$ admits an execution starting at initial state $S[S_H]$, executing the rules and methods specified in $\vec{ex}$, and resulting in the final state $S'[S'_H]$, where $S[\,]$ and $S'[\,]$ are the initial and final state of $M[\,]$, and S_H and S'_H are the initial and final state of M_H. The rules and methods in $\vec{ex}$ are no different from how they are defined in Chapter 5 for the module $M[M_H]$.

6.4 Modular Verification Theorem

Now that we have defined a module context, we can look at the relation of modules $M[M_I]$ and $M[M_S]$ provided $M_I \sqsubseteq M_S$. This gives us the Modular Verification Theorem stated below.

Theorem 6.2 (Modular Verification Theorem)**.** *Given $M_I \sqsubseteq M_S$ via simulation relation R, then for all contexts $M[\,]$, $M[M_I] \sqsubseteq M[M_S]$ via simulation relation*

$$\{(S[S_I], S[S_S]) \mid (S_I, S_S) \in R,\ S \in \mathsf{States}(M[\,])\}\,.$$

In order to prove the Modular Verification Theorem, it is useful to use semantics for module contexts that are independent of the hole module implementation. The next section presents semantics for module contexts, and then the Modular Verification Theorem is proven in Section 6.6.

6.5 Modular Semantics

We want to describe the possible behaviors of a module context without specifying a module definition for the hole submodule. This means we need a way to describe the semantics of the context when it fires a rule or method that calls a method of the hole.

We handle hole method calls using *hole assumption labels*. Hole assumption labels have the same structure as step labels, but hole assumption label are used in modular semantics

to represent what assumptions are made about a hole module within an action or during a step. When a hole method is called from an action, it is unknown what value it will return. The semantics are free to choose any return value provided it returns a hole assumption label representing the assumption it made about the hole method call's return value.

The notion of hole assumption labels is related to Kami's labels for called methods [33], but unlike in Kami, hole assumption labels only refer to methods of one module and are kept separate from the rest of the method labels.

For example, if an action calls hole action method f_a with argument 5, the action semantics can choose any return value y provided it also returns the corresponding hole assumption label $f_a(5) \rightarrow_a y$.

This gives each step transition two labels: one corresponding to the fired rule or method of the context module, and the other is the hole assumption label. The judgement for a step of a module context is

$$M[\,] \vdash S[\,] \xrightarrow[\ell_H]{\ell} S'[\,]$$

where $M[\,]$ is the module context, $S[\,]$ is the initial state with a placeholder for the initial state of the hole, ℓ is the rule or method of $M[\,]$ being executed, ℓ_H is the hole assumption label, and $S[\,]$ is the final state with a placeholder for the final state for the hole.

The modular semantics are designed to have the property that when a module has a step transition with the same label as a hole assumption label, then the step semantics can be "plugged into" the modular semantics to produce step semantics for a module context with a filled-in hole.

For example, if there exists a module M_H with the step judgement

$$M_H \vdash S_H \xrightarrow{\ell_H} S'_H,$$

then this step judgement of M_H can be combined with the step judgement of $M[\,]$ from above to produce the step judgement of $M[M_H]$ below:

$$M[M_H] \vdash S[S_H] \xrightarrow{\ell} S'[S'_H].$$

This property of modular semantics is proven later in this section.

6.5.1 Modular Action Semantics

The modular action semantics are very similar to the original action semantics, except for the presence of hole method calls and hole assumption labels. A hole method call can return any arbitrary value and produce a corresponding hole assumption label. All other actions propagate hole assumption labels as necessary.

If actions depend on the semantics of two subactions, the corresponding hole assumption labels are joined using the $\oplus$ operator from Chapter 5. This operator combines a pair of labels into a single label as long as each label is either a method label or an empty label, and there is at most one action method between the two labels.

The modified action semantics for a module context take the form

$$M[\,] \vdash (a, S[\,], B) \downarrow (U, v, \ell_H)$$

which matches the original action semantics except for the hole assumption label ℓ_H. The updated action semantics are shown in Figure 6-1.

6.5.2 Modular Step Semantics

For the most part, the modular step semantics follow directly from using the modular action semantics in the original step semantics and putting the corresponding action's hole assumption label below the arrow in the modular step judgement. The only modular step semantics that does not follow that pattern is executing a rule inside a hole.

When executing a rule inside a hole, the action of the rule is unknown, and the state change of the rule would be limited to the hole, so there is not much to do. The only thing to do for hole rules is to specify in the hole assumption label that the hole fired the rule.

The full modular step semantics are shown in Figure 6-2. Note that $S[\,][U]$ is the context state $S[\,]$ updated using the update map U.

state-wr

$$\frac{[\![e]\!](S[\,],B) = v}{M[\,] \vdash (\langle s \texttt{ <= } e\rangle, S[\,], B) \downarrow ([s \mapsto v], \varepsilon, \varepsilon)}$$

if-true

$$\frac{[\![e]\!](S[\,],B) = True \qquad M[\,] \vdash (a_1, S[\,], B) \downarrow (U, v, \ell)}{M[\,] \vdash (\langle \texttt{if } e \texttt{ then } a_1 \texttt{ else } a_2\rangle, S[\,], B) \downarrow (U, v, \ell)}$$

if-false

$$\frac{[\![e]\!](S[\,],B) = False \qquad M[\,] \vdash (a_2, S[\,], B) \downarrow (U, v, \ell)}{M[\,] \vdash (\langle \texttt{if } e \texttt{ then } a_1 \texttt{ else } a_2\rangle, S[\,], B) \downarrow (U, v, \ell)}$$

guard

$$\frac{[\![e]\!](S[\,],B) = True}{M[\,] \vdash (\langle \texttt{guard(}e\texttt{)}\rangle, S[\,], B) \downarrow ([\,], \varepsilon, \varepsilon)}$$

par

$$\frac{M[\,] \vdash (a_1, S[\,], B) \downarrow (U_1, v_1, \ell_1) \qquad M[\,] \vdash (a_2, S[\,], B) \downarrow (U_2, v_2, \ell_2)}{M[\,] \vdash (\langle a_1 \texttt{ ; } a_2\rangle, S[\,], B) \downarrow (U_1 \uplus U_2, v_2, \ell_1 \oplus \ell_2)}$$

a-let

$$\frac{\begin{array}{c} M[\,] \vdash (a_1, S[\,], B) \downarrow (U_1, v_1, \ell_1) \\ M[\,] \vdash (a_2, S[\,], B[t \mapsto v_1]) \downarrow (U_2, v_2, \ell_2) \end{array}}{M[\,] \vdash (\langle \texttt{let } t \texttt{ = } a_1 \texttt{ in } a_2\rangle, S[\,], B) \downarrow (U_1 \uplus U_2, v_2, \ell_1 \oplus \ell_2)}$$

expr

$$\frac{[\![e]\!](S[\,],B) = v}{M[\,] \vdash (\langle e\rangle, S[\,], B) \downarrow ([\,], v, \varepsilon)}$$

action-method-call (submodule)

$$\frac{\begin{array}{c} \langle \texttt{instance } m \;\; M_{\mathsf{sub}}\rangle \in \mathsf{SubmodInsts}(M[\,]) \\ [\![e]\!](S[\,],B) = x \qquad M_{\text{sub}} \vdash S[\,](m) \xrightarrow{f_a(x) \to_a y} \Omega \end{array}}{M[\,] \vdash (\langle m.f_a(e)\rangle, S[\,], B) \downarrow ([m \mapsto \Omega], y, \varepsilon)}$$

value-method-call (submodule)

$$\frac{\begin{array}{c} \langle \texttt{instance } m \;\; M_{\mathsf{sub}}\rangle \in \mathsf{SubmodInsts}(M[\,]) \\ [\![e]\!](S[\,],B) = x \qquad M_{\text{sub}} \vdash S[\,](m) \xrightarrow{f_v(x) \to_v y} S[\,](m) \end{array}}{M[\,] \vdash (\langle m.f_v(e)\rangle, S[\,], B) \downarrow ([\,], y, \varepsilon)}$$

action-method-call (hole)

$$\frac{\langle \texttt{instance } m \;\; \square\rangle \in \mathsf{SubmodInsts}(M[\,]) \qquad [\![e]\!](S[\,],B) = x}{M[\,] \vdash (\langle m.f_a(e)\rangle, S[\,], B) \downarrow ([\,], y, f_a(x) \to_a y)}$$

value-method-call (hole)

$$\frac{\langle \texttt{instance } m \;\; \square\rangle \in \mathsf{SubmodInsts}(M[\,]) \qquad [\![e]\!](S[\,],B) = x}{M[\,] \vdash (\langle m.f_v(e)\rangle, S[\,], B) \downarrow ([\,], y, f_v(x) \to_v y)}$$

Figure 6-1: Modular action semantics judgements

6.5.3 Modular Execution Semantics

We define the judgement for the execution of a module context as

$$M[\,] \vdash S[\,] \bullet \xrightarrow[\overrightarrow{ex_H}]{\overrightarrow{ex}} S'[\,]$$

$$\text{context-rule} \quad \frac{\langle \texttt{rule}\ r\ =\ a\rangle \in \mathsf{Rules}(M[\,]) \qquad M[\,] \vdash (a, S[\,], [\,]) \downarrow (U, v, \ell_H)}{M[\,] \vdash S[\,] \xrightarrow[\ell_H]{r} S[\,][U]}$$

$$\text{context-action-method} \quad \frac{\langle \texttt{amethod}\ f_a\ =\ \lambda t.a\rangle \in \mathsf{Methods}(M[\,]) \qquad M[\,] \vdash (a, S[\,], [t \mapsto x]) \downarrow (U[\,], y, \ell_H)}{M[\,] \vdash S[\,] \xrightarrow[\ell_H]{f_a(x) \to_a y} S[\,][U]}$$

$$\text{context-value-method} \quad \frac{\langle \texttt{vmethod}\ f_v\ =\ \lambda t.a\rangle \in \mathsf{Methods}(M[\,]) \qquad M[\,] \vdash (a, S[\,], [t \mapsto x]) \downarrow ([\,], y, \ell_H)}{M[\,] \vdash S[\,] \xrightarrow[\ell_H]{f_v(x) \to_v y} S[\,]}$$

$$\text{context-submodule-rule} \quad \frac{\langle \texttt{instance}\ m'\ M'\rangle \in \mathsf{SubmodInsts}(M) \qquad M' \vdash S[\,](m') \xrightarrow{r} \Omega}{M[\,] \vdash S[\,] \xrightarrow[\varepsilon]{m'.r} S[\,][m' \mapsto \Omega]}$$

$$\text{hole-rule} \quad \frac{\langle \texttt{instance}\ m\ \square\rangle \in \mathsf{SubmodInsts}(M)}{M[\,] \vdash S[\,] \xrightarrow[r]{m.r} S[\,]}$$

$$\text{context-concurrent-methods} \quad \frac{M \vdash S \xrightarrow[\ell_{H,1}]{\ell_1} S \qquad \ell_1 \text{is a value method} \qquad M \vdash S \xrightarrow[\ell_{H,2}]{\ell_2} S' \qquad \ell_2 \text{is one or more methods}}{M \vdash S \xrightarrow[\ell_{H,1} \oplus \ell_{H,2}]{\ell_1 \oplus \ell_2} S'}$$

Figure 6-2: Modular step semantics

where the hole assumption sequence $\overrightarrow{ex_H}$ is the sequence of hole assumption labels corresponding to the execution sequence $\overrightarrow{ex}$. The modular execution semantics can be seen in Figure 6-3.

6.5.4 Applying Modular Semantics

As mentioned before, the modular semantics are designed to have the property that when a module has a step transition with the same label as a hole assumption label from modular step semantics, the step semantics can be "plugged into" the modular semantics to produce step semantics for a module context with a filled-in hole. The following lemma and theorem

$$\text{context-reset} \quad \frac{\langle \texttt{reset}\ e\rangle = \mathsf{Reset}(M) \qquad \mathcal{E}[\![e]\!](S[\,],\emptyset) = True}{M[\,] \vdash S[\,] \bullet \xrightarrow[\varepsilon]{\varepsilon} S[\,]}$$

$$\text{context-step} \quad \frac{M[\,] \vdash S[\,] \bullet \xrightarrow[\vec{ex}_H]{\vec{ex}} S'[\,] \qquad M[\,] \vdash S'[\,] \xrightarrow[\ell_H]{\ell} S''[\,]}{M[\,] \vdash S[\,] \bullet \xrightarrow[\vec{ex}_H\ \ell_H]{\vec{ex}\ \ell} S''[\,]}$$

Figure 6-3: Modular execution semantics

are used to prove this property.

Lemma 6.3. *Let $M[\,]$ be a module context with a hole for submodule instance m, and let M_H be a module that can be plugged in the hole of $M[\,]$. If $M[\,] \vdash (a, S[\,], B) \downarrow (U, v, \ell_H)$ and $M_H \vdash S_H \xrightarrow{\ell_H} S'_H$, then*

- *if ℓ_H contains an action method, then $M[M_H] \vdash (a, S[S_H], B) \downarrow (U[m \mapsto S'_H], v)$.*
- *if ℓ_H does not contain an action method, then $M[M_H] \vdash (a, S[S_H], B) \downarrow (U, v)$.*

Proof. Let $M[\,] \vdash (a, S[\,], B) \downarrow (U, v, \ell_H)$ and $M_H \vdash S_H \xrightarrow{\ell_H} S'_H$. We want to show that if ℓ_H contains an action method, then $M[M_H] \vdash (a, S[S_H], B) \downarrow (U[m \mapsto S'_H], v)$, otherwise $M[M_H] \vdash (a, S[S_H], B) \downarrow (U, v)$. This theorem is proved by structural induction over the action a.

- $(a = \langle s\ \texttt{<=}\ e\rangle)$ - Since $M[\,] \vdash (\langle s\ \texttt{<=}\ e\rangle, S[\,], B) \downarrow (U, v, \ell_H)$, the modular action semantics gives $U = [\texttt{s} \mapsto [\![e]\!](S[\,], B)]$, $v = \varepsilon$, and $\ell_H = \varepsilon$. From the normal action semantics, $M[M_H] \vdash (\langle s\ \texttt{<=}\ e\rangle, S[S_H], B) \downarrow (\texttt{s} \mapsto [\![e]\!](S[S_H], B)], \varepsilon)$, so the theorem holds for this case.
- $(a = \langle \texttt{if}\ e\ \texttt{then}\ a_1\ \texttt{else}\ a_2\rangle)$ - The semantics of this action is either equal to the semantics of a_1 or the semantics of a_2, so this case holds nearly directly from the structural induction hypothesis.
- $(a = \langle \texttt{guard(}e\texttt{)}\rangle)$ - Since $M[\,] \vdash (\langle \texttt{guard(}e\texttt{)}\rangle, S[\,], B) \downarrow (U, v, \ell_H)$, the modular action semantics gives $U = [\,]$, $v = \varepsilon$, and $\ell_H = \varepsilon$, and it also implies that $[\![e]\!](S[\,], B) = True$. Since $[\![e]\!](S[\,], B) = True$ we get that $[\![e]\!](S[S_H], B) = True$ as well. From

the normal action semantics, and the fact that $[\![e]\!](S[S_H], B) = True$, we get that $M[M_H] \vdash (\langle \texttt{guard(}e\texttt{)} \rangle, S[S_H], B) \downarrow ([\,], \varepsilon)$, so the theorem holds for this case.

- $(a = \langle a_1 \ \texttt{;} \ a_2 \rangle)$ - Let $M[\,] \vdash (a_1, S[\,], B) \downarrow (U_1, v_1, \ell_{H,1})$ and let $M[\,] \vdash (a_2, S[\,], B) \downarrow (U_2, v_2, \ell_{H,2})$. Therefore $M[\,] \vdash (\langle a_1 \ \texttt{;} \ a_2 \rangle, S[\,], B) \downarrow (U_1 \uplus U_2, v_2, \ell_{H,1} \oplus \ell_{H,2})$, and thus $U = U_1 \uplus U_2$, $v = v_2$, and $\ell_H = \ell_{H,1} \oplus \ell_{H,2}$.

  Assume $\ell_{H,1} \oplus \ell_{H,2}$ has an action method in it, therefore exactly one of $\ell_{H,1}$ or $\ell_{H,2}$ has an action method in it. Assume without loss of generality that $\ell_{H,2}$ has an action method in it. Therefore $M_H \vdash S_H \xrightarrow{\ell_{H,1} \oplus \ell_{H,2}} S'_H$ implies that $M_H \vdash S_H \xrightarrow{\ell_{H,1}} S_H$ and $M_H \vdash S_H \xrightarrow{\ell_{H,2}} S'_H$.

  The structural induction hypothesis implies that $M[M_H] \vdash (a_1, S[S_H], B) \downarrow (U_1, v_1, \ell_{H,1})$ and $M[M_H] \vdash (a_2, S[S_H], B) \downarrow (U_2[m \mapsto S'_H], v_2, \ell_{H,2})$. Using these two with the normal action semantics gives $M[M_H] \vdash (\langle a_1 \ \texttt{;} \ a_2 \rangle, S[S_H], B) \downarrow (U_1 \uplus U_2[m \mapsto S'_H], v_2)$, and therefore the theorem holds for this case.

  If $\ell_{H,1} \oplus \ell_{H,2}$ has no action method in it, the proof is similar except for the judgements $M_H \vdash S_H \xrightarrow{\ell_{H,2}} S_H$ and $M[M_H] \vdash (a_2, S[S_H], B) \downarrow (U_2, v_2, \ell_{H,2})$, and the resulting semantics are $M[M_H] \vdash (\langle a_1 \ \texttt{;} \ a_2 \rangle, S[S_H], B) \downarrow (U_1 \uplus U_2, v_2)$.

- $(a = \langle \texttt{let } t \texttt{ = } a_1 \texttt{ in } a_2 \rangle)$ - This case is nearly identical to $a_1 \ \texttt{;} \ a_2$ except the bindings of the semantics for a_2 include the value from the semantics of a_1.

- $(a = \langle e \rangle)$ - Since $M[\,] \vdash (\langle e \rangle, S[\,], B) \downarrow (U, v, \ell_H)$, the modular action semantics gives $U = [\,]$, $v = [\![e]\!](S[\,], B)$, and $\ell_H = \varepsilon$. Furthermore $v = [\![e]\!](S[\,], B)$ gives that $v = [\![e]\!](S[S_H], B)$. From the normal action semantics and $v = [\![e]\!](S[\,], B)$ we get that $M[M_H] \vdash (\langle e \rangle, S[S_H], B) \downarrow ([\,], [\![e]\!](S[S_H], B))$, so the theorem holds for this case.

- $(a = \langle m' \,.\, f_a\texttt{(}e\texttt{)} \rangle$ where $m' \neq m)$ - Since $M[\,] \vdash (\langle m' \,.\, f_a\texttt{(}e\texttt{)} \rangle, S[\,], B) \downarrow (U, v)$, we know that for some x, $[\![e]\!](S[\,], B) = x$. We also get that there exist y and Ω such that $M' \vdash S[\,](m') \xrightarrow{f_a(x) \to_a y} \Omega$. Furthermore this gives $U = [m' \mapsto \Omega]$ and $v = y$.

  Using the normal action semantics, $M[M_H] \vdash (\langle m' \,.\, f_a\texttt{(}e\texttt{)} \rangle, S[S_H], B) \downarrow ([m' \to \Omega], y)$, and therefore the theorem holds in this case.

- $(a = \langle m' \,.\, f_e(e) \rangle$ where $m' \neq m)$ - This case is nearly identical to the case above except $U = [\,]$.

- $(a = \langle m \,.\, f_a(e) \rangle)$ - Since $M[\,] \vdash (\langle m \,.\, f_a(e) \rangle, S[\,], B) \downarrow (U, v, \ell_H)$, we know that there exist x and y such that $[\![e]\!](S[\,], B) = x$, $v = y$, and $\ell_H = f_a(x) \rightarrow_a y$. Furthermore, we get that $U = [\,]$.

  We can substitute the value of ℓ_H in the assumed M_H step judgement to get $M_H \vdash S_H \xrightarrow{f_a(x) \rightarrow_a y} S'_H$.

  Using the normal action semantics, $M[M_H] \vdash (\langle m \,.\, f_a(e) \rangle, S[S_H], B) \downarrow ([m \mapsto S'_H], y)$, so the theorem holds in this case.

- $(a = \langle m \,.\, f_e(e) \rangle)$ - This case is proved in a nearly identical manner as the previous case.

Since the structural induction holds for all cases, this theorem is true for all actions. □

Theorem 6.4 (Modular Semantics Application)**.** *Let $M[\,]$ be a module context with a hole for submodule instance m, and let M_H be a module that can be plugged into $M[\,]$. If*

$$M[\,] \vdash S[\,] \xrightarrow[\ell_H]{\ell} S'[\,]$$

is a legal modular step judgement and

$$M_H \vdash S_H \xrightarrow{\ell_H} S'_H$$

is a legal step judgement, then the combination of the two:

$$M[M_H] \vdash S[S_H] \xrightarrow[\ell_H]{\ell} S'[S'_H]$$

is a legal step judgement.

Proof. Let $M[\,] \vdash S[\,] \xrightarrow[\ell_H]{\ell} S'[\,]$ and $M_H \vdash S_H \xrightarrow{\ell_H} S'_H$. We prove this theorem by structural induction over the label ℓ. This includes six cases: ℓ is an action method, ℓ is a value

method, ℓ is a compound method label, ℓ is a context rule, ℓ is a hole rule, or ℓ is a context submodule rule. In the case that ℓ is a compound method label $\ell_1 \oplus \ell_2$, because of the structural induction, we assume the theorem holds for ℓ_1 and ℓ_2 when proving it holds for ℓ.

- $(\ell = f_a(x) \to_a y)$ - Let action method f_a be the action $\lambda t.a$. Since $M[\,] \vdash S[\,] \xrightarrow[\ell_H]{f_a(x) \to y} S'[\,]$, therefore $M[\,] \vdash (a, S[\,], [t \mapsto x]) \downarrow (U, y, \ell_H)$ where $S[\,][U] = S'[\,]$. Using the previous lemma along with $M_H \vdash S_H \xrightarrow{\ell_H} S'_H$ gives $M \vdash (a, S[S_H], [t \mapsto x]) \downarrow (U[m \mapsto S'_H], y)$. From that, $M[M_H] \vdash S[S_H] \xrightarrow{f_a(x) \to_a y} S'[S'_H]$.

- $(\ell = f_v(x) \to_v y)$ - Let value method f_v be the action $\lambda t.a$. Since $M[\,] \vdash S[\,] \xrightarrow[\ell_H]{f_v(x) \to_v y} S[\,]$, therefore $M \vdash (a, S[\,], [t \mapsto x]) \downarrow ([\,], y, \ell_H)$. Since f_v is a value method, it can only call other value methods, therefore ℓ_H is made up of only value methods. Therefore in the step judgement of M_H, $S'_H = S_H$, so $M_H \vdash S_H \xrightarrow{\ell_H} S_H$. Using the previous lemma along with $M_H \vdash S_H \xrightarrow{\ell_H} S_H$ gives $M \vdash (a, S[S_H], [t \mapsto x]) \downarrow ([\,], y)$. From that, $M[M_H] \vdash S[S_H] \xrightarrow{f_v(x) \to_v y} S[S_H]$.

- $(\ell = \ell_1 \oplus \ell_2)$ - By the structural induction hypothesis, we assume that the theorem holds for ℓ_1 and ℓ_2.

  Assume that ℓ contains an action method, and assume without loss of generality that ℓ_2 contains the action method. From the step semantics of $M[\,]$, there exist transitions $M[\,] \vdash S[\,] \xrightarrow[\ell_{H,1}]{\ell_1} S[\,]$ and $M[\,] \vdash S[\,] \xrightarrow[\ell_{H,2}]{\ell_2} S'[\,]$ such that $\ell_H = \ell_{H,1} \oplus \ell_{H,2}$.

  From this, we can take $M_H \vdash S_H \xrightarrow{\ell_{H,1} \oplus \ell_{H,2}} S'_H$ and split it into $M_H \vdash S_H \xrightarrow{\ell_{H,1}} S_H$ and $M_H \vdash S_H \xrightarrow{\ell_{H,2}} S'_H$. Finally, applying the induction hypothesis gives us $M[M_H] \vdash S[S_H] \xrightarrow{\ell_1} S[S_H]$ and $M[M_H] \vdash S[S_H] \xrightarrow{\ell_2} S'[S'_H]$, and putting these together gives $M[M_H] \vdash S[S_H] \xrightarrow{\ell_1 \oplus \ell_2} S'[S'_H]$.

  The case where ℓ does not contain an action method is proven in a nearly identical manner as the previous case.

- $(\ell = r)$ - Let rule r be the action a. Since $M[\,] \vdash S[\,] \xrightarrow[\ell_H]{r} S'[\,]$, therefore $M[\,] \vdash (a, S[\,], [\,]) \downarrow (U, v, \ell_H)$ where $S[\,][U] = S'[\,]$. Using the previous lemma along with $M_H \vdash S_H \xrightarrow{\ell_H} S'_H$, we get $M[M_H] \vdash (a, S[S_H], [\,]) \downarrow (U[m \mapsto S'_H], v)$. From that, $M[M_H] \vdash S[S_H] \xrightarrow{r} S'[S'_H]$.

- ($\ell = m.r$) - Since $\ell = m.r$ is a hole rule, then $\ell_H = r$, so we have $M[\,] \vdash S[\,] \xrightarrow[r]{m.r} S[\,]$. If we have $M_H \vdash S_H \xrightarrow{r} S'_H$, then that means we have $M[M_H] \vdash S[S_H] \xrightarrow{m.r} \Omega$ where $\Omega = S[S_H][m \mapsto S'_H]$. Ω can be simplified to $S[S'_H]$, so from that we get $M[M_H] \vdash S[S_H] \xrightarrow{m.r} S[S'_H]$.

- ($\ell = m'.r$ where $m' \neq m$) - Since ℓ is a transition for a rule of a submodule that is not the hole, the submodule is unable to call methods on the hole, so ℓ_H must be the empty label ε, and thus we have $M[\,] \vdash S[\,] \xrightarrow[\varepsilon]{m'.r} S'[\,]$. Since $\ell_H = \varepsilon$, the step judgement for M_H must have $S'_H = S_H$ so $M_H \vdash S_H \xrightarrow{\varepsilon} S_H$. Looking at the semantics of $M[M_H]$, we get $M[M_H] \vdash S[S_H] \xrightarrow{m'.r} S'[S_H]$.

Since the structural induction holds for all cases, the theorem is true. □

6.5.5 Separating Semantics into Modular and Hole Semantics

The opposite action of the previous theorem is separating semantics of a module $M[M_H]$ into the modular semantics of $M[\,]$ and the semantics of M_H. This process is shown in the following theorem and is useful in proofs that use modular semantics.

Theorem 6.5 (Modular Semantics Separation)**.** *For all steps $M[M_H] \vdash S[S_H] \xrightarrow{\ell} S'[S'_H]$, there exists a ℓ_H such that $M[\,] \vdash S[\,] \xrightarrow[\ell_H]{\ell} S'[\,]$ and $M_H \vdash S_H \xrightarrow{\ell_H} S'_H$.*

Proof. This theorem is proven by applying the modular semantics of $M[\,]$ in tandem with the semantics of $M[M_H]$ and using the semantics of M_H to get the labels that make up ℓ_H. □

6.6 Proving the Modular Verification Theorem

Now that we have the modular semantics and associated theorems, we can use them to prove the modular verification theorem:

> Given $M_I \sqsubseteq M_S$ via simulation relation R, for all contexts $M[\,]$, $M[M_I] \sqsubseteq M[M_S]$ via simulation relation $\{(S[S_I], S[S_S]) \mid (S_I, S_S) \in R,\ S \in \mathsf{States}(M[\,])\}$.

Proof. Assume M_I and M_S are modules such that $M_I \sqsubseteq M_S$ via simulation relation R, and let $M[\,]$ be an arbitrary module context. We are going to show that

$$R' = \{(S[S_I], S[S_S]) \mid (S_I, S_S) \in R,\ S \in \mathsf{States}(M[\,])\}$$

is a simulation relation that makes $M[M_I] \sqsubseteq M[M_S]$.

Let $(S[S_I], S[S_S])$ be an arbitrary element in R', and therefore $(S_I, S_S) \in R$. Let $S'[S'_I]$ be a state of $M[M_I]$ and let ℓ be a label such that $M[M_I] \vdash S[S_I] \xrightarrow{\ell} S'[S'_I]$. Using Theorem 6.5, let ℓ_H be the label such that $M[\,] \vdash S[\,] \xrightarrow[\ell_H]{\ell} S'[\,]$ and $M_I \vdash S_I \xrightarrow{\ell_H} S'_I$. The proof splits into cases depending on the type of ℓ_H:

- (ℓ_H is a method label) Since $M_I \sqsubseteq M_S$ via simulation relation R, there exist S'_S and S''_S such that $M_S \vdash S_S \overset{\varepsilon}{\Rightarrow} S''_S$, $M_S \vdash S''_S \xrightarrow{\ell_H} S'_S$, and $(S'_I, S'_S) \in R$.

  The execution sequence $M_S \vdash S_S \overset{\varepsilon}{\Rightarrow} S''_S$ corresponds to a sequence of rules of M_S, so each rule step can be combined with $M[\,] \vdash S[\,] \xrightarrow[r]{m.r} S[\,]$ where r is the name of each rule to produce $M[M_S] \vdash S[S_S] \overset{\varepsilon}{\Rightarrow} S[S''_S]$.

  Next, the step $M_S \vdash S''_S \xrightarrow{\ell_H} S'_S$ can be combined with $M[\,] \vdash S[\,] \xrightarrow[\ell_H]{\ell} S'[\,]$ using Theorem 6.4 to get $M[M_S] \vdash S[S''_S] \xrightarrow{\ell} S'[S'_S]$.

  Finally, since $(S'_I, S'_S) \in R$, therefore $(S'[S'_I], S'[S'_S]) \in R'$.

- (ℓ_H is a rule label) Since $M_I \sqsubseteq M_S$ via simulation relation R, there exist S'_S such that $M_S \vdash S_S \overset{\varepsilon}{\Rightarrow} S'_S$ and $(S'_I, S'_S) \in R$.

  The execution sequence $M_S \vdash S_S \overset{\varepsilon}{\Rightarrow} S'_S$ corresponds to a sequence of rules of M_S, so each rule step can be combined with $M[\,] \vdash S[\,] \xrightarrow[r]{m.r} S[\,]$, where r is the name of each rule, to produce $M[M_S] \vdash S[S_S] \overset{\varepsilon}{\Rightarrow} S[S'_S]$.

  Since ℓ_H is a rule label, ℓ must be a hole rule label making S' equal to S according to the hole rule semantics. That means the original $M[M_I]$ step can be written $M[M_I] \vdash S[S_I] \xrightarrow{\ell} S[S'_I]$. Finally, since $(S'_I, S'_S) \in R$, therefore $(S[S'_I], S[S'_S]) \in R'$.

The last thing to show is that for every initial state S_0 of $M[M_I]$ there exists an initial state S'_0 of $M[M_S]$ such that $(S_0, S'_0) \in R'$. Let $S_{I,0}$ be an initial state of M_I. Let $S_{M,0}$ be the

reset state of M_S such that $(S_{I,0}, S_{S,0}) \in R$ implied by $M_I \sqsubseteq M_S$. Let $S_0[\,]$ be an arbitrary reset state of $M[\,]$, so $S_0[S_{I,0}]$ is a reset state of $M[M_I]$, $S_0[S_{S,0}]$ is a reset state of $M[M_S]$, and $(S_0[S_{I,0}], S_0[S_{S,0}]) \in R'$.

□

6.7 Using the Modular Verification Theorem

Now that we have proven the modular verification theorem, let's understand it can be used during verification of large modules.

Consider the verification of a processor that contains a cached memory system as a submodule. Our specification for the cached memory system is an uncached memory system that accesses a simplified model of main memory directly. This specification is simpler to use during verification because there is less internal state and fewer rules required for memory accesses on average.

For the purposes of this example, assume the processor has two methods: a `start` method that takes in a program and a `getResult` method that returns a single output for each program.

For simplicity, we use $Proc[\,]$ to refer to the processor as a module context with a hole for the memory system. We use $Cache$ to denote the cached memory system, and we use Mem to denote the uncached memory specification. Therefore in this example we are trying to verify $Proc[Cache]$ is correct.

We want to show that $Proc[Cache]$ always returns the right answer for each program, but the $Cache$ module in the processor complicates the verification due to its complexity.

Once we verify that $Cache \sqsubseteq Mem$ via some simulation relation R, we can use the modular verification theorem to show that $Proc[Cache] \sqsubseteq Proc[Mem]$ via the simulation relation $R' = \{(S_P[S_C], S_P[S_M]) \mid (S_C, S_M) \in R\}$. With this, if we can show that $Proc[Mem]$ returns the right answer for each program, then $Proc[Cache]$ will return the right answers as well.

We can also verify properties about the states of $Proc[\,]$ using Mem in place of $Cache$. For example, let's say we want to make sure that the number of instructions in the pipeline

stages between scoreboard insert and remove is equal to the number of elements in the scoreboard. This can be written as a property P on states of $Proc[\ ]$.

The simulation relation R' implies the only reachable $Proc[\ ]$ states of $Proc[Cache]$ are reachable in $Proc[Mem]$ as well. Therefore if we prove that P holds for all states of $Proc[Mem]$, it will hold for all states of $Porc[Cache]$ as well.

Another way the modular verification theorem is used is for refining designs that have already been proven correct. For example, consider a new version of the cache module $Cache'$. If we can show that $Cache' \sqsubseteq Cache$, then the previously mentioned verification for $Proc[Cache]$ will apply for $Proc[Cache']$ as well.

6.8 Conclusion

In this chapter we introduced the implements relation, the notion of a module context, and the modular verification theorem. We then presented modular semantics for module contexts in order to prove the modular verification theorem. Finally we showed how the modular verification theorem can be used during actual verification tasks.

In the upcoming chapters, we will present more details about how to perform this verification, including how to verify $M_I \sqsubseteq M_S$.

Chapter 7

Assertions

To aid in verification of modules, we introduce different kinds of assertions to Spec 'n' Check that can be used to show desired rule-level properties hold in a design by checking that asserted expressions are always true. These assertions can either be checked dynamically during the execution of a module, or they can be checked statically without executing the module to show the assertions hold for all possible executions.

In this chapter we introduce the different kinds of assertions and how they are used. This includes how assertions interact with the implements relation and the modular verification theorem. Finally we will present the rule-level miter module construction that can be used to show a module implements a specification by showing all the assertions in the module hold.

7.1 Introduction

A fundamental part of verifying properties on hardware designs is the inclusion of assertions in the code of the design. Assertions are expressions that should evaluate to true during the execution of the module if the module is correct.

For rule-based verification, we only want to check properties that make sense in the rule-based abstraction. In RTL verification, it does not make sense to try to verify properties about physical aspects of the design (area, power, etc.) because those only exist below the RTL abstraction. Similarly, in rule-based verification, it does not make sense to try to verify

properties about clock-cycle timing or concurrency, because those notions do not exist in the one-rule-at-a-time abstraction of our design language. It's fine for designs to include assertions that check these types of properties for completeness, but the verification of these assertions does not make sense until RTL has been produced for the rule-based abstraction either automatically or manually.

7.1.1 Property Restrictions for Rule-Based Verification

We restrict the verification to rule-level properties because we want to verify properties that are composable, *i.e.* still hold when the the module is used in a larger outer module, and do not depend on the RTL implementation of the design. To achieve these, we restrict the properties we are checking by restricting the language used in assertions.

First, we do not allow assertions to reference what happens in the next state transition, or transitively, a future state transition (*i.e.* **X** in LTL). For example, consider the verification of `mkThreeF` in Figure 3-2. Looking at this module in isolation, we could prove the property that the `applyF` rule always fires after the `start(v)` method. Once we use this module as a submodule in another module, the property may no longer hold because the outer module may have other rules or methods that can fire between `start(v)` and `applyF`. Note that this restriction does not prevent a rule from containing assertions about a value the rule is writing to a state variable.

Second, we do not allow assertions to say an event will finally happen (*i.e.* **F**, **U**, or **M** in LTL). Properties of this type do not hold up under composition or compilation to RTL. For example, again consider the verification of `mkThreeF` from Figure 3-2. Looking at this module in isolation, we could prove the property that when `start(v)` is called, eventually `getResult` is ready assuming a rule is always fired if there is at least one rule ready. Unfortunately, in order for this property to hold in RTL, we need to recheck the property considering the schedule chosen to produce the RTL. This property may not hold if the RTL implementation does not always schedule the `applyF` rule to fire when it is ready.[1]

[1]This may seem like an adversarial case, but there could be practical reasons for this. For example, if $f(x)$ is a very large combinational circuit that is used in other places in the design, then the RTL may share $f(x)$ between rules, introducing a structural conflict between the rules. If the other uesrs of $f(x)$ get priority over `applyF`, then `applyF` may never fire.

In short, we want assertions to hold their value if a state within an execution is repeated (in case of modular composition) or if the execution is truncated (in case of an undesirable scheduler). If $A(S_{i,j})$ is notation that assertion A holds for states i to j, then these constraints can be written as follows:

$$\forall i \leq j \leq k, A(S_{i,k}) \implies A(S_{i,j} +\!\!+ S_{j,k})$$
$$\forall i \leq j \leq k, A(S_{i,k}) \implies A(S_{i,j})$$

where $S_{i,j} +\!\!+ S_{j,k}$ is the concatenation of the two subsequences of states such that the state j is repeated.

Although we are restricting the properties we are verifying over rule-based designs, we are not compromising verification. All the properties we do not check on rule-based designs can be checked later in the design process once there is RTL available. This is similar to checking lower-level properties of Verilog designs only once they are synthesized to gates.

7.2 Assertion Semantics

In Spec 'n' Check, assertions are Boolean expressions on the state of the system that are expected to be true whenever they are active, but they have no effect on the behavior of the module; they are for verification purposes only. Because of this, assertions can be added to or removed from a module without changing the semantics of it. Note that Kami has an `assert` statement in its language, but it is not the same as assertions here. Instead, Kami's `assert` statement is the same as Spec 'n' Check's `guard` action.

In Spec 'n' Check, there are different kinds of assertions that vary in two dimensions: when they are checked and what constructs their expressions can contain. Assertions can either be checked between each rule or top-level method firing, or every time a certain rule or method fires. Assertions either use the same syntax as expressions, or they use an extended expression syntax that allows for more power in writing assertions. These assertions have different purposes, but they all have the same goal: to check that an expression is true for verification purposes.

7.2.1 Assertions by When They are Checked

Classifying assertions by when they are checked splits assertions into two classes: state assertions and step assertions.

State Assertions

A *state assertion* is an assertion that is checked between each rule or top-level method firing. State assertions can also be viewed as being checked on each reachable state of a module.

When state assertions are checked dynamically, the expression of the assertion is evaluated for the initial state and all following states in the execution. If the expression ever evaluates to false for some state in the execution, then the assertion is said to be violated for that state and execution.

A state assertion can be used to check state invariants in a module. For example, a FIFO implemented with a circular buffer may have an enqueue pointer `enqP`, a dequeue pointer `deqP`, and flags `full` and `empty` for specifying if the FIFO is full or empty respectively. This circular buffer can only be full or empty when `enqP` is equal to `deqP`, therefore `full` and `empty` should be False when `enqP` is not equal to `deqP`. Furthermore exactly one of `full` and `empty` should be true when `enqP` is equal to `deqP`. These invariants can be covered by the state-assertion expression

```
(enqP == deqP) ? (full != empty) : (!full && !empty).
```

Step Assertions

A *step assertion* is an action statement that is written in the body of a rule or a method anywhere an action can be written. These assertions are only checked in steps where the associated rule or method is fired, and furthermore, if the action appears within a branch of an if statement, the action is only checked if the branch of the if statement it is on is executed.

Step assertions may seem similar to guards, but the two play different roles. Guards are used to prevent rules and methods from firing when certain conditions are false, so therefore each time a rule or method fires, all the guards are true. For example, a typical `getResult`

method has a guard to prevent it from being called until the result is ready. On the other hand, an assertion does not change whether a method or rule can be called. Instead it is just checking a condition that is expected to be true each time the method or rule fires and all the guards are true. For example, a typical assertion may check that a `getResult` method returns the expected value.

When a module is in development, it is common for assertions to be violated in executions as bugs are ironed out. Preventing rules from firing because of violated assertions would hide these bugs.

7.2.2 Assertions by Expression Constructs

Classifying assertions by what expression constructs they can use splits assertions into two classes: simple assertions and extended assertions.

Simple Assertions

A *simple assertion* is an assertion that uses the standard expression constructs as used in the rest of Spec 'n' Check in order to write the expression being asserted. The set of expressions that can be asserted are the same as the set of expressions that can be written to a state variable. Therefore simple assertions can be implemented using normal Spec 'n' Check and state variables.

Since "simple" only refers to the expression of the assertion, simple assertions can either be state assertions or step assertions.

Extended Assertions

An *extended assertion* is an assertion that uses an extended expression language for assertions that goes beyond the capabilities of Spec 'n' Check. This specifically includes breaking the module abstraction so that extended assertions can get state values within submodules and the guards of submodule methods. Since extended assertions break the module abstraction, they cannot be implemented using Spec 'n' Check, and they can detect things that would be undetectable by Spec 'n' Check.

Extended assertions are required in order to write a module with assertion that checks if a submodule implements a specification. A naive way to try to verify a module against a specification is to construct a wrapper module that always calls the methods of the module and the specification together with assertions that make sure they produce the same result. This module is insufficient because this module only explores executions that are legal for both the module and the specification and cannot detect if there is a method that can be called on the module but not be called on the specification. With extended assertions, an additional assertion can be added to this module to make sure that each time a method in the implementation is ready, its corresponding method in the specification is ready as well.

7.3 Assertions in Modular Verification

Chapter 6 introduces the notion of modular verification with the implements relation and the modular verification theorem. The following theorem shows how simple assertions interact with modular verification.

The first case is how assertions contained within $M[\ ]$ of the module $M[M_I]$ can be verified using $M[M_S]$ for some module M_S such that $M_I \sqsubseteq M_S$.

Theorem 7.1 (Modular Verification Assertion Theorem)**.** *Given modules M_I and M_S such that $M_I \sqsubseteq M_S$, for all module contexts $M[\ ]$, if all the simple assertions of $M[\ ]$ hold for all executions of $M[M_S]$, then those same assertions hold for all executions of $M[M_I]$.*

Proof. To prove this we look at a reachable state of $M[M_I]$ and construct a corresponding reachable state of $M[M_S]$ such that all the simple state and step assertions have the same value.

Let $M_I \sqsubseteq M_S$ via simulation relation R, and let $M[\ ]$ be an arbitrary module context.

Let $S[S_I]$ be an arbitrary reachable state of $M[M_I]$. There exists an initial state $S_0[S_{I,0}]$ and execution sequence $\overrightarrow{ex_I}$ such that

$$M[M_I] \vdash S_0[S_{I,0}] \bullet\!\!\xrightarrow{\overrightarrow{ex_I}}\!\!\!\!\rightarrow S[S_I]$$

The modular verification theorem implies that $M[M_I] \sqsubseteq M[M_S]$ via the simulation relation

$R' = \{(S[S_I], S[S_S]) \mid (S_I, S_S) \in R,\ S \in \mathsf{States}(M[\,])\}$, so therefore there exists an initial state $S_0[S_{S,0}]$ and execution sequence $\overrightarrow{ex_S}$ such that

$$M[M_S] \vdash S_0[S_{S,0}] \bullet\!\!\xrightarrow{\overrightarrow{ex_S}}\!\!\!\!\rightarrow S[S_S]$$

where $(S_I, S_S) \in R$.

We now prove the value of each assertion A of $M[\,]$ in state $S[S_I]$ matches the value of assertion A in state $S[S_S]$.

- (A is a state assertion) - If A is a state assertion in $M[\,]$, then it can only depend on state variables within $M[\,]$ and therefore can be written as a function on states $S[\,]$ of $M[\,]$. Since $S[S_I]$ and $S[S_S]$ have the same state value for the states of $M[\,]$, any state assertion A has the same value for $S[S_I]$ and $S[S_S]$.

- (A is a step assertion) - If A is a step assertion in $M[\,]$, then it can be found in any step that is not a hole rule. Let $S'[S'_I]$ be a state of $M[M_I]$ and ℓ be a transition label of $M[\,]$ that does not correspond to a hole rule such that $M[M_I] \vdash S[S_I] \xrightarrow{\ell} S'[S'_I]$. By Theorem 6.5, there exists a hole label ℓ_H such that $M[\,] \vdash S[\,] \xrightarrow[\ell_H]{\ell} S'[\,]$ and $M_I \vdash S_I \xrightarrow{\ell_H} S'_I$. Since $M_I \sqsubseteq M_S$ via R, there exist S'_S and S''_S such that $M_S \vdash S_S \overset{\varepsilon}{\Rightarrow} S''_S$ and $M_S \vdash S''_S \xrightarrow{\ell_H} S'_S$. Since $M_S \vdash S_S \overset{\varepsilon}{\Rightarrow} S''_S$ and $S[S_S]$ is a reachable state of $M[M_S]$, $S[S''_S]$ is a reachable state of $M[M_S]$. Since $M_S \vdash S''_S \xrightarrow{\ell_H} S'_S$, $M[M_S] \vdash S[S''_S] \xrightarrow{\ell} S'[S'_S]$. The transitions we have presented for ℓ from $S[S_I]$ and $S[S''_S]$ have the same behavior with respect to the context $M[\,]$ since they both use the same hole assumption label ℓ_H, so therefore any step assertion of $M[\,]$ in ℓ has the same value in the two modules.

□

Remark. This theorem also holds for extended assertions as long as they do not depend on the guards of any hole methods or states internal to the hole, but in practice, extended assertions are only used for invariants to help model checking prove that simple assertions hold for all reachable states or that a module implements a specification.

The second case using assertions in modular verification is using assertions of M_S that

are always true in the context $M[M_S]$ to prove things about reachable states of $M[M_I]$ for some module M_I such that $M_I \sqsubseteq M_S$.

Theorem 7.2 (Relating Assertions)**.** *Given modules M_I and M_S such that $M_I \sqsubseteq M_S$ via simulation relation R. If an assertion A of M_S holds for all reachable states of $M[M_S]$, then if state S_I is not mapped by R to a state that satisfies A, then $S[S_I]$ is not a reachable state of $M[M_I]$.*

Proof. This is proved easily by contradiction. If such a state $S[S_I]$ was reachable, then there would be a state S_S such that $(S_I, S_S) \in R$. □

7.4 Miter Module Construction

In traditional RTL verification, a miter module is a module that instantiates two implementations of the same module and performs equivalence checking to make sure the modules have the same cycle-by-cycle behavior [81]. In our rule-based verification, we can construct a different style of rule-level miter module that uses extended assertions to show that a module implements a specification.

Definition 7.1 (Rule-Level Miter Module)**.** If M_I and M_S are modules with the same action and value methods, the miter module for M_I and M_S, or $\mathsf{Miter}(M_I, M_S)$, is a module construction used to show $M_I \sqsubseteq M_S$. The $\mathsf{Miter}(M_I, M_S)$ module:

- Contains instances m_I and m_S of M_I and M_S respectively.
- Has a method for each method of m_I and m_S that calls the corresponding method of each submodule with the same argument, asserts the results are equal, and returns the result from the method call of m_I.
- Contains extended state assertions for each method that confirm that a method is ready for m_I, the corresponding method is ready for m_S.

As an example, Figure 7-1 shows a miter module construction in Spec 'n' Check for modules `mkThreeFBuffered` and `mkFuncUnitConcurrentSpec` from Chapter 3. The goal of

this module is to show that `mkThreeFBuffered` $\sqsubseteq$ `mkFuncUnitConcurrentSpec`. As seen in this example, `eassert` is used to denote extended assertions, and `READY` is used to get if a specified method is ready or not.

```
module mkMiter =
    instance mi mkThreeFBuffered
    instance ms mkFuncUnitConcurrentSpec

    eassert(!READY(mi.start) || READY(ms.start))
    eassert(!READY(mi.getResult) || READY(ms.getResult))

    method start = λ x. mi.start(x) ; ms.start(x)

    method getResult = λ _.
        let x = mi.getResult() in
        let y = ms.getResult() in
        assert(x == y) ;
        x
```

Figure 7-1: Example miter module construction

Theorem 7.3 (Miter Module Theorem)**.** *If the assertions in* $\mathsf{Miter}(M_I, M_S)$ *hold for all reachable states, then* $M_I \sqsubseteq M_S$ *via the simulation relation*

$$R = \{(S_I, S_S) \mid [m_I \mapsto S_I, m_S \mapsto S_S] \text{ is a reachable state of } \mathsf{Miter}(M_I, M_S)\}\,.$$

Proof. Let M_I and M_S be modules such that the assertions in $\mathsf{Miter}(M_I, M_S)$ hold for all reachable states. We will show that the set R constructed by making pairs out of the reachable states of $\mathsf{Miter}(M_I, M_S)$ is a simulation relation between M_I and M_S and therefore $M_I \sqsubseteq M_S$.

Let (S_I, S_S) be an arbitrary element in R. Therefore $S_{\text{Miter}} = [m_I \mapsto S_I, m_S \mapsto S_S]$ is a reachable state of $\mathsf{Miter}(M_I, M_S)$.

- Let S'_I be a state and ℓ_m be a method label such that $M_I \vdash S_I \xrightarrow{\ell_m} S'_I$. Since all the assertions in the miter hold, the method in ℓ_m is ready for state S_S of M_S and it produces the same result, so therefore there exists a state S'_S such that $M_S \vdash S_S \xrightarrow{\ell_m} S'_S$. As a result $[m_I \mapsto S'_I, m_S \mapsto S'_S]$ is a reachable state of the miter so $(S'_I, S'_S) \in R$.

- Let S'_I be a state and r be a rule label such that $M_I \vdash S_I \xrightarrow{r} S'_I$. Therefore the transition $m_I.r$ is a legal step transition for the miter module from state S_{Miter} and results in the state $[m_I \mapsto S'_I, m_S \mapsto S_S]$, so therefore $(S'_I, S_S) \in R$.

- The initial states of $\mathsf{Miter}(M_I, M_S)$ are any state $[m_I \mapsto S_I, m_S \mapsto S_S]$ such that S_I is an initial state of M_I and S_S is an initial state of M_S, so therefore for every initial state $S_{I,0}$ of M_I there exists an initial state $S_{S,0}$ of M_S such that $(S_{I,0}, S_{S,0}) \in R$.

Therefore by the definition of the implements relation: $M_I \sqsubseteq M_S$. □

Note that the inverse of this theorem is not true. If there exists an execution of the miter module $\mathsf{Miter}(M_I, M_S)$ that violates assertions, $M_I \sqsubseteq M_S$ may still be true. This is most often the case when M_S has an internal rule that must fire between certain method calls. There are two ways to solve this problem.

The first solution is to use a different specification module M_S that does not include the internal rule. This is the ideal solution because removing the internal rule also improves verification performance.

The second solution is to make a customized miter module (outside the capabilities of Spec 'n' Check) that executes the internal rules of M_S at the right time in order to enable the M_S module to match the method calls of the M_I module. This can be faked sometimes by merging the rules of M_S into the bodies of methods they must be called before, but this is effectively the first solution because it makes a new module with no internal rules. This process is similar to the custom testbench used in some of the processor examples in Chapter 4.

7.5 Assertions in Practice

In practice, assertions are used in three main ways: verifying a desired property holds, verifying a module implements a specification (as in a miter module), and adding invariants to a design to aid in the success of rule-based model checking methods.

In Chapter 3, testbenches are used to make sure modules behave as expected. These testbenches have simple step assertions in them to make sure the return values match the

expected values. This is one of the most common cases of using an assertion to check a known property. When these assertions are simple assertions, submodules can be substituted by specification modules with fewer internal rules to simplify the verification.

In this chapter, the miter module construction is introduced as a way to check if a module implements a specification. Miter modules are used often in rule-based verification to show a module can be substituted by a specification for verification purposes.

Later in this thesis, Chapter 9 introduces SMT-based model checking of rule-based designs. When showing a property holds for all executions, it is sometimes necessary to add assertions as invariants to restrict the search space for the induction step. In this case, extended assertions that can use internal states are very useful. This is explained in more detail in Chapter 9.

7.5.1 Alternate Flavors of Assertions

There are other flavors of assertions that are useful when verifying a module. One of the most useful variant is the assumption for assume-guarantee reasoning [20]. Assumptions are like assertions in that they are expressions that should always be true when active, but they are used differently during verification to enable modular verification.

For example, if the correctness of a submodule depends on it being used the right way (*e.g.* its methods are never called with "illegal" arguments), then the conditions that determine that the submodule is used the right way are assumed using using `assume` statements. When verifying assertions within the submodule, all the assume statements are assumed to be true, that is, we are verifying there are no executions which satisfy all assume statements but violate an assertion. Then when the submodule is used in an outer module, the submodule assumptions are treated as assertions to make sure the outer module satisfies all the assumptions about how the submodule should be used.

If the `assume` statements are hard to verify in the context of a larger design, then the `assume` statements can be related to assertions in a specification module, and Theorem 7.2 can be used to show that if the specification assertions hold, then the implementation assumptions hold.

In RTL-based model checking, assume statements are very important to make sure inputs

follow expected conventions, but in rule-based model checking, method guards take care of most of that.

7.6 Conclusion

This chapter introduced different kinds of assertions and some theory about how they can be used. The upcoming chapters will show how SMT-based model checking can be done to verify these assertions hold for all reachable states.

Chapter 8

Symbolic Semantics of Spec 'n' Check

In order to verify properties about our rule-based designs, we want to be able to produce a symbolic representation for our designs that can be used in SMT solvers to perform SMT-based verification. This means it is desirable to have symbolic representations of the semantics that can be used to prove properties about designs. The symbolic representations should be consistent with the semantics, so when the symbolic representations are evaluated with concrete values, the results should match the semantics.

This chapter introduces symbolic representations that are derived from, and proven consistent with, the semantics of Spec 'n' Check, and therefore we call these symbolic representations the *symbolic semantics*.

8.1 Symbolic Expressions and Data Structures

In order to present the symbolic semantics of Spec 'n' Check, we must introduce the language for symbolic expressions that will be used in the symbolic semantics. This also includes symbolic versions of various data structures used in the semantics

8.1.1 Symbolic Expressions

Symbolic expressions are used to represent expressions with unknown values or free variables in them. They can also be viewed as functions that take in maps of values for symbolic

variables and return evaluated values of expressions where symbolic variables get their values from the input maps. The relevant definitions for symbolic expressions can be seen in Figure 8-1.

8.1.2 Guarded Symbolic Expressions

There are multiple places in the semantics where a symbolic expression only makes sense when a certain condition holds. For example, when looking at a guarded rule, the semantics are only legal when the guard holds. As another example, when looking at the map of state updates in the symbolic semantics, if a state is only written to under certain conditions, then the value written to the state element only makes sense under those conditions.

To express a symbolic expression that is only valid under certain conditions, we introduce guarded symbolic expressions.

Definition 8.1 (Guarded Symbolic Expression)**.** A guarded symbolic expression is a symbolic expression $\langle g, v \rangle$ where g and v are symbolic expressions and g is the guard of v. If g evaluates to true, then $\langle g, v \rangle$ evaluates to the evaluation of v. If g evaluates to false, then the interpretation of $\langle g, v \rangle$ depends on the context; commonly it means the value or semantics are not valid.

Guarded symbolic expressions are used in two main places: expressing rule and method guards in the semantics (action, rule, method, step, and execution semantics) and expressing conditional state updates in the updates map (action semantics).

8.1.3 Symbolic Maps

Associative arrays, or maps, are used in the semantics to hold the state values, the variable bindings, and the state updates. In the symbolic semantics, it is necessary to have symbolic representations of these structures as well.

Symbolic Maps for State Values and Variable Bindings

Symbolic state values and symbolic variable bindings can be implemented as a standard associative array with symbolic values. This is because whether or not a key is in these

Symbolic Expression Constructors:

$$
\begin{array}{rcll}
s & ::= & \mathbf{c} & \text{(const)} \\
 & | & \mathbf{v} & \text{(var)} \\
 & | & [\mathbf{m}.]^*\mathbf{v} & \text{(hierarchical var)} \\
 & | & \mathbf{apply}(f, s_1, \ldots, s_n) & \text{(apply function)} \\
 & | & \mathbf{mux}(s_p, s_t, s_f) & \\
 & | & \mathbf{and}(s_1, s_2) & \\
 & | & \mathbf{or}(s_1, s_2) & \\
 & | & \mathbf{not}(s) &
\end{array}
$$

Symbolic Expression Eval:
Evaluates a symbolic expression in an environment E. E is a hierarchical map containing values for symbolic variables $\mathbf{v}$. Note that eval assumes all necessary variables are defined in E.

$$
\begin{array}{l}
\mathsf{Eval}(\mathbf{c}, E) = c \\
\mathsf{Eval}(\mathbf{v}, E) = E(v) \\
\mathsf{Eval}(\mathbf{m.x}, E) = \mathsf{Eval}(\mathbf{x}, E(m)) \qquad \text{(hierarchical var)} \\
\mathsf{Eval}(\mathbf{apply}(f, s_1, \ldots, s_n), E) = \\
\quad \underline{f}(\mathsf{Eval}(s_1, E), \ldots, \mathsf{Eval}(s_n, E)) \\
\mathsf{Eval}(\mathbf{mux}(s_p, s_t, s_f), E) = \\
\quad \mathsf{Eval}(s_p, E) \ ? \ \mathsf{Eval}(s_t, E) : \mathsf{Eval}(s_f, E) \\
\mathsf{Eval}(\mathbf{and}(s_1, s_2), E) = \mathsf{Eval}(s_1, E) \wedge \mathsf{Eval}(s_2, E) \\
\mathsf{Eval}(\mathbf{or}(s_1, s_2), E) = \mathsf{Eval}(s_1, E) \vee \mathsf{Eval}(s_2, E) \\
\mathsf{Eval}(\mathbf{not}(s), E) = \neg \mathsf{Eval}(s, E)
\end{array}
$$

Adding Modular Hierarchy to Symbolic Expressions:
The $\mathsf{Prefix}(s, a)$ function adds the module prefix a to all symbolic variables in symbolic expression s. This function is defined as follows:

$$
\begin{array}{lcl}
\mathsf{Prefix}(\mathbf{c}, a) & = & \mathbf{c} \\
\mathsf{Prefix}(\mathbf{v}, a) & = & \mathbf{a.v} \\
\mathsf{Prefix}(\mathbf{m.x}, a) & = & \mathbf{a.m.x} \\
\mathsf{Prefix}(\mathbf{C}(s_1, \ldots, s_n), a) & = & \mathbf{C}(\mathsf{Prefix}(s_1, a), \ldots, \mathsf{Prefix}(s_n, a))
\end{array}
$$

where $\mathbf{C}$ corresponds to the other symbolic expression constructors

Symbolic Expression Substitution:

$$
\begin{array}{lcl}
\mathsf{Substitute}(\mathbf{v}, E) & = & \text{if } \mathbf{v} \in E \text{ then } E(\mathbf{v}) \text{ else } \mathbf{v} \\
\mathsf{Substitute}(\mathbf{C}(s_1, \ldots, s_n), E) & = & \mathbf{C}(\mathsf{Substitute}(s_1, E), \ldots, \mathsf{Substitute}(s_n, E))
\end{array}
$$

where $\mathbf{C}$ corresponds to the other symbolic expression constructors

Figure 8-1: Symbolic expression definitions

maps does not depend on a symbolic expression. Therefore there is little difference in the structure of these maps between the concrete semantics and the symbolic representation.

Symbolic Maps for State Updates

On the other hand, the symbolic representation of the state update map must be a truly symbolic map where which keys are in the map can depend on a symbolic expression. For example, consider the action

```
if p then x <= 1 else y <= 2.
```

If `p` is true, then the resulting update map should be $[\mathtt{x} \mapsto 1]$, but if `p` is false, then the update map should be $[\mathtt{y} \mapsto 2]$. In order to have a symbolic representation of the update map without adding maps as a primitive symbolic value, we need a representation for symbolic maps that can evaluate to the two separate maps when the value of `p` is specified.

We define symbolic maps for state updates using what is effectively a list of key-value pairs where the keys are (not necessarily unique) state names and the values are guarded symbolic expressions. The guard of each value represents if the key-value pair is valid or not. If the guard evaluates to false for a certain environment, then it is as if the key-value pair does not exist for that environment.

If a key appears multiple times in the symbolic map, then as long as at most one guard is true per environment, then there is effectively only one value for the key, and there is no problem. If there are multiple valid values for a key for a given environment, then this corresponds to a double-write error in the action semantics for that environment. The action semantics do not allow multiple writes, so in the symbolic semantics, we check for double writes when joining updates together.

Therefore with this definition of symbolic map, the update map for the previous example would be

$$U = [\mathtt{x} \mapsto \langle \mathtt{p}, 1 \rangle, \mathtt{y} \mapsto \langle \mathbf{not}(\mathtt{p}), 2 \rangle]$$

Definition 8.2 (Symbolic Map Membership)**.** If M is a symbolic map, then we define the

membership function as

$$\begin{aligned} x \in M[t \mapsto \langle g, v \rangle] &= ((x = t) \wedge g) \vee x \in M \\ x \in \emptyset &= false \end{aligned}$$

Definition 8.3 (Symbolic Map Lookup)**.** If M is a symbolic map, then the lookup function $M(x)$ is defined as

$$\begin{aligned} M[t \mapsto \langle g, v \rangle](x) &= \text{if } x = t \text{ then } \mathbf{mux}(g, \langle true, v \rangle, M(x)) \text{ else } M(x) \\ \emptyset(x) &= \langle false, \_ \rangle \end{aligned}$$

Definition 8.4 (Symbolic Map Eval)**.** For simplicity, we define Eval to work on symbolic maps. If M is a symbolic map and E is an environment for evaluating the symbolic expressions in M, then $\mathsf{Eval}(M, E)$ is defined as

$$\begin{aligned} \mathsf{Eval}(M[k \mapsto \langle g, v \rangle], E) &= \text{if } \mathsf{Eval}(g, E) \text{ then } \mathsf{Eval}(M, E)[k \mapsto \mathsf{Eval}(v, E)] \text{ else } \mathsf{Eval}(M, E) \\ \mathsf{Eval}(\emptyset, E) &= \emptyset \end{aligned}$$

Definition 8.5 (Double Write)**.** If M is a symbolic map, then $\mathsf{DoubleWrite}(M)$ returns a symbolic expression corresponding to if M has a key with two valid values defined as

$$\begin{aligned} \mathsf{DoubleWrite}(M[t \mapsto \langle g, v \rangle]) &= (g \wedge (t \in M)) \vee \mathsf{DoubleWrite}(M) \\ \mathsf{DoubleWrite}(\emptyset) &= false \end{aligned}$$

Definition 8.6 (Symbolic Map Mux)**.** For simplicity, we define **mux** to work on symbolic maps as well. If s is a symbolic Boolean expression and M_1 and M_2 are symbolic maps, then $\mathbf{mux}(s, M_1, M_2)$ is defined as

$$\mathbf{mux}(s, M_1, M_2) = \mathsf{WithCondition}(M_1, s) +\!\!+ \mathsf{WithCondition}(M_2, \mathbf{not}(s))$$

where $+\!\!+$ concatenates lists of key-value pairs and $\mathsf{WithCondition}(M, p)$ is defined as

$$\begin{array}{lcl} \mathsf{WithCondition}(M[k \mapsto \langle g, v\rangle], p) & = & \mathsf{WithCondition}(M, p)[k \mapsto \langle \mathbf{and}(g, p), v\rangle] \\ \mathsf{WithCondition}(\emptyset, p) & = & \emptyset \end{array}$$

Lemma 8.1. *For all Boolean symbolic values s and symbolic maps M_1 and M_2,* $\mathit{Eval}(\mathbf{mux}(s, M_1, M_2), S) = \mathit{if}\ \mathit{Eval}(s, S)\ \mathit{then}\ \mathit{Eval}(M_1, S)\ \mathit{else}\ \mathit{Eval}(M_2, S)$.

This lemma is proved by first showing that

$$\mathsf{Eval}(\mathsf{WithCondition}(M, s), S) = \text{if } \mathsf{Eval}(s, S) \text{ then } \mathsf{Eval}(M, S) \text{ else } [\,].$$

After that, it is trivial to show that this lemma holds.

8.2 Expression Semantics

We start looking at the symbolic semantics of modules by looking at the symbolic semantics of expressions.

The symbolic semantics of expressions are denoted using the following function:

$$\mathsf{symbExpr}(e, B) = v$$

This symbolic semantics function says that expression e with symbolic variable bindings B is equal to symbolic expression v. Note that this function does not take in values for state variables. Instead, all state variables are treated as symbolic variables.

The symbolic semantics of expressions can be seen in Figure 8-2. The consistency of these semantics with the concrete expression semantics is stated and proved in Section 8.8.1.

8.3 Action Semantics

The symbolic semantics for actions are denoted using the following function:

$$\mathsf{symbAction}(M, a, B) = \langle g, (U, v)\rangle$$

state-rd	$\mathsf{symbExpr}(\langle s \rangle, B) = \mathbf{s}$
const	$\mathsf{symbExpr}(\langle c \rangle, B) = \underline{c}$
var	$\mathsf{symbExpr}(\langle t \rangle, B) = B[t]$
func	$\mathsf{symbExpr}(\langle f(e_1, \ldots, e_n) \rangle, B)$: $\forall i,\ v_i = \mathsf{symbExpr}(e_i, B)$ return $\mathbf{apply}(f, v_1, \ldots, v_n)$
e-let	$\mathsf{symbExpr}(\langle \texttt{let } t \texttt{ = } e_1 \texttt{ in } e_2 \rangle, B)$: $v_1 = \mathsf{symbExpr}(e_1, B)$ return $\mathsf{symbExpr}(e_2, B[t \mapsto v_1])$

Figure 8-2: Symbolic expression semantics

The symbolic semantics function says that in module M, action a with symbolic variable bindings B results in the symbolic state updates found in U and the return value v provided the expression g is true.

The symbolic action semantics can be seen in Figure 8-3. For method calls, we use the modular action semantics, so they depend on the symbolic semantics for methods; therefore they are presented later after the symbolic semantics for methods.

Since much of the complexity of symbolic semantics is contained within the symbolic action semantics, these semantics are worth explaining in detail.

- (state-wr) - A state write action is always ready, and it always results in a register update. Therefore the guard of the return value is true, and the guard of the value in the returned update map is also true.

- (if) - The concrete semantics for if actions only depend on the action of the taken branch. For symbolic semantics, the semantics of if actions always depend on both branches, and the resulting semantics are produced by muxing the results from the two branches. Most of the complexity of this function is contained within the update map mux which is defined in Definition 8.6.

- (guard) - Since guards are only ready when the expression evaluates to true, the guard of the returned symbolic semantics is the symbolic semantics of the expression.

state-wr $\mathsf{symbAction}(M, \langle s \texttt{ <= } e\rangle, B)$:
$v = \mathsf{symbExpr}(e, B)$
return $\langle true, ([s \mapsto \langle true, v\rangle], \varepsilon)\rangle$

if $\mathsf{symbAction}(M, \langle \texttt{if } e \texttt{ then } a_1 \texttt{ else } a_2\rangle, B)$:
$v_e = \mathsf{symbExpr}(e, B)$
$\langle g_1, (U_1, v_1)\rangle = \mathsf{symbAction}(M, a_1, B)$
$\langle g_2, (U_2, v_2)\rangle = \mathsf{symbAction}(M, a_2, B)$
return $\langle \mathbf{mux}(v_e, g_1, g_2), (\mathbf{mux}(v_e, U_1, U_2), \mathbf{mux}(v_e, v_1, v_2))\rangle$

guard $\mathsf{symbAction}(M, \langle \texttt{guard(}e\texttt{)}\rangle, B)$:
$v = \mathsf{symbExpr}(e, B)$
return $\langle v, ([\,], \varepsilon)\rangle$

par $\mathsf{symbAction}(M, \langle a_1 \texttt{ ; } a_2\rangle, B)$:
$\langle g_1, (U_1, v_1)\rangle = \mathsf{symbAction}(M, a_1, B)$
$\langle g_2, (U_2, v_2)\rangle = \mathsf{symbAction}(M, a_2, B)$
$U = U_1 +\!\!+ U_2$
$dw = \mathsf{doubleWrite}(U)$
return $\langle \mathbf{and}(g_1, g_2, \mathbf{not}(dw)), (U, v_2)\rangle$

a-let $\mathsf{symbAction}(M, \langle \texttt{let } t \texttt{ = } a_1 \texttt{ in } a_2\rangle, B)$:
$\langle g_1, (U_1, v_1)\rangle = \mathsf{symbAction}(M, a_1, B)$
$\langle g_2, (U_2, v_2)\rangle = \mathsf{symbAction}(M, a_2, B[t \mapsto v_1])$
$U = U_1 +\!\!+ U_2$
$dw = \mathsf{doubleWrite}(U)$
return $\langle \mathbf{and}(g_1, g_2, \mathbf{not}(dw)), (U, v_2)\rangle$

expr $\mathsf{symbAction}(M, \langle e\rangle, B) = \langle True, ([\,], \mathsf{symbExpr}(e, B))\rangle$

Figure 8-3: Symbolic action semantics (without method calls)

- (par) - The concrete semantics for par requires the two actions to be ready, and it requires their updates to be disjoint. The symbolic semantics does a nondisjoint union of the symbolic update maps and produces a symbolic expression dw corresponding to the condition in which the union was not disjoint. The guard of the symbolic semantics returned by this function is equal to the guards of the two actions anded together with the condition dw.

- (a-let) - The symbolic semantics for this action are nearly identical to par except for the value of B used to compute the semantics of a_2.

- (expr) - This just lifts an expression to an action, so the semantics are always ready and just correspond to returning the symbolic semantics of the expression.

8.4 Step Semantics

The symbolic step semantics are separated into functions for specific steps (*i.e.* specific rules, action methods, or value methods) and a generic function that supports a step of a rule or method determined by a symbolic variable. Symbolic semantics are used differently than the concrete semantics; one result of this is there is no need for a generic function that supports arbitrary compound method steps (this is thanks to unambiguous one-method-at-a-time semantics).

The symbolic step semantics for specific steps are produced using the following functions:

Rule:	$\mathsf{symbRule}(M, r) = \langle g, S' \rangle$
Action Method:	$\mathsf{symbAMethod}(M, f_a, x) = \langle g, (S', y) \rangle$
Value Method:	$\mathsf{symbVMethod}(M, f_v, x) = \langle g, y \rangle$

The symbolic rule semantics function takes in the module M and the concrete rule name r and returns a guarded next state $\langle g, S' \rangle$. The symbolic action method semantics function takes in the module M, action method name f_a, and symbolic argument x and returns a guarded next state and return value $\langle g, (S', y) \rangle$. The symbolic value method semantics function takes in the module M, value method name f_v, and symbolic argument x and returns a guarded return value $\langle g, y \rangle$.

rule	$\mathsf{symbRule}(M, r)$: $\langle \texttt{rule}\ r\ \texttt{=}\ a \rangle \in \mathsf{Rules}(M)$ $\langle g, (U, \_) \rangle = \mathsf{symbAction}(M, a, \emptyset)$ $S' = \hat{S}[U]$ (where $\hat{S}$ is the symbolic identity mapping) return $\langle g, S' \rangle$
submod-rule	$\mathsf{symbRule}(M, m.r)$: $\langle \texttt{instance}\ m\ M_{\mathsf{sub}} \rangle \in \mathsf{SubmodInsts}(M)$ $\langle g, \Omega \rangle = \mathsf{Prefix}(\mathsf{symbRule}(M_{\mathrm{sub}}, r), m)$ $S' = \hat{S}[m \mapsto \Omega]$ (where $\hat{S}$ is the symbolic identity mapping) return $\langle g, S' \rangle$
action-method	$\mathsf{symbAMethod}(M, f_a, x)$: $\langle \texttt{amethod}\ f_a\texttt{(t)}\ \texttt{=}\ \lambda t.a \rangle \in \mathsf{AMethods}(M)$ $\langle g, (U, y) \rangle = \mathsf{symbAction}(M, a, [t \mapsto x])$ $S' = \hat{S}[U]$ (where $\hat{S}$ is the symbolic identity mapping) return $\langle g, (S', y) \rangle$
value-method	$\mathsf{symbVMethod}(M, f_v, x)$: $\langle \texttt{vmethod}\ f_v\texttt{(t)}\ \texttt{=}\ \lambda t.a \rangle \in \mathsf{VMethods}(M)$ $\langle g, (\_, y) \rangle = \mathsf{symbAction}(M, a, [t \mapsto x])$ return $\langle g, y \rangle$

Figure 8-4: Symbolic rule and method semantics

The symbolic rule and method semantics can be seen in Figure 8-4. Note that there are two definitions for symbRule. When symbRule is called, the definition used depends on whether the rule argument is a top-level rule name r or a modular rule name $m.r$.

Now that we have the symbolic step semantics for specific steps, we can present a generic function that supports a step of an arbitrary rule or method. The generic function for symbolic step semantics is

$$\mathsf{symbStep}(M, \ell_{\mathrm{id}}, x) = \langle g, (S', y) \rangle$$

The symbolic step semantics function takes in the module M, a symbolic value corresponding to the name of a rule or method ℓ_{id}, and a symbolic argument value x. The function returns a guarded pair including a next state value and a return value $\langle g, (S', y) \rangle$. The symbolic step semantics can be seen in Figure 8-5.

step $\mathsf{symbStep}(M, \ell_{\text{id}}, x)$:
$\alpha = \langle False, (S, \varepsilon) \rangle$
for $\langle \texttt{rule}\ r\ \texttt{=}\ a \rangle \in \mathsf{Rules}(M)$: (including submodule rule names)
$\langle g, S' \rangle = \mathsf{symbRule}(M, r)$
$\alpha = \mathbf{mux}(\ell_{\text{id}} = r, \langle g, (S', \varepsilon) \rangle, \alpha)$
for $\langle \texttt{amethod}\ f_a\ \texttt{=}\ \lambda t.a \rangle \in \mathsf{AMethods}(M)$:
$\langle g, (S', y) \rangle = \mathsf{symbAMethod}(M, f_a, x)$
$\alpha = \mathbf{mux}(\ell_{\text{id}} = f_a, \langle g, (S', y) \rangle, \alpha)$
for $\langle \texttt{vmethod}\ f_v\ \texttt{=}\ \lambda t.a \rangle \in \mathsf{VMethods}(M)$:
$\langle g, y \rangle = \mathsf{symbVMethod}(M, f_v, x)$
$\alpha = \mathbf{mux}(\ell_{\text{id}} = f_v, \langle g, (\hat{S}, y) \rangle, \alpha)$ (where $\hat{S}$ is the symbolic identity mapping)
return α

Figure 8-5: Symbolic step semantics

8.5 Method Call Semantics

Now that the symbolic semantics for methods have been presented, we can return back to the symbolic semantics of actions and present the symbolic semantics of method-call actions. These semantics are presented in Figure 8-6.

These semantics are mostly straightforward applications of symbAMethod and symbVMethod similar to the concrete semantics. One major difference is the presence of the Prefix function. The Prefix function is used to take symbolic variables corresponding to the states of the submodule and prefix them with the instance name so they become unique symbolic variables in the outer module. This is done to avoid name conflicts between submodule instances of the same module type and to avoid conflicts between submodules that contain the same state variable names as the parent module.

8.6 Execution Semantics

The final piece of the symbolic semantics is the execution semantics. The symbolic execution semantics are split into two functions:

action-method-call $\mathsf{symbAction}(M, m\,.\,f_a(e), B)$:
$\langle \texttt{instance}\ \ m\ \ M_{\texttt{sub}} \rangle \in \mathsf{SubmodInsts}(M)$
$x = \mathsf{symbExpr}(e, B)$
$\langle g, (\Omega, y) \rangle = \mathsf{symbAMethod}(M_{\text{sub}}, f_a, x)$
$\Omega' = \mathsf{Prefix}(\Omega, m)$
$y' = \mathsf{Prefix}(y, m)$
return $\langle g, ([m \mapsto \langle true, \Omega' \rangle], y') \rangle$

value-method-call $\mathsf{symbAction}(M, m\,.\,f_v(e), B)$:
$\langle \texttt{instance}\ \ m\ \ M_{\texttt{sub}} \rangle \in \mathsf{SubmodInsts}(M)$
$x = \mathsf{symbExpr}(e, B)$
$\langle g, y \rangle = \mathsf{symbVMethod}(M_{\text{sub}}, f_v, x)$
$y' = \mathsf{Prefix}(y, m)$
return $\langle g, ([\,], y') \rangle$

Figure 8-6: Symbolic action method call semantics

Reset: $\mathsf{symbReset}(M) = p$

Execution: $\mathsf{symbExecution}(M, S, \overrightarrow{ex}) = \langle g, S' \rangle$

The symbReset function returns a symbolic Boolean predicate p corresponding to when a state is a legal initial state for module M. The concrete state S is a legal initial state if and only if the expression $\mathsf{Eval}(\mathsf{symbReset}(M), S)$ evaluates to true. The symbExecution function takes in a module M, initial state S, and an execution sequence $\overrightarrow{ex}$ and returns a guarded final state $\langle g, S' \rangle$; the guard is false if $\overrightarrow{ex}$ is not a legal execution sequence starting from state S.

For the symbolic semantics, we assume the execution sequence $\overrightarrow{ex}$ is a list of tuples (ℓ_{id}, x, y) where ℓ_{id} is the symbolic name of the rule or method, x is the symbolic argument, and y is the symbolic return value. When ℓ_{id} evaluates to a rule, x and y are both expected to evaluate to ε.

The definitions of symbReset and symbExecution are shown in Figure 8-7.

8.7 Symbolic Representation of Assertions

The assertions presented in Chapter 7 can also be represented symbolically using symbExpr and a custom version of symbAction. One way this can be done is to treat each step assertion

reset $\mathsf{symbReset}(M)$:
  $p = \mathsf{symbExpr}(\mathsf{Reset}(M), [\,])$
  for each $\langle$`instance m' M'`$\rangle \in \mathsf{SubmodInsts}(M)$:
    $p = \mathbf{and}(p, \mathsf{Prefix}(\mathsf{symbReset}(M'), m'))$
  return p

execution $\mathsf{symbExec}(M, S, \overrightarrow{ex})$:
  if $\overrightarrow{ex}$ matches $[\,]$:
    return $\mathsf{Substitute}(\mathsf{symbReset}(M), S)$
  if $\overrightarrow{ex}$ matches $\overrightarrow{ex'} +\!\!+ [(\ell_{\mathrm{id}}, x, y)]$:
    $\langle g', S' \rangle = \mathsf{symbExec}(M, S, \overrightarrow{ex'})$
    $\langle g'', (S'', y'') \rangle = \mathsf{Substitute}(\mathsf{symbStep}(M, \ell_{\mathrm{id}}, x), S')$
    return $\langle \mathbf{and}(g', g'', y = y''), S'' \rangle$

Figure 8-7: Symbolic execution semantics

as a state variable, and each time there is an assertion, return an update map where the state variable `assert` gets the symbolic value corresponding to the assertion expression.

For example, the symbolic semantics for the action `assert(p == 2)` would be just like a register write `assert <= p == 2`. It would return the update map

$$U = [\texttt{assert} \mapsto \langle \mathit{true}, \mathbf{apply}(\mathit{eq}, \mathbf{p}, 2) \rangle]$$

assuming `p` is a state variable. Furthermore, in the following example

```
1 if foo == 7 then
2     assert(p == 2) ;
3     m.bar(1)
4 else
5     m.bar(0)
```

the update map would have the following entry for assert:

$$\texttt{assert} \mapsto \langle \mathbf{apply}(\mathit{eq}, \mathbf{foo}, 7), \mathbf{apply}(\mathit{eq}, \mathbf{p}, 2) \rangle.$$

This assertion is violated if the guard evaluates to true and the value evaluates to false. If the guard evaluates to false, the assertion does not matter, because it was on the inactive branch of the if action.

8.8 Symbolic Semantics Consistency

In order to use the symbolic semantics for effective SMT-based verification, we need to know the symbolic semantics are consistent with the concrete semantics presented in Chapter 5. The general idea behind this consistency is that there is a correspondence between the concrete semantics and evaluating the symbolic semantics with the same values as used in the concrete semantics.

The consistency between concrete semantics and symbolic semantics is formalized through the lemmas in this section.

8.8.1 Expression Consistency

Lemma 8.2 (Expression Semantics Consistency)**.** *For all expressions e, if the maps S and B contain the values necessary to evaluate $[\![e]\!](S, B)$, and B' is a symbolic map such that $\mathit{Eval}(B', S) = B$, then $[\![e]\!](S, B) = \mathit{Eval}(\mathit{symbExpr}(e, B'), S)$.*

Proof. Assume S and B contain the mappings necessary to evaluate $[\![e]\!](S, B)$, and assume B' is a symbolic map such that $\mathsf{Eval}(B', S) = B$. We prove this lemma by structural induction over the constructors of expression e.

- $(e = \langle s \rangle)$ - $[\![s]\!](S, B) = S(s)$ and $\mathsf{Eval}(\mathsf{symbExpr}(\langle s \rangle, B'), S) = \mathsf{Eval}(\mathbf{s}, S) = S(s)$, so therefore $[\![s]\!](S, B) = \mathsf{Eval}(\mathsf{symbExpr}(\langle s \rangle, B'), S)$.

- $(e = \langle c \rangle)$ - $[\![c]\!](S, B) = \underline{c}$ and $\mathsf{Eval}(\mathsf{symbExpr}(\langle c \rangle, B'), S) = \mathsf{Eval}(\mathbf{c}, S) = \underline{c}$, so therefore $[\![c]\!](S, B) = \mathsf{Eval}(\mathsf{symbExpr}(\langle c \rangle, B'), S)$.

- $(e = \langle t \rangle)$ - $[\![t]\!](S, B) = B(t)$ and $\mathsf{Eval}(\mathsf{symbExpr}(\langle t \rangle, B'), S) = \mathsf{Eval}(B'(t), S)$. Since $\mathsf{Eval}(B', S) = B$, then $\mathsf{Eval}(B'(t), S) = B(t)$, so therefore $\mathsf{Eval}(\mathsf{symbExpr}(\langle t \rangle, B'), S) = B(t)$ and thus $[\![t]\!](S, B) = \mathsf{Eval}(\mathsf{symbExpr}(\langle t \rangle, B'), S)$.

- $(e = \langle f(e_1, \ldots, e_n) \rangle)$ - For all $i \in \{1, \ldots, n\}$, let $[\![e_i]\!](S, B) = v_i$. From the structural induction hypothesis, for all $i \in \{1, \ldots, n\}$, $\mathsf{Eval}(\mathsf{symbExpr}(e_i, B'), S) = v_i$. Now looking

at the semantics of $f(e_1, \ldots, e_n)$, $[\![f(e_1, \ldots, e_n)]\!](S, B) = \underline{f}(v_1, \ldots, v_n)$ and

$$\begin{aligned}
&\mathsf{Eval}(\mathsf{symbExpr}(\langle f(e_1, \ldots, e_n)\rangle, B'), S) \\
&\quad = \mathsf{Eval}(\mathbf{apply}(f, \mathsf{symbExpr}(e_1, B'), \ldots, \mathsf{symbExpr}(e_n, B')), S) \\
&\quad = \underline{f}(\mathsf{Eval}(\mathsf{symbExpr}(e_1, B'), S), \ldots, \mathsf{Eval}(\mathsf{symbExpr}(e_n, B'), S)) \\
&\quad = \underline{f}(v_1, \ldots, v_n).
\end{aligned}$$

Therefore $[\![f(e_1, \ldots, e_n)]\!](S, B) = \mathsf{Eval}(\mathsf{symbExpr}(\langle f(e_1, \ldots, e_n)\rangle, B'), S)$

- $(e = \langle \texttt{let } t \texttt{ = } e_1 \texttt{ in } e_2\rangle)$ - Let $v_1 = [\![e_1]\!](S, B)$ and let $v_2 = [\![e_2]\!](S, B[t \mapsto v_1])$, so therefore $[\![\texttt{let } t \texttt{ = } e_1 \texttt{ in } e_2]\!](S, B) = v_2$. Since $\mathsf{Eval}(B', S) = B$, the induction hypothesis gives $\mathsf{Eval}(\mathsf{symbExpr}(e_1, B'), S) = v_1$. Let $B'' = B'[t \mapsto \mathsf{symbExpr}(e_1, B')]$. Looking at $\mathsf{Eval}(B'', S)$ gives

$$\begin{aligned}
\mathsf{Eval}(B'', S) &= \mathsf{Eval}(B'[t \mapsto \mathsf{symbExpr}(e_1, B')], S) \\
&= \mathsf{Eval}(B', S)[t \mapsto \mathsf{Eval}(\mathsf{symbExpr}(e_1, B'), S)] \\
&= B[t \mapsto v_1].
\end{aligned}$$

Therefore $\mathsf{Eval}(\mathsf{symbExpr}(e_2, B''), S) = v_2$.

Now looking at the semantics of $\texttt{let } t \texttt{ = } e_1 \texttt{ in } e_2$ gives $[\![\texttt{let } t \texttt{ = } e_1 \texttt{ in } e_2]\!](S, B) = v_2$ and $\mathsf{Eval}(\mathsf{symbExpr}(\langle \texttt{let } t \texttt{ = } e_1 \texttt{ in } e_2\rangle, B'), S) = \mathsf{Eval}(\mathsf{symbExpr}(e_2, B''), S) = v_2$. Therefore $[\![\texttt{let } t \texttt{ = } e_1 \texttt{ in } e_2]\!](S, B) = \mathsf{Eval}(\mathsf{symbExpr}(\langle \texttt{let } t \texttt{ = } e_1 \texttt{ in } e_2\rangle, B'), S)$.

This covers all the cases, so by induction the lemma holds for all expressions. □

8.8.2 Action and Method Consistency

Since the action and method symbolic semantics depend on each other (through method calls), their consistency lemmas are proved at the same time, but they are presented separately for simplicity.

Lemma 8.3 (Action Semantics Consistency)**.** *For all actions a, if the maps S and B contain values for the states and unbound variables in a, and B' is a symbolic map such that*

$\mathsf{Eval}(B', S) = B$, *then for all* (U, v), $M \vdash (a, S, B) \downarrow (U, v)$ *if and only if* $\mathsf{Eval}(\mathsf{symbAction}(M, a, B'), S) = (U, v)$.

Lemma 8.4 (Action Method Semantics Consistency)**.** *For all modules* M*, action methods* f_a*, states* S_1 *and* S_2*, arguments* x*, and return values* y*,* $M \vdash S_1 \xrightarrow{f_a(x) \to_a y} S_2$ *if and only if* $\mathsf{Eval}(\mathsf{symbAMethod}(M, f_a, x), S_1) = (S_2, y)$.

Lemma 8.5 (Value Method Semantics Consistency)**.** *For all modules* M*, value methods* f_v*, states* S_1*, arguments* x*, and return values* y*,* $M \vdash S_1 \xrightarrow{f_v(x) \to_v y} S_1$ *if and only if* $\mathsf{Eval}(\mathsf{symbVMethod}(M, f_v, x), S_1) = y$.

The proofs of these lemmas are tedious to write due to all the cases to cover, but none of the cases are difficult. The proof is structural induction over actions and module depth. The induction hypothesis is that Lemma 8.3 holds for all subactions of the current action in the current module and Lemmas 8.4 and 8.5 hold for all submodule methods.

One of the necessary details in the proof is to ensure the symbolic guard g corresponds to whether or not the premises hold for the corresponding deduction rule in the concrete semantics. In the simplest example, for the guard action, g is equal to the symbolic value of e. In more complicated cases, the deduction rule uses the disjoint union on set updates (*i.e.* $U_1 \uplus U_2$) to ensure updates are disjoint. This is implemented in the symbolic semantics using $\mathsf{doubleWrite}(U_1 +\!\!+ U_2)$. If the result has a double write, then the semantics return an invalid value.

One final detail that must be handled in the proof is the use of Prefix in the value and action method call semantics in order to take symbolic expressions from submodules and add prefixes to them so that they refer to submodule state instead of top-level state.

8.8.3 Rule Consistency

The rule semantics consistency lemma states the consistency of both the top-level rule and submodule rule symbolic semantics.

Lemma 8.6 (Rule Semantics Consistency)**.** *For all modules* M*, rules* r *(either top-level or submodule rules), and states* S_1 *and* S_2*,* $M \vdash S_1 \xrightarrow{r} S_2$ *if and only if* $\mathsf{Eval}(\mathsf{symbRule}(M, r), S_1) = S_2$.

The proof of this lemma follows directly from Lemma 8.3.

8.8.4 Step Consistency

Unlike the previous lemmas, the step consistency lemma just states that the step semantics evaluate to the same thing as the corresponding rule or method symbolic semantics. Since Lemmas 8.4, 8.5, and 8.6 relate the rule and method symbolic semantics to their concrete counterparts, this relates the symbolic step semantics to its concrete counterparts as well.

Lemma 8.7 (Step Semantics Consistency)**.** *For all states S, symbolic rule or method names ℓ_{id}, and arguments x, let $\langle g, (S', y)\rangle = \mathsf{symbStep}(M, \ell_{id}, x)$. If E is an environment containing values for symbolic variables in ℓ_{id} and x that are not found in S, then:*

- *If $\mathsf{Eval}(\ell_{id}, S \cup E) = r$ for some rule r, then $\mathsf{Eval}(\mathsf{symbRule}(M, r), S \cup E) = \mathsf{Eval}(\langle g, S'\rangle, S \cup E)$.*
- *If $\mathsf{Eval}(\ell_{id}, S \cup E) = f_a$ for some action method f_a, then $\mathsf{Eval}(\mathsf{symbAMethod}(M, f_a, x), S \cup E) = \mathsf{Eval}(\langle g, (S', y)\rangle, S \cup E)$.*
- *If $\mathsf{Eval}(\ell_{id}, S \cup E) = f_v$ for some value method f_v, then $\mathsf{Eval}(\mathsf{symbVMethod}(M, f_v, x), S \cup E) = \mathsf{Eval}(\langle g, S'\rangle, S \cup E)$.*

The proof of this lemma follows directly from the definition of symbStep.

8.8.5 Execution Consistency

The execution semantics consistency lemma relates the symbolic execution semantics to the concrete execution semantics. This is the final lemma that states the consistency between the symbolic and concrete semantics.

Lemma 8.8 (Execution Semantics Consistency)**.** *For all modules M, states S and S', and execution sequences $\vec{ex}$, if S_{sym} and $\overrightarrow{ex_{sym}}$ are symbolic versions of S and $\vec{ex}$ such that for some environment E, $\mathsf{Eval}(S_{sym}, E) = S$ and $\mathsf{Eval}(\overrightarrow{ex_{sym}}, E) = \vec{ex}$, $M \vdash S \overset{\vec{ex}}{\bullet\!\!\rightarrow} S'$ if and only if $\mathsf{Eval}(\mathsf{symbExec}(M, S_{sym}, \overrightarrow{ex_{sym}}), E) = S'$.*

The proof of this lemma requires proving the equivalence of the reset semantics and applying Lemma 8.7.

8.9 Example Symbolic Semantics

As an example of the symbolic semantics, consider `mkThreeFBuffered` from Figure 5-2. This is the same module used in the example in Chapter 5 as well. In this example, the function `f` is treated as a symbolic variable so its value can be supplied later. The function `f` can be kept as a symbolic variable and passed to an SMT solver as an uninterpreted function to prove properties that hold for all possible functions `f`. Instead of considering the semantics of computing $f(f(f(x)))$ for a specific x (as was done in Chapter 5), this example will construct symbolic representations of the steps of `mkThreeFBuffered` along with the symbolic representation of the initial state.

The symbolic representation of the initial state of `mkThreeFBuffered` is given by

$$\mathsf{symbReset}(\texttt{mkThreeFBuffered}) = \mathbf{and}(\texttt{count} == 0, \mathbf{not}(\texttt{out\_fifo.v1}), \mathbf{not}(\texttt{out\_fifo.v2})).$$

This expression is derived by evaluating the reset expressions of `mkThreeFBuffered` and `mkFIFO2` and adding the necessary prefix to the reset expression of `mkFIFO2` so the state names correspond to the instance `out_fifo`. This expression can be evaluated with a given state S to determine if that state is a legal reset state.

The symbolic semantics for the rules and methods of `mkThreeFBuffered` require getting the symbolic action semantics of the bodies of each rule and method. To see how symbolic action semantics work, consider the body of the step rule:

```
1 let next_val = f(val) in
2 if count == 3 then
3     out_fifo.enq(next_val) ; count <= 0
4 else
5     val <= next_val ; count <= count + 1
```

This action is a let action with an "if" action nested in its body. Evaluating the let action is equivalent to evaluating the body with the symbolic variable bindings obtained from the symbolic semantics of `f(val)`:

$$B = [\texttt{next\_val} \mapsto \mathbf{apply}(f, \mathbf{val})]$$

The symbolic semantics of the "if" action requires getting the symbolic semantics of each branch and then joining them together with a mux. The true action has the update map

$$U_t = [\texttt{out\_fifo} \mapsto \langle true, \Omega \rangle, \texttt{count} \mapsto \langle true, 0 \rangle]$$

where Ω is the result of the call to `out_fifo.enq` (Ω is used as a placeholder to keep the complexity of this example low). The false action has the update map

$$U_f = [\texttt{val} \mapsto \langle true, \mathbf{apply}(f, \mathbf{val}) \rangle, \texttt{count} \mapsto \langle true, \mathbf{apply}(add, \mathbf{count}, 1) \rangle].$$

In the symbolic semantics of the "if" action, the update maps are muxed together using the **mux** function on symbolic maps as defined in Definition 8.6. This gives the following update map for the "if" action (after applying some trivial optimizations):

$$\begin{aligned} U \quad = \quad & [\texttt{out\_fifo} \mapsto \langle \mathbf{apply}(eq, \mathbf{count}, 3), \Omega \rangle, \\ & \texttt{count} \mapsto \langle \mathbf{apply}(eq, \mathbf{count}, 3), 0 \rangle, \\ & \texttt{val} \mapsto \langle \mathbf{not}(\mathbf{apply}(eq, \mathbf{count}, 3)), \mathbf{apply}(f, \mathbf{val}) \rangle, \\ & \texttt{count} \mapsto \langle \mathbf{not}(\mathbf{apply}(eq, \mathbf{count}, 3)), \mathbf{apply}(add, \mathbf{count}, 1) \rangle]. \end{aligned}$$

This update map has two separate entries for `count`, but the guards of the values are mutually exclusive, so they will never trigger a double-write error. If desired, these two entries can be combined into the single entry

$$\texttt{count} \mapsto \langle true, \mathbf{mux}(\mathbf{apply}(eq, \mathbf{count}, 3), 0, \mathbf{apply}(add, \mathbf{count}, 1)) \rangle.$$

The full symbolic semantics of the rules and methods of `mkThreeFBuffered` are presented in Figure 8-8.

8.10 Conclusion

With the symbolic semantics, it is possible to make claims about the semantics and verify them with an SMT solver. In the next chapter, we introduce how rule-based model checking

Start Method

$\mathsf{symbAMethod}(\texttt{mkThreeFBuffered}, \texttt{start}, \mathbf{x}) =$

$\langle \mathbf{apply}(eq, \mathbf{count}, 0), ([\texttt{val} \mapsto \mathbf{x}, \texttt{count} \mapsto 1, \texttt{out\_fifo} \mapsto [$
$\texttt{d1} \mapsto \mathbf{out\_fifo.d1}, \texttt{d2} \mapsto \mathbf{out\_fifo.d2},$
$\texttt{v1} \mapsto \mathbf{out\_fifo.v1}, \texttt{v2} \mapsto \mathbf{out\_fifo.v2}]], \varepsilon)\rangle$

Step Rule

$\mathsf{symbRule}(\texttt{mkThreeFBuffered}, \texttt{step}) =$

$\langle \mathbf{and}(\mathbf{count} \geq 1, \mathbf{mux}(\mathbf{apply}(eq, \mathbf{count}, 3), \mathbf{not}(\mathbf{out\_fifo.v2}), true)),$
$[\texttt{val} \mapsto \mathbf{mux}(\mathbf{apply}(eq, \mathbf{count}, 3), \mathbf{val}, \mathsf{apply}(f, \mathbf{val})),$
$\texttt{count} \mapsto \mathbf{mux}(\mathbf{apply}(eq, \mathbf{count}, 3), 0, \mathbf{apply}(add, \mathbf{count}, 1)),$
$\texttt{out\_fifo} \mapsto [$
$\texttt{d1} \mapsto \mathbf{out\_fifo.d1},$
$\texttt{d2} \mapsto \mathbf{mux}(\mathbf{apply}(eq, \mathbf{count}, 3), \mathsf{apply}(f, \mathbf{val}), \mathbf{out\_fifo.d2}),$
$\texttt{v1} \mapsto \mathbf{out\_fifo.v1},$
$\texttt{v2} \mapsto \mathbf{mux}(\mathbf{apply}(eq, \mathbf{count}, 3), true, \mathbf{out\_fifo.v2})]$
$]\rangle$

Canonicalize Rule

$\mathsf{symbRule}(\texttt{mkThreeFBuffered}, \texttt{out\_fifo.canonicalize}) =$

$\langle \mathbf{and}(\mathbf{out\_fifo.v2}, \mathbf{not}(\mathbf{out\_fifo.v1})),$
$[\texttt{val} \mapsto \mathbf{val}, \texttt{count} \mapsto \mathbf{count},$
$\texttt{out\_fifo} \mapsto [$
$\texttt{d1} \mapsto \mathbf{out\_fifo.d2},$
$\texttt{d2} \mapsto \mathbf{out\_fifo.d2},$
$\texttt{v1} \mapsto \mathbf{out\_fifo.v2},$
$\texttt{v2} \mapsto false]$
$]\rangle$

GetResult Method

$\mathsf{symbAMethod}(\texttt{mkThreeFBuffered}, \texttt{getResult}, \mathbf{t}) =$

$\langle \mathbf{out\_fifo.v1},$
$([\texttt{val} \mapsto \mathbf{val}, \texttt{count} \mapsto \mathbf{count},$
$\texttt{out\_fifo} \mapsto [$
$\texttt{d1} \mapsto \mathbf{out\_fifo.d1},$
$\texttt{d2} \mapsto \mathbf{out\_fifo.d2},$
$\texttt{v1} \mapsto false,$
$\texttt{v2} \mapsto \mathbf{out\_fifo.v2}]$
$], \mathbf{out\_fifo.d1})\rangle$

Figure 8-8: Symbolic semantics of `mkThreeFBuffered`

can be performed using an SMT solver to prove properties such as assertions holding for all reachable states.

Chapter 9

SMT-Based Verification

This chapter introduces the process of SMT-based verification of Spec 'n' Check modules. We want to prove that all the assertions in a design hold for all reachable states or that a module implements a specification. The first step is to present how the symbolic semantics of Chapter 8 are used to create symbolic representations of modules for SMT solvers. This includes taking advantage of term abstraction opportunities allowed by the SMT solver. The next step is to create SMT queries for checking properties. This includes bounded and unbounded model checking, and it also includes checking for deadlock. Finally, an example of performing SMT-based verification on a Spec 'n' Check module is shown.

9.1 SMT Introduction

SMT is the problem of taking a predicate expression defined over free variables of various types and determining if it is *satisfiable*, *i.e.* there exist assignments for variables that make the predicate expression evaluate to true. This is similar to SAT, except SAT requires the predicate to be a function over Boolean types while SMT allows the use of various *theories* including natural numbers with linear arithmetic, bit vectors, arrays, uninterpreted functions, and abstract types.

SMT solvers are programs that take in such predicate expressions and return whether or not they are satisfiable. For a satisfiable predicate expression, an SMT solver can also return a set of values for the variables that satisfy the predicate. SMT solvers can be used to

verify predicates hold for all inputs by checking if the negation of the predicate is satisfiable; if it is not satisfiable, then the predicate holds for all inputs, but if it is satisfiable, then the satisfying solution gives inputs which result in a counterexample.

9.1.1 Motivating Example: Verifying Guards of a GCD Module

As a simple motivating example, consider verifying the property that there is at least one ready rule or method for all reachable states in the `mkGCD` module shown in Figure 9-1. Since there are no implicit guards caused by method calls in any of the rules or methods, the guards are just the explicit guards written as part of the rule and method definitions.

```
module mkGCD(FuncUnit#(Tuple2#(Bit#(8), Bit#(8)), Bit#(8));
  State#(Bit#(32)) x <- mkState;
  State#(Bit#(32)) y <- mkState;
  State#(Bool) busy <- mkState;
  reset(!busy);
  rule swap(busy && ((x > y) || (x == 0)) && (y != 0));
    x <= y;
    y <= x;
  endrule
  rule subtract(busy && ((x <= y) && (x != 0)) && (y != 0));
    y <= y - x;
  endrule
  method start(Tuple2#(Bit#(8), Bit#(8)) in) if (!busy);
    x <= tpl_1(in);
    y <= tpl_2(in);
    busy <= True;
  endmethod
  method getResult() if (busy && (y == 0));
    return x;
  endmethod
endmodule
```

Figure 9-1: `mkGCD` module

To show that there is always at least one ready rule or method, we construct this as the Boolean predicate

$$g_{\text{swap}} \vee g_{\text{subtract}} \vee g_{\text{start}} \vee g_{\text{getResult}}$$

where

$$\begin{aligned} g_{\text{swap}} &= \text{busy} \wedge ((x > y) \vee (x = 0)) \wedge (y \neq 0) \\ g_{\text{subtract}} &= \text{busy} \wedge ((x \leq y) \wedge (x \neq 0)) \wedge (y \neq 0) \\ g_{\text{start}} &= \neg\text{busy} \\ g_{\text{getResult}} &= \text{busy} \wedge (y = 0). \end{aligned}$$

We want this predicate to hold for all reachable states of `mkGCD`, but we can simplify the verification process if we instead just show the predicate holds for all possible values of the states of `mkGCD`. This is simpler because it is not necessary to worry about which states are reachable.

To verify this with an SMT solver, we feed the solver the negation of the property we want to show:

$$\neg(g_{\text{swap}} \vee g_{\text{subtract}} \vee g_{\text{start}} \vee g_{\text{getResult}}).$$

If the solver finds a solution, then that means there is a combination of state values that violates the property we are trying to prove. If the counterexample is a reachable state, then the property is false, but if it is not a reachable state, then the results are inconclusive, and the property may still hold for all reachable states.

This verification can be done automatically from the source of the designs if the guards from the symbolic semantics of each rule and step can be used in an SMT solver. The next section shows how the symbolic expressions used in Chapter 8 can be easily translated to the standard input language for SMT solvers.

Furthermore, if this property doesn't hold for some unreachable state, then unrolling techniques and additional invariants can be used to restrict the search space to fewer unreachable states. This process is addressed later in this chapter.

9.2 Representing Modules for SMT Solvers

In order to check properties about our rule-based modules using SMT-based model checking, we need a representation of our module in a format acceptable to SMT solvers. A standard format adopted by most SMT solvers is the SMT-LIB input language [17]. SMT-LIB uses a Lisp-like syntax to express free variables, symbolic expressions, assertions, and commands

to the SMT solver to perform checks. Note that the reference implementation presented in Chapter 10 uses the Python API of the Z3 SMT solver which has more features than SMT-LIB, but for generality, we assume capabilities common to SMT-LIB for examples in this chapter.

9.2.1 Handling Restrictions in SMT-LIB

The symbolic semantics used in Chapter 8 can be supported by SMT-LIB after two minor changes.

First, hierarchical variables such as $\mathbf{m.x}$ are not supported in SMT-LIB, so they must be flattened with hierarchical-looking names like `m_x`. This is not a problem because hierarchical variable names are just used as a convenience in the symbolic semantics to simplify working with state variables from different submodules at the same time.

Second, SMT-LIB requires all expressions to be well-typed. In the semantics, all expressions are assumed to be well-typed, so this is not a problem for the expression or action symbolic semantics. On the other hand, well-typed expressions are a problem for the symbolic step semantics produced by the $\mathsf{symbStep}(M, \ell_{\text{top}}, x)$ function because the symbolic variable x is used as the argument to all methods. If two methods expect arguments of different types, then x is not well-typed. To fix this problem, we split x into separate variables, one for each method that expects an argument. A similar fix has to be applied to $\mathsf{symbExec}(M, \overrightarrow{ex})$ to make all the terms $\overrightarrow{ex}$ well-typed, but $\overrightarrow{ex}$ contains output variables from methods, so they have to be split into one value per method as well.

Other than the noted changes, the symbolic semantics in Chapter 8 can be easily translated to SMT-LIB.

9.2.2 SMT Representation for a Single Step

Now we look at the details of constructing SMT-LIB expressions for model checking, starting with a single step. As described before, we need to add multiple argument inputs to symbStep to support method calls whose arguments are different types. We make two further changes to symbStep for verification and efficiency purposes.

First, we add the symbolic expressions corresponding to all the state and step assertions in the design. The state assertion is applied to the input state, and the step assertions are considered as outputs for each rule and method. The step assertions are only considered when the rule or method is ready, so by or'ing each assertion with the negation of the guard, we get an expression for each rule and method that should always be true if the assertion holds in the corresponding rule or method. These modified assertions are and'ed together with the state assertion to get a summary of all the assertions for the entire step; this assertion summary is false if and only if there is a violated assertion. By checking this assertion summary is always true for each step, this simplifies the SMT search because it gives the solver more power to easily find a counterexample.

The second change is that the expression for the next state always assumes a rule or action method is fired. If there is no ready rule or action method for the current state, then we say the module is *deadlocked*. Even though it may still be possible to call value methods from a deadlocked state, there is no way for the state to change since all rules and methods that can change the state are not ready. The guards of all the rules and action methods are or'ed together to get a readiness summary expression which is true if there is a ready rule or action method and false if the module is deadlocked.

Value methods and empty steps are not considered for steps because these do not change the state. The step assertions found within value methods are checked as part of the assertion summary expression, so the value methods do not have to be explicitly executed to check the method's step assertions.

We can present the SMT representation for a step in a circuit-like manner shown in Figure 9-2. The module represented in this figure has an action method f_a, a value method f_v, and two rules r_1 and r_2. This representation takes in a symbolic state S, an action method or rule to fire ℓ_{id}, and arguments x_1 and x_2 for the potential method calls. This representation produces a symbolic next state S' and guard g for the step picked by ℓ_{id}, and it also produces the assertion summary a and the readiness summary rdy.

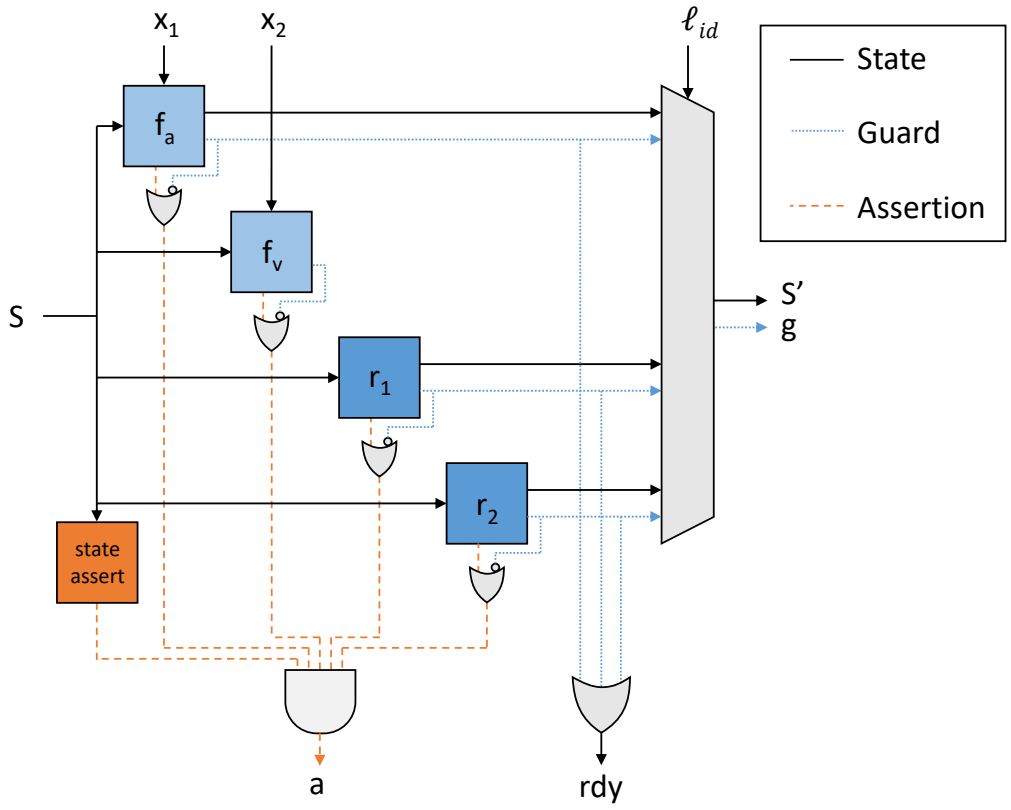

Figure 9-2: SMT Representation of a step

9.2.3 SMT Representation of a Sequence of Steps

Multiple instances of SMT representations of steps can be chained together to produce the SMT representation of an unrolled sequence of steps. An SMT representation of a step unrolled n times is shown in Figure 9-3.

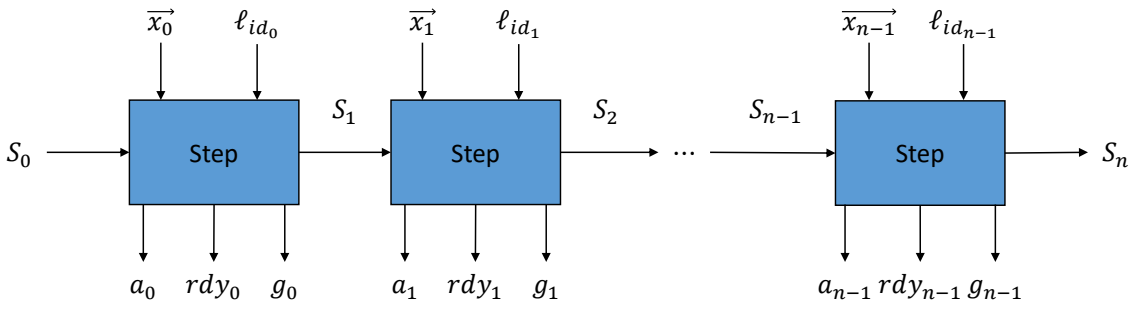

Figure 9-3: SMT representation of a sequence of steps

The states in this sequence are S_0 to S_n. For each step i, the selected rule or action method is $\ell_{\text{id},i}$, and the arguments for the methods in the step are $\vec{x}_i$. Each step produces

the guard of the selected step g_i, the assertion summary a_i, and the readiness summary rdy_i. For ease of use in some model-checking routines, sometimes an n-step execution adds an extra step after S_n just for checking the expressions a_n and rdy_n to check if the assertions hold and if there is a ready rule for the final state.

Note that this sequence of steps is just back-to-back step semantics, and if any of the guards are not satisfied, the sequence of steps is not legal.

One issue with this unrolling is that if each step uses the output state of the previous step as its input state, then there is a potential for the size of expressions to get exponentially large as the number of steps in the unrolling increases. To avoid this, the n-step unrolling can be implemented as n separate 1-step unrollings with free variables as inputs for each step and SMT constraints to ensure the output state of each step is equal to the input state free variable of the next step. By implementing the unrolling this way, an unrolling can be easily extended in either direction without changing the symbolic expression for each step. This is especially beneficial for k-induction as introduced later in this chapter.

9.2.4 Getting Legal Executions

The only thing required to get a legal execution from the SMT representation of a sequence of steps presented in Figure 9-3 is SMT constraints (implemented using `assert` in SMT-LIB). By putting proper constraints on the inputs and outputs of a sequence of steps, it is possible to restrict all solutions to the constraints to be legal executions of the module.

The necessary constraints are that S_0 is a legal reset state, and for each step i, g_i is true. The reset predicate can be represented as a symbolic expression using symbReset, so symbReset can be used to produce the function reset(S) that returns a symbolic expression denoting if S is a legal initial state.

If there is no solution to these constraints, then there are no legal executions that fire n consecutive action methods or rule.

9.2.5 Using Abstractions

SMT solvers support two abstractions that are very beneficial for SMT-based model checking of hardware: abstract types and uninterpreted functions. Abstract types are types with an unspecified number of values. The only functions defined on abstract types ar equality and `ite()`, *i.e.* if-then-else or mux. Uninterpreted functions are functions without definitions. The only thing known about uninterpreted functions is that if f is an uninterpreted function, then for all x and y, $x = y \implies f(x) = f(y)$. It is possible for uninterpreted functions to use abstract types as argument types or return types, so the two abstractions can be used together.

The most natural place to use these abstractions is in place of parameters. FIFO definitions are typically parameterized by the type of data stored in the FIFO, so when constructing a symbolic representation of a FIFO for SMT-based model checking, the data type can be an abstract type. If the FIFO is verified with the abstract type as its type parameter, then the verification results hold for all possible types.

The functional units in Chapter 3 are all parameterized by a function parameter f. This function parameter also implies a type parameter t. Therefore these modules can be represented for SMT using abstract types and uninterpreted functions. Any model-checking results that hold using these abstractions also hold for any specific parameters used in place of these abstractions.

The other common place to use abstractions in SMT-based model checking is for term abstraction [30]. Term abstraction is the process of using uninterpreted functions and other abstractions in place of their full hardware definitions in order to simplify the work done by the SMT solver during verification. This process can result in large increases in performance for the SMT solver, especially when used in verifying a miter module where the same abstractions are used in the implementation and the specification.

9.3 Bounded Model Checking

Now that we have a way to symbolically express legal executions of n steps starting at reset, we can check if there exists such a sequence that violates an assertion. This process is known as *bounded model checking*.

To perform bounded model checking of depth n, we get the symbolic expressions for n consecutive steps (Figure 9-3) and we assume the constraints used to construct a legal execution (*i.e.* $\mathsf{reset}(S_0)$ and $\bigwedge_{i=0}^{n-1} g_i$). To see if assertions are ever violated, we add a constraint that at least one assertion is violated, *i.e.* $\bigvee_{i=0}^{n} \neg a_i$. This gives the following SMT constraints:

initial state is a reset state:	$\mathsf{reset}(S_0)$
selected rules and methods are ready:	$\bigwedge_{i=0}^{n-1} g_i$
there is a violated assertion:	$\bigvee_{i=0}^{n} \neg a_i$

When the SMT solver is run to find a solution for these constraints, if it finds a solution, then that solution is a legal execution that violates at least one assertion, thus implying the assertions do not hold for all possible executions. On the other hand, if the SMT solver returns unsat (meaning there is no solution to the constraints) then there are no executions containing n rules or methods that violate an assertion.

9.3.1 Considering all Shorter Executions

These results do not hold for shorter sequences due to the possibility of deadlock. If there exists a k-step execution with $k < n$ that ends in a state that violates the state assertion and is deadlocked, this would not meet the constraints for a legal execution of n steps because there is no way for g_k to be true.

There are two ways to do bounded model checking of depth n and include all shorter executions. The first way is to adjust the constraints so that g_i can be false if there is no ready rule or method (*i.e.* rdy_i is false). When rdy_i is false, all g_k for $k \geq i$ and all a_j for $j > i$ are ignored. This is done by changing the constraints to the following:

initial state is a reset state:	$\mathsf{reset}(S_0)$
selected rules and methods are ready:	$\bigwedge_{i=0}^{n-1} \left(g_i \vee \neg \bigwedge_{j=0}^{i} rdy_j \right)$
there is a violated assertion:	$\bigvee_{i=0}^{n} \left(\neg a_i \wedge \bigwedge_{j=0}^{i-1} rdy_j \right)$

The second way to do bounded model checking of depth n and include all shorter executions is to perform the model checking incrementally with increasing depth. At each step $i \leq n$, you are only checking for assertion violations in sequences of steps including i rules or action methods. Since you are checking for each length of sequence, you only need to check for violated assertions in the last step; any violated assertions in earlier steps are covered by shorter sequences of steps. Therefore the constraints become the following:

for all $i \leq n$:

initial state is a reset state:	$\mathsf{reset}(S_0)$
selected rules and methods are ready:	$\bigwedge_{j=0}^{i-1} g_j$
the final state has a violated assertion:	$\neg a_i$

9.3.2 Checking for Deadlock

Since it is possible to detect deadlock through the rdy_i expressions, it is possible to check for deadlock using bounded model checking by treating rdy_i as an assertion the SMT solver is trying to violate. Checking for deadlock can significantly simplify searching for violated assertions with shorter traces because any case where rdy_i is false is an error. The first formulation in Section 9.3.1 can easily by changed to search for deadlocked cases; the resulting constraints are:

initial state is a reset state:	$\mathsf{reset}(S_0)$
selected rules and methods are ready:	$\bigwedge_{i=0}^{n-1} \left(g_i \vee \neg \bigwedge_{j=0}^{j \leq i} rdy_j \right)$
there is a violated assertion or deadlock:	$\bigvee_{i=0}^{n} (\neg a_i \vee \neg rdy_i)$

Note that the constraint that selected rules and methods are ready can be replaced with the constraint

$$\bigwedge_{i=0}^{n-1} (g_i \vee \neg rdy_i) .$$

This constraint is easier to write, but it puts more constraints on the SMT solver, requiring it to pick a ready rule or action method each time one exists, even after the design reached a deadlocked state and a nonready rule or action method was selected to execute.

9.3.3 Incremental Bounded Model Checking Algorithm

Many SMT solvers can benefit from incremental solving, that is, using results from previous SMT searches to improve the performance of later searches with related constraints. When using incremental solving, SMT constraints are added and removed using a stack. Z3-inspired pseudocode for the full incremental bounded model checking algorithm is shown below.

```
s = Solver()
s.assert(reset(S_0))
for i in 0 to n:
    s.push()
    s.assert(¬a_i ∨ ¬rdy_i)
    if s.check() == sat:
        print('Violated assertion or deadlock')
        return False
    s.pop()
    s.assert(g_i)
    # we know this is true from previous check,
    # so we add it here to help the SMT solver
    s.assert(a_i)
print('No violated assertion or deadlock found')
return True
```

This procedure performs bounded model checking of depth n to find all possible executions of length n or less which result in violated assertions or deadlock. In this procedure, we start with 0-step sequences (*i.e.* all possible reset states) and incrementally build up until the entire n-step sequence is considered. At each step, a search is done to find a legal execution that ends with a final state that either has a violated assertion or no ready rule or action method. If there is no such state, then the constraint looking for a violated assertion or

deadlock is removed by popping the stack, and the search continues to longer sequences of steps until the limit n is reached.

9.3.4 Applying Bounded Model Checking Results to Unbounded Executions

A successful bounded model checking verification with bound n implies that the checked assertions hold for the first n steps of any execution of the module. If the module has a finite number of reachable states, then it is possible to choose an n large enough so the bounded model checking includes all possible reachable states. This is not typically a feasible way to verify assertions hold for all reachable states because the exact value of n can be hard to compute [22], and for many interesting designs, its value is too large for bounded model checking.

A simpler version of this strategy can be done by detecting if all the traces under some bound n return to a reset state. If that is true, then bounded model checking with depth n covers all reachable states. While many modules do not have this behavior, it is common for nonpipelined functional units such as `mkThreeF` from Figure 3-2 to return to a reset state between uses.

It is possible to detect if all the traces under a given length return to a reset state by doing a form of bounded model checking that asserts that all states other than the initial state do not satisfy the reset predicate. This is done by performing the SMT search with the following constraints:

initial state is a reset state:	$\mathsf{reset}(S_0)$
no other state is a reset state:	$\bigwedge_{i=1}^{n} \neg\mathsf{reset}(S_i)$
selected rules and methods are ready:	$\bigwedge_{i=0}^{n-1} g_i$

If there is no execution that satisfies these constraints, then all traces of length n return back to reset, and therefore successful bounded model checking of length n implies the assertions hold for all reachable states.

The check for executions returning to a reset state can be integrated into the incremental

bounded model checking algorithm to produce a new bounded model checking algorithm that may return that an assertion holds for all reachable states instead of just the first n states. The new algorithm is shown below:

```
s = Solver()
s.assert(reset(S_0))
for i in 0 to n:
    if i > 0:
        s.assert(¬reset(S_i))
        if s.check() == unsat:
            print('Assertions holds for all reachable states')
            return True
    s.push()
    s.assert(¬a_i ∨ ¬rdy_i)
    if s.check() == sat:
        print('Violated assertion or deadlock')
        return False
    s.pop()
    s.assert(g_i)
    # we know this is true from previous check,
    # so we add it here to help the SMT solver
    s.assert(a_i)
print('No violated assertion or deadlock found')
return True
```

This algorithm performs an additional SMT search each loop iteration, so it should only be used if there is a chance the module always returns to reset.

For modules that do not always return to a reset state, we rely on induction-driven unbounded model checking as presented in the next section.

9.4 Unbounded Model Checking

Bounded model checking can check for the presence of bounded-length executions with violated assertions. Ideally we would like to verify a module for all possible executions of any, potentially unbounded length. To do this, we combine bounded model checking with a form of induction to get unbounded model checking [44, 74].

9.4.1 Standard Induction

In the simplest case of induction, we define the property $P(i)$ as "in the i^{th} step of all executions of our module to verify, all the assertions hold and the module is never deadlocked". We want to prove $P(i)$ is true for all $i \geq 0$. When proving this property by induction, the base case is $P(0)$, and the induction step is $\forall i \geq 0,\ P(i) \implies P(i+1)$; if we can prove each of these, then the property holds for all $i \geq 0$.

The base case in this induction is simply the degenerate case of bounded model checking where $n = 0$. This is just a check for if there is a reset state that violates an assertion or is deadlocked. The constraints for verifying the base case are below:

Base case:

initial state is a reset state:	$\mathsf{reset}(S_0)$
initial state violates an assertion or is deadlocked:	$\neg a_0 \vee \neg rdy_0$

If there is no solution to these constraints, then $P(0)$ holds.

The induction step can be proven using an SMT search on a single step of the module with the following constraints:

Induction step:

initial state satisfies assertion:	a_0
step is ready:	g_0
final state violates an assertion or is deadlocked:	$\neg a_1 \vee \neg rdy_1$

If these constraints produce no results, then they imply that for all $i \geq 0,\ P(i) \implies P(i+1)$. Combining with the base case, this proves that $P(i)$ holds for all $i \geq 0$, and thus, for all reachable states of the module, the assertions hold and there is no deadlock.

9.4.2 Shortcomings of Standard Induction

Checking the induction step constraints presented in the previous section is not exactly the same as verifying $\forall i \geq 0,\ P(i) \implies P(i+1)$. The induction step $\forall i \geq 0,\ P(i) \implies P(i+1)$

has a notion of *reachability* from the predicates $P(i)$ since $P(i)$ is that the assertions hold for the i^{th} element in all legal executions. On the other hand, the induction-step SMT constraints are equivalent to checking if the assertions are an *inductive invariant*, *i.e.* if the assertions hold for *any* state (reachable or not), then the assertions will hold for the next state as well.

Due to this difference, if the induction-step SMT search produces a counterexample trace that satisfies the constraints, it does not imply that there exists an execution that violates an assertion, it just implies the assertions do not make up an inductive invariant. Induction-step counterexamples produce inconclusive results towards verifying if the assertions hold for all reachable states. If the initial state in the result produced by the SMT solver is a reachable state, then the assertion violation is reachable, meaning the assertion does not hold for all reachable states. Otherwise if the initial state is not reachable, then the SMT solver produced a spurious counterexample, and more work must be done to determine if the assertions hold or not.

As an example, consider the verification of a counter module that counts modulo 10, specifically verifying that values outside the range 0 to 9 are never reached. The implementation of this module is shown in Figure 9-4. This implementation has 3 different state assertions.

```
1  module mkCountMod10 =
2      state x
3      reset(x == 0)
4      assert(x < 10)
5      assert(x != 10)
6      assert(x != 15)
7
8      amethod step = λ _.
9          if (x == 9)
10             then x <= 0
11             else x <= x + 1 ;
12         x
```

Figure 9-4: Module for counter modulo 10

The first state assertion is the strictest; this assertion asserts that every state that should

not be reachable is not reachable. The other two are weaker, one just saying 10 is not reachable and the other saying 15 is not reachable. Both of these are implied by the first assertion, but verifying them without using the first assertion take a different amount of effort.

The assertions `x < 10` and `x != 10` are each inductive invariants, *i.e.* if we consider any state that satisfies these assertions, the next state after executing `step` will also satisfy these assertions. Since these are inductive invariants, the SMT-based induction presented in the previous section can be used to prove the assertions always hold.

On the other hand, the assertion `x != 15` is not an inductive invariant because if `x` is 14 (which satisfies the assertion), then firing the `step` method will result in `x` being 15 which violates this assertion. We present two ways to fix this problem so it is possible to prove that `x != 15` for all reachable states.

The first way is to strengthen the assertion (make it constrain more) so it becomes an inductive invariant. The strengthened assertion must at least include `x != 14` so the previous counterexample is no longer valid. Just adding `x != 14` produces the assertion `x != 15 && x != 14` which is still not an inductive invariant because `x` being 13 satisfies the assertion. Continuing this process results in a strengthened assertion that says `x` is not in the range from 10 to 15. Another possible strengthening is replacing `x != 15` with the much stronger assertion `x < 10` from the start instead of adding states one-by-one to the assertion.

The second way is to look at longer sequences of steps in the induction step. In the example of verifying the assertion `x != 15`, if we looked at sequences of steps that were 6 steps long, then it would be impossible to construct a sequence of steps that violates `x != 15` in the final state. The next section presents this idea in its general form: *k-induction.*

9.4.3 K-Induction

K-induction is a stronger form of induction that uses sequences of steps in the induction step instead of just a single step. For example, 2-induction uses the induction step

$$\forall i \geq 0,\ (P(i) \wedge P(i+1)) \implies P(i+2).$$

In order to complete the induction, k-induction requires the base case to also include a sequence of steps. In the 2-induction example, the base case is $P(0) \wedge P(1)$.

Formally, k-induction is used to prove $\forall i \geq 0,\ P(i)$ using the following implication:

$$\underbrace{\left(\bigwedge_{i=0}^{k-1} P(i)\right)}_{\text{base case}} \wedge \underbrace{\forall i \geq 0, \left(\left(\bigwedge_{j=i}^{i+k-1} P(j)\right) \implies P(i+k)\right)}_{\text{induction step}} \implies \forall i \geq 0,\ P(i)$$

When verifying k-induction with an SMT solver, the base case is bounded model checking with depth $n-1$.

The induction step of k-induction can be verified using an SMT solver by looking for a solution for the following constraints using the same unrolling used by bounded model checking:

Induction step:

initial states satisfy assertions:	$\bigwedge_{i=0}^{k-1} a_0$
selected rules and methods are ready:	$\bigwedge_{i=0}^{k-1} g_0$
final state violates an assertion or is deadlocked:	$\neg a_k \vee \neg rdy_k$

The benefit of k-induction comes from the extra execution history considered when doing the induction step. This extra history helps filter out unreachable states and therefore helps make the induction step hold when verifying with an SMT solver. For example, when verifying the mod 10 counter from before, the assertion `x != 15` was not an inductive invariant, but when looking at the induction step of 6-induction, the SMT search returns no solutions, meaning the induction step holds. This is because the 6-step unrolling has enough history to easily show that `x` cannot reach 15.

9.4.4 Incremental Unbounded Model Checking Routine

We can combine bounded model checking and k-induction into a single incremental SMT-based proof procedure with max depth n. This procedure performs perform bounded model checking and k-induction for increasing depths starting with depth 0, so if the verification is successful, it is done with the minimum depth required for the k-induction to hold.

This procedure works by interleaving k-induction and bounded model checking of increasing depth. At each step, if the k-induction is successful, the property holds for all steps. If the bounded model checking fails, then a counterexample to the property was found. If the routine reaches the maximum depth without either happening, then the results are inconclusive.

```
bmc_s = Solver()
kind_s = Solver()
bmc_s.assert(reset(S_0))
for i in 0 to n:
    # K-Induction step
    if i == 0:
        kind_s.add(¬a_{n-i} ∨ rdy_{n-i})
    else:
        kind_s.add(a_{n-i} ∧ g_{n-i})
    if kind_s.check() == unsat:
        print('Assertions hold for all reachable states')
        return True
    if i == n:
        break

    # BMC Step
    bmc_s.push()
    bmc_s.add(¬a_i ∨ ¬rdy_i)
    if bmc_s.check() == sat:
        print('Assertion violation found')
        return False
    bmc_s.pop()
    bmc_s.add(a_i ∧ g_i)
print('Inconclusive results')
return None
```

When this process yields inconclusive results (`None` in the procedure above), it is because the k-induction step kept finding potential counterexamples. If these examples use unreachable states, then the assertions may still hold for all reachable states, and just like the simple induction case it is necessary to either strengthen the assertion to make the k-induction step successful or increase the maximum depth of the search.

9.5 Example SMT-Based Model Checking

As an example of SMT-based model checking, we consider verifying `mkThreeFBuffered` shown in Figure 5-2 against the specification `mkFuncUnitConcurrentSpec` from Section 3.6. To simplify the narrative in this example, we will ignore the internal implementation of `mkFIFOSpec` and assume it is a correctly implemented FIFO specification that can hold at least two elements.

Since we are verifying a module against a specification, we use a rule-based miter module construction to compare the two modules, specifically the miter in Figure 7-1, and we are trying to show the miter's assertions hold for all reachable states to prove `mkThreeFBuffered` $\sqsubseteq$ `mkFuncUnitConcurrentSpec`.

In a real verification situation, we would use an uninterpreted function for `f` and an abstract type for the input and output type of `f` and the type of the elements in the FIFOs. For the purposes of this illustrative example, we will assume `f` is the function $f(x) = x + 1$, and the data type is unsigned 8-bit integer.

To start out, we do bounded model checking to make sure there is no easily detected bug in `mkThreeFBuffered` that makes it behave differently than `mkFuncUnitConcurrentSpec`. When bounded model checking with depth 10 is performed, the SMT solver finds no executions that violate assertions, so we have confidence that the module behaves as expected. Note that the module `mkThreeFBuffered` can execute forever without returning to a reset state (it can always keep at least one element in the FIFO), so the bounded model checking algorithm that checks for executions that return to reset is not able to verify the assertions in the miter for all reachable states.

To show the assertions in the miter hold for all reachable states, we use k-induction. Running k-induction for depth 3 produces inconclusive results because the SMT solver finds a counterexample execution for the k-induction step. One such execution is shown below: (Note that in these execution diagrams, the specification's FIFO is shown as a list of the elements in the FIFO where the first element in the list is the oldest value in the FIFO.)

state number	executed step	implementation						specification
		val	count	out_fifo				fifo
				d1	d2	v1	v2	
0	`step`	8	2	0	0	false	false	[255]
1	`step`	9	3	0	0	false	false	[255]
2	`canonicalize`	9	0	0	10	false	true	[255]
3	—	9	0	10	10	true	false	[255]

In this execution, at state 3, calling `getResult` on the implementation produces the result 10 (seen in `out_fifo.d1`), but the specification produced the result 255. This should not be possible because if the implementation and the specification are given the same argument, then the two modules should produce the same result. Since this execution does not include the start method being called, there is nothing that forces the implementation and the specification to be computing the same result. Therefore we want to increase the depth used for k-induction.

We increase the depth used for k-induction until the counterexample includes the `start` method that matches the final `getResult`. At depth 11, all executions that end with `getResult` must also include the corresponding `start` method call. Performing k-induction with depth 11 still produces a counterexample. The last four states of one such counterexample are shown below:

state number	executed step	implementation						specification
		val	count	out_fifo				fifo
				d1	d2	v1	v2	
...								
8	`step`	8	2	0	0	false	false	[255, 10]
9	`step`	9	3	0	0	false	false	[255, 10]
10	`canonicalize`	9	0	0	10	false	true	[255, 10]
11	—	9	0	10	10	true	false	[255, 10]

Inspecting this counterexample reveals that the implementation and the specification are out-of-sync; the implementation has 1 pending result while the specification has two.

This should not be possible, so we add an assertion to the miter module that says the number of elements in the implementation's `out_fifo` matches the number of elements in the specification's `fifo`. This requires an extended state assertion because the assertion must use state variables within the implementation and the specification.

Once the assertion has been added, repeating the k-induction produces successful results implying the miter assertions hold for all reachable states. Therefore `mkThreeFBuffered` $\sqsubseteq$ `mkThreeFBufferedSpec` is proven by SMT-based model checking the miter.

9.6 Conclusion

In this chapter we presented how an SMT solver can be used to perform various kinds of model checking on Spec 'n' Check designs using the symbolic representations of modules from Chapter 8. In the next chapter we present our prototype implementation of Spec 'n' Check including routines for automating the model checking shown in this chapter.

Chapter 10

ProtoRMC: Prototype Implementation of Rule-Based Hardware Model Checking

In this chapter we introduce ProtoRMC, the prototype implementation of our rule-based hardware model checking framework. ProtoRMC is a prototype implementation of Spec 'n' Check written in Python as an embedded domain-specific language (EDSL), so therefore it differs in syntax from Spec 'n' Check but not in semantics. ProtoRMC also contains the tools necessary to produce a symbolic representation of the module for use with SMT solvers. Furthermore, ProtoRMC has functions to perform model checking and some SMT-based visualizations to better understand and verify the design.

10.1 ProtoRMC Structure

ProtoRMC consists of multiple layers to implement the semantics of Spec 'n' Check and the tools for model checking as seen in Figure 10-1. At the core is the Z3 SMT solver [43]. To better suit hardware designs, ProtoRMC has a wrapper around the Z3 Python API called Z3W to support features that are necessary for hardware design. The types and functions in Z3W are the basis of the expression language of the rule-based hardware design EDSL. The module descriptions are then turned into symbolic representations with the SymbolicModule class. The SymbolicModule class has methods for model checking (including multiple variants of bounded model checking and k-induction), and it has methods for visualizations.

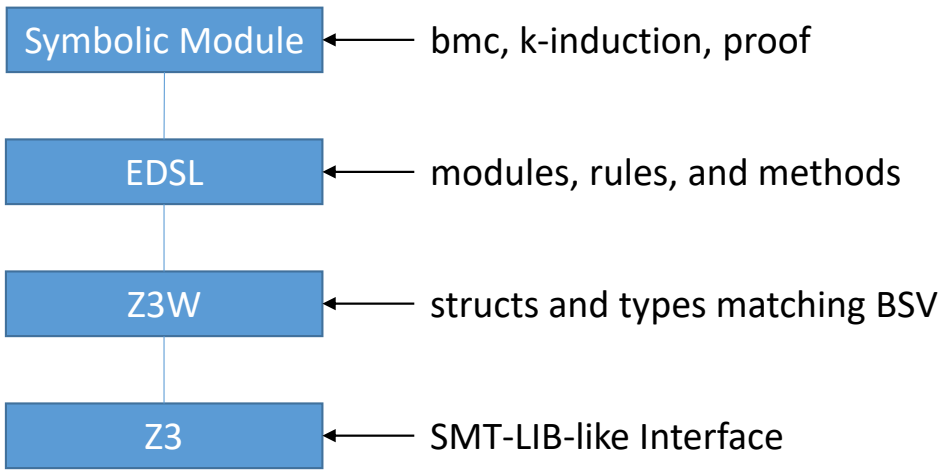

Figure 10-1: Components of ProtoRMC

In the following sections we will go through each component of ProtoRMC in more detail, starting with Z3W and continuing up the stack.

10.2 Z3W: Wrapper for Z3's Python API

The Z3 API provides an interface to the Z3 solver that matches the types and functions in SMT-LIB. Our wrapper for Z3, Z3W, provides types, functions, and an overall interface that is better suited for a hardware EDSL.

The majority of the types and functions exposed by Z3's Python API are the same types and functions that are part of SMT-LIB [17]. That means there is no distinction between signed and unsigned bit vectors; there is just a single bit vector type and separate functions for signed and unsigned operations on those bit vectors. On the other hand, BSV has three types of bit vectors: `Bit#(n)`, `Int#(n)`, and `UInt#(n)` (Bit is nearly equivalent to UInt). These types are all n-bit vectors, but functions such as comparison operators are defined differently depending on whether the type corresponds to a signed number or an unsigned number. Therefore the implementation of `x < y` in BSV depends on the type of `x` and `y`. With this distinction, a class of arithmetic bugs can be avoided through type checking.

When Z3W wraps Z3's bit vectors, it exposes two types of bit vectors, `UBitVec(n)` for unsigned bit vectors and `SBitVec(n)` for signed bit vectors. This is to avoid arithmetic bugs where a signed operator is performed on unsigned bit vectors or vice versa. This is done by

constructing signed and unsigned wrapper classes for bit vectors. These classes define all the desired operators on bit vectors, and each operator calls the correct function from the Z3 API depending on the type of the class.

Another feature included in Z3W is structs. Z3W allows for typed structs to be defined as Python classes using type hint syntax. For example, consider the struct for the `FetchToDecode` type used in a simple processor. This struct defines the data passed from the fetch stage to the decode stage. In BSV, the `FetchToDecode` struct is defined as follows:

```
typedef struct {
    Bit#(32) pc;
    Bit#(32) ppc;
    Bool epoch;
    Bit#(2) prv;
} FetchToDecode deriving (Bits);
```

Using Z3W, the same structure is defined as:

```
class FetchToDecode(Struct):
    pc : UBitVec(32)
    ppc : UBitVec(32)
    epoch : Bool
    prv : UBitVec(2)
```

Field values of Z3W structs are accessed the same way as in BSV. For example, accessing the `pc` field of `x` is written `x.pc`.

Z3W also includes vectors as an extension of structs. Vectors are structs where the items are given automatically enumerated names, and entries can be accessed using square bracket notation (`x[1]`) where the field in the square bracket is either an integer constant or something that can be compared to an integer constant.

Z3W also provides other means to simplify writing Z3 expressions.

10.3 Hardware Design EDSL

The EDSL for rule-based hardware design is a mixture of Python, BSV, and Spec 'n' Check syntax. Despite resembling Python syntax, it is not executed as a Python program. Instead, modules are parsed using Python's built-in `ast` library, and then the AST is processed

into a symbolic representation. This approach is necessary in order to support Python `if` statements and the Booolean operators `and`, `or`, and `not`.

10.3.1 Structure of a Module

An example modulus-3 counter written in the EDSL can be seen below:

```
class mkMod3Counter(Module):
    val : UBitVec(2)

    @reset
    def reset(self):
        assert self.val == 0

    @method
    def increment(self):
        if self.val == 2:
            self.val <= 0
        else:
            self.val <= self.val + 1

    @method
    def getValue(self) -> UBitVec(2):
        return self.val

    @predicate
    def checkVal(self):
        assert self.val < 3
```

Modules are written as Python classes extending the ProtoRMC class `Module`. Modules are made up of two parts: declarations and functions.

States and submodules are declared at the top of the module using the syntax *`name : type`*, so in the above example, the line `val : UBitVec(2)` declares state `val` as a 2-bit unsigned bit vector.

Functions inside the module are used to define the reset predicate, methods, rules, and predicates used for verification. The decorators `reset`, `method`, `rule`, `predicate`, and `verif` are used to label the type of each function.

Functions decorated with `reset` are used to specify the reset predicate of the module. The assertions in this function are treated as the reset predicate for the module, and the

return value is ignored.

Rules are denoted as `rule`-decorated functions that take no arguments and have no return values. Such a function describes a state transition function, along with a condition when the state transition is legal (guard) and sets of assertions and assumptions that are expected to hold. Guards are expressed using the same syntax as Spec 'n' Check.

Methods are denoted as `method`-decorated functions that may take arguments and may have return values. Other than arguments and return values, methods are very similar to rules. ProtoRMC determines if methods are value methods or action methods automatically. If there are any state updates in the method, it is an action method; otherwise it is a value method.

The `predicate`-decorated functions are functions that just contain predicates, or assertions, to check within the design. In the example above, the `checkVal` predicate ensures that for each reachable state, `val` is less than 3. Predicate functions are used to implement state assertions.

The final type of function is `verif`-decorated functions. These functions are used to simplify the verification process by making it easier to write extended assertions that look into the internal state of a submodule. Instead of writing expressions that depend on multiple state variables within a submodule, the submodule can have a `verif` function that returns values computed from the internal state that are useful for extended assertions but are not included as an interface method. Because of how they are used, `verif` functions cannot contain any guards, state updates, or method calls (but they can call other `verif` functions of submodules).

As an example of a `verif` function, consider verifying a module that contains FIFO submodules that should always have the same number of elements in each. To simplify the extended assertion that ensures this property, each FIFO can expose a `num_elements verif` function that returns the number of elements in the FIFO, and the extended assertion in the outer module would just be `fifo1.num_elements() == fifo2.num_elements()`.

10.3.2 Function Bodies

A wide range of syntax can be used in the functions within a module class. This includes "if" statements, the Boolean operators `and`, `or`, and `not`, nonblocking assignment operator for state updates (`<=`), guard functions (`guard()`), assertions (`assert`), assumptions (`assumption()`), and standard Python expressions and function calls (this includes Z3W expressions and function calls as well). This means rules and methods in the EDSL can be written in a very similar way as the corresponding rules or methods in BSV, except for the absence of loops and the addition of guard functions, assertions, and assumptions.

The `guard()` function is a way to add an explicit guard to a rule or method. In BSV, rules and methods include explicit guards as part of the syntax before the body. Since the EDSL is built using Python's ASTs, mimicking what BSV does would require changes to Python's syntax. Instead the EDSL includes the `guard()` function to add explicit guards. To match an explicit guard in BSV, the rule or method in EDSL would include the explicit guard in a guard function in the first line of the rule or method's body.

The `assert` statement and `assume()` functions are used to add assertions and assumptions for verification purposes. When they appear in the body of an "if" statement, the assertion or assumption is only checked when the corresponding predicate is true.

As an example of the similarities between the bodies of BSV rules and methods and the bodies of the EDSL's rules and methods, consider state updates (register writes in BSV). Both adopt the same syntax and semantics for writing to a state. That means states are updated with the `<=` symbol, and the updates are not visible until the end of the corresponding rule or method.

10.3.3 Printf Actions

When debugging hardware, it can be useful to have print statements that can display what the hardware is doing and what the values mean in a human-friendly manner. For example, in a processor hardware design, it is common to have print statements from each pipeline stage that say whether each stage has a valid instruction in it, and if so, what is the instruction [86, 10, 5, 3]. These print statements can also display a human-readable disassembly of the

instruction bits to significantly simplify debugging.

ProtoRMC adds `printf` actions to the EDSL in order to support print-statement debugging during verification. A `printf` action takes in a format string and a number of expressions matching the placeholders in the format string. These `printf` statements can be printed for a specific execution by evaluating whether or not the `printf` statement was reached during evaluating the action of the corresponding rule or method, and if so, printing the message with evaluated values plugged into it.

In our model-checking-based verification, `printf` statements are especially useful when analyzing k-induction counterexamples. Since `printf` statements can be used to give a human-friendly summary of what rules and methods are doing, it is easier to detect inconsistencies between rules that reveal the counterexamples should not be reachable.

10.3.4 Parameterization

Parameterization is a useful feature for the design and verification of hardware modules. Parameterization is implemented in ProtoRMC by wrapping the parameterized module with a Python function that takes in parameters as arguments and returns the module definition. These returned modules behave like closures because they contain the bindings for the parameters provided by the wrapper function. An example of a FIFO parameterized by data type `t` and size `n` is shown below:

```
def mkFIFO(t, n):
    class FIFO(Module):
        data : Vector(n, t)
        enqP : UBitVec(log2(n))
        deqP : UBitVec(log2(n))
        full : Bool
        # rest of FIFO definition
    return FIFO
```

In this way of parameterization, modules cannot be used without fully specified parameters. Fortunately, if the right abstract parameter values are used, then verification of the module with abstract parameters can imply the correctness of the module for all possible parameters. For example, in the parameterized FIFO, if a newly declared abstract type is passed in as the `t` parameter, then verification of the FIFO implies the FIFO is correct

for all possible types `t`. Similarly, if an uninterpreted functions is passed in as a function parameter, then verification with the uninterpreted function implies the module is correct for all possible function parameters.

10.4 Symbolic Module Construction

Once the module's description has been written in the EDSL, it can be converted into a symbolic representation using the `SymbolicModule` class. The `SymbolicModule` class parses the Python code of the module and constructs the necessary Z3W expressions corresponding to all the methods, rules, reset functions, and predicates in the module. This class applies the symbolic semantics from Chapter 8 to construct the building blocks seen in Figure 9-2.

In addition to the symbolic output expressions shown in Figure 9-2 (*i.e.* next state, guard, assertion summary, and readiness summary), `SymbolicModule` also produces intermediate symbolic values for other flavor of assertions (*e.g.* assumptions), symbolic read and write sets per rule, and `printf` actions.

Instead of using an enumeration of the rules and action methods to select the executed rule or action method in the step, a one-hot bitvector of will-fire signals is used. This simplifies many of the symbolic expressions in the step, but it requires an additional constraint during model checking to ensure exactly one of the bits is set.

Additionally, using a bitvector instead of a custom enumeration allows the entire model-checking problem to be contained within the theory of bitvectors if the module itself is contained within the bitvector theory.

To keep the size of expressions down, the steps in the unrolled execution are not explicitly chained together. Instead, the input states for each step are a free variable, and additional constraints are added to force the input of each step to be equal to the previous step. This significantly improves the verification performance for larger modules unrolled to larger depths. As mentioned in Section 9.2.3, this has the added benefit that unrolled executions can be further unrolled backwards in time which is essential for incremental SMT solving with k-induction.

10.5 Model Checking

The `SymbolicModule` provides methods for various SMT-based model checking procedures. This includes bounded model checking to show assertions hold for all bounded sequences and k-induction-based unbounded model checking to show assertions hold for all reachable states. These model-checking formulations use Z3 to find solutions to the constraints presented in Chapter 9 for each form of model checking.

ProtoRMC includes a number of changes and enhancements made to the model-checking formulations that are described below.

10.5.1 Abstraction

A number of abstractions can be used to improve the performance of model checking. Some of these abstractions can be stated explicitly in the model, and some can be introduced by ProtoRMC. The biggest opportunity for abstraction comes from the use of uninterpreted functions.

In ProtoRMC, functions can be used in the bodies of methods and rules to simplify the code. These functions can either be defined external from the module (using the `function` decorator), or they can be undefined and treated as uninterpreted functions.

In ProtoRMC, functions marked with the `function` decorator are treated as candidates for abstraction through the use of uninterpreted functions, even though they have definitions. In many cases, especially when a function appears in both modules in a miter, the success of verification does not depend on the function definition. In these cases, the function can be expressed in the SMT solver as an uninterpreted function to speed up verification.

By default, ProtoRMC plays a trick in function encoding to try to take advantage of some of the benefits provided by uninterpreted functions without losing function definitions. In ProtoRMC, a function marked with the `function` decorator is encoded as an uninterpreted function with an added constraint that the output is equal to the interpreted function value. Even though this encoding is mathematically equal to the function definition, the use of an uninterpreted function causes Z3 to try to consider it as an uninterpreted function

during SMT solving. In many cases, this works nearly as well as treating the function as entirely uninterpreted with the added benefit that if the function needs to be interpreted for the verification to be successful, then the definition is available to the SMT solver.

In practice, this method of treating functions as interpreted and uninterpreted works well for smaller examples, but as examples get large, then a different approach is required. ProtoRMC has an automatic abstraction-refinement routine that can be used during model checking. This approach assumes all functions marked with the `function` decorator are uninterpreted, and each time a counterexample is found in bounded model checking, function interpretations are added in to see which function definitions are required to make the counterexample infeasible (if possible). This process is related to counterexample-guided abstraction refinement [37]. Unfortunately this process is very slow because it requires many uses of the SMT solver, so in many cases, the best approach for now is for the designer to manually pick the functions to treat as uninterpreted.

10.5.2 Execution Symmetry Reduction

A common strategy in reducing the complexity of SMT and SAT formulas is symmetry reduction, or removing symmetric states. The same techniques used for general SMT/SAT formulas can be applied here as well, but we can take advantage of higher-level structure to exploit symmetry, specifically symmetry in the step sequences considered. This is a form of partial-order reduction where transitions are removed instead of states [40].

In ProtoRMC's execution symmetry reduction, it looks for pairs of rules and methods which produce the same result no matter which order they are fired in. This notion is the same as for conflict-free rules and methods in BSV, but ProtoRMC uses a more aggressive detection methodology than the Bluespec compiler. Instead of considering all states that appear as members of the rule's read and write sets, ProtoRMC looks at state-dependent read and write sets and only introduces conflicts between rules if there exists a state such that the two rules are ready and will touch the same state. This state-dependent approach uses an SMT solver to detect the conflicts.

Once ProtoRMC finds rules and methods that can be fired in any order and produce the same result, a canonical ordering is chosen, and it is added as a constraint to any model-

checking formulations produced for the module. That means if a module has rules `a` and `b` which are conflict-free, ProtoRMC chooses a canonical ordering for these rules, say `a` before `b`, and adds a constraint that if `b` fires in step i, `a` cannot fire in step $i + 1$.

This can significantly simplify the search space for model-checking tasks, and as a result, it can speed up their execution.

10.5.3 Parallelization Opportunities

The model-checking routines provide multiple opportunities for parallelizing the verification across multiple processes.

First, the assertions to verify can be divided across processes. There are two flavors of each model-checking function in `SymbolicModule`: a full version and a partial version. The full version checks all assertions at once, and the partial version only checks one assertion at a time. The partial variant assumes that the other assertions are being checked in parallel, so the other assertions are assumed to hold for states in which the current assertion is assumed to hold. The partial formulation allows for easily dividing the model checking across multiple processes to perform the verification in parallel.

The model-checking functions also support dividing verification by execution sequences, so different processes consider different execution sequences. For bounded model checking, executions are divided by prefixes, and for k-induction, executions are divided by suffixes; this takes advantage of the incremental SMT-solving capabilities of Z3.

10.6 Inspection and Visualization

Symbolic representations of rule-based hardware designs provide new opportunities for visualization of rule-based hardware systems. These can be used to understand how a module works, or they can be used as a sanity check to make sure the module behaves as expected.

10.6.1 State-Machine Diagrams

The first visualization example is extracting a state-machine diagram from a module. State-machine diagrams are commonly used in hardware design to construct the logic necessary to get the desired behavior. Typically these diagrams only show a simplified description of the state space because a one-to-one mapping would produce too many states to be useful. For example, if the state-machine diagram is written for a 4-element 32-bit FIFO, and the different possible values in the FIFO are represented as different states in the state-machine diagram, then the diagram would have more than 2^{64} states.

Instead it is useful to present a simplified or abstract state machine that describes the behavior of a machine at a high level using either a simplified or abstracted view of the state. For example, a simplified state machine would only consider a subset of the state elements in the system, while an abstract state machine considers a projection of the states to abstract states. These reduced state machines are easier to look at but are harder to turn into real hardware.

As an example of simplified and abstracted state machines, consider the state machine for the control logic of a 4-element FIFO implemented as a 4-element circular buffer with enqueue and dequeue pointers. If a simplified state machine is made using just the enqueue and dequeue pointers, then there are 16 states in total; if we differentiate empty and full FIFOs for the states where the pointers are equal, then it goes up to 20 states. This large number of states can make it hard to see what is going on, and it can be hard to tell if there are any extraneous or missing transitions. An example representation of this state machine can be seen in Figure 10-2.

On the other hand, if an abstract state machine is made using the number of elements in the FIFO as its state, then this state machine is much easier to visualize since it only has 5 states with a simple topology of state transitions, but it no longer is easily translatable to hardware. An example representation of this state machine can be seen in Figure 10-3.

With ProtoRMC, we can automatically produce state-machine diagrams from rule-based designs where the state space is a function of the states. This allows us to produce easy-to-visualize state machines for different aspects of designs to see if they behave as

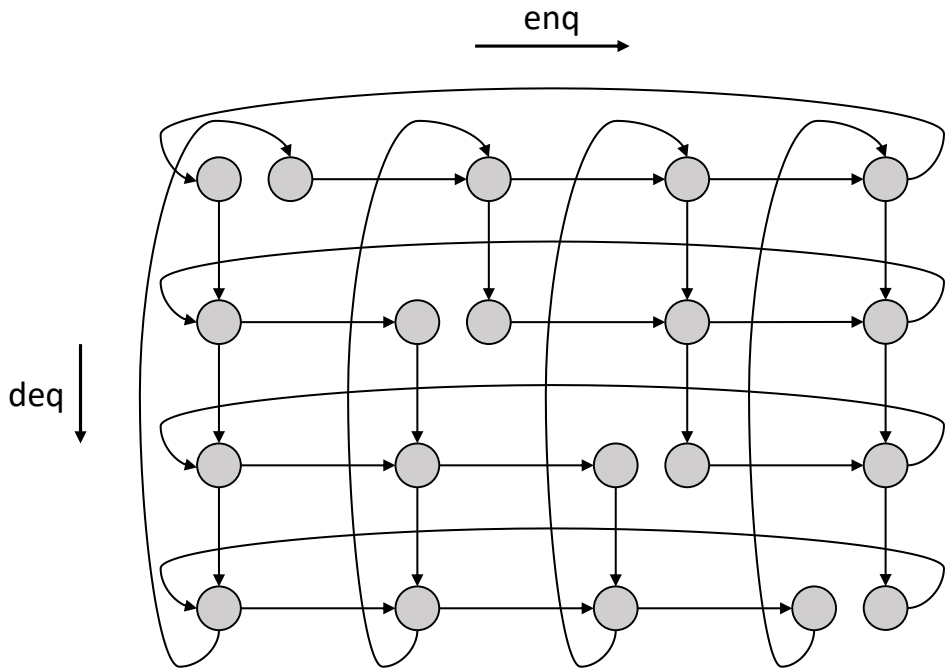

Figure 10-2: State machine for control logic of 4-element FIFO. States are determined by the enqueue and dequeue pointers (top-left pair is (0,0)). For the cases where the pointers are equal, there are two states: one that corresponds to a full FIFO (left) and one that corresponds to an empty FIFO (right).

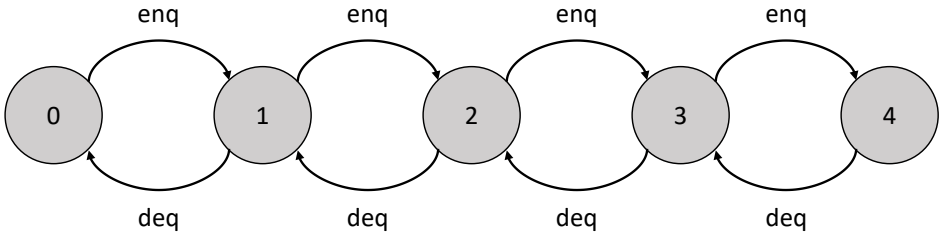

Figure 10-3: State machine for control logic of 4-element FIFO using number of elements in the FIFO as the state

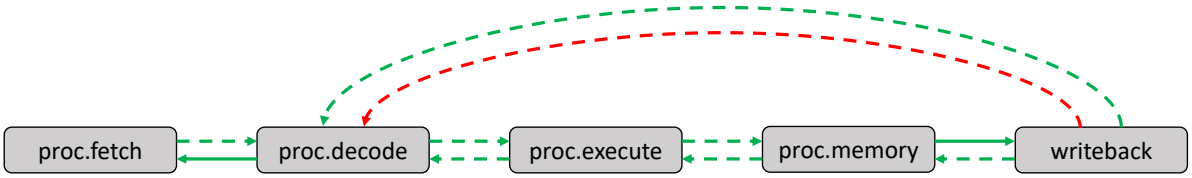

Figure 10-4: Rule-guard relations for a 5-stage pipelined processor

expected. Therefore when constructing the 4-element FIFO described earlier, we can use ProtoRMC to automatically produce the state machine seen in Figure 10-3 to manually verify there are no missing or extraneous state transitions.

10.6.2 Rule-Guard Relations

Another visualization technique provided by ProtoRMC for rule-based hardware designs is visualizing how rule or method firings allow or disallow other rules and methods to fire. For example, in our five-stage pipelined processor, the decode rule always enables the fetch rule, meaning if the fetch rule is not ready, firing the decode rule will always make it ready. Also in our processor, the writeback rule sometimes disables the decode rule from firing, so when the decode rule is ready and the writeback rule fires, sometimes the guard of the decode rule becomes false.

ProtoRMC can visualize this information in a directed graph where the nodes are the rules and methods, and the arrows correspond to the relations always enables (solid green), sometimes enables (dashed green), sometimes disables (dashed red), and always disables (solid red). An example graph for the five-stage pipelined processor can be seen in Figure 10-4.

10.7 Modular Verification

In order to perform modular verification, we need to be able to check if a module implements a specification; this is most easily done with the construction of a miter module. ProtoRMC provides the necessary elements to easily construct miter modules, specifically the `READY` macro that allows for extended assertions that depend on the readiness of a method.

An example miter module implemented in ProtoRMC can be seen in Figure 10-5. This

```
def mkFuncUnitMiter(mkImpl, mkSpec):
    class FuncUnitMiter(Miter):
        impl : mkImpl
        spec : mkSpec

        @method
        def start(self, x : t):
            self.impl.enq(x)
            self.spec.enq(x)

        @method
        def getResult(self) -> t:
            impl_val = self.impl.getResult()
            spec_val = self.spec.getResult()
            assert impl_val == spec_val
            return impl_val

        @predicate
        def guards_match(self, x : t):
            assert Implies(READY(self.impl.start(x)),
                           READY(self.spec.start(x)))
            assert Implies(READY(self.impl.getResult()),
                           READY(self.spec.getResult()))
    return FuncUnitMiter
```

Figure 10-5: Example miter module construction for functional units in ProtoRMC

is a parameterized miter construction that checks if a module that implements the `FuncUnit` interface from Chapter 3 matches a corresponding specification. This is a parameterized version of the miter in Figure 7-1 written in ProtoRMC.

In order to complete the verification, the `guards_match` method implements the state assertions to make sure that each time a method in the implementation is ready to call, the corresponding method in the specification is ready to call. This requires looking at the guards of methods, so we use the `READY` macro to get if the corresponding method is ready. The `READY` macro takes in the syntax for a method call and returns if the guard for that method would be ready or not. The method call inside the macro is not called, and therefore there are no corresponding state changes associated with those macro invocations.

Now if model checking is performed with this miter and the assertions always hold, then

the provided implementation implements the specification.

10.8 Examples

To show some of the capabilities of ProtoRMC, we are going to look at verifying example modules that use the `FuncUnit` interface from Chapter 3. Despite the similarities between the examples, they are still able to show off a wide range of features of ProtoRMC and the verification procedure.

10.8.1 Functional Unit Specifications

To start, we look at specifications for functional units that compute pure functions. This includes implementing `mkFuncUnitSpec` from Figure 3-5 and `mkFuncUnitConcurrentSpec`. These specifications are shown in Figure 10-6, and the `mkFIFOSpec` module used in `mkFuncUnitConcurrentSpec` is shown in Figure 10-7.

Both specifications compute the result in the start method, but `mkFuncUnitSpec` stores the result in a state that can only hold one result, while `mkFuncUnitConcurrentSpec` stores the result in a `mkFIFOSpec` (an infinite-sized specification of a FIFO). The implementation of `mkFIFOSpec` can be seen in Figure 10-7. Note that the `Int` type used in `mkFIFOSpec` is an unbounded integer, so there is no limit to the number of elements that can be in the FIFO specification. This makes the FIFO specification able to specify FIFOs of all sizes, but it also makes it not synthesizable to hardware.

10.8.2 `mkThreeF`

The first example is verification of the `mkThreeF` module from Figure 3-2. The ProtoRMC implementation of this module is shown in Figure 10-8. This module takes in a function parameter `f` and implements `f(f(f(x)))` in a folded manner.

This module can be verified against the specification `mkThreeFSpec` using the functional unit miter and the script in Figure 10-9. This verification script initializes the abstract type and uninterpreted function used in `mkThreeF`, constructs a symbolic module for the miter,

```
def mkFuncUnitSpec(f):
    argT = f.arg_sorts[0]
    resultT = f.arg_sorts[1]
    class FuncUnitSpec(Module):
        result_data : resultT
        result_valid : Bool

        @reset
        def reset(self):
            assert not self.result_valid
        @method
        def start(self, x : argT):
            guard(not self.result_valid)
            self.result_data <= f(x)
            self.result_valid <= True
        @method
        def getResult(self) -> resultT:
            guard(self.result_valid)
            self.result_valid <= False
            return self.result_data
        @verif
        def active(self) -> Bool:
            return self.result_valid
    return FuncUnitSpec

def mkFuncUnitConcurrentSpec(f):
    argT = f.arg_sorts[0]
    respT = f.arg_sorts[1]
    class FuncUnitConcurrentSpec(Module):
        result_fifo : mkFIFOSpec(resultT)

        @method
        def start(x : argT):
            self.result_fifo.enq(f(x))
        @method
        def getResult() -> resultT:
            val = self.result_fifo.deq()
            return val
        @verif
        def numActive() -> Int:
            return result_fifo.numElements()
    return FuncUnitConcurrentSpec
```

Figure 10-6: Functional unit specifications in ProtoRMC

```
def mkFIFOSpec(t):
    class FIFOSpec(Module):
        enqP : Int
        deqP : Int
        data : Array(Int, t)

        @reset
        def reset(self):
            assert self.enqP == 0 and self.deqP == 0
        @method
        def enq(self, value : t):
            self.data <= Store(self.data, self.enqP, value)
            self.enqP <= self.enqP + 1
        @method
        def deq(self) -> t:
            guard(self.deqP < self.enqP)
            self.deqP <= self.deqP + 1
            return self.data[self.deqP]
        @predicate
        def reachableInvariant(self):
            assert self.enqP >= self.deqP
            assert self.enqP >= 0 and self.deqP >= 0
        @verif
        def numElements(self) -> Int:
            return self.enqP - self.deqP
```

Figure 10-7: Infinite-sized FIFO specification in ProtoRMC

```
def mkThreeF(f)
    t = f.arg_sorts[0]
    # f.arg_sorts[1] should also be t
    class ThreeF(Module):
        x : t
        count : z3w.UBitVec(3)

        @reset
        def reset(self):
            assert self.count == 0
        @rule
        def step(self):
            guard(self.count > 0 and self.count < 4)
            self.x <= f(self.x)
            self.count <= self.count + 1
        @method
        def start(self, v : t):
            guard(self.count == 0)
            self.x <= v
            self.count <= 1
        @method
        def getResult(self) -> t:
            guard(self.count == 4)
            self.count <= 0
            return self.x
        @verif
        def active(self) -> Bool:
            return self.count != 0
    return ThreeF
```

Figure 10-8: Implementation of `mkThreeF` in ProtoRMC

```
# Create an abstract type t
t = DeclareSort('t')
# Create an uninterpreted function of type t -> t
f = Function('f', t, t)
def g(x):
    return f(f(f(x)))
miter = SymbolicModule(mkFuncUnitMiter(mkThreeF(f),
                                       mkFuncUnitSpec(g)))
miter.proof(10)
```

Figure 10-9: Verification script for `mkThreeF` in ProtoRMC

and then runs a k-induction-based unbounded model-checking proof routine with maximum depth. When this proof script is run, the proof routine `miter.proof(10)` prints that the unbounded model checking was successful, and the assertions hold for all reachable steps of the miter. Therefore `mkThreeF(f)` implements `mkFuncUnitSpec(g)`.

10.8.3 mkPipeline

The next example of a functional unit using ProtoRMC is a simple arithmetic pipeline module called `mkPipeline`. In this example, we compute the value of the polynomial $f(x) = x^8 + 4x^6 + 10x^4 + 12x^2 + 12$ in a three-stage pipeline where each stage squares a value and adds a constant. The functions performed in each pipeline stage are $f_1(x) = x^2 + 1$, $f_2(x) = x^2 + 2$, and $f_3(x) = x^2 + 3$. This module is intended to implement the specification `mkFuncUnitConcurrentSpec(f)`. The implementation of this module can be seen below.

```
@function
def f1(x : t) -> t:
    return (x * x) + 1

@function
def f2(x : t) -> t:
    return (x * x) + 2

@function
def f3(x : t) -> t:
    return (x * x) + 3

class mkPipeline(Module):
    s1 : t
```

```
    s1_valid : z3w.Bool
    s2 : t
    s2_valid : z3w.Bool
    s3 : t
    s3_valid : z3w.Bool
    s4 : t
    s4_valid : z3w.Bool

    @reset
    def reset(self):
        assert not (self.s1_valid or self.s2_valid
                or self.s3_valid or self.s4_valid)
    @method
    def start(self, x : t):
        guard(not self.s1_valid)
        self.s1 <= x
        self.s1_valid <= True
    @rule
    def stage1(self):
        guard(self.s1_valid and not self.s2_valid)
        self.s1_valid <= False
        self.s2 <= f1(self.s1)
        self.s2_valid <= True
    @rule
    def stage2(self):
        guard(self.s2_valid and not self.s3_valid)
        self.s2_valid <= False
        self.s3 <= f2(self.s2)
        self.s3_valid <= True
    @rule
    def stage3(self):
        guard(self.s3_valid and not self.s4_valid)
        self.s3_valid <= False
        self.s4 <= f3(self.s3)
        self.s4_valid <= True
    @method
    def getResult(self) -> t:
        guard(self.s4_valid)
        self.s4_valid <= False
        return self.s4
```

Naively trying to prove the assertions in the miter `mkFuncUnitMiter(mkPipeline, mkFuncUnitConcurrentSpec(f))` using model checking is not computational feasible. The

complexity arises from the combination of multiplication being a very expensive operation in model checking and verifying a pipeline requiring a significant amount of unrolling for model checking. All that unrolling increases the number of multiplications found in the SMT constraints, resulting in the computational infeasibility. Therefore, in order to verify `mkPipeline` implements `mkFuncUnitConcurrentSpec(f)`, we need a new approach.

The key to simplifying the verification problem is recognizing that there are two independent things being verified: first, the pipeline control logic is implemented correctly, and second, that $f_3(f_2(f_1(x))) = f(x)$. In order to split these verification problems, we introduce a new intermediate specification: `mkFuncUnitConcurrentSpec(g)` where `g(x) = f3(f2(f1(x)))`. With this transformation, we can treat `f1`, `f2`, and `f3` as uninterpreted functions, completely removing the computational complexity introduced by multiplication.

After proving that `mkPipeline` implements `mkFuncUnitConcurrentSpec(g)`, we can then prove that `mkFuncUnitConcurrentSpec(g)` is equivalent to `mkFuncUnitConcurrentSpec(f)` by showing that `f` is equal to `g`. We can use an SMT solver to show that `f` is equal to `g` for all inputs in a fraction of a second. Therefore we have taken a computationally infeasible verification problem and made it feasible by introducing an intermediate specification that splits the verification problem into two much simpler, independent pieces.

The script to perform this verification is shown below:

```
def g(x):
    return f3(f2(f1(x)))
miter = SymbolicModule(
            mkFuncUnitMiter(mkPipeline,
                            mkFuncUnitConcurrentSpec(g)))
miter.proof(20)
# show that f == g
s = Solver()
x = Const('x', t)
s.add( f(x) != g(x) )
result = s.check()
assert result == unsat
```

10.8.4 mkMultiFuncUnit

For the next example, consider the `mkMultiFuncUnit` module from Figure 3-11.

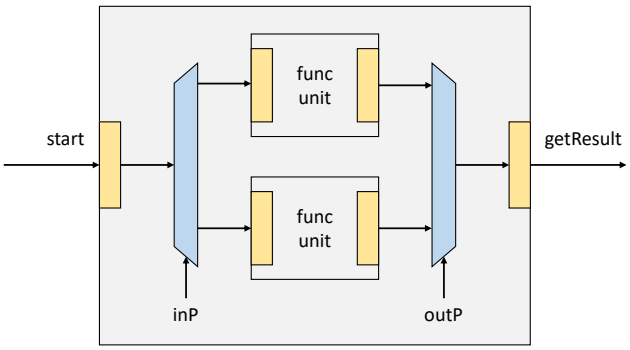

Figure 10-10: `mkMultiFuncUnit`

This module takes in a functional unit module definition as a parameter and produces a functional unit with double the throughput of the original module. This is done by instantiating two of the functional unit parameter as submodules and using the two submodules in an alternating manner. The alternation is managed by two one-bit state variables, `inP` and `outP`. Each time the `start` method is called on the top module, `inP` is used to select the submodule to call `start` on, and `inP` is incremented. Each time the `getResult` method is called on the top module, `outP` is used to select the submodule to call `getResult` on, `outP` is incremented, and the value returned from the submodule's `getResult` method is returned by the top module's `getResult` method. The overall structure of this module can be seen in Figure 10-10, and the implementation of this module can be seen in Figure 10-11.

The specification for this module depends on the module passed in as `funcUnit`. If `funcUnit` implements `mkFuncUnitSpec(f)` for some `f`, then `mkMultiFuncUnit(funcUnit)` should implement the specification `mkFuncUnitSpecConcurrent(f)`. This can be verified by showing the slightly stronger property that `mkMultiFuncUnit(mkFuncUnitSpec(f))` implements `mkFuncSpecConcurrent(f)`. The verification script for proving this is shown in Figure 10-12. Unfortunately, running this verification script as-is results in inconclusive results, because counterexamples are found for all k-induction steps.

When the k-induction step of `miter.proof()` finds a counterexample for each depth, the final counterexample trace is written to a VCD file, and any `printf` statements active during the counterexample are printed. Examining this information reveals that in the counterexample, the two modules are out-of-sync because they have different numbers of active computations in them. Therefore, an additional invariant must be added to the

```
def mkMultiFuncUnit(funcUnit):
    argT = funcUnit.start.arg_type[0]
    resultT = funcUnit.getResult.return_type
    class MultiFuncUnit(Module):
        units : Vector(2, funcUnit)
        in_unit : UBitVec(1)
        out_unit : UBitVec(1)

        @reset
        def reset(self):
            assert in_unit == out_unit

        @method
        def start(self, x:  argT):
            self.units[self.in_unit].start(x)
            self.in_unit <= self.in_unit + 1

        @method
        def getResult(self) -> resultT:
            result = self.units[self.out_unit].getResult()
            self.out_unit <= self.out_unit + 1
            return result

        @verif
        def numActive(self) -> Int:
            if self.units[0].active() and
                        self.units[1].active():
                return 2
            elif self.units[0].active() or
                        self.units[1].active():
                return 1
            else:
                return 0
    return MultiFuncUnit
```

Figure 10-11: Implementation of `mkMultiFuncUnit` in ProtoRMC

```
argT = DeclareSort('argT')
retT = DeclareSort('retT')
f = Function('f', argT, retT)
miter = FuncUnitMiter(mkMultiFuncUnit(
            mkFuncUnitSpec(f)), mkFuncUnitConcurrentSpec(f))
miter.proof(10)
```

Figure 10-12: Verification script for `mkMultiFuncUnit` in ProtoRMC

design that says the numbers of active computations in the two modules are equal. To do this, an assertion is added to the miter that asserts that the `numActive verif` functions of `mkmkMultiFuncUnit` and `mkFuncUnitConcurrentSpec` always produce the same result. After adding this invariant, the proof is successful. Note that in practice, `verif` functions like `numActive` are only added when they are deemed to be necessary, so getting a k-induction step to be successful may involve adding `verif` functions for extended assertions.

10.8.5 `mkGCD`

As a more advanced example, consider a module that computes the GCD of two 32-bit numbers using the Euclidean algorithm. The implementation for this module can be seen in Figure 10-13:

The module `mkGCD` should implement the specification `mkFuncUnitSpec(gcd)` where `gcd` is a function that computes the GCD of two numbers. Unlike other example functional units, it is not feasible to write the full definition of the function being implemented (`gcd`) in a way that can be used by the SMT backend. This makes writing the specification significantly harder.

To overcome this, we define an uninterpreted function for the GCD function `gcd_uf` and add axioms as constraints to the model checking that define `gcd_uf` axiomatically. These axioms are the same axioms used in the Euclidean algorithm to compute the GCD:

```
# these axioms hold for all x and y
def gcd_axioms(x : t, y : t) -> Bool:
    return And(
            gcd_uf(x, 0) == x, gcd_uf(0, y) == y,
            Implies(x >= y, gcd_uf(x, y) == gcd_uf(x-y, y)),
            Implies(x <= y, gcd_uf(x, y) == gcd_uf(x, y-x)),
```

```
class mkGCD(Module):
    x : UBitVec(32)
    y : UBitVec(32)
    started : z3w.Bool

    @reset
    def reset_predicate(self):
        assert self.started == False
    @rule
    def swap(self):
        guard(self.started and (self.x > self.y or
                self.x == 0) and self.y != 0)
        self.x <= self.y
        self.y <= self.x
    @rule
    def subtract(self):
        guard(self.started and self.x <= self.y
                and self.x != 0 and self.y != 0)
        self.y <= self.y - self.x
    @method
    def start(self, in : (t, t)):
        guard(not self.started)
        self.x <= in[0]
        self.y <= in[1]
        self.started <= True
    @method
    def getResult(self) -> t:
        guard(self.started and self.y == 0)
        self.started <= False
        return self.x
```

Figure 10-13: Implementation of `mkGCD` in ProtoRMC

```
            gcd_uf(x, y) == gcd_uf(y, x)
        )
```

At this point, the statement `ForAll([x,y], gcd_axioms(x,y))` could be added as a constraint to the SMT solver, but adding universal quantifiers to SMT solvers complicates the solving process. Instead it is sufficient to have instances of the axioms only for the arguments observed in the model checking. Therefore the axioms are only needed for the state variables `x` and `y` of `mkGCD`. This is done by adding a `predicate` function to `mkGCD` that assumes the axioms.

```
    # added to mkGCD
    @predicate
    def axioms(self):
        assume(gcd_axioms(self.x, self.y))
```

Since GCD operations can take many cycles, it is impossible to unroll the `mkGCD` module enough to cover the duration of all possible GCD computations. Because of this, we need a stronger invariant to make k-induction work. Therefore we add a `verif` method that returns the expected result of `mkGCD` when it is in the middle of computation. This is done by taking the gcd of `x` and `y`.

```
    # added to mkGCD
    @verif
    def expectedResult(self) -> UBitVec(32):
        return gcd_uf(self.x, self.y)
```

If an assertion is added to the miter that says `expectedResult` of `mkGCD` matches the expected result of `mkFuncUnitSpec(gcd_uf)`, then the unbounded model checking will be successful. Therefore this proves that `mkGCD` implements `mkFuncUnitSpec(gcd)` given that `gcd_uf` is equivalent to `gcd` given the axioms specified by `gcd_axioms`.

10.8.6 mkMultiGCD

Now as a final example, consider the verification of the module `mkMultiGCD`. The module `mkMultiGCD` uses two `GCD` modules in an alternating manner to produce a higher-throughput GCD module. Since we already have a module that does this transformation for us, we define `mkMultiGCD` as `mkMultiFuncUnit(mkGCD)`.

We want to prove that `mkMultiFuncUnit(mkGCD)` implements `ConcurrentFuncUnit(gcd)`. We can actually verify this without doing any model checking and just reusing previous results along with the modular verification theorem.

From the verification of `mkMultiFuncUnit` we have

$$\forall \texttt{f},\ \texttt{mkMultiFuncUnit}(\texttt{mkFuncUnitSpec}(\texttt{f})) \sqsubseteq \texttt{mkFuncUnitConcurrentSpec}(\texttt{f}).$$

From the verification of `mkGCD` we have that `mkGCD` $\sqsubseteq$ `mkFuncUnitSpec(gcd)`. Using the Modular Verification Theorem (Theorem 6.2), we get

$$\texttt{mkMultiFuncUnit}(\texttt{mkGCD}) \sqsubseteq \texttt{mkMultiFuncUnit}(\texttt{mkFuncUnitSpec}(\texttt{gcd})).$$

Finally, using the transitive property of $\sqsubseteq$ we get

$$\texttt{mkMultiGCD} = \texttt{mkMultiFuncUnit}(\texttt{mkGCD}) \sqsubseteq \texttt{mkFuncUnitConcurrentSpec}(\texttt{gcd})$$

thus `mkMultiGCD` implements its specification `mkFuncUnitConcurrentSpec(gcd)`.

10.9 Evaluation

To evaluate the performance of ProtoRMC and the effectiveness of the abstractions and optimizations provided as part of it, we verify some of the previously shown examples with different configurations.

10.9.1 Evaluating Verification of `mkThreeF`

First consider the verification of `mkThreeF` from the previous example. We verify this module using an uninterpreted function as the function parameter, and we compare the results to verifying `mkThreeF` using two different interpreted function parameters: $f(x) = x + 1$ and $f(x) = x^2$. Since the interpreted functions require a concrete type, we use unsigned bitvectors of various widths. The results of this verification are shown in Figure 10-14. The solid lines represent the verification of `mkThreeF` with the specified interpreted function parameter,

and the dashed lines represent the verification of `mkThreeF` with an uninterpreted function parameter.

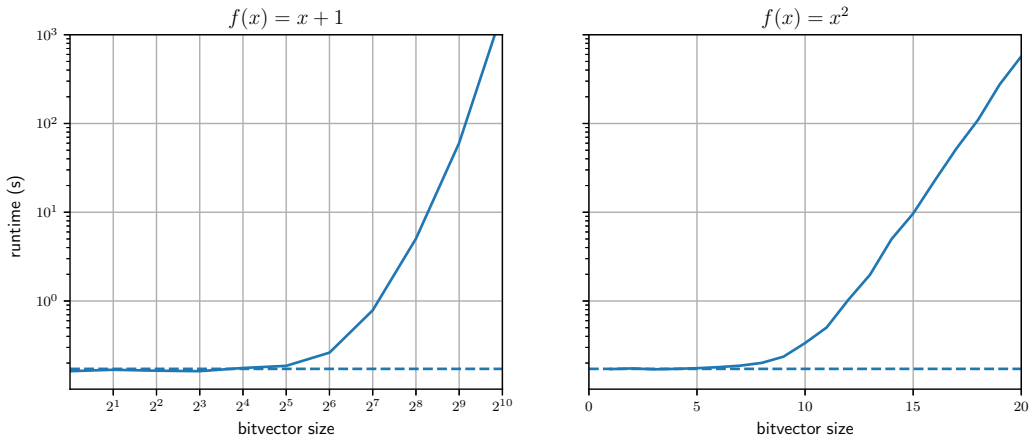

Figure 10-14: Runtime for verifying different configurations of `mkThreeF`. The dashed line in each figure represents verifying the module using an uninterpreted function.

These results show that verification using an uninterpreted function as the function parameter takes the same amount of time as verification using either of the interpreted functions with a small bitwidth. This means there are no overheads introduced by using uninterpreted functions in this module's verification. Another thing to notice is that there are points for both interpreted functions where increasing the bitwidth causes the runtime to increase significantly. This point is different for the two functions because of their relative complexity. Addition is a simpler function than multiplication, so verification using $f(x) = x + 1$ can be done efficiently for larger bitwidths than verification using $f(x) = x^2$.

10.9.2 Evaluating Verification of `mkPipeline`

Next consider the verification of `mkPipeline`. Previously we showed how `mkPipeline` can be easily verified against an intermediate specification (`mkFuncUnitConcurrentSpec(g)` where `g(x) = f3(f2(f1(x)))`) and then the intermediate specification can be verified against full specification (`mkFuncUnitConcurrentSpec(f)` where $f(x) = x^8 + 4x^6 + 10x^4 + 12x^2 + 12$). Now we look at the performance of this verification for various bitwidths of the values used in the pipeline. We compare the runtime of verifying against an intermediate specification to the runtime of verifying against the full specification directly. We also compare each to the

runtime of verifying a fully-abstract representation of the pipeline against the intermediate specification. These verifications are done with and without the execution-symmetry reduction optimization to see the effects of this optimization too. The results of this evaluation are in Figure 10-15. The dashed lines in the figure correspond to verification runs that use execution-symmetry reduction.

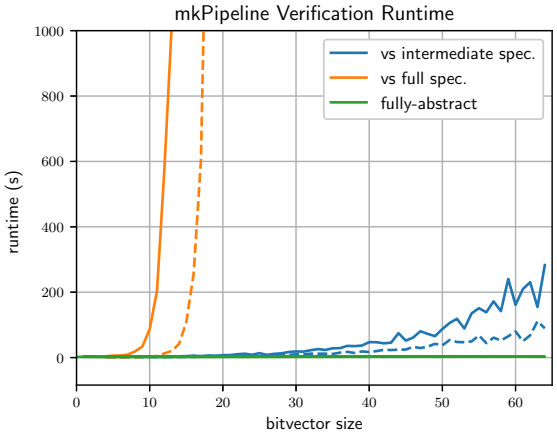

Figure 10-15: Runtime for verifying `mkPipeline`

These results show that verifying directly against the full specification is not feasible once the bitwidth passes a certain limit. This limit depends on whether or not execution-symmetry reduction is used; using this optimization allows for larger bitwidths to be verified. If the verification is limited to a 10 minute runtime, adding the optimization allows the verification to go from checking the pipeline for 12-bit bitvectors to checking it for 17-bit bitvectors in the same time.

These results also show that ProtoRMC's default treatment for functions (combining an uninterpreted function with the interpretation) works well for the pipelined example, but starts paying a noticeable penalty over the fully-abstract representation of the pipeline once bitwidths get above about 30. Once the module to verify gets above a certain level of complexity, it becomes beneficial to treat functions as fully uninterpreted if possible instead of relying on ProtoRMC's default treatment for functions.

10.9.3 Evaluating Verification of `mkMultiFuncUnit`

Now consider the verification of `mkMultiFuncUnit`. Since `mkMultiFuncUnit` takes in a functional unit as a parameter, when we verify it, we pass in a specification for a functional unit as the parameter. This evaluation explores the importance of modular verification by verifying `mkMultiFuncUnit` with different modules passed in as the functional unit.

First consider verifying `mkMultiFuncUnit` with a functional unit specification that must fire an extraneous internal rule for a number of steps between calls to `start` and `getResult`. The results of this verification is shown in Figure 10-16. Notice that without execution-symmetry reduction, the verification sees a massive spike in runtime when the specification must fire its internal rule 8 times for each computation. With execution-symmetry reduction, `mkMultiFuncUnit` can be verified for a much larger number of required internal rule firings. Execution-symmetry reduction works very well in this case to mitigate the overhead of the extra rule firings, but it is not a substitute for modular verification as shown in the next example.

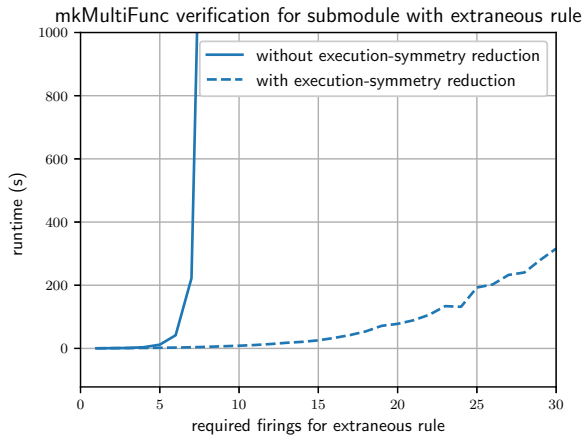

Figure 10-16: Runtime for verifying `mkMultiFunc` with a submodule that requires an internal rule to fire a number of times for each computation

Now consider verifying `mkMultiFuncUnit(mkThreeF(f))` for the same function parameters as used in Figure 10-14. We compare the runtime of this verification to the runtime of verifying just `mkThreeF(f)` for $f(x) = x + 1$ and $f(x) = x^2$. The results of this verification are shown in Figure 10-17. The dashed lines in the figure correspond to verification runs

that use execution-symmetry reduction.

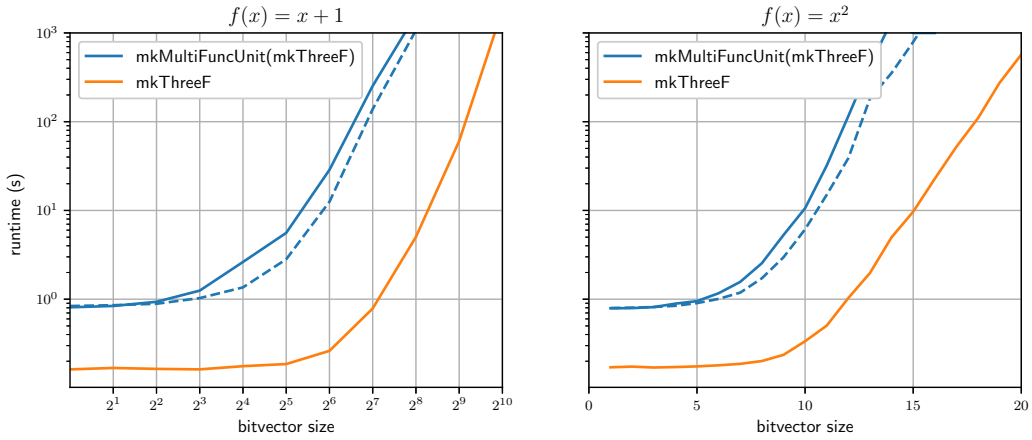

Figure 10-17: Comparing runtimes for verifying `mkMultiFunc(mkThreeF(f))` and `mkThreeF(f)`

From this figure we see that verifying `mkMultiFunc(mkThreeF(f))` for simple functions, *i.e.* small bitwidths, takes just under an order of magnitude longer than verifying `mkThreeF(f)` alone, even with execution-symmetry reduction. As the bitwidth increases, the gap in runtime between verifying `mkMultiFunc(mkThreeF(f))` and verifying `mkThreeF(f)` grows to multiple orders of magnitude. This emphasizes the importance of modular verification since the verification of `mkMultiFuncUnit` with a functional unit specification takes just 0.3 seconds. Therefore using modular verification makes the verification of `mkMultiFunc(mkThreeF(f))` only take 0.3 seconds longer than the verification of `mkThreeF(f)` instead of orders of magnitude longer.

10.10 Conclusion

In this chapter we presented ProtoRMC, a prototype implementation of Spec 'n' Check embedded in Python along with various routines for model checking and visualization. After presenting the features of ProtoRMC, we showed a few examples of how it can be used to express and verify modules. Finally we presented an evaluation for some of the examples that shows how complexity, abstraction, and optimizations affect verification runtime.

Chapter 11

Conclusion

In this thesis we showed how SMT-based verification of a rule-based HDL can efficiently verify designs in a modular manner by leveraging multiple opportunities for refinement and abstraction that combat the state-space-explosion problem.

We showed that our rule-based HDL, Spec 'n' Check, provides improvements over other HDLs that make it easier to specify and verify modules, specifically thanks to unambiguous one-method-at-a-time semantics. We also showed how the modularity of Spec 'n' Check can be leveraged in modular verification including the definition of the implements relation ($\sqsubseteq$) and the modular refinement theorem.

We presented symbolic semantics for Spec 'n' Check that are consistent with the dynamic semantics. The symbolic semantics are symbolic representations for Spec 'n' Check modules that can be used in SMT solvers. We also showed how the symbolic semantics can be used with an SMT solver to perform both bounded model checking and unbounded model checking, including multiple varieties of each.

Finally we presented ProtoRMC, our prototype implementation of Spec 'n' Check that contains tools for SMT-based verification and visualization. A wide range of examples of ProtoRMC were presented to show its use.

11.1 Future Work

This thesis presented a good base for performing SMT-based verification in a rule-based HDL; this initial work opens the door to many more opportunities.

11.1.1 Direct Integration into BSV

The most attractive piece of future work is integrating SMT-based model checking directly with the Bluespec Compiler. Now that the Bluespec compiler is open-source with a license that allows modification [2], it is possible to add a hook to the compiler to emit the SMT equations necessary for model checking. In order to take advantage of the modular verification presented in this thesis, it would be necessary to either modify the semantics of BSV or restrict the input BSV to a subset where the semantics match the semantics of Spec 'n' Check.

11.1.2 Verification of Rule-Based Designs against RTL

Verified rule-level hardware designs are not useful unless the RTL produced for the design actually matches the semantics of the rule-based description. If we have a verified, or at least trusted, compiler, then we can get verified (or trusted) RTL from the rule-level design automatically, but in some cases it is desirable to produce manual RTL corresponding to rule-level designs. This can either be from wanting RTL-level optimizations that are not available from a rule-level compiler or importing existing RTL as a rule-based module. In these cases, we need a way to formally verify the RTL matches the rule-level description.

11.1.3 Automatic Invariant Inference

One of the biggest benefits of model checking over proof-based methods is the automation, but unfortunately unbounded model checking is not an entirely automated process in most cases because in order to get the induction step to be successful, it is necessary to add invariants. Some techniques exist to add invariants in certain cases [78, 18] so it would be

nice to incorporate those and research further ways to make the model checking done in this thesis entirely automatic.

Bibliography

[1] BlueCheck. https://github.com/CTSRD-CHERI/bluecheck.

[2] Bluespec compiler. https://github.com/B-Lang-org/bsc.

[3] Flute. https://github.com/bluespec/Flute.

[4] Ivy. https://github.com/kenmcmil/ivy.

[5] Piccolo. https://github.com/bluespec/Piccolo.

[6] RISC-V instruction set manual. https://github.com/riscv/riscv-isa-manual.

[7] riscv-formal. https://github.com/SymbioticEDA/riscv-formal.

[8] RISCV Sail model. https://github.com/rems-project/sail-riscv.

[9] riscv-semantics. https://github.com/mit-plv/riscv-semantics.

[10] Rocket chip generator. https://github.com/chipsalliance/rocket-chip. Accessed: 2021-05-04.

[11] Zaher S. Andraus and Karem A. Sakallah. Automatic abstraction and verification of Verilog models. In *Proceedings of the 41st Annual Design Automation Conference*, DAC '04, page 218–223, New York, NY, USA, 2004. Association for Computing Machinery.

[12] Alasdair Armstrong, Thomas Bauereiss, Brian Campbell, Alastair Reid, Kathryn E Gray, Robert Norton-Wright, Prashanth Mundkur, Mark Wassell, Jon French, Christopher Pulte, et al. ISA semantics for ARMv8-a, RISC-v, and CHERI-MIPS. 2019.

[13] Arvind and Xiaowei Shen. Using term rewriting systems to design and verify processors. *IEEE Micro*, 19(3):36–46, 1999.

[14] H. S. Baird and Y. E. Cho. An artwork design verification system. In *Proceedings of the 12th Design Automation Conference*, DAC '75, page 414–420. IEEE Press, 1975.

[15] U. Banerjee, C. Juvekar, A. Wright, Arvind, and A. P. Chandrakasan. An energy-efficient reconfigurable DTLS cryptographic engine for end-to-end security in IoT applications. In *2018 IEEE International Solid-State Circuits Conference - (ISSCC)*, pages 42–44, Feb 2018.

[16] Utsav Banerjee, Andrew Wright, Chiraag Juvekar, Madeleine Waller, Anantha P Chandrakasan, et al. An energy-efficient reconfigurable DTLS cryptographic engine for securing Internet-of-Things applications. *IEEE Journal of Solid-State Circuits*, 54(8):2339–2352, 2019.

[17] Clark Barrett, Aaron Stump, Cesare Tinelli, et al. The SMT-LIB standard: Version 2.0. In *Proceedings of the 8th international workshop on satisfiability modulo theories (Edinburgh, England)*, volume 13, page 14, 2010.

[18] Saddek Bensalem and Yassine Lakhnech. Automatic generation of invariants. *Formal Methods in System Design*, 15(1):75–92, 1999.

[19] 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.

[20] Sergey Berezin, Sérgio Campos, and Edmund M Clarke. Compositional reasoning in model checking. In *International Symposium on Compositionality*, pages 81–102. Springer, 1997.

[21] Armin Biere, Alessandro Cimatti, Edmund Clarke, and Yunshan Zhu. Symbolic model checking without BDDs. In *International conference on tools and algorithms for the construction and analysis of systems*, pages 193–207. Springer, 1999.

[22] Armin Biere, Marijn Heule, and Hans van Maaren. *Handbook of satisfiability*, volume 185. IOS press, 2009.

[23] Johannes Birgmeier, Aaron R Bradley, and Georg Weissenbacher. Counterexample to induction-guided abstraction-refinement (CTIGAR). In *International Conference on Computer Aided Verification*, pages 831–848. Springer, 2014.

[24] Andrew D Booth. A signed binary multiplication technique. *The Quarterly Journal of Mechanics and Applied Mathematics*, 4(2):236–240, 1951.

[25] Thomas Bourgeat, Ian Clester, Andres Erbsen, Samuel Gruetter, Andrew Wright, and Adam Chlipala. A multipurpose formal RISC-V specification, 2021.

[26] Thomas Bourgeat, Ilia A. Lebedev, Andrew Wright, Sizhuo Zhang, Arvind, and Srinivas Devadas. MI6: secure enclaves in a speculative out-of-order processor. *CoRR*, abs/1812.09822, 2018.

[27] 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.

[28] Aaron R. Bradley. SAT-based model checking without unrolling. In Ranjit Jhala and David Schmidt, editors, *Verification, Model Checking, and Abstract Interpretation*, pages 70–87, Berlin, Heidelberg, 2011. Springer Berlin Heidelberg.

[29] 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.

[30] 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.

[31] Randal E Bryant. Formal verification of pipelined Y86-64 microprocessors with UCLID5. Technical report, Technical Report CMU-CS-18-122, 2018.

[32] 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.

[33] 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.

[34] Alessandro Cimatti, Alberto Griggio, Sergio Mover, and Stefano Tonetta. Infinite-state invariant checking with IC3 and predicate abstraction. *Formal Methods in System Design*, 49(3):190–218, 2016.

[35] Koen Claessen and John Hughes. QuickCheck: A lightweight tool for random testing of Haskell programs. In *Proceedings of the Fifth ACM SIGPLAN International Conference on Functional Programming*, ICFP '00, page 268–279, New York, NY, USA, 2000. Association for Computing Machinery.

[36] Edmund Clarke, Armin Biere, Richard Raimi, and Yunshan Zhu. Bounded model checking using satisfiability solving. *Formal methods in system design*, 19(1):7–34, 2001.

[37] Edmund Clarke, Orna Grumberg, Somesh Jha, Yuan Lu, and Helmut Veith. Counterexample-guided abstraction refinement for symbolic model checking. *J. ACM*, 50(5):752–794, September 2003.

[38] Edmund M. Clarke, Orna Grumberg, Hiromi Hiraishi, Somesh Jha, David E. Long, Kenneth L. McMillan, and Linda A. Ness. Verification of the Futurebus+ cache coherency protocol. *Formal Methods in System Design*, 6:217–232, 1995.

[39] Edmund M. Clarke, T. A. Henzinger, Helmut Veith, and Roderick P. Bloem. *Handbook of Model Checking*. Springer, 2018.

[40] Edmund M Clarke Jr, Orna Grumberg, Daniel Kroening, Doron Peled, and Helmut Veith. *Model checking*. MIT press, 2018.

[41] N. Dave, M. Katelman, M. King, Arvind, and J. Meseguer. Verification of microarchitectural refinements in rule-based systems. In *Ninth ACM/IEEE International Conference on Formal Methods and Models for Codesign (MEMPCODE2011)*, pages 61–71, July 2011.

[42] Nirav Dave, Arvind, and Michael Pellauer. Scheduling as rule composition. In *Proceedings of the 5th IEEE/ACM International Conference on Formal Methods and Models for Codesign*, MEMOCODE '07, page 51–60, USA, 2007. IEEE Computer Society.

[43] Leonardo De Moura and Nikolaj Bjørner. Z3: An efficient SMT solver. In *International conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 337–340. Springer, 2008.

[44] Leonardo De Moura, Harald Rueß, and Maria Sorea. Bounded model checking and induction: From refutation to verification. In *International Conference on Computer Aided Verification*, pages 14–26. Springer, 2003.

[45] Alan Edelman. The mathematics of the Pentium division bug. *SIAM review*, 39(1):54–67, 1997.

[46] N. Een, A. Mishchenko, and R. Brayton. Efficient implementation of property directed reachability. In *2011 Formal Methods in Computer-Aided Design (FMCAD)*, pages 125–134, Oct 2011.

[47] Aman Goel and Karem Sakallah. Model checking of Verilog RTL using IC3 with syntax-guided abstraction. In *NASA Formal Methods Symposium*, pages 166–185. Springer, 2019.

[48] Evguenii I Goldberg, Mukul R Prasad, and Robert K Brayton. Using SAT for combinational equivalence checking. In *Proceedings Design, Automation and Test in Europe. Conference and Exhibition 2001*, pages 114–121. IEEE, 2001.

[49] Roberto Gorrieri and Cristian Versari. *Introduction to concurrency theory: transition systems and CCS*. Springer, 2015.

[50] Dick Hamlet. When only random testing will do. In *Proceedings of the 1st international workshop on Random testing*, pages 1–9, 2006.

[51] Chris Hawblitzel, Jon Howell, Manos Kapritsos, Jacob R Lorch, Bryan Parno, Michael L Roberts, Srinath Setty, and Brian Zill. IronFleet: proving practical distributed systems correct. In *Proceedings of the 25th Symposium on Operating Systems Principles*, pages 1–17, 2015.

[52] Gage Hills, Christian Lau, Andrew Wright, Samuel Fuller, Mindy Bishop, Tathagata Srimani, Pritpal Kanhaiya, Rebecca Ho, Aya Amer, Yosi Stein, Denis Murphy, Arvind, Anantha Chandrakasan, and Max Shulaker. A modern microprocessor built from complementary carbon nanotube transistors. *Nature*, 572(7771):595–602, 2019.

[53] Y. Ho, A. Mishchenko, and R. Brayton. Property directed reachability with word-level abstraction. In *2017 Formal Methods in Computer Aided Design (FMCAD)*, pages 132–139, Oct 2017.

[54] Ramin Hojati and Robert K. Brayton. Automatic datapath abstraction in hardware systems. In Pierre Wolper, editor, *Computer Aided Verification*, pages 98–113, Berlin, Heidelberg, 1995. Springer Berlin Heidelberg.

[55] 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.

[56] Daniel Jackson. Alloy: a lightweight object modelling notation. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 11(2):256–290, 2002.

[57] Alfred Koelbl, Reily Jacoby, Himanshu Jain, and Carl Pixley. Solver technology for system-level to RTL equivalence checking. In *2009 Design, Automation & Test in Europe Conference & Exhibition*, pages 196–201. IEEE, 2009.

[58] Hari Govind Vediramana Krishn an, Yakir Vizel, Vijay Ganesh, and Arie Gurfinkel. Interpolating strong induction. In *International Conference on Computer Aided Verification*, pages 367–385. Springer, 2019.

[59] Leslie Lamport. The temporal logic of actions. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 16(3):872–923, 1994.

[60] Suho Lee and Karem A Sakallah. Unbounded scalable verification based on approximate property-directed reachability and datapath abstraction. In *International Conference on Computer Aided Verification*, pages 849–865. Springer, 2014.

[61] Charles E Leiserson, Flavio M Rose, and James B Saxe. Optimizing synchronous circuitry by retiming (preliminary version). In *Third Caltech conference on very large scale integration*, pages 87–116. Springer, 1983.

[62] Panagiotis Manolios and Sudarshan K Srinivasan. A refinement-based compositional reasoning framework for pipelined machine verification. *IEEE transactions on very large scale integration (VLSI) systems*, 16(4):353–364, 2008.

[63] Kenneth McMillan. Modular specification and verification of a cache-coherent interface. In *Proceedings of the 16th Conference on Formal Methods in Computer-Aided Design*, FMCAD '16, page 109–116, Austin, Texas, 2016. FMCAD Inc.

[64] 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.

[65] Kenneth L. McMillan. A methodology for hardware verification using compositional model checking. *Science of Computer Programming*, 37(1-3):279–309, 2000.

[66] Kenneth L McMillan and Oded Padon. Ivy: a multi-modal verification tool for distributed algorithms. In *International Conference on Computer Aided Verification*, pages 190–202. Springer, 2020.

[67] Nicholas Moore and Mark Lawford. Correct safety critical hardware descriptions via static analysis and theorem proving. pages 58–64, 05 2017.

[68] Matthew Naylor and Simon Moore. A generic synthesisable test bench. In *2015 ACM/IEEE International Conference on Formal Methods and Models for Codesign (MEMOCODE)*, pages 128–137. IEEE, 2015.

[69] R. Nikhil. Bluespec System Verilog: efficient, correct RTL from high level specifications. In *Proceedings. Second ACM and IEEE International Conference on Formal Methods and Models for Co-Design, 2004. MEMOCODE '04.*, pages 69–70, June 2004.

[70] Oded Padon, Kenneth L McMillan, Aurojit Panda, Mooly Sagiv, and Sharon Shoham. Ivy: safety verification by interactive generalization. In *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 614–630, 2016.

[71] D. Price. Pentium FDIV flaw-lessons learned. *IEEE Micro*, 15(2):86–88, 1995.

[72] Dominic Richards and David Lester. A monadic approach to automated reasoning for Bluespec SystemVerilog. *Innovations in Systems and Software Engineering*, 7(85), 2011.

[73] S. A. Seshia and P. 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, Oct 2018.

[74] Mary Sheeran, Satnam Singh, and Gunnar Stålmarck. Checking safety properties using induction and a SAT-solver. In *International conference on formal methods in computer-aided design*, pages 127–144. Springer, 2000.

[75] Xiaowei Shen. *Modular Verification of Hardware Systems*. Massachusetts Institute of Technology, 2000.

[76] G. Singh and S. K. Shukla. Model checking Bluespec specified hardware designs. In *2007 Eighth International Workshop on Microprocessor Test and Verification*, pages 39–43, Dec 2007.

[77] Gaurav Singh and Sandeep K. Shukla. Verifying compiler based refinement of Bluespec specifications using the SPIN model checker. In Klaus Havelund, Rupak Majumdar, and Jens Palsberg, editors, *Model Checking Software*, pages 250–269, Berlin, Heidelberg, 2008. Springer Berlin Heidelberg.

[78] 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.

[79] Pramod Subramanyan, Bo-Yuan Huang, Yakir Vizel, Aarti Gupta, and Sharad Malik. Template-based parameterized synthesis of uniform instruction-level abstractions for SoC verification. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 37(8):1692–1705, 2017.

[80] Pramod Subramanyan, Yakir Vizel, Sayak Ray, and Sharad Malik. Template-based synthesis of instruction-level abstractions for SoC verification. In *2015 Formal Methods in Computer-Aided Design (FMCAD)*, pages 160–167. IEEE, 2015.

[81] Berkeley Verification and Synthesis Research Center. ABC: A system for sequential synthesis and verification. https://people.eecs.berkeley.edu/~alanmi/abc/abc.htm, 2012.

[82] Muralidaran Vijayaraghavan. *Modular Verification of Hardware Systems*. PhD thesis, Massachusetts Institute of Technology, 2016.

[83] Muralidaran Vijayaraghavan, Adam Chlipala, Arvind, and Nirav Dave. Modular deductive verification of multiprocessor hardware designs. In Daniel Kroening and Corina S. Păsăreanu, editors, *Computer Aided Verification*, pages 109–127, Cham, 2015. Springer International Publishing.

[84] Bruce Wile, John Goss, and Wolfgang Roesner. *Comprehensive functional verification: The complete industry cycle*. Morgan Kaufmann, 2005.

[85] C. Wolf. End-to-end formal ISA verification of RISC-V processors with riscv-formal. http://www.clifford.at/papers/2017/riscv-formal/slides.pdf.

[86] Sizhuo Zhang, Andrew Wright, Thomas Bourgeat, and Arvind. Composable building blocks to open up processor design. In *2018 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 68–81, Oct 2018.

[87] Sizhuo Zhang, Andrew Wright, Thomas Bourgeat, and Arvind. Composable building blocks to open up processor design. *IEEE Micro*, 39(3):47–55, 2019.