COMET: Tractable Reactive Program Synthesis

by

Christopher Chen

B.S., Portland State University, 2018

A THESIS SUBMITTED IN PARTIAL FULFILLMENT
OF THE REQUIREMENTS FOR THE DEGREE OF

Master of Science

in

THE FACULTY OF GRADUATE AND POSTDOCTORAL STUDIES
(Computer Science)

The University of British Columbia
(Vancouver)

January 2021

The following individuals certify that they have read, and recommend to the Faculty of Graduate and Postdoctoral Studies for acceptance, the thesis entitled:

COMET: Tractable Reactive Program Synthesis

submitted by **Christopher Chen** in partial fulfillment of the requirements for the degree of **Master of Science** in **Computer Science**.

Examining Committee:

Mark Greenstreet, Professor, Computer Science, UBC
Co-Supervisor

Margo Seltzer, Professor, Computer Science, UBC
Co-Supervisor

Alexander J. Summers, Associate Professor, Computer Science, UBC
Additional Examiner

Abstract

Reactive programs are programs that process external events, such as signals and messages. Device drivers and cloud microservices are examples of reactive programs. Systems built from reactive programs are concurrent and exhibit a high degree of nondeterminism, making non-exhaustive testing inadequate. A higher-level language can be used to write a *specification*: a formal definition of a program's desired behaviour. Such specifications may be easier to reason about, but proofs of the specification say nothing of a low-level implementation.

Program synthesis is one way to bridge this gap. A synthesizer searches a space of candidate programs for an implementation that satisfies the specification. In general, the number of candidate programs is infinite, making synthesis undecidable. Even a bounded search, e.g., on program length, soon becomes intractable as that search space grows exponentially.

This work introduces COMET, a system for synthesizing reactive programs from *unbounded nondeterministic iterative transformations* (UNITY) specifications. Recent work in symbolic execution and solver-aided synthesis has advanced the state of the art, but symbolic techniques also lead to exponential blowup. COMET seeks to avoid exponential blowup by constraining the search space and synthesizing in small steps. COMET synthesizes non-trivial programs for sequential and concurrent languages by applying three techniques: symbolic execution of the specification into guarded traces, intermediate target trace synthesis, and recursive synthesis of target expressions. To evaluate this approach, I synthesize Arduino and Verilog programs from UNITY specifications of the Paxos consensus proposer and acceptor roles.

Lay Summary

Reactive programs are computer programs that react to external inputs, e.g., automated banking machine software. It is difficult for humans to predict a reactive program's behaviour for all possible inputs, but by writing a program in a more abstract *specification language*, a programmer can use automated tools to analyze a program's complete behaviour. Unfortunately, computer systems are designed to run programs written in *low-level languages* and cannot run specifications directly.

COMET is a system for finding low-level reactive programs whose behaviour matches a specification. Finding programs is difficult because the number of programs to search through is huge. COMET breaks the search process into incremental steps and minimizes the complexity at each step. To demonstrate its efficacy, I use COMET to find low-level programs following specifications of Paxos, an algorithm for obtaining consensus across a group of networked computers.

Preface

All of the work presented henceforth was done at the Integrated Systems Design and Systopia laboratories at the University of British Columbia, Vancouver campus. This thesis is original, unpublished work by Christopher Chen, done under the supervision of Margo Seltzer and Mark Greenstreet.

Table of Contents

List of Tables

List of Figures

Glossary

API application programming interface

CAD computer-aided design

CSP communicating sequential processes

DES data encryption standard, a standard encyption algorithm from 1977

FIFO first-in-first-out, as in a queue

FPGA field programmable gate array, a programmable hardware device

HDL hardware description language

IEEE Institute of Electrical and Electronics Engineers

JIT just-in-time compiler

SMT satisfiability modulo theories

SVM symbolic virtual machine

TLA+ temporal logic of actions

UNITY unbounded nondeterministic iterative transformations

VLSI very large-scale integration, an integrated circuit

Acknowledgements

Writing a thesis is ordinarily difficult. Writing a thesis during the maelstrom of 2020 was especially hard and seemingly impossible at times. If you're reading this in the future, and it's hard in your time, know that I see you.

None of this would have come to be without the love and support of my parents and my partner Rachel: thank you. To Mark Greenstreet and Margo Seltzer: thank you for engaging in this messy effort with warmth, humour, and dedication. To Ivan Sutherland, Marly Roncken, and Warren Hunt: thank you for your mentorship and friendship, giving me a chance, and launching me on my way. To William J. Bowman: thank you for the poems. To Yan Peng and Justin Reiher: thank you for making my first two years in the Integrated Systems Design Laboratory so much fun. To Puneet Mehrotra: thank you for always listening. To Eric Lu and Crystal Hu: thank you for your comments and advice. To Adam Geller and Paulette Koronkevich: thank you for bravely and graciously using an early, bug-ridden version of this work as the basis for a project. I am so sorry. To the (late) Lewis Mitchell and Pete Stoelzl, my mentors at M&H Type: thank you for teaching me everything I know about typesetting and more. Finally, to Harriet (*Felis catus*) who spent countless hours by my side: thank you for reminding me to feed you and give you scritches.

CHAPTER 1

Introduction

Designing and implementing reactive software in a concurrent setting is a perilous task. Components operate independently: to an external observer, a system can exhibit a large degree of non-determinism. Each reactive component, by definition, *reacts* to an external environment outside of that component's control. This combination makes for systems that are difficult to reason about and diagnose, and traditional test and debugging practices are often inadequate for the task. Indeed, the term *heisenbug* entered the concurrent programming jargon to describe errors that disappear under investigation, because added debug code or diagnostic tools used to diagnose the problem cause subtle changes to the interleaving of events [17]. The problem seems to magically disappear, only to reemerge when the observer e.g., debug code is removed.

A stronger guarantee is provided by a formal analysis that accounts for the behaviour of a *formal model* of a system under all possible interleavings and all possible inputs. These models can be constructed using a process calculus such as the π-calculus or a specification language such as *communicating sequential processes* (CSP) or *temporal logic of actions* (TLA+) [14, 22, 29]. I refer to the high-level model of a system as its *specification*.

Once formalized, analysis tools, e.g., model checkers or theorem provers can be used prove that a specification satisfies desired correctness, safety, or liveness properties [6, 7, 18, 31]. Correctness in this case means that the program or algorithm arrives at the *correct* result. Safety can be informally described

as *nothing can go wrong*, while liveness can be described as *eventually something good happens*.

While it's nice to prove that a property holds for some specification, it's cold comfort if the specification and implementation are separate artifacts. To formally prove that implementations satisfy those desired properties, it's necessary to formalize the semantics of the implementation platform, then prove that an artifact implements the abstract specification, usually by showing some correspondence between the two. This so-called *proof burden* can be significant.

I propose an alternative approach. Given a semantics for a specification and implementation language, the problem of finding an implementation that follows a specification can be posed as a problem of program synthesis. Informally, the synthesizer answers and provides a witness for the question: given a specification, does there exist a program in the implementation language that for all input states, reacts in a way that satisfies the specification? A program synthesizer appears from the outside to be the same as a compiler: a function from programs to programs. The salient feature that separates compilation from synthesis is how the outputs are produced. Where a traditional compiler applies one or more mapping translations, a synthesizer searches over the space of output programs to find one whose semantics match some criteria. To a large degree, a synthesizer's effectiveness is governed by how efficiently it can search the space of programs, which can grow exponentially.

As described in Chapter 2, there has been much interest in compiler correctness, program synthesis, reactive systems, and symbolic execution. My work combines aspects of each to synthesize both concurrent and sequential programs that communicate with each other across the hardware/software divide. I present COMET, a method that: extracts *guarded traces* from *unbounded nondeterministic iterative transformations* (UNITY) specifications and synthesizes concurrent and sequential reactive programs from those guarded traces [28]. Synthesis occurs in two general phases: trace synthesis from UNITY guarded traces to low-level guarded traces, and program synthesis from low-level guarded traces to a low-level program. Trace synthesis allows the designer to express correctness criteria for concurrent and sequential programs in ways natural for each execution model.

My contributions are as follows:

1. Formalizations of subsets of: UNITY, the Arduino C++ *application programming interface* (API), and the Verilog *hardware description language* (HDL) in Rosette [1, 16, 37].

2. A symbolic interpreter for UNITY specifications that yields guarded traces.

3. Trace equivalence relations for synthesizing traces under concurrent and sequential models that are a refinement of the specification.

4. A recursive expression synthesizer for handling arbitrarily deep expressions.

I show that given a mapping between the domain of implementation state values to UNITY state values, it's possible to synthesize interoperable Arduino and Verilog implementations of reactive programs that are refinements of their specifications. Arduino and Verilog artifacts are further compiled and programmed onto microcontroller and *field programmable gate array* (FPGA) boards respectively. To validate the approach, I show the synthesis of the proposer and acceptor roles for the Paxos consensus algorithm for a 4-member ensemble.

The thesis is organized as follows: Chapter 2 discusses related work and provides a background for this thesis. Chapter 3 introduces UNITY, the language for specification, Arduino and Verilog, the languages for implementation, and Rosette, the language for formalization and synthesis. Chapter 4 goes further and provides a formal description of each language's syntax and how each is modelled in Rosette. Chapter 5 formalizes what it means for a Verilog or Arduino program to be correct with respect to a UNITY specification. Chapter 6 discusses why tractable program synthesis is so challenging, and the design and implementation of COMET. Chapter 7 introduces the Paxos consensus algorithm and the application of COMET to the synthesis of Arduino and Verilog implementations. Chapter 8 concludes the thesis and explores future avenues for work. Appendix A lists specification code in UNITY and synthesized implementation code in Arduino and Verilog.

CHAPTER 2

Related Work

At a high level, I focus on using program synthesis to translate executable specifications written in UNITY to low-level language implementations that satisfy correctness properties. We consider correct compilation because because we use similar definitions of correctness between a source program: the specification and the target program: the implementation. Program behaviours at the specification and implementation levels are characterized by their traces, or their record of value assignments made over their execution; this motivates us to consider trace-based methods. COMET synthesizes programs that run on microcontrollers and FPGAs; synthesis methods that target these hardware devices are discussed here. General and reactive program synthesis work germane to this work is presented. Finally, the use of solver-aided programming in COMET requires us to consider work in symbolic execution.

2.1 Correct compilation

The translation aspect of my work can be characterized as high-level compilation, with details left for program synthesis to fill in. With this in mind, I have found it useful to frame the synthesis problem in terms of compiler correctness and directed translation from specification languages to target implementations.

Moore was the first to provide a mechanical proof of correctness for compilation from an abstract assembly language Piton [30] to machine code for

Hunt's formally-verified FM8502 microprocessor [15]. Leroy's CompCert [25] was the first practical, multi-pass optimizing compiler, a 2 person-year effort yielding a 4-pass compiler back end from the Cminor intermediate language to PowerPC assembly. CompCert, written in the Coq proof assistant, provides a machine-checked proof of correctness that each compilation pass preserves observational equivalence between source and target programs and that equivalence composes over passes. More precisely, the correctness property states that if the source program has a well-defined behaviour, and the compiler successfully emits a target program, then the source and target are observationally equivalent. Work for the front end from a subset of C to Cminor took an additional person-year. I use the same simulation arguments and commutativity diagrams as those used by Moore and Leroy in relating the specification and the synthesized implementation.

Direct compilation is an alternative approach to synthesis. Zhang, Costa, and Do take this approach to the problem of translating specifications to implementations of distributed programs with Modular PlusCal and PGo [8, 10, 38]. Modular PlusCal is an extension of Lamport's PlusCal [23] specification language, and PGo is a compiler and runtime that emits Go programs. Their work allows Modular PlusCal specifications to be model checked to prove correctness properties, but neither the compiler, compiled implementations, or the runtime are verified. My work provides a certified implementation, by synthesis under correctness constraints and the verification of the final program.

2.2 Trace-based methods

A trace is a record of the observable actions generated by a program's execution. Using traces as source material for synthesis or compilation is widely investigated in the literature. One feature that sets my work apart from the following approaches is the search-based synthesis of equivalent low-level *traces* as opposed to *programs* in the first step of synthesis.

Biermann in 1972 was the first to show how complete traces of state machine behaviour could be used to synthesize Turing machines using a backtracking search [4]. He suggested that search-based program synthesis is pos-

sible using just input-output examples but feared that the number of examples required would prove cumbersome. Follow-on work in trace-based synthesis, so-called *programming by example* approaches usually try to generalize user-supplied traces, as described by Lau and Weld [24]. Programming by example infers a potentially incomplete specification from user-provided examples, whereas COMET consumes complete specifications.

Recent applications of trace-based compilation began with the trace-based *just-in-time* (JIT) compiler for JavaScript by Gal et al. [13]. Existing method-based JIT compilation strategies, which work well for statically type-checked languages, don't align well with dynamically type-checked languages such as JavaScript, due to the difficulties in generating specialized machine code when types are only known at runtime. Gal et al. showed how instrumenting the JavaScript interpreter to capture execution traces combined with a method for identifying hot loops allowed for the compilation of specialized sections of code that yielded substantial speedups.

2.3 High-level synthesis

Part of my work translates a higher-level specification into a digital circuit specified in the Verilog HDL [16]. This translation is referred to in the *very large-scale integration* (VLSI) *computer-aided design* (CAD) domain as *high-level synthesis* [9]. Existing approaches, however, achieve this translation via directed compilation and not search-based synthesis.

2.4 Inductive program synthesis

The program synthesis task from low-level guarded traces to a low-level program requires searching over the space of program expressions in the low-level language. In my work, the space of expressions, and therefore programs, is encoded as a tree structure, where the choice of possible children for a node is taken from the grammar of the language. In addition, many of the expressions we are interested in synthesizing involve bit-wise manipulations and comparisons.

Solar-Lezama et al. provide the first example of *sketch-based* program synthesis, referring to their technique as compilation by constraint-solving [36]. They approached the problem of correct compilation of bit-streaming programs for cryptographic algorithm implementations by providing the user with a dataflow language for specification, and a method for writing a partial program, or *sketch*. The sketch contains *holes* that the program synthesizer fills to satisfy constraints. By solving for the constraints posed by the specification and the sketch, they were able to generate complete, optimized implementations of the *data encryption standard* (DES) and Serpent cryptographic algorithms that were as fast or faster than existing implementations.

The number of branches, representing the number of different choices in constructing a program, grows exponentially in the depth of a tree. Shah, Kulal, and Bodik present an method to represent this search space as a *symbolic syntax graph*. A symbolic syntax graph minimizes the number of nodes in the tree using typing information, mutability constraints, and collapsing common nodes. The reduction in the search space can be significant in certain applications, up to an order of magnitude for a change counting benchmark. I extend these techniques in Chapter 6 with expression splitting to find smaller targets for synthesis, with the expectation that these synthesized subexpressions can be composed to solve the larger problem.

2.5 Reactive program synthesis

Initial reactive systems synthesis work focused on inferring transition systems from specifications. Madhusudan argued instead for a focus on synthesizing reactive programs as a more compact and useful artifact [27]. His work showed that the synthesis of boolean imperative recursive and non-recursive programs from specifications given as a set of traces is decidable and that the decision procedure could be constructed in exponential time.

Finkbeiner, Klein, Piskac, and Satolucito synthesize concrete functional reactive programs from specifications given in temporal stream logic [11, 12]. This work is similar to mine in that it targets software and hardware but targets

higher level functional frameworks, and is unconcerned with lower-level concerns such as preserving refinement.

Termite and Termite-2 by Ryzhyk et al. propose a game-theoretic approach to synthesizing device drivers from specifications of hardware and operating system interfaces [34, 35]. Termite takes device and operating system specifications given as state machines and combines them into a composite state machine that defines their interactions. The synthesis task is to find a *winning strategy* in that composite state machine that preserve both safety and liveness in the face of an adversarial environment. This winning strategy is fed into a code generator, producing a device driver in C. While Termite is an automatic, *push-button* synthesizer, Termite-2 is a user-guided synthesis and verification approach to the development of device drivers. In addition to the device and operating system specifications, Termite-2 requires an initial program sketch of the device driver. Like Termite, Termite-2 assumes an adversarial environment and tries to find a similar winning strategy within the constraints of the existing program sketch. If synthesis fails, Termite-2 provides feedback for the programmer to refine or repair the sketch or specification. In COMET, the state machine to be synthesized is given in the UNITY specifications, so my concern is preserving the semantics of that abstract specification in low-level hardware and software implementations, as opposed to synthesizing a state machine interposed between hardware and software. Unlike Termite and Termite-2, COMET provides no liveness guarantees.

2.6 Symbolic execution

My approach to synthesis relies upon satisfying an equivalence relation between the symbolic executions of a specification and a candidate program or trace in the implementation language.

Symbolic execution, pioneered by King, models a program's execution as a logical formula, which allows for the verification of correctness conditions [19]. This technique requires a symbolic interpreter or compiler for a language of interest, which may not be readily available. Torlak and Bodik's Rosette provides a language for building symbolic interpreters on top of a *symbolic virtual*

machine (SVM) [37]. Rosette provides support for verification and program synthesis on top of the SVM which I use for my work.

Modelling symbolic state for a program execution with even modest conditional control flow leads to an exponential blowup in the *path conditions* that must be tracked. Rosette, like many modern symbolic execution engines, supports *state merging* to minimize the number of path conditions in an execution. This optimization can be deleterious to performance, as the path conditions themselves are become more complex, as discussed by Kuznetsov et al. [20]. My work disables state merging in the symbolic interpreter for UNITY specifications, to keep guarded transitions distinct, making it possible to handle each separately. The control of state merging and its implications for symbolic execution performance in Rosette is further discussed by Porncharoenwase, Bornholt, and Torlak [32].

CHAPTER 3

Preliminaries

This chapter provides an introduction to the working languages for this thesis. UNITY is the language used to specify reactive systems. Arduino and Verilog are low-level languages used for implementation [1, 16]. Executable models for all three languages are written in Rosette, whose symbolic virtual machine translates assertions into *satisfiability modulo theories* (SMT) formulae [37]. These models are described in Chapter 4.

I present a single-value *first-in-first-out* (FIFO) boolean inverter in UNITY, Arduino, and Verilog. The UNITY example works over two channels: one input and one output. A channel is $\texttt{Empty}$ or $\texttt{Full}(value)$, where $value$ is a boolean value. A sender can transition a channel from $\texttt{Empty} \rightarrow \texttt{Full}(value)$, and the receiver $\texttt{Full}(value) \rightarrow \texttt{Empty}$.

A channel is implemented in Arduino and Verilog with three physical *input/output* (I/O) wires: req (request), ack (acknowledge), and $value$. This implementation is a *non-return-to-zero* protocol: channel state is determined by the relative difference between req and ack, not their absolute values. When $req = ack$, the channel is $\texttt{Empty}$, and when $req \neq ack$, the channel is $\texttt{Full}(value)$. The sender controls the req and $value$ wire and the receiver controls the ack wire.

```
program INV-FIFO
  declare
    in: recv-channel
    out : send-channel

  initially
    out := empty

  assign
    in, out := empty, full(not(value(in)))
      if (full?(in) and empty?(out))
```

Figure 3.1: A single-value boolean FIFO inverter in UNITY

3.1 UNITY

UNITY by Chandy and Misra is a language and associated program logic for the design and analysis of parallel programs [28]. The design language leaves many details about execution unspecified, leaving implementation decisions to the *mappings* to different parallel architectures, e.g., shared-memory multiprocessors, message passing computers, and distributed systems.

At a high level, UNITY's semantics reflect an abstract state machine. Each state transition is a conditional or unconditional multi-assignment, where expressions are evaluated using values from before the transition. A conditional multi-assignment is a piece-wise function with mutually-exclusive guard expressions. The evaluation schedule for the guards is subject to a *weak fairness* constraint: a multi-assignment whose guard is persistently enabled will execute eventually. An additional way of combining transitions is by nondeterministic choice, where the program can select a transition from a set or conditional of unconditional multi-assignments.

An execution of a UNITY program yields an *execution sequence*, which is an infinite sequence of pairs, $(\mathtt{state}, \mathtt{label})$, where $\mathtt{state}$ describes the current state of the program and $\mathtt{label}$ indicates the transition selected for execution. A non-trivial program can yield an infinite set of execution sequences. A UNITY program logic allows one to prove safety and liveness properties over a program's executions. The program logic is not used in this thesis.

The example in Figure 3.1 implements the example FIFO inverter and illustrates the three sections of a UNITY program:

- The **declare** section specifies the program's variables. In this case, receive channel `in` and send channel `out` are declared. Channels are an addition to the UNITY language and are defined in Chapter 4.
- The **initially** section specifies the initial state of the program. Here, the output channel `out` is set to `empty`.
- The **assign** section specifies iterative state assignments. The boolean expression `(full?(in) and empty?(out))` guards the single assignment. Multiple assignment is respective, concurrent, and atomic, e.g., the right-hand values `empty` and `full(value(in))` are evaluated before assignment to `in` and `out` respectively.

3.2 Wiring/Arduino

Wiring [3] is an open-source hardware and software platform by Barragán that extends the Processing [33] programming environment to enable low-cost experimentation with physical computing devices: those that interact with the physical world via sensors and actuators. Arduino is derived from Wiring, and supports a broad range of hardware devices, from low-cost 8-bit microcontrollers to higher-performance microprocessors with auxiliary FPGAs [1]. The Arduino language is an API in C++, with functions for reading and writing from input/output hardware pins in digital or analog modes. A substantial amount of the environment consists of compilers for each of the targeted platforms and tools to program, or write, compiled binaries to the hardware.

The example in Figure 3.2 implements the example FIFO inverter and illustrates the three sections of an Arduino program:

- Constants and variable declarations are made at the top level. Here symbolic names are mapped to physical I/O pin numbers.

```
const in_req = 0;
const in_ack = 1;
const in_data = 2;
const out_req = 3;
const out_ack = 4;
const out_data = 5;

void setup() {
  pinMode(in_req, INPUT);
  pinMode(in_ack, OUTPUT);
  pinMode(in_data, INPUT);
  pinMode(out_req, OUTPUT);
  pinMode(out_ack, INPUT);
  pinMode(out_data, OUTPUT);
  // set output empty
  digitalWrite(out_req, digitalRead(out_ack));
}

void loop() {
  if ((digitalRead(in_req) != digitalRead(in_ack)) &&
      (digitalRead(out_req) == digitalRead(out_ack))) {
    digitalWrite(out_data, !digitalRead(in_data));
    digitalWrite(out_req, !digitalRead(out_ack));
    digitalWrite(in_ack, digitalRead(in_req));
  }
}
```

Figure 3.2: A single-value boolean FIFO inverter in Arduino

- A **setup** routine runs at initialization time and completes the setup of I/O pins. Pins are set to their input or output mode, and `out_req` is set to the value of `out_ack`, emptying the output channel.
- An event **loop** runs continuously. The `if` statement is a translation of the **assign** section of the UNITY program, with a safe ordering of the `digitalWrite` statements. Safety is discussed in Chapter 5.

3.3 Verilog

Verilog is a HDL defined by the *Institute of Electrical and Electronics Engineers* (IEEE) Standard 1364-2005, intended for the design and simulation of digital hardware systems [16]. To that end, Verilog contains constructs not typical of

```
module fifo_inv(in_ack, in_req, in_data,
                out_ack, out_req, out_data, clock, reset);

  input wire in_req;
  output reg in_ack;
  input wire in_data;
  output reg out_req;
  input wire out_ack;
  output reg out_data;
  input wire clock;
  input wire reset;

  always @(posedge clock or posedge reset)
    begin
      if (reset)
        begin
          out_req <= out_ack;
        end
      else
        begin
          if ((in_req != in_ack) && (out_req == out_ack))
            begin
              out_data <= !in_data;
              out_req <= !out_ack;
              in_ack <= in_req;
            end
        end
    end
endmodule
```

Figure 3.3: A single-value boolean FIFO inverter in Verilog

"normal" programming languages. While Verilog contains blocking statements that can be used to encode sequencing in an imperative fashion, the language itself allows for a high degree of concurrency between statements, reflecting the concurrent nature of digital circuits. Timing constraints can be made explicit, to model the delays caused by long wires. Although Verilog is intended for designing digital systems, part of the language exists for testing and simulation. *Synthesizable Verilog* refers to the subset of the language that can be translated or synthesized into digital circuits.

The example in Figure 3.3 implements the example FIFO inverter, and shows the three sections of a Verilog module:

- The **module signature** declares the module name and lists the names exposed to other modules for interfacing.
- **Type declarations** inside the module specify the data types of names used. This is analogous to the **declare** section of a UNITY program or the variable declaration statements of an Arduino program. Behaviour modes for input or output are specified as part of the declaration.
- An **always** block is evaluated every time a *triggering event* occurs. Here, the block is evaluated every time `clock` or `reset` transition from a negative to positive, also called *edge triggering*. If `reset` is positive, then the *nonblocking* assignment `out_req <= out_ack` empties the output channel. Otherwise, the next `if` statement, similar in structure to the Arduino implementation, is evaluated. The behaviour of nonblocking assignments are similar to UNITY assignment: all right-hand values are evaluated first, and all assignments occur concurrently at the end of the always block.

3.4 Rosette

Rosette is a language implemented in Racket that provides support for symbolic execution and solver-aided programming, which include verification, synthesis, bug-fixing, and superoptimization [37].

At the core of Rosette is the SVM which executes a program with symbolic inputs, tracking *path conditions* that occur at conditional program points and attempting *state merging* when multiple conditional paths rejoin. If a symbolic value depends on a path condition, it is encoded as a *symbolic union*: a set of $[guard, value]$ pairs where each guard represents a path condition. Figure 3.4 shows the symbolic unions that result from two conditional expressions. The second symbolic union is the result of the expression `(if X (if Y 'left 'right) (if Y 'left 'right))`. It is a simple demonstration of state merging: the four possible paths are merged into a symbolic union of two elements. A programmer engages in *symbolic reflection* by examining the guards and values of a symbolic union much like a programmer engages in object-

```
> ;; Symbolic booleans
> (define-symbolic X Y boolean?)

> ;; Conditional expression
> (if X 'left 'right)

;; Symbolic union
{[X left]
 [(! X) right]}

> ;; More complex conditional expression
> (if X (if Y 'left 'right) (if Y 'left 'right))

;; State-merged symbolic union
{[(|| (&& X (! Y)) (&& (! X) (! Y))) right]
 [(|| (&& X Y) (&& Y (! X))) left]}
```

Figure 3.4: Symbolic execution in Rosette

oriented reflection by examining an object's class hierarchy. COMET uses symbolic reflection to decompose synthesis tasks in Chapter 6.

For *solver-aided programming*, assertions are encoded into satisfiability queries and sent to a SMT solver, as in Figure 3.5. A verification query asks for a counterexample to the assertion given. In the case of De Morgan's laws, no counterexample can be found, so an unsatisifiable result is returned. In the case of the second, bad verification query, a counterexample model is returned, providing values that violate the assertion.

This introduction concludes with synthesis, illustrated in 3.6. Given a language over the integers with one variable and two binary operators, `add` and `sub`, defined as addition and subtraction, synthesize an expression equivalent to multiplying that variable by 3. The example begins by defining the syntax of the language and an evaluator for expressions. A function is defined to generate arbitrarily deep search spaces. A 1-deep search space shows how the search space is encoded: a symbolic union, guarded by the boolean variables `x?$0` and `x?$1`. To synthesize, a search space is defined, and a query is made: for all values of `X`, find an expression in the search space that evaluates to the same value as multiplying `X` by 3. When successful, the response is a model, an assignment to the boolean variables that constitute the guards in the search space. When

```
> ;; Symbolic booleans
> (define-symbolic X Y boolean?)

> ;; Verification query: De Morgan's laws
> (verify (assert (eq? (not (and X Y))
                       (or (not X) (not Y)))))

;; Unsatisifiable result: the assertion is verified
(unsat)

> ;; Bad verification query
> (verify (assert (eq? (not (and X Y))
                       (not (or X Y)))))

;; Satisfiable result: the model is the counterexample
(model
 [X #t]
 [Y #f])
```

Figure 3.5: Verification in Rosette

the model is applied to the search space, a satisfying expression results: `(add (add X X) X)`.

```
> (require rosette/lib/angelic
           rosette/lib/match)

> ;; Define grammar
> (struct add (l r) #:transparent)
> (struct sub (l r) #:transparent)

> ;; Define semantics
> (define (evaluate-expr expr)
    (match expr
      [(add l r) (+ (evaluate-expr l)
                    (evaluate-expr r))]
      [(sub l r) (- (evaluate-expr l)
                    (evaluate-expr r))]
      [e e]))

> ;; Symbolic integer
> (define-symbolic X integer?)

> ;; Build a choose tree of depth N
> (define (choose-tree depth)
    (if (zero? depth)
        X
        (let ([l-expr (choose-tree (sub1 depth))]
              [r-expr (choose-tree (sub1 depth))])
          (choose* X
                   ((choose* add sub) l-expr r-expr)))))

> ;; 1-deep choose-tree
> (choose-tree 1)

;; 1-deep choose-tree: a symbolic union
{[x?$1 X]
 [(&& x?$0 (! x?$1)) #(struct:add X X)]
 [(&& (! x?$0) (! x?$1)) #(struct:sub X X)]}

> ;; Synthesize an expression equal to (* 3 X)
> (let* ([sketch (choose-tree 2)]
         [model (synthesize
                 #:forall X
                 #:guarantee (assert (eq? (* 3 X)
                                          (evaluate-expr sketch))))])
    (evaluate sketch model))

;; The synthesized expression
(add (add X X) X)
```

Figure 3.6: Synthesis in Rosette

CHAPTER 4

Language Formalization

> *Remember that all models are wrong; the practical question is how wrong do they have to be to not be useful.*
>
> George Box

This chapter introduces the mechanized formal models in Rosette for UNITY, Arduino, and Verilog. Common features and patterns are covered first. Each language's specific data types, semantics, and mechanizations in Rosette are covered individually.

Formatting note: Rosette is a dialect of Lisp, and COMET uses S-expressions internally, with operator prefix notion enclosed in parentheses: (operator $\mathtt{arg}_0 \ldots \mathtt{arg}_n$). Native syntax presented in this chapter uses operator($\mathtt{arg}_0, \ldots \mathtt{arg}_n$) notation. Individual type annotations are presented as variable: type and parameterized type annotations are presented as variable: list[type].

4.1 Common features

Each model consists of a type context, a state, an expression evaluator, and a statement interpreter. A *type context* is a mapping from variables to data types: it defines the legal variables for a program and each variable's type. A *state* is an append-only mapping from variables to values: it defines the current value for

a variable. Type contexts and states are often handled together: an *environment* is a pair containing a type context and a state. An *expression* yields a value when evaluated over an environment. A *statement* yields an environment when interpreted over an environment. For example, for the integers x bound to 1 and y bound to 2:

$$context = \{(x \mapsto \mathsf{integer}), (y \mapsto \mathsf{integer})\} \tag{4.1}$$

$$state = \{(x \mapsto 1), (y \mapsto 2)\} \tag{4.2}$$

$$environment = \langle context, state \rangle \tag{4.3}$$

Evaluating the expression x + y over $environment$ yields the value 3:

$$evaluate(\mathsf{x} + \mathsf{y}, environment) \;=\; 3 \tag{4.4}$$

Statements modify the type context by declaring variables, modify the state by making assignments or taking output actions, or take different paths of execution, depending on the values of expressions. For example, declaring an integer w over $environment$ yields a new environment with a new context:

$$\begin{aligned} &interpret(\mathsf{integer\ w}, environment) \;= \\ &\quad \langle \{(x \mapsto \mathsf{integer}), (y \mapsto \mathsf{integer}), (w \mapsto \mathsf{integer})\}, \\ &\qquad state \rangle \end{aligned} \tag{4.5}$$

or assigning $x+y$ to x over $environment$ yields a new environment with a new state:

$$\begin{aligned} &interpret(\mathsf{x := x + y}, environment) \;= \\ &\quad \langle context, \\ &\qquad \{(x \mapsto 1), (y \mapsto 2), (x \mapsto 3)\} \rangle \end{aligned} \tag{4.6}$$

Note that new type context and state are created by appending values to copies of the previous type context or state. The mutability of the type context depends on the language model: for UNITY and Verilog the type context is set before ex-

Type	Rosette Representation
boolean	boolean
natural	integer
recv-buffer	buffer(cursor: integer, values: list[boolean])
send-buffer	buffer(cursor: integer, values: list[boolean])
recv-channel	channel(valid: boolean, value: boolean)
send-channel	channel(valid: boolean, value: boolean)

Table 4.1: UNITY data types and Rosette representations

ecution begins, while for Arduino the type context can change throughout the execution of the program. In practice however, COMET only synthesizes Arduino programs where the context is set at the beginning, before the execution of any state-altering statements. The state's append-only nature makes it a trace, or history, of values as execution proceeds.

4.2 UNITY

A subset of Chandy and Misra's UNITY language is presented below [28]. This section covers data types, syntax and semantics, and the mechanization of the language in Rosette.

4.2.1 UNITY data types

UNITY data types and their representations in Rosette are given in Table 4.1. Boolean and natural number types are implemented with Rosette's boolean and integer types. Buffer and channel types are represented by Rosette *structs*, a record datatype.

COMET uses buffers for sending and receiving natural numbers over boolean channels: the cursor value points to the *current index* in a list of boolean values, where the low-order bit comes first. The size of the list is fixed when the buffer is declared. For example, a buffer of two values representing the number two with a cursor pointing at zero-count element one, or true:

Operator(s)	Type
and or eq	$boolean \times boolean \rightarrow boolean$
+	$natural \times natural \rightarrow natural$
< =	$natural \times natural \rightarrow boolean$
recv-buf-put	$recv\text{-}buffer \times boolean \rightarrow recv\text{-}buffer$
nat-to-send-buf	$natural \times natural \rightarrow send\text{-}buffer$

Table 4.2: UNITY binary operator types

$$buffer(1, list(\mathsf{false}, \mathsf{true})) \tag{4.7}$$

COMET uses channels for communication between UNITY programs: the valid boolean determines if the channel is full or empty. For example, a full channel containing the value false:

$$channel(\mathsf{true}, \mathsf{false}) \tag{4.8}$$

Note that send and receive variants of buffers and channels are distinct types, even though they share the same underlying representation.

4.2.2 UNITY expressions

UNITY expression syntax is presented in Figure 4.1. A type summary of the binary operators is given in Table 4.2. A type summary of the unary operators is given in Table 4.3. Boolean and arithmetic expressions are standard, but buffer and channel expressions deserve some explanation.

Buffer expressions

There are two buffer predicates: recv-buf-full? for recv-buffers and send-buf-empty? for send-buffers. To understand their motivation, consider each buffer's use case. If recv-buf-full? is true, the buffer is ready for conversion to a natural number. Otherwise, the buffer requires additional values. If send-buf-empty?

```
bool       ∈  BOOLEAN
var        ∈  VARIABLE
nat        ∈  NATURAL

expression ⊢  bool | nat | var | empty
           |  ( expr binop-infix expr )
           |  binop-prefix ( expr expr )
           |  unop ( expr )

binop-infix ⊢  and | or | eq
            |  + | < | =

binop-prefix ⊢  recv-buf-put | nat-to-send-buf

unop       ⊢  not
           |  empty? | full?
           |  recv-buf-full? | send-buf-empty?
           |  message | value
           |  empty-recv-buf | empty-send-buf | send-buf-next
           |  recv-buf-to-nat | send-buf-get
```

Figure 4.1: UNITY expression syntax

is true, all buffer values have been sent, and the buffer is ready for reuse. Otherwise, the buffer has values that need to be sent. The empty-recv-buf constructor creates a new recv-buffer of a set size, with a pre-populated list of false values, as in this example:

$$\begin{aligned} &evaluate(\text{empty-recv-buf}(4), \ldots) = \\ &\quad buffer(0, list(\text{false}, \text{false}, \text{false}, \text{false})) \end{aligned} \tag{4.9}$$

The $environment$ argument to the evaluation function is elided for brevity. In contrast, the empty-send-buf constructor creates a new send-buffer of a set size, with a pre-populated list of false values and a cursor set to indicate an *exhausted*

Operator(s)	Type
not	$boolean \rightarrow boolean$
empty?	$send\text{-}channel \rightarrow boolean$
full?	$recv\text{-}channel \rightarrow boolean$
recv-buf-full?	$recv\text{-}buffer \rightarrow boolean$
send-buf-empty?	$send\text{-}buffer \rightarrow boolean$
message	$boolean \rightarrow send\text{-}channel$
value	$recv\text{-}channel \rightarrow boolean$
empty-recv-buf	$natural \rightarrow recv\text{-}buffer$
empty-send-buf	$natural \rightarrow send\text{-}buffer$
send-buf-next	$send\text{-}buffer \rightarrow send\text{-}buffer$
recv-buf-to-nat	$recv\text{-}buffer \rightarrow natural$
send-buf-get	$send\text{-}buffer \rightarrow boolean$

Table 4.3: UNITY unary operator types

buffer, as in this example:

$$evaluate(\text{empty-send-buf}(4), \ldots) = \\ buffer(4, list(\text{false}, \text{false}, \text{false}, \text{false})) \tag{4.10}$$

The nat-to-send-buf operator constructs a new send-buffer given a buffer size from a natural number value, as in this example:

$$evaluate(\text{nat-to-send-buf}(4, 2), \ldots) = \\ buffer(0, list(\text{false}, \text{true}, \text{false}, \text{false})) \tag{4.11}$$

The recv-buf-put operator inserts a boolean value into an existing non-full recv-buffer and returns a new buffer with the cursor incremented, as in this example:

$$evaluate(\text{recv-buf-put}(buffer(0, list(\text{false}, \text{false})), \text{true}), \ldots) = \\ buffer(1, list(\text{true}, \text{false})) \tag{4.12}$$

The send-buf-next operator increments the cursor, while send-buf-get extracts the current value of a non-empty send-buffer. A full recv-buffer's natural number representation is extracted with recv-buf-to-nat.

Channel expressions

There are two channel predicates: empty? for send-channels and full? for recv-channels. Note that empty? is not defined over recv-channels and full? is not defined over send-channels. To understand why, consider that both empty recv-channels and full send-channels are channel states that can change at any time due to external action. The message constructor creates a new send-channel value, in this case containing the value false:

$$evaluate(\mathsf{message}(\mathsf{false}), \ldots) = channel(\mathsf{true}, \mathsf{false}) \tag{4.13}$$

When a recv-channel is full, the receiver extracts its contents with value:

$$evaluate(\mathsf{value}(channel(\mathsf{true}, \mathsf{false})), \ldots) = \mathsf{false} \tag{4.14}$$

The empty reserved word creates an empty recv-channel value:

$$evaluate(\mathsf{empty}, \ldots) = channel(\mathsf{false}, \mathsf{null}) \tag{4.15}$$

4.2.3 UNITY statements

UNITY statement syntax is presented in Figure 4.2. Statements consist of a list of unconditional or conditional assignments. Unconditional assignments evaluate an expression list to a value list of the same length; those values are applied respectively to a list of variables. Conditional assignments apply expression values to their list of variables if the guard expression is true. In both cases, new value mappings are appended to the previous state and the new state is returned. The assignments are considered concurrent and atomic: there are no intermediate states. COMET models a deterministic subset of UNITY. Inter-

```
    expr  ∈  Expression
     var  ∈  Variable

statement  ⊢  list[assignment]

assignment  ⊢  unconditional
            |  conditional

unconditional  ⊢  vars := exprs
  conditional  ⊢  vars := list[case]

         case  ⊢  exprs if expr

         vars  ⊢  var , vars
               |  var

        exprs  ⊢  expr , exprs
               |  expr
```

Figure 4.2: UNITY statement syntax

pretation of statements is defined only for single unconditional assignments or mutually exclusive conditional assignments.

4.2.4 UNITY programs

UNITY program syntax is presented in Figure 4.3. Program initialization begins by populating the type context, specified in the **declare** section, followed by the initial assignment, specified by the unconditional assignments in the **initially** section. The program then repeatedly iterates over the statements in the **assign** section.

4.2.5 Mechanizing UNITY

Mechanizing the UNITY language in Rosette involves defining:

1. An encoding for type contexts and states

```
      var  ∈  VARIABLE
statement  ∈  STATEMENT
unconditional  ∈  UNCONDITIONAL

  program  ⊢  program var declare initially assign

  declare  ⊢  declare list[declaration]
initially  ⊢  initially unconditional
   assign  ⊢  assign statement

declaration  ⊢  var : type

     type  ⊢  natural | boolean
           |  recv-buffer | send-buffer
           |  recv-channel | send-channel
```

Figure 4.3: UNITY program syntax

2. Struct record types to reflect the language syntax

3. An *evaluator function* to implement the expression semantics

4. An *interpreter function* to implement the statement semantics

The interpreter function is used in two wrapper functions: an initializer function to set the environment from the **declare** and **initially** sections, and an assignment function to take a single iterative step from the **assign** section.

I will be using the receiver program listed in Figure 4.4 as a motivating example. The program receives boolean values over an input channel, populating a receive buffer until it fills up. When the buffer fills up, its natural number representation is extracted and stored in a local variable and the receive process begins again.

Type contexts and states

Type contexts and states are represented in Rosette as lists of *cons* pairs, each pair representing a mapping from the first element to the second. Variables

```
program RECEIVER
  declare
    in: recv-channel
    buf: recv-buffer
    val: natural

  initially
    buf, val := empty-recv-buf(8), 0

  assign
    in, buf := empty, recv-buf-put(buf, value(in))
      if (full?(in) and not(recv-buf-full?(buf)))
    buf, val := empty-recv-buf(8), recv-buf-to-nat(buf)
      if recv-buf-full?(buf)
```

Figure 4.4: A natural number receiver in UNITY

and types are represented using Rosette *symbols*, appearing as strings prefixed with an apostrophe. These symbols are Lisp/Scheme data objects as opposed to Rosette symbolic values, and can be thought of as immutable strings. For example, a type context for the receiver program:

$$\{(\mathtt{in} \mapsto \mathsf{recv\text{-}channel}), (\mathtt{buf} \mapsto \mathsf{recv\text{-}buffer}), (\mathit{val} \mapsto \mathsf{natural})\} \tag{4.16}$$

is represented in Rosette with this list:

```
(list (cons 'val 'natural)
      (cons 'buf 'recv-buffer)
      (cons 'in 'recv-channel))
```

Note the *reverse ordering*: lists in Rosette are efficiently constructed by adding elements to the *head* of the list rather than appending as is the customary presentation introduced in Section 4.1. For example, the receiver's initial state:

$$\{(\mathtt{buf} \mapsto \mathtt{buffer}(0, \mathtt{list}(\mathsf{false}, \mathsf{false}, \mathsf{false}, \mathsf{false}, \mathsf{false}, \mathsf{false}, \mathsf{false}, \mathsf{false}))),$$
$$(\mathit{val} \mapsto 0)\} \tag{4.17}$$

is represented in Rosette with this list:

```
(list (cons 'val 0)
      (cons 'buf (buffer* 0 (list #f #f #f #f #f #f #f #f))))
```

Rosette boolean literals are `#t` and `#f` for true and false. UNITY integer literals are represented by Rosette integer literals. Adding a full `in` channel with the literal `#t` yields:

```
(list (cons 'in (channel* #t #t))
      (cons 'val 0)
      (cons 'buf (buffer* 0 (list #f #f #f #f #f #f #f #f))))
```

Because many names collide with built-in Racket/Rosette functions, COMET struct names all have an asterisk suffix.

Syntax

The abstract syntax tree for a UNITY program is constructed in Rosette with nested structs. Structs are defined for each syntactic element: expression operators, conditional assignments, unconditional assignments, etc. A struct definition includes at the minimum the struct name and a list of fields. The definition can include optional annotations such as transparency, covered in Chapter 6. For example, defining structs for the and, not, full?, and recv-buf-full?:

```
(struct and* (left right))

(struct not* (expr))

(struct full?* (chan))

(struct recv-buf-full?* (buf))
```

For example, the following syntax tree corresponds to the first guard in the receiver's assignment:

```
(and* (full?* 'in)
      (not* (recv-buf-full?* 'buf)))
```

The struct representing the UNITY assignment operator `:=`, has two fields corresponding to the operator's operands: a list of variables and *either* a list of expressions for the unconditional case or a list of cases:

```
(struct :=* (vars values-or-cases))

(struct case* (values guard))
```

For example, the following syntax tree corresponds to the receiver program's initial assignment:

```
(:=* (list 'buf 'val)
     (list (empty-recv-buf* 8) 0)))
```

and the receiver program's iterative assignment:

```
(list
    (:=* (list 'in 'buf)
         (case* (list 'empty (recv-buf-put* 'buf (value* 'in)))
                (and* (full?* 'in)
                      (not* (recv-buf-full?* 'buf)))))
    (:=* (list 'buf 'val)
         (case* (list (empty-recv-buf* 8) (recv-buf-to-nat* 'buf))
                (recv-buf-full?* 'buf))))
```

Expression semantics

Expression semantics are defined and mechanized in Rosette by the following:

```
(define (evaluate-expr expr context state)
    (match expr
        [(not* e) (not (evaluate-expr e context state))]
        [(and* l r) (and (evaluate-expr l context state)
                         (evaluate-expr r context state))]
        ...
        [(recv-buf-full?* b) (eval-recv-buf-full?
                                  (evaluate-expr b context state))]
        ...
        [(full?* c) (channel*-valid (evaluate-expr c context state))]
        [(value* c) (channel*-value (evaluate-expr c context state))]
        ...
        [term (cond [(eq? term 'empty) (channel* #f null)]
                    [(boolean? term) term]
                    [(natural? term) term]
                    [(symbol? term) (get-mapping term state)])]))
```

The evaluator is a recursive function that uses pattern matching to decompose expressions. The first two pattern matching clauses on lines 3 and 4 show how simple boolean operators are handled, by applying the boolean operator to the evaluated values of the subexpressions. For more complex operators, helper

functions are used, as on line 7. The `eval-recv-buf-full?` helper evaluates to true if the cursor is greater than or equal to the length of the values list:

```
1  (define (eval-recv-buf-full? buf)
2      (match buf
3          [(buffer* cursor values)
4              (>= cursor (length values))]))
```

For channel operators `full?` and `value` on lines 10 and 11, the evaluator uses field accessor functions to extract the appropriate field from the channel struct. The final pattern match on line 13 handles terminals of the syntax tree: a conditional expression handles cases where the `'empty` reserved word is used, a literal boolean or natural number value is given, or if an variable is used. Variables are evaluated by returning the latest value for the variable in the state. For example, evaluating this expression under this context and state yields the value true:

```
1  (evaluate-expr
2      ;; The expression to evaluate
3      (and* (full?* 'in)
4            (not* (recv-buf-full?* 'buf)))
5      ;; The type context
6      (list (cons 'val 'natural)
7            (cons 'buf 'recv-buffer)
8            (cons 'in 'recv-channel))
9      ;; The state
10     (list (cons 'in (channel* #t #t))
11           (cons 'val 0)
12           (cons 'buf (buffer* 0 (list #f #f #f #f #f #f #f #f)))))
```

Statement semantics

Statement semantics are defined and mechanized in Rosette by the following:

```
(define (interpret-assign-stmts assignments context state)
    (define (build-trace vars exprs)
        (map (lambda (v e)
                 (cons v
                       (evaluate-expr e context state)))
             vars
             exprs))

    (define (expand-cases vars cases)
        (map (lambda (c)
               (let ([exps (car c)]
                     [guard (cdr c)])
                  (guard-trace (evaluate-expr guard context state)
                               (build-trace vars exps))))
             cases))

    (define (regularize-assign assign)
        (match assign
               [(:=* vars exps)
                (match exps
                       [(case* cases) (expand-cases vars cases)]
                       [_ (guard-trace #t
                                       (build-trace vars exps))])]))

    (define (apply-traces traces)
        (match traces
               ['() (stobj state)]
               [(cons head tail)
                (match head
                       [(guard-trace guard? trace)
                        (if guard?
                            (append trace state)
                          (apply-traces tail))])]))

    (let ([guard-traces (flatten (map regularize-assign assignments))])
        (apply-traces guard-traces)))
```

The interpreter begins by regularizing the assignments into independent *guarded traces*. A guarded trace contains a guard and *next* state *if* the guard is true. For unconditional assignments, the guard is set to `#t`. For conditional case assignments, the cases are expanded and a guarded trace is generated for each. When regularization is complete, the guarded traces are applied and the first guarded trace with a true guard is returned as the new state. For example, interpreting the iterative assignment under a context and state:

```
(interpret-stmt
    ;; The statement to evaluate
    (list
        (:=* (list 'in 'buf)
             (case* (list 'empty (recv-buf-put* 'buf (value* 'in)))
                    (and* (full?* 'in)
                          (not* (recv-buf-full?* 'buf)))))
        (:=* (list 'buf 'val)
             (case* (list (empty-recv-buf* 8) (recv-buf-to-nat* 'buf))
                    (recv-buf-full?* 'buf))))
    ;; The type context
    (list (cons 'val 'natural)
          (cons 'buf 'recv-buffer)
          (cons 'in 'recv-channel))
    ;; The state
    (list (cons 'in (channel* #t #t))
          (cons 'val 0)
          (cons 'buf (buffer* 0 (list #f #f #f #f #f #f #f #f)))))
```

Yields the following state, with new values for `'in` and `'buf`:

```
(list (cons 'buf (buffer* 1 (list #t #f #f #f #f #f #f #f)))
      (cons 'in (channel* #f null))
      (cons 'in (channel* #t #t))
      (cons 'val 0)
      (cons 'buf (buffer* 0 (list #f #f #f #f #f #f #f #f))))
```

The underlying UNITY model of execution is iterative: programs sample their environment and take action when they can. UNITY programs do not *halt* in the traditional sense. Rather, a program that halts is one that infinitely iterates about a fixed point. This is why the interpreter returns the original state when none of the guards are satisfied. Recall that COMET implements a deter-

Type	Rosette Representation
byte	bitvector(8)
pin-input	boolean
pin-output	boolean

Table 4.4: UNITY data types and Rosette representations

ministic subset of UNITY. Interpreting statements where multiple guards are satisfied is unsupported. Any future work to support non-deterministic specifications would require ensuring that the interpreter satisfies UNITY's weak fairness constraint.

Program semantics

Complete program semantics are provided by initializer and assignment wrapper functions:

$$initialize : program \times state \rightarrow environment \tag{4.18}$$

$$assign : program \times environment \rightarrow environment \tag{4.19}$$

Initialization takes a program and starting state and interprets declare and initially sections, yielding an environment with a new type context and state. Assignment takes a program and environment, and interprets the assign section, yielding an environment with the existing type context and a new state.

4.3 Arduino

The subset of the Arduino language used for synthesis is presented here. This section covers data types, syntax and semantics, and the mechanization of the language in Rosette.

4.3.1 Arduino data types

Arduino data types and their representations in Rosette are given in Table 4.4. Input/output pins are represented with Rosette's boolean type. Byte values

$$
\begin{array}{rcl}
\mathit{byte} & \in & \textsc{Byte} \\
\mathit{var} & \in & \textsc{Variable} \\
\\
\mathit{expression} & \vdash & \mathit{byte} \mid \mathit{var} \mid \mathsf{HIGH} \mid \mathsf{LOW} \\
& \mid & \mathit{expr}\ \mathit{binop}\ \mathit{expr} \\
& \mid & \mathit{unop}\ \mathit{expr} \\
& \mid & \mathsf{read}(\mathit{pin}) \\
\\
\mathit{binop} & \vdash & \&\& \mid || \mid < \mid == \mid \& \mid \mid \mid \hat{} \mid << \mid >> \mid + \\
\mathit{unop} & \vdash & ! \mid \sim \\
\\
\mathit{pin} & \vdash & 0 \mid 1 \mid \ldots \mid 21
\end{array}
$$

Figure 4.5: Arduino expression syntax

are represented using Rosette's bitvector type, parameterized to eight bits. The low-order bit comes first and bitvector values are modelled as unsigned integers. Input pins, as their name suggests, allow for reading, while output pins allow for both reading and writing. The mechanized Arduino model does not type check during execution, but depends on the synthesizer to emit well-typed values, expressions, and statements.

4.3.2 Arduino expressions

Arduino expression syntax is presented in Figure 4.5. All binary expressions are $\mathit{byte} \times \mathit{byte} \to \mathit{byte}$. All unary expressions are $\mathit{byte} \to \mathit{byte}$. The read operator is $\mathit{pin} \to \mathit{byte}$.

Logical operators follow the C semantics for truth values: 0 for false, nonzero for true. Logical binary operators AND (&&), OR (||), less than (<), and equals (==) yield byte 1 if true and byte 0 if false. Arithmetic addition (+) is unsigned. Bitwise operators AND (&), OR (|), XOR (^), shift right (<<), and shift left (>>) are of the logical variety: zeroes are shifted in. Logical NOT (!) yields byte 1 if true and byte 0 if false. Bitwise NOT (~) inverts each bit in the bitvector.

```
    expr  ∈  Expression
     var  ∈  Variable

   block  ⊢  { statement }

statements  ⊢  list[stmt]

    stmt  ⊢  if ( expr ) block
          |  if ( expr ) block else block
          |  write( pin, expr );
          |  var = expr ;
          |  pinMode( pin , mode );
          |  byte var ;

     pin  ⊢  0 | 1 | ... | 21
 pinMode  ⊢  INPUT | OUTPUT
```

Figure 4.6: Arduino statement syntax

4.3.3 Arduino statements

Arduino statement syntax is presented in Figure 4.6. The Arduino's statement semantics reflects a sequential, control-flow oriented execution model. The state is modified by assignment (=) and setting output pin values (write). The type context is modified by declaration of variables (byte) and pin modes (pinMode). Type contexts for variables are *lexically scoped*: any variable declarations inside of a block are valid for interpretation only in the remainder of the block. Conditional if statements operate conventionally, with and without else blocks. Internally, COMET represents all if statements as if-then-else statements. Statements are interpreted sequentially, with type context or state changes taking effect immediately, with changes visible to the next statement.

4.3.4 Arduino programs

Arduino program syntax is presented in Figure 4.7. The declarations section contains variable declaration statements that remain in scope for both setup

$$
\begin{array}{rcl}
\mathit{block} & \in & \textsc{Block} \\
\mathit{declarations} & \in & \textsc{Statements} \\
 & & \\
\mathit{program} & \vdash & \mathit{declarations}\ \mathit{setup}\ \mathit{loop} \\
 & & \\
\mathit{setup} & \vdash & \mathsf{setup()}\ \mathit{block} \\
\mathit{loop} & \vdash & \mathsf{loop()}\ \mathit{block}
\end{array}
$$

Figure 4.7: Arduino program syntax

and loop blocks. Setup block statements are interpreted once at the beginning of program execution. Loop block statements are interpreted iteratively.

4.3.5 Mechanizing Arduino

The mechanizations of UNITY and Arduino in Rosette are similar in structure. Semantics for bitvector expressions use Rosette's bitvector library. The main difference with the UNITY model is statement interpretation: the UNITY interpreter makes one atomic step, whereas the Arduino interpreter applies each statement sequentially.

This is the statement interpreter for the Arduino model:

```
1  (define (interpret-stmt statement context state)
2      (match statement
3          ['() (environment* context state)]
4          [(cons head tail)
5              (match head
6                  [(byte* id)
7                      (interpret-stmt tail
8                                      (add-mapping id 'byte context)
9                                      state)]
10                 [(pin-mode* pin mode)
11                     (interpret-stmt tail
12                                     (add-mapping pin
13                                                  (case mode
14                                                      ['INPUT 'pin-input]
15                                                      ['OUTPUT 'pin-output
                                                            ↪ ])
16                                                  context)
```

```
                                  state)]
                [(write* pin expr)
                    (let ([val (evaluate-expr expr context state)])
                        (interpret-stmt tail
                                        context
                                        (add-mapping pin
                                                     (bitvector->bool val
                                                         ↪ )
                                                     state)))]
                [(:=* var expr)
                    (let ([val (evaluate-expr expr context state)])
                        (interpret-stmt tail
                                        context
                                        (add-mapping var val state)))]
                [(if* test left right)
                    (let* ([test-val (evaluate-expr test context state)]
                           [branch-to-take (if (bitvector->bool test-val)
                                               left
                                               right)]
                           [taken-env (interpret-stmt branch-to-take
                                                      context
                                                      state)]
                           [taken-cxt (environment*-context taken-env)]
                           [taken-st (environment*-state taken-env)])
                        (interpret-stmt tail taken-cxt taken-st))]))]))
```

The interpreter is written in a tail-recursive style, and processes lists of statements. If the list is empty, the interpreter returns an environment struct containing the type context and state with which it was called. Otherwise, pattern matching is applied to the first statement in the list. The first two matches cover byte variable and pin mode declarations. A recursive call is made with the type context changed with `add-mapping`. The third and fourth matches cover pin writes and assignments. The value of the expression `expr` is evaluated in the `let` expression, and a recursive call is made with the state changed with `add-mapping`. In the case of pin writes, the byte value is converted into a boolean value with Rosette's `bitvector->bool` function. The final match covers if statements with a then branch `left` and an else branch `right`. The sequence of evaluations in the `let*` expression show the evaluation of the test

Type	Rosette Representation
reg	boolean
reg[7:0]	bitvector(8)
wire	boolean
wire[7:0]	bitvector(8)

Table 4.5: Verilog data types and Rosette representations

expression, branch selection based on the test value, interpretation of the chosen branch, unpacking of the returned environment struct, and a recursive call with the latest type context and state.

4.4 Verilog

The subset of the Verilog language used for synthesis is presented here. This section covers data types, syntax and semantics, and the mechanization of the language in Rosette.

4.4.1 Verilog data types

Verilog data types and their representations in Rosette are given in Table 4.5. Every variable is either a read-write reg or a read-only wire type. By default, types are 1-bit wide. Wider orderings are specified with large and small indices: reg[7:0] is eight bits with the low-order bit first. Multiple-bit values are unsigned by default. Rosette booleans model 1-bit wide types with bitvectors modelling the rest. By default, variables are internal: external inputs and outputs are declared with input and output. Like the Arduino model, the mechanized Verilog model does not type check during execution, but depends on the synthesizer to emit well-typed values, expressions, and statements.

Note: as a HDL, standard Verilog operates according to 4-state logic: 0 (false), 1 (true), Z (disconnected/high impedance), and X (unknown). The model mechanized here only admits 2-state (true/false) logic, which is sufficient because COMET programs interact exclusively with other 2-state logic COMET programs, and the circuits are designed such that signal levels are 2-state.

```
byte   ∈  BITVECTOR
var    ∈  VARIABLE

event  ⊢  event or event
       |  posedge var
       |  negedge var

expression ⊢  byte | var | 0 | 1
           |  ( expr binop expr )
           |  unop ( expr )

binop  ⊢  && | || | == | < | & | | | ^ | << | >> | +
unop   ⊢  ! | ~
```

Figure 4.8: Verilog expression syntax

4.4.2 Verilog expressions

Verilog expression syntax is presented in Figure 4.8. Expression syntax and semantics are similar to the Arduino model, including the logical nature of bit shift operators << and >>. Event expressions posedge and negedge are used to guard *edge-triggered* statements. An edge-triggered statement executes at the edge transition of the named input, such as a clock signal.

4.4.3 Verilog statements

Verilog statement syntax is presented in Figure 4.6. Declarations are either for input/output *ports* or internal variables. The bitwidth syntax specifies a vector where the low-order bit comes first. Always statements execute a block of statements whenever an edge-triggered event occurs. If statements operate conventionally. Internally, COMET represents all if statements as if-then-else statements.

The nonblocking assignment (<=) operator deserves some attention. Standard Verilog defines two kinds of assignments: blocking (=) and nonblocking. Blocking assignments define an ordering of assignments, whereas nonblocking assignments specify no ordering and are allowed to occur concurrently. For ex-

```
    event  ∈  Event
     expr  ∈  Expression
      var  ∈  Variable
      nat  ∈  Natural

declaration  ⊢  port-decl | var-decl

port-decl  ⊢  input var-decl
           |  output var-decl

var-decl  ⊢  type var ;

     type  ⊢  reg
           |  reg[ nat : 0 ]
           |  wire
           |  wire[ nat : 0 ]

statement  ⊢  always @( event ) block
           |  if expr block
           |  if expr block else block
           |  var <= expr ;

    block  ⊢  begin list[statement] end
```

Figure 4.9: Verilog statement syntax

ample, in an always block, a nonblocking assignment `z <= x + y;` evaluates the right hand expression `w + x` immediately but the *actual assignment* to `z` only occurs at the end of the always block, along with all the other nonblocking assignments. Nonblocking assignments are used here because the goal is to synthesize concurrent Verilog programs. The use of nonblocking assignment means that transitions between pre-states and post-states during interpretation are observationally atomic [1].

[1] Whether or not two events can happen simultaneously is a matter for the physicists to consider. See future work.

$$
\begin{array}{rcl}
decl & \in & \textsc{Declaration} \\
var & \in & \textsc{Variable} \\
stmt & \in & \textsc{Statement} \\
prog & \vdash & \text{module } var \text{ (} vars \text{);} \\
 & & \quad list[decl] \\
 & & \quad list[stmt] \\
 & & \text{endmodule} \\
 & & \\
vars & \vdash & var \text{ , } vars \\
 & | & var
\end{array}
$$

Figure 4.10: Verilog program syntax

4.4.4 Verilog programs

Verilog program syntax is presented in Figure 4.10. The module is the top-level structure. The signature of the module lists variables and ports, i.e., inputs and outputs. A module contains port and variable declarations, followed by a list of statements.

At initialization time, declarations are read into the type context. Establishing initial state is handled by the always statement. Using the example from Figure 3.3:

```
always @(posedge clock or posedge reset)
  begin
    if (reset)
      begin
        out_req <= out_ack;
      end
    ...
  end
```

The first if statement in the always block checks for the reset signal, and assigns values appropriately. Initialization then requires populating the type context, then interpreting the always block with reset asserted.

4.4.5 Mechanizing Verilog

The mechanizations of UNITY, Arduino and Verilog in Rosette are similar in structure. Semantics for bitvector expressions use Rosette's bitvector library. The statement interpreter for the Verilog model combines elements of imperative control flow, like Arduino, with concurrent assignment, like UNITY:

```
(define (interpret-stmts statements state)
    (define (helper stmts next-state)
        (match stmts
            ['() next-state]
            [(cons stmt tail)
                (match stmt
                    [(always* guard branch)
                        (let* ([guard-val (evaluate-expr guard state)]
                               [always-state (if guard-val
                                                 (helper branch
                                                     ↪ next-state)
                                                 next-state)])
                            (helper tail always-state))]
                    [(if* guard branch-t branch-f)
                        (let* ([guard-val (evaluate-expr guard state)]
                               [branch-chosen (if guard-val branch-t
                                   ↪ branch-f)]
                               [if-state (helper branch-chosen next-state
                                   ↪ )])
                            (helper tail if-state))]
                    [(<=* l r)
                        (let* ([r-val (evaluate-expr r state)])
                            (helper tail
                                    (add-mapping l r-val next-state)))])
                                        ↪ ]))

    (helper statements state))
```

All processing is done by the tail-recursive helper function, that maintains the `next-state` state. All value evaluations are made with regards to the pre-state `state` from the outer scope. For always statements, the event guard is evaluated. If the guard is satisfied, `always-state` is the result of interpreting the block. Otherwise, `always-state` is unchanged. The helper then recur-

sively continues with `always-state`. If and nonblocking assignment statements are interpreted similarly.

CHAPTER 5

Defining Correctness

At a high level, COMET synthesizes programs whose traces satisfy a correctness constraint with regards to a specification trace. This chapter provides motivation and introduces the two parts of the correctness constraint: the *refinement mapping* and the *simulation relation*.

5.1 A motivating example for trace correctness

Solver-aided synthesis with Rosette requires a correctness constraint, a boolean function that compares an implementation against a specification. Consider the example UNITY inverter FIFO program from Chapter 3:

```
1  program INV-FIFO
2    declare
3      in: recv-channel
4      out: send-channel
5
6    initially
7      out := empty
8
9    assign
10     in, out := empty, full(not(value(in)))
11       if (full?(in) and empty?(out))
```

Because the language models differ in their data types, representations, and execution models, it is not possible to directly compare between high-level UNITY traces and low-level Arduino or Verilog traces. We require a method

UNITY	Arduino	Verilog
x: boolean	byte x	reg x
x: natural	byte x	reg[7:0] x
x: recv-buffer	byte x_cursor	reg[7:0] x_cursor
	byte x_val	reg[7:0] x_val
x: send-buffer	byte x_cursor	reg[7:0] x_cursor
	byte x_val	reg[7:0] x_val
x: recv-channel	pinMode(pin_{req}, pin-input)	input wire x_req
	pinMode(pin_{ack}, pin-output)	output reg x_ack
	pinMode(pin_{val}, pin-input)	input wire x_val
x: send-channel	pinMode(pin_{req}, pin-output)	output wire x_req
	pinMode(pin_{ack}, pin-input)	input reg x_ack
	pinMode(pin_{val}, pin-output)	output wire x_val

Table 5.1: UNITY type context translation scheme

UNITY type	Arduino variable(s)	UNITY value expression
boolean	x	`(bitvector->bool x)`
natural	x	`(bitvector->natural x)`

recv/send-buffer	x_cursor
	x_val
		`(buffer*`
		`  (bitvector->natural x_cursor)`
		`  (map bitvector->bool`
		`  (bitvector->bits x_val)))`

recv/send-channel	pin_{req}
	pin_{ack}
	pin_{val}
		`(if (eq?` pin_{req} pin_{ack}`)`
		`  (channel* #f null)`
		`  (channel* #t` pin_{val}`))`

Table 5.2: Arduino to UNITY value translation scheme

for translating type contexts and states between models as part of our correctness constraint.

UNITY type	Verilog variable(s)	UNITY value expression
boolean	x	x
natural	x	(bitvector->natural x)
recv/send-buffer	x_cursor x_val	(buffer* (bitvector->natural x_cursor) (map bitvector->bool (bitvector->bits x_val)))
recv/send-channel	x_req x_ack x_val	(if (eq? x_req x_ack) (channel* #f null) (channel* #t x_val))

Table 5.3: Verilog to UNITY value translation scheme

5.1.1 Type contexts and data types

The example program has the type context:

$$\{(\mathtt{in} \mapsto \text{recv-channel}), (\mathtt{out} \mapsto \text{send-channel})\} \tag{5.1}$$

UNITY channels are abstract types and neither Arduino nor Verilog support them natively. They are implemented in COMET using types native to each low-level language.

For example, COMET implements channels at the low-level using a *non-return-to-zero* protocol with three boolean communication lines: *request* from sender to receiver, *acknowledge* from receiver to sender, and *value* from sender to receiver. Recall that a UNITY channel is full or empty. A full channel contains a value that the reader can access. Only the receiver can modify a full channel: by emptying it. Conversely, an empty channel has no value, so it is inaccessible to the reader. Only the sender can modify an empty channel: by filling it. When the request and acknowledge lines are equal, $\langle \mathtt{req} = \mathtt{true}, \mathtt{ack} = \mathtt{true} \rangle$, $\langle \mathtt{req} = \mathtt{false}, \mathtt{ack} = \mathtt{false} \rangle$: the channel is empty. When they differ, $\langle \mathtt{req} = \mathtt{true}, \mathtt{ack} = \mathtt{false} \rangle$, $\langle \mathtt{req} = \mathtt{false}, \mathtt{ack} = \mathtt{true} \rangle$: the channel is full. Signalling is efficient: a sender toggles the request line indicate that an empty channel is

full, and a receiver toggles the acknowledge line to indicate that the contents of a full channel have been read.

A *translation scheme*, in Table 5.1, formalizes the first part of the implementation: the relation between types. The scheme is used to translate UNITY contexts to their low-level equivalents. Each channel is represented by three I/O Arduino pins or Verilog ports. Variables $\mathtt{pin}_{req} \ldots \mathtt{pin}_{val}$ represent available Arduino pin numbers. For example, translating the UNITY type context in equation 5.1 to Arduino yields:

$$\{(\mathtt{in}_{req} \mapsto \text{pin-input}), (\mathtt{in}_{ack} \mapsto \text{pin-output}), (\mathtt{in}_{val} \mapsto \text{pin-input}),$$
$$(\mathtt{out}_{req} \mapsto \text{pin-output}), (\mathtt{out}_{ack} \mapsto \text{pin-input}), (\mathtt{out}_{val} \mapsto \text{pin-output})\} \quad (5.2)$$

A *value translation scheme*, in Tables 5.2 and 5.3 for Arduino and Verilog respectively formalizes the second part of the implementation: the relation between values. For example, the following expression yields the corresponding UNITY value for the $\mathtt{in}$ channel from an Arduino state:

```
1 (if (eq? in_req in_ack)
2     (channel* #f null)
3     (channel* #t in_val))
```

Recall that UNITY channels are represented as structs with two fields: validity for full/empty and value for the contents of the channel. The direction of the value mapping from low-level to UNITY is motivated by the desire to prove safety properties for intermediate low-level states, covered next.

5.1.2 States

Consider this UNITY pre-state:

$$\{(\mathtt{in} \mapsto \text{channel*(\#t, \#f)}), (\mathtt{out} \mapsto \text{channel*(\#f, null)})\} \quad (5.3)$$

Interpreting the example program over this pre-state yields the following UNITY post-state, with the value on $\mathtt{out}$ inverted:

$$\{(\mathtt{in} \mapsto \mathsf{channel^*(\#t,\ \#f)}), (\mathtt{out} \mapsto \mathsf{channel^*(\#f,\ null)}),$$
$$\boxed{(\mathtt{out} \mapsto \mathsf{channel^*(\#t,\ \#t)}), (\mathtt{in} \mapsto \mathsf{channel^*(\#f,\ null)})}\}$$ (5.4)

These two states represent a simple specification. Note that the UNITY semantics for assignment specify an instantaneous transition between pre- and post-states. There are no *intermediate* UNITY states such as the following:

$$\{(\mathtt{in} \mapsto \mathsf{channel^*(\#t,\ \#f)}), (\mathtt{out} \mapsto \mathsf{channel^*(\#f,\ null)}),$$
$$\boxed{(\mathtt{out} \mapsto \mathsf{channel^*(\#t,\ \#t)})}\}$$ (5.5)

Consider this Arduino pre-state, compliant with the translated type context in equation 5.2:

$$\{(\mathtt{in}_{req} \mapsto \mathsf{\#f}), (\mathtt{in}_{ack} \mapsto \mathsf{\#t}), (\mathtt{in}_{val} \mapsto \mathsf{\#f}),$$
$$(\mathtt{out}_{req} \mapsto \mathsf{\#f}), (\mathtt{out}_{ack} \mapsto \mathsf{\#f}), (\mathtt{out}_{val} \mapsto \mathsf{\#f})\}$$ (5.6)

The values here map to the same ones in the UNITY pre-state: $(\mathtt{in} \mapsto \mathsf{channel(\#t,\ \#f)})$ and $(\mathtt{out} \mapsto \mathsf{channel^*(\#f,\ null)})$. Next, consider a problematic Arduino post-state that toggles $\mathtt{out}_{req}$ *before* writing to $\mathtt{out}_{val}$:

$$\{(\mathtt{in}_{req} \mapsto \mathsf{\#f}), (\mathtt{in}_{ack} \mapsto \mathsf{\#t}), (\mathtt{in}_{val} \mapsto \mathsf{\#f}),$$
$$(\mathtt{out}_{req} \mapsto \mathsf{\#f}), (\mathtt{out}_{ack} \mapsto \mathsf{\#f}), (\mathtt{out}_{val} \mapsto \mathsf{\#f}),$$
$$\boxed{(\mathtt{out}_{req} \mapsto \mathsf{\#t}), (\mathtt{out}_{val} \mapsto \mathsf{\#t}), (\mathtt{in}_{ack} \mapsto \mathsf{\#f})}\}$$ (5.7)

The values in the problematic post-state map to the same ones in the UNITY post-state: $(\mathtt{out} \mapsto \mathsf{channel^*(\#t,\ \#t)})$ and $(\mathtt{in} \mapsto \mathsf{channel^*(\#f,\ null)})$, so it seems like this post-state satisfies the specification. The problem lies in the intermediate states. While the UNITY model moves between pre- and post-states with a single atomic transition, the Arduino model executes sequentially, meaning

that the following intermediate state *is* observable:

$$\begin{aligned}&\{(\mathtt{in}_{req} \mapsto \#\mathsf{f}), (\mathtt{in}_{ack} \mapsto \#\mathsf{t}), (\mathtt{in}_{val} \mapsto \#\mathsf{f}),\\ &\ (\mathtt{out}_{req} \mapsto \#\mathsf{f}), (\mathtt{out}_{ack} \mapsto \#\mathsf{f}), (\mathtt{out}_{val} \mapsto \#\mathsf{f}),\\ &\ \boxed{(\mathtt{out}_{req} \mapsto \#\mathsf{t})}\}\end{aligned} \tag{5.8}$$

Values in this intermediate state map to UNITY values $(\mathtt{in} \mapsto \mathsf{channel}^*(\#\mathsf{t}, \#\mathsf{f}))$ which matches the pre-state but $(\mathtt{out} \mapsto \mathsf{channel}^*(\#\mathsf{f}, \#\mathsf{f}))$ is a value not described by the specification.

Correctness between traces requires that a low-level pre-state maps to a UNITY pre-state, a low-level post-state maps to a UNITY post-state, and that, for sequential executions, intermediate states map each *externally-visible* value to a UNITY pre- or post-state value. Channels are the only externally-visible values used in this thesis. A more formal definition of correctness follows.

5.2 Correctness by refinement simulation

The low-level programs that COMET synthesizes are *refinements* of their UNITY specifications. This means that the specification and implementation are related by a *refinement simulation relation* $\leqslant_R$ as introduced by Lynch and Vaandrager [26]. Before we define the relation, let us consider what we mean by *simulation*.

Informally, a *simulator* is some system that faithfully exhibits the behaviour of another system while abstracting away irrelevant details. A flight simulator used for pilot training exhibits the behaviour of an airplane, including known failure modes. We say that a simulator is *sufficient* if it emulates the behaviours that we consider important. We could also say that a simulator is *lacking* if there are important behaviours not accounted for by the simulator. For example, pilots may want to practice scenarios where the landing gear gets stuck halfway through deployment. A simulator that only stores landing gear state with a boolean value would be unable to emulate this behaviour. This notion of simulation is an elegant way to relate an abstract specification with a low-level implementation.

Lynch and Vaandrager's refinement simulation relation $\leqslant_R$ is is defined over state machines. Specification $S = \langle \textsc{Spec}, P_s, \textsc{Start}_s \rangle$ is a state machine over a set of states $\textsc{Spec}$, where $P_s : \textsc{Spec} \to \textsc{Spec}$ is the state transition function and $\textsc{Start}_s \subseteq \textsc{Spec}$ is the set of start states. Implementation $I = \langle \textsc{Impl}, P_i, \textsc{Start}_i \rangle$ is a state machine over a set of states $\textsc{Impl}$, where $P_i : \textsc{Impl} \to \textsc{Impl}$ is the state transition function and $\textsc{Start}_i \subseteq \textsc{Impl}$ is the set of start states. We say that $S \leqslant_R I$, or S *simulates* I if there exists a *refinement mapping* function $r : \textsc{Impl} \to \textsc{Spec}$ such that for all states in $\textsc{Start}_i$ their refinement mapping is in $\textsc{Start}_s$, and that for any *externally-visible* i.e., channel state transition in P_i, the refinement mapping of the pre- and post-states is permitted by P_s, either by a state transition or by state repetition, also known as *stuttering equivalence*, denoted as $\equiv_r$:

$$\begin{aligned} S \leqslant_R I \iff & (\forall s_i \in \textsc{Start}_i . r(s_i) \in \textsc{Start}_s) \wedge \\ & \forall \langle s_{pre}, s_{post} \rangle \in P_i . \\ & (\langle r(s_{pre}), r(s_{post}) \rangle \in P_s \vee \\ & r(s_{pre}) \equiv_r r(s_{post})) \end{aligned} \tag{5.9}$$

A simple positive and negative example of state repetition follows:

$$\{(x \mapsto 1), (x \mapsto 2), (x \mapsto 2), (x \mapsto 3)\} \equiv_r \{(x \mapsto 1), (x \mapsto 2), (x \mapsto 3)\} \tag{5.10}$$

$$\{(x \mapsto 1), (x \mapsto 2), (x \mapsto 3), (x \mapsto 2)\} \not\equiv_r \{(x \mapsto 1), (x \mapsto 2), (x \mapsto 3)\} \tag{5.11}$$

A *simulation diagram* illustrates this relation, where $\xrightarrow{P_s}{}^{?}$ indicates that the simulation makes a transition in P_s or stutters:

$$\begin{array}{ccc} r(s_{pre}) & \xrightarrow{P_s}{}^{?} & r(s_{post}) \\ \uparrow_r & & \uparrow_r \\ s_{pre} & \xrightarrow{P_i} & s_{post} \end{array} \tag{5.12}$$

To illustrate the relation, consider the example UNITY and Arduino pre-states with r following the channel value mapping given in Table 5.2 on page 46:

$$\begin{aligned} s_{pre} = \{&(in_{req} \mapsto \#f), (in_{ack} \mapsto \#t), (in_{val} \mapsto \#f), \\ &(out_{req} \mapsto \#f), (out_{ack} \mapsto \#f), (out_{val} \mapsto \#f)\} \end{aligned} \tag{5.13}$$

$$r(s_{pre}) = \{(in \mapsto \text{channel*(\#t, \#f)}), (out \mapsto \text{channel*(\#f, null)})\} \tag{5.14}$$

Recall that the UNITY execution model transitions instantaneously between pre- and post-states. This means that the ordering of UNITY post-state assignments *does not* specify an assignment ordering. As seen before, the proposed Arduino post-state exposed an external state not permitted by the UNITY pre- or post-states. Let's continue with a correct intermediate Arduino state s^0_{post}, compared to the the problematic state s_{bug}:

$$\begin{aligned} s^0_{post} = \{&(in_{req} \mapsto \#f), (in_{ack} \mapsto \#t), (in_{val} \mapsto \#f), \\ &(out_{req} \mapsto \#f), (out_{ack} \mapsto \#f), (out_{val} \mapsto \#f), \\ &\boxed{(out_{val} \mapsto \#t)}\} \end{aligned} \tag{5.15}$$

$$s_{bug} = \{\ldots (out_{req} \mapsto \#t)\} \tag{5.16}$$

The refinement mapping of the buggy state, $r(s_{bug})$ violates the simulation relation because the mapped value for out is not permitted by P_s nor is it a stuttering step:

$$\begin{aligned} r(s_{bug}) = \{&(in \mapsto \text{channel*(\#t, \#f)}), (out \mapsto \text{channel*(\#f, null)}), \\ &\boxed{(out \mapsto \text{channel*(\#t, \#f)})}\} \end{aligned} \tag{5.17}$$

On the other hand, the refinement mapping, $r(s^0_{post})$ is permitted by the specification because the mapped value for out is unchanged from the pre-state:

$$\begin{aligned} r(s^0_{post}) = \{&(in \mapsto \text{channel*(\#t, \#f)}), (out \mapsto \text{channel*(\#f, null)}), \\ &\boxed{(out \mapsto \text{channel*(\#f, null)})}\} \end{aligned} \tag{5.18}$$

We continue with the assignment to out_{req} to yield a new state s^1_{post}:

$$\begin{aligned} s^1_{post} = \{&(\mathrm{in}_{req} \mapsto \#\mathsf{f}), (\mathrm{in}_{ack} \mapsto \#\mathsf{t}), (\mathrm{in}_{val} \mapsto \#\mathsf{f}), \\ &(\mathrm{out}_{req} \mapsto \#\mathsf{f}), (\mathrm{out}_{ack} \mapsto \#\mathsf{f}), (\mathrm{out}_{val} \mapsto \#\mathsf{f}), \\ &\boxed{(\mathrm{out}_{val} \mapsto \#\mathsf{t}), (\mathrm{out}_{req} \mapsto \#\mathsf{t})}\} \end{aligned} \tag{5.19}$$

The new refinement mapping, $r(s^1_{post})$ is permitted by the specification because the mapped value for out matches the post-state:

$$\begin{aligned} r(s^1_{post}) = \{&(\mathrm{in} \mapsto \mathsf{channel^*(\#t, \#f)}), (\mathrm{out} \mapsto \mathsf{channel^*(\#f, null)}), \\ &\boxed{(\mathrm{out} \mapsto \mathsf{channel^*(\#t, \#t)})}\} \end{aligned} \tag{5.20}$$

We finish with the final Arduino post-state s^2_{post} and mapped UNITY post-state $r(s^2_{post})$, along with the post-state from the specification S_{spec}:

$$\begin{aligned} s^2_{post} = \{&(\mathrm{in}_{req} \mapsto \#\mathsf{f}), (\mathrm{in}_{ack} \mapsto \#\mathsf{t}), (\mathrm{in}_{val} \mapsto \#\mathsf{f}), \\ &(\mathrm{out}_{req} \mapsto \#\mathsf{f}), (\mathrm{out}_{ack} \mapsto \#\mathsf{f}), (\mathrm{out}_{val} \mapsto \#\mathsf{f}), \\ &\boxed{(\mathrm{out}_{val} \mapsto \#\mathsf{t}), (\mathrm{out}_{req} \mapsto \#\mathsf{t}), (\mathrm{in}_{ack} \mapsto \#\mathsf{f})}\} \end{aligned} \tag{5.21}$$

$$\begin{aligned} r(s^2_{post}) = \{&(\mathrm{in} \mapsto \mathsf{channel^*(\#t, \#f)}), (\mathrm{out} \mapsto \mathsf{channel^*(\#f, null)}), \\ &\boxed{(\mathrm{out} \mapsto \mathsf{channel^*(\#f, null)}), (\mathrm{out} \mapsto \mathsf{channel^*(\#t, \#t)}),} \\ &\boxed{(\mathrm{in} \mapsto \mathsf{channel^*(\#f, null)})}\} \end{aligned} \tag{5.22}$$

$$\begin{aligned} s_{spec} = \{&(\mathrm{in} \mapsto \mathsf{channel^*(\#t, \#f)}), (\mathrm{out} \mapsto \mathsf{channel^*(\#f, null)}), \\ &\boxed{(\mathrm{out} \mapsto \mathsf{channel^*(\#t, \#t)}), (\mathrm{in} \mapsto \mathsf{channel^*(\#f, null)}),}\} \end{aligned} \tag{5.23}$$

The remainder of this chapter covers the mechanization of the refinement mapping and the refinement simulation relation.

5.3 Mechanizing the refinement mapping

The refinement mapping is constructed by traversing the specification type context. Depending on the type of the UNITY variable, one or more implementation variables are allocated, their types determined by the scheme in Table 5.1.

At the same time a function from an implementation state to a value for the UNITY variable is constructed. This function extracts the relevant values from the implementation state and applies a translation according to a scheme. For Arduino, the scheme is listed in Table 5.2. The scheme for Verilog is listed in Table 5.3. At the end of the traversal, the individual value translation functions are combined into a single state to state function.

For Arduino booleans, the translation follows C semantics: non-zero for true, zero for false. Verilog booleans require no translation. For naturals, bitvectors are interpreted as unsigned integers. For buffers, the complex buffer struct is created by translating the cursor bitvector as a natural, and splitting the value bitvector into a list of single-bit bitvectors, then translating each one as a boolean. Channels have been covered previously.

5.4 Mechanizing the refinement simulation relation

Recall that our interpretation of concurrency allows us to consider the value assignments in the UNITY post-state in any order. With this in mind, the relation function tests each variable in isolation. For internal variables such as booleans, naturals, and buffers, a simple pre- and post-state match is all that is required. For external variables such as channels, intermediate states must be considered. Each external variable is checked to make sure its mapped value corresponds to either the UNITY pre- or post-state, and that once transitioned to the post-state, that it stays there. This *monotonicity* property is implied by the refinement simulation relation. The relation function does this by testing the implementation post-state, then repeatedly removing single value mappings from the end of the state and testing until the trimmed state matches the implementation pre-state.

CHAPTER 6

Implementation of COMET

Theoretically, a procedure *could* search a constrained (e.g., by maximum program length or expression depth) space of low-level programs and, using the refinement mapping and simulation relation constraints from Chapter 5, find a program that satisfies a UNITY specification. Unfortunately, such one-step synthesis is intractable for all but the most trivial specifications. This chapter discusses why this is the case and introduces the COMET approach to synthesizing non-trivial programs.

6.1 The limits of one-step synthesis

The synthesis example presented in Chapter 3, page 18 is a one-step example: the synthesizer searches over the space of *whole* addition or subtraction expressions to find one equivalent to multiplying a number by 3. While succinct, this approach scales poorly when applied to non-trivial UNITY specifications. To understand why, consider the refinement simulation diagram from Chapter 5 that defines the refinement simulation relation:

$$\begin{array}{ccc} r(s_{pre}) & \xrightarrow{P_s}{}^{?} & r(s_{post}) \\ \uparrow_r & & \uparrow_r \\ s_{pre} & \xrightarrow{P_i} & s_{post} \end{array} \tag{6.1}$$

Remember that this means if implementation program, P_i, makes an externally-visible state transition from s_{pre} to s_{post} then specification program P_s must

allow a transition from $r(s_{pre})$ to $r(s_{post})$ where r is the refinement mapping function, or $r(s_{pre}) \equiv_r r(s_{post})$ where $\equiv_r$ is state repetition, also known as stuttering equivalence.

Initial examples of expression evaluation, statement interpretation, refinement mapping, and simulation relations were all over concrete states and values. To express and evaluate programs over all possible states, synthesis requires *symbolic states*. All the primary functions mentioned above must be symbolically executed, which requires a different accounting for complexity.

Let us refine the simulation diagram to reflect the symbolic executions that occur during one-step synthesis:

$$\begin{array}{ccc} r(s_{pre}) & \xrightarrow{P_s} & (r(s_{pre}))_{post} \\ & & \leqslant_R \\ \uparrow_r & & r(s_{post}) \\ & & \uparrow_r \\ s_{pre} & \xrightarrow{P_i} & s_{post} \end{array} \tag{6.2}$$

This augmented simulation diagram shows two execution paths starting from the low-level pre-state s_{pre}. The UNITY path maps to $r(s_{pre})$, then interprets the specification to yield $(r(s_{pre}))_{post}$. The low-level path interprets the *search space* of low-level programs to yield s_{post}, then maps to $r(s_{post})$. Unfortunately, $(r(s_{pre}))_{post}$ and $r(s_{post})$ are so complex that solving for the correctness constraint $\leqslant_R$ becomes intractable.

The complexity of $(r(s_{pre}))_{post}$ arises from conditionals in the mapping function and from the guards in the specification. As the number of guarded assignments in a UNITY specification grows, so do the number of alternate assignments in $(r(s_{pre}))_{post}$. In addition, recall that Rosette encodes data structures, such as symbolic states, in *symbolic unions*, sets of $\langle guard, value \rangle$ pairs corresponding to the conditional cases encountered during symbolic execution. By default, Rosette minimizes the number of $\langle guard, value \rangle$ pairs in a symbolic union by *state merging*. By analogy, if a symbolic union is a C switch statement:

```
switch (test) {
  case true:
    x = a;
    break;
  case false:
    x = b;
    break;
}
```

State merging collapses cases into a single statement using the ternary operator: `x = test ? a : b`. The end result is a reduction in $\langle guard, value \rangle$ pairs but an increase of the complexity of each merged $value$. In the best case, state merging avoids the state explosion that comes from lengthy control flow, but in the worst case, it obscures the structure of the original specification and harms performance due to complex merged constraints.

The complexity of $r(s_{post})$ arises from the size of the search space and from conditionals in the mapping function. In the worst case, the search space grows exponentially with expression depth and statement count. Note that in general, more complex UNITY synthesis targets necessitate more complex low-level expressions and statements, which in turn require larger search spaces. In short, $r(s_{post})$ quickly blows up.

6.2 COMET overview

COMET synthesizes programs by a multi-stage process that breaks the UNITY post-state synthesis target into tractable pieces. A synthesis workflow is presented in Figure 6.1. Our UNITY interpreter uses *symbolic reflection* to control state merging, yielding *guarded traces* that reflect the cases written into the specification. Like the name suggests, each guarded trace consists of a boolean expression *guarding* the case and its corresponding trace. Each guard is a target for *guard synthesis*, which uses *recursive expression synthesis* to find a low-level expression equivalent to the guard. Each trace is a target for *trace value synthesis*, which targets each value in the target trace independently, yielding a set of partial traces. Each partial trace is a target for *statement fragment synthesis*, and the fragments are assembled into low-level statements that satisfy the guarded

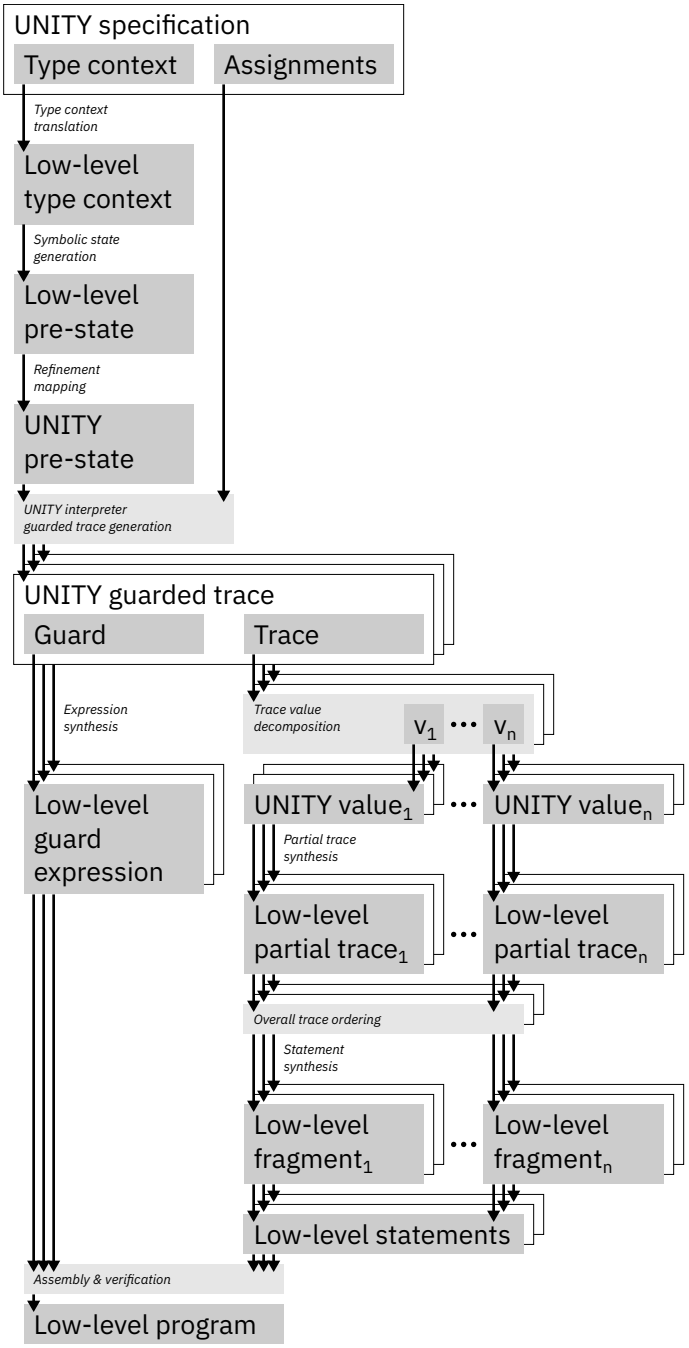

Figure 6.1: COMET synthesis workflow

```
1  program INV-FIFO
2    declare
3      in: recv-channel
4      out: send-channel
5
6    initially
7      out := empty
8
9    assign
10     in, out := empty, full(not(value(in)))
11       if (full?(in) and empty?(out))
```

Figure 6.2: A single-value boolean FIFO inverter in UNITY

trace. Finally, the synthesized low-level guard expressions and statements from each guarded trace are assembled into a whole program, which is then verified against the specification.

To motivate understanding, we continue with the UNITY inverter FIFO program from Chapter 3, presented again in Figure 6.2, and synthesize an Arduino implementation.

6.3 Preliminary steps

Recall that the simulation diagrams use the low-level start-state s_{pre} as the starting point. COMET generates an Arduino pre-state, starting with the inverter specification type context:

$$\{(\mathtt{in} \mapsto \mathsf{recv\text{-}channel}), (\mathtt{out} \mapsto \mathsf{send\text{-}channel})\} \tag{6.3}$$

Using the type context translation scheme introduced in Chapter 5, COMET derives an Arduino type context:

$$\begin{aligned}&\{(\mathtt{in}_{req} \mapsto \mathsf{pin\text{-}input}), (\mathtt{in}_{ack} \mapsto \mathsf{pin\text{-}output}), (\mathtt{in}_{val} \mapsto \mathsf{pin\text{-}input}),\\ &\ (\mathtt{out}_{req} \mapsto \mathsf{pin\text{-}output}), (\mathtt{out}_{ack} \mapsto \mathsf{pin\text{-}input}), (\mathtt{out}_{val} \mapsto \mathsf{pin\text{-}output})\}\end{aligned} \tag{6.4}$$

The final step to generating the Arduino pre-state requires creating symbolic values for each variable declared in the type context. Each Arduino pin is mod-

elled with a Rosette boolean. For example, the following declares a symbolic boolean variable `in_req`:

```
1 (define-symbolic in_req boolean?)
```

where further references to `in_req` refer to a symbolic boolean. In this way, COMET builds a low-level pre-state where each value is symbolic:

$$\{(\mathrm{in}_{req} \mapsto \mathrm{in\_req}), (\mathrm{in}_{ack} \mapsto \mathrm{in\_ack}), (\mathrm{in}_{val} \mapsto \mathrm{in\_val}),$$
$$(\mathrm{out}_{req} \mapsto \mathrm{out\_req}), (\mathrm{out}_{ack} \mapsto \mathrm{out\_ack}), (\mathrm{out}_{val} \mapsto \mathrm{out\_val})\} \tag{6.5}$$

Mapping to a UNITY symbolic pre-state

The final preliminary step is mapping the Arduino symbolic pre-state to a UNITY symbolic pre-state. COMET uses the refinement mapping introduced in Chapter 5 to derive a mapped UNITY symbolic pre-state:

$\{(\mathrm{in} \mapsto$ `(channel*`
`(! (<=> in_req in_ack))` (6.6)
`{`
`[(<=> in_req in_ack) null]` (6.7)
`[(! (<=> in_req in_ack)) in_val]` (6.8)
`})),`
$(\mathrm{out} \mapsto$ `(channel*`
`(! (<=> out_req out_ack))` (6.9)
`{`
`[(<=> out_req out_ack) null]` (6.10)
`[(! (<=> out_req out_ack)) out_val]` (6.11)
`})`$)\}$

This is an example of state merging.

Recall the channel translation scheme:

```
1 (if (eq? pin_req pin_ack)
2     (channel* #f null)
3     (channel* #t pin_val))
```

The Rosette SVM attempts to minimize path conditions by merging the constraints for each of the `channel*` structs from the branches of the if expression. Boolean equality is interpreted as bi-implication: `<=>`. The channel validity fields at 6.6 and 6.9 are read as not-equals. The channel value fields are symbolic unions containing two guard-value pairs. The first guard at 6.7 or 6.10 is satisfied if request and acknowledge are equal, and the value is `null`. Otherwise the second guard at 6.8 or 6.11 is satisfied and the value is `in_val` or `out_val` respectively. State merging for the initial state is useful: COMET can work with symbolic unions that can be further simplified with assumptions e.g., in the case where `full?(in)`, COMET can assume `(! (<=> in_req in_ack))` and reduce the value of `in` to `(channel* #t in_val)`.

6.4 Guarded trace synthesis

The inverter specification has one case in the assign section: empty the input and fill the output with the inverted input when the input is full and the output is empty. There is also one implicit no-op case if the case isn't satisfied. A naïve symbolic interpretation would yield a post-state with intermingled symbolic values from the assign case and the no-op case. For a more complex specification with many cases, state merging for the post-state yields a symbolic union with a minimal number of highly-complex guard and value constraints. Because the UNITY model's interpreter executes a single step, the number of path conditions *cannot* explode. Instead, what we desire are guard and value constraints as simple as possible.

The COMET interpreter for UNITY processes each case separately and encloses each resulting post-state in an *opaque* struct. This prevents the Rosette SVM from examining the enclosed state values and merging them. For the inverter, the interpreter returns a symbolic union with two guard-value pairs, reflecting the assign case and the no-op case. A guarded trace is made for each

$\langle guard, value \rangle$ pair allowing each case to be synthesized independently. For example, here is the guarded trace for the assign case:

```
(guarded-trace
  (&& (! (<=> in_req in_ack))
    (<=> out_req out_ack))                                    (6.12)
  {(in ↦ (channel*
           (! (<=> in_req in_ack))
           {
             [(<=> in_req in_ack) null]
             [(! (<=> in_req in_ack)) in_val]
           })),
  (out ↦ (channel*
           (! (<=> out_req out_ack))
           {
             [(<=> out_req out_ack) null]
             [(! (<=> out_req out_ack)) out_val]
           })),
  (in ↦ (channel* #f null)),                                  (6.13)
  (out ↦ (channel* #t (! in_val)))} )                         (6.14)
```

The expression at 6.12 is the guard: `(full?(in) and empty?(out))`. The remainder of the guarded trace is the post-state: note the new assignments to in at 6.13 and out at 6.14. Combined with the guarded trace generated from the `initially` section, there are two guarded traces to target for further synthesis:

1. The case from the assign section
2. The initial state, with a trivial guard: true

6.4.1 Guard synthesis

Guard synthesis searches for an equivalent boolean low-level expression to the guard in a guarded trace. Equivalence means that evaluating the low-level expression with the low-level pre-state yields the same boolean value as evaluating the UNITY guard with the UNITY pre-state. Recall that the states refer to the same symbolic values: the UNITY pre-state is the refinement mapping of the low-level pre-state.

The synthesis search space is a generated *choose tree*: a symbolic expression over syntax. To recap, a choose expression in Rosette allows the programmer to encode alternatives, and a choose tree is a tree of choose expressions. During synthesis, the solver searches for the alternative that satisfies the correctness constraint. For example, the expression `(choose* 'HIGH 'LOW)` encodes the choice between two Racket symbols modelling the Arduino reserved words `HIGH` and `LOW`. It evaluates to a symbolic union: `{ [x?$0 HIGH] [(! x?$0) LOW] }`. The variable `x?$0` is a unique boolean variable allocated by the Rosette runtime. Syntax is modelled with structs, so the alternative between logical AND, logical OR, and logical XOR is expressed as `((choose* &&* ||* ^*) lt rt)` where `lt` and `rt` represent arguments such as concrete values or other choose trees. The evaluation of this example is similar but slightly more involved:

```
{  [x?$1 (&&* lt rt)]
   [(&& x?$2 (! x?$1)) (||* lt rt)]
   [(&& (! x?$2) (! x?$1)) (^* lt rt)]  }        (6.15)
```

Note the additional unique variables `x?$1` and `x?$2` to encode the choice between the three alternatives.

The generator function for a choose tree over the expression syntax is recursive and generates choose trees of an arbitrary depth. For the base case, the choose tree is a choice over the low-level *terminals*: literal booleans, literal bitvectors, variable names, and Arduino `read(pin)` expressions, where `pin` is a pin identifier. For the recursive case, the choose tree is a choice over the terminals, binary expressions with two recursive choose trees, or unary ex-

pressions with one recursive choose tree. Note that allowing terminals in the recursive case allows for asymmetrical syntax trees: the depth parameter is an upper bound only.

The synthesis approach we take is iterative: we start with a choose tree of depth zero, then increase the depth if synthesis is unsuccessful. The complexity of choose trees is exponential with regards to their depth, so expression synthesis performance declines rapidly as choose trees get deeper. Unfortunately, this places limitations on the complexity of UNITY guards we are able to synthesize. To overcome this limitation, we use recursive expression synthesis.

Recursive expression synthesis

Recursive expression synthesis is motivated by the insight that it is easier to synthesize complex boolean expressions if we start from the bottom and propagate synthesized sub-expressions as *hints* to subsequent synthesis rounds. These hints are included as additional terminals, and if useful, each synthesis step needs to consider choose trees of only depth one or two.

To illustrate, consider the guard portion of the guarded trace:

```
(&& (! (<=> in_req in_ack))
    (<=> out_req out_ack))                    (6.16)
```

The synthesizer recognizes the && operator and descends into the sub-expressions: `(! (<=> in_req in_ack))` and `(<=> out_req out_ack)`. We will follow the first branch, `(! (<=> in_req in_ack))`. At the leaves, the synthesizer finds `read(`$\mathtt{in}_{\mathtt{req}}$`)` for `in_req` and `read(`$\mathtt{in}_{\mathtt{ack}}$`)` for `in_ack`. The synthesized expressions are passed up, and with a one-deep choose tree, the synthesizer finds `(read(`$\mathtt{in}_{\mathtt{req}}$`) == read(`$\mathtt{in}_{\mathtt{ack}}$`))` for `(<=> in_req in_ack)`. The synthesized expression *and* previous hints are passed up, and with a one-deep choose tree, the synthesizer finds an XOR expression `(read(`$\mathtt{in}_{\mathtt{req}}$`) ^ read(`$\mathtt{in}_{\mathtt{ack}}$`))` for `(! (<=> in_req in_ack))`. Note that the synthesizer was not restricted to using hints returned from recursive calls. Finally, using hints and a one-deep choose tree, the synthesizer arrives at the

complete guard:

```
((read(in_req) ^ read(in_ack)) &&
  (read(out_req) == (read(out_ack))))                    (6.17)
```

6.4.2 Trace synthesis

Recall the trace portion of the guarded trace:

```
{(in ↦ (channel*
         (! (<=> in_req in_ack))
         {
           [(<=> in_req in_ack) null]
           [(! (<=> in_req in_ack)) in_val]
         })),
 (out ↦ (channel*
         (! (<=> out_req out_ack))
         {
           [(<=> out_req out_ack) null]
           [(! (<=> out_req out_ack)) out_val]
         })),
 (in ↦ (channel* #f null)),                              (6.18)
 (out ↦ (channel* #t (! in_val)))}                       (6.19)
```

The result of trace synthesis is a low-level trace whose refinement mapping satisfies the refinement simulation relation with the above trace. Note that the new value mappings 6.18 and 6.19 depend only on symbolic values present in the pre-state, namely `in_val`. This property is a consequence of UNITY's atomic multi-assignment: post-state values only depend on pre-state values. We use this insight to break the trace synthesis problem into two partial trace synthesis problems: one that targets 6.18 and one that targets 6.19.

Partial trace synthesis encounters two challenges when targeting sequential models: unsafe orderings and dependency cycles. Unsafe orderings occur when assignments *invalidate* pre-state values required by subsequent assignments. For example, in before out is an unsafe ordering: assigning a new value to in *invalidates* `in_val`; this makes out's assignment unsafe. COMET excludes unsafe orderings via *partial trace ordering* and *overall trace ordering*. Dependency cycles occur when the sequential ordering of assignments generates a cyclic dependency, as happens when swapping two values. Specifications with dependency cycles are not supported by COMET: we leave this for future work.

Trace value synthesis

Trace value synthesis finds a partial trace that maps to a single UNITY trace value. Consider the new values in the guarded trace:

```
(guarded-trace ...
  (in ↦ (channel* #f null)),
  (out ↦ (channel* #t (! in_val)))} )
```
(6.20)

Trace value synthesis targets in and out independently. When targeting in, the synthesizer determines the low-level values that map to it: $\mathtt{in}_{req}$, $\mathtt{in}_{ack}$, and $\mathtt{in}_{val}$. However, among the three, only $\mathtt{in}_{ack}$ is writable in the type context. The synthesizer searches for a value for for $\mathtt{in}_{ack}$, represented by $\boxed{?}$, appended to the low-level pre-state, such that in the mapped state, (in ↦ `(channel* #f null)`):

$$\begin{aligned}&\{(\mathtt{in}_{req} \mapsto \mathtt{in\_req}), (\mathtt{in}_{ack} \mapsto \mathtt{in\_ack}), (\mathtt{in}_{val} \mapsto \mathtt{in\_val}),\\ &(\mathtt{out}_{req} \mapsto \mathtt{out\_req}), (\mathtt{out}_{ack} \mapsto \mathtt{out\_ack}), (\mathtt{out}_{val} \mapsto \mathtt{out\_val})\\ &(\mathtt{in}_{ack} \mapsto \boxed{?})\}\end{aligned} \tag{6.21}$$

Note the distinction between $\mathtt{in}_{ack}$ the variable and $\mathtt{in\_ack}$ the symbolic value.

The search space is over a restricted set of Rosette. Terminals consist of the symbolic values $\mathtt{in\_req} \ldots \mathtt{out\_val}$, literal booleans, and literal bitvectors.

Expressions consist of Rosette boolean and bitvector functions. The choice to synthesize intermediate values using native Rosette functions is driven by the desire to avoid another layer of symbolic execution.

In this case of $\mathtt{in}$, with a zero deep choose tree, the synthesizer finds that when $(\mathtt{in}_{ack} \mapsto \mathtt{in\_req})$, the mapped state satisfies $(\mathtt{in} \mapsto \texttt{(channel* \#f null)})$. When targeting $\mathtt{out}$, the synthesizer determines the writable low-level values that map to it: $\mathtt{out}_{req}$ and $\mathtt{out}_{val}$. At this point, the low-level trace looks like this, with $\boxed{?_{req}}$ and $\boxed{?_{val}}$ representing values for $\mathtt{out}_{req}$ and $\mathtt{out}_{val}$ the synthesizer needs to fill concurrently.

$$\begin{aligned}&\{(\mathtt{in}_{req} \mapsto \mathtt{in\_req}), (\mathtt{in}_{ack} \mapsto \mathtt{in\_ack}), (\mathtt{in}_{val} \mapsto \mathtt{in\_val}),\\ &\ (\mathtt{out}_{req} \mapsto \mathtt{out\_req}), (\mathtt{out}_{ack} \mapsto \mathtt{out\_ack}), (\mathtt{out}_{val} \mapsto \mathtt{out\_val})\\ &\ (\mathtt{out}_{req} \mapsto \boxed{?_{req}}), (\mathtt{out}_{val} \mapsto \boxed{?_{val}})\}\end{aligned} \tag{6.22}$$

After trying and failing to find a solution in the search space of zero-depth trees, the synthesizer expands the search space one deeper, where the solver finds a model: $(\mathtt{out}_{req} \mapsto \texttt{(! out\_ack)})$ and $(\mathtt{out}_{val} \mapsto \texttt{(! in\_val)})$ maps to $(\mathtt{out} \mapsto \texttt{(channel* \#t (! in\_val))})$.

At the end of trace value synthesis, we have partial traces for $\mathtt{in}$ and $\mathtt{out}$:

$$\{\ldots(\mathtt{in}_{ack} \mapsto \mathtt{in\_req})\} \tag{6.23}$$

$$\{\ldots(\mathtt{out}_{req} \mapsto \texttt{(! out\_ack)}), (\mathtt{out}_{val} \mapsto \texttt{(! in\_val)})\} \tag{6.24}$$

Note that the partial trace for $\mathtt{out}$ has the same ordering problem described in detail in Chapter 5: toggling request before writing to value exposes a potentially incorrect intermediate state. This invalid ordering is corrected in the next phase.

Partial trace ordering

Trace value synthesis finds the state values that map to the desired UNITY post-state, but without regard to intermediate states, as may occur in the Arduino model. Partial trace ordering finds a correct ordering of the synthesized partial trace that satisfies the correctness constraint. To recap, the constraint ensures

that intermediate states map to either the UNITY pre- or post-state. Additionally, for sequential models such as the Arduino model, partial trace ordering must satisfy the safety property that assignments do not invalidate pre-state values used in subsequent assignments.

Partial trace ordering is *per partial trace*. The solver searches over the permutations of the new values. The partial trace for $\mathtt{in}$ is trivially correct because there are no intermediate states. With regards to the partial trace for $\mathtt{out}$, there are two orderings to consider:

$$\{\ldots(\mathtt{out}_{req} \mapsto \texttt{(! out\_ack)}), (\mathtt{out}_{val} \mapsto \texttt{(! in\_val)})\} \tag{6.25}$$

$$\{\ldots(\mathtt{out}_{val} \mapsto \texttt{(! in\_val)}), (\mathtt{out}_{req} \mapsto \texttt{(! out\_ack)})\} \tag{6.26}$$

Only the second partial trace is correct, as the intermediate state maps to the pre-state. To convince yourself, recall that the guard from the guarded trace places constraints on the pre-state:

```
(&& (! (<=> in_req in_ack))
    (<=> out_req out_ack))
```
(6.27)

Because of the guard, we know that the pre-state value for $\mathtt{out}$ is `(channel* #f null)`, so changing the value of $\mathtt{out}_{val}$ has no effect on the mapped value. Only when $\mathtt{out}_{req}$ changes to `(! out_ack)` does the mapped value transition to the post-state value `(channel* #t (! in_val))`.

At the end of partial trace ordering, we have two properly ordered partial traces:

$$\{\ldots(\mathtt{in}_{ack} \mapsto \mathtt{in\_req})\} \tag{6.28}$$

$$\{\ldots(\mathtt{out}_{val} \mapsto \texttt{(! in\_val)}), (\mathtt{out}_{req} \mapsto \texttt{(! out\_ack)})\} \tag{6.29}$$

Overall trace ordering

After partial trace ordering, each partial trace is correct with respect to its corresponding UNITY value. When synthesizing for a sequential model such as

the Arduino model, an overall ordering of partial traces must satisfy the safety property that no partial trace invalidates a pre-state value used by a subsequent partial trace. Note that the *internal ordering* of each partial trace remains fixed. Valid orderings are found by filtering out permutations of the concatenation of partial traces with invalid Rosette symbolic value dependency graphs. The first valid ordering is used. For the inversion example, there are only two orderings to consider. Note that the write to $\mathtt{out}_{val}$ depends on the value of $\mathtt{in}_{val}$, so the fragment for out must come before in:

$$\{\ldots(\mathtt{out}_{val} \mapsto (!\ \mathtt{in\_val})), (\mathtt{out}_{req} \mapsto (!\ \mathtt{out\_ack}))\} \tag{6.30}$$

$$\{\ldots(\mathtt{in}_{ack} \mapsto \mathtt{in\_req})\} \tag{6.31}$$

6.4.3 Statement synthesis

The result of statement synthesis is a sequence of low-level statements that, when interpreted over the low-level pre-state, yield a post-state whose refinement mapping satisfies the correctness constraint. This is achieved by synthesizing statement fragments for each partial trace then collating them into a complete sequence of low-level statements.

Statement fragment synthesis

Statement fragment synthesis is done on a per-partial trace basis. Recall the partial trace corresponding to out:

$$\begin{aligned}&\{(\mathtt{in}_{req} \mapsto \mathtt{in\_req}), (\mathtt{in}_{ack} \mapsto \mathtt{in\_ack}), (\mathtt{in}_{val} \mapsto \mathtt{in\_val}),\\ &\ (\mathtt{out}_{req} \mapsto \mathtt{out\_req}), (\mathtt{out}_{ack} \mapsto \mathtt{out\_ack}), (\mathtt{out}_{val} \mapsto \mathtt{out\_val})\\ &\ (\mathtt{out}_{val} \mapsto (!\ \mathtt{in\_val})), (\mathtt{out}_{req} \mapsto (!\ \mathtt{out\_ack}))\}\end{aligned} \tag{6.32}$$

The goal is to find statements for $\mathtt{out}_{val}$ and $\mathtt{out}_{req}$ that when interpreted over the pre-state, yield this partial trace. The overall syntax for the statements is determined by the model and the type context. For Arduino, = is used for internal variables, while `write` is used for writable pins. For Verilog, non-blocking assignment <= is used for all variables. To start, the synthesizer constructs the

appropriate partial program sketch for $\mathtt{out}_{val}$ and $\mathtt{out}_{req}$ with $\boxed{?_{val}}$ and $\boxed{?_{req}}$ representing expressions:

$$\begin{aligned}&\texttt{write(}\ \mathtt{out}_{val}\ \texttt{,}\ \boxed{?_{val}}\ \texttt{);}\\&\texttt{write(}\ \mathtt{out}_{req}\ \texttt{,}\ \boxed{?_{req}}\ \texttt{);}\end{aligned} \tag{6.33}$$

Each statement is synthesized independently. For $\mathtt{out}_{val}$, the synthesizer searches for an Arduino expression equivalent to `(! in_val)`. The previously-discussed recursive expression synthesis method with sub-expression hints is used here, where the synthesizer finds $\texttt{read(}\mathtt{in}_{val}\texttt{)}$ for `in_val`, then finds $\texttt{!read(}\mathtt{in}_{val}\texttt{)}$ for `(! in_val)`. Statements for $\mathtt{out}_{req}$ and $\mathtt{in}_{ack}$ are synthesized in a similar fashion, resulting in two statement fragments:

$$\begin{aligned}&\texttt{write(}\ \mathtt{out}_{val}\ \texttt{,}\ \texttt{!read(}\mathtt{in}_{val}\texttt{));}\\&\texttt{write(}\ \mathtt{out}_{req}\ \texttt{,}\ \texttt{!read(}\mathtt{out}_{ack}\texttt{));}\end{aligned} \tag{6.34}$$

$$\texttt{write(}\ \mathtt{in}_{ack}\ \texttt{,}\ \texttt{!read(}\mathtt{in}_{req}\texttt{));} \tag{6.35}$$

Statement synthesis concludes by concatenating the statement fragments in the order determined by the partial trace ordering:

$$\begin{aligned}&\texttt{write(}\ \mathtt{out}_{val}\ \texttt{,}\ \texttt{!read(}\mathtt{in}_{val}\texttt{));}\\&\texttt{write(}\ \mathtt{out}_{req}\ \texttt{,}\ \texttt{!read(}\mathtt{out}_{ack}\texttt{));}\\&\texttt{write(}\ \mathtt{in}_{ack}\ \texttt{,}\ \texttt{!read(}\mathtt{in}_{req}\texttt{));}\end{aligned} \tag{6.36}$$

6.5 Assembly and verification

With guarded trace assembly complete, the guard expressions and statements are assembled into a sequence of `if` statements, following the order taken by the UNITY interpreter. The initial state is synthesized from a guarded trace with a trivial guard. Type declaration statements are generated from the translated type context. All these elements are assembled into the complete

```
const in_req = 0;
const in_ack = 1;
const in_val = 2;
const out_req = 3;
const out_ack = 4;
const out_val = 5;

void setup() {
  pinMode(in_req, INPUT);
  pinMode(in_ack, OUTPUT);
  pinMode(in_val, INPUT);
  pinMode(out_req, OUTPUT);
  pinMode(out_ack, INPUT);
  pinMode(out_val, OUTPUT);
  digitalWrite(out_req, digitalRead(out_ack));
}

void loop() {
  if ((digitalRead(in_req) ^ digitalRead(in_ack)) &&
      (digitalRead(out_req) == digitalRead(out_ack))) {
    digitalWrite(out_val, !digitalRead(in_val));
    digitalWrite(out_req, !digitalRead(out_ack));
    digitalWrite(in_ack, digitalRead(in_req));
  }
}
```

Figure 6.3: The synthesized FIFO inverter in Arduino

low-level program in Figure 6.3. Finally, the entire program is interpreted over the low-level pre-state, and verified against the UNITY post-state.

CHAPTER 7

Application

This chapter covers a non-trivial application of COMET: the synthesis of Paxos proposer and acceptor roles.

7.1 Paxos consensus

In the context of this thesis, Paxos refers to Lamport's *single-decree synod* consensus algorithm [21]. The problem Paxos solves is one of *distributed consensus*: how to get a collection of distributed processes to agree on a value. Because processes are distributed, they operate in a *shared nothing* environment: there are no shared memories/storage. The only way for processes to interact with each other is through message passing.

The basic Paxos protocol defines three classes of participants: proposers, acceptors, and learners. Proposers and acceptors are active participants and learners are passive. Proposers initiate a protocol round by sending *prepare* messages to a majority of the acceptors. The acceptors reply with *promise* messages, promising to accept a proposed value. Once a proposer receives promise replies from the majority of the acceptors, it sends *accept* messages to acceptors to commit a value. Acceptors reply to the accept message with an *accepted* message, indicating that the value is committed and the round is complete. After a value is accepted by an acceptor, additional *accepted* messages are sent from acceptors to learners: this propagates the consensus value.

The safety guarantee of the Paxos algorithm ensures that once a value has been chosen, that value will remain stable. The algorithm guarantees this by associating each protocol round with a *ballot number*. Proposer's *prepare* messages are required to have a ballot number greater than that of any existing prepare request. Acceptors are required to inform proposers in promise messages if they have already accepted a value, along with the associated ballot number. When an acceptor sends a *promise* reply, it promises to ignore any requests with lesser ballot numbers. Proposers are required to propose the previously accepted value with the greatest ballot number. This ensures that once a value is accepted, it remains so.

7.2 Specification of the roles

Specifications for the proposer and acceptor in UNITY use a pair of channels between each proposer and acceptor. Each specification is written for an Arduino MKR Vidor 4000 development board containing a microprocessor to run compiled Arduino programs and an FPGA to run compiled Verilog programs [2]. Each development board has 22 I/O pins, which limits the size of the system we can specify. Two-way communication between boards requires two channels, requiring six pins in total. Each board can then communicate back and forth with three others, using 18 pins. The specification defines a topology with one proposer and three acceptors. The acceptor and proposer specifications contain 14 and 34 clauses respectively. Specifications are listed in Appendix A.

Topologies containing up to three proposers and three acceptors are possible. A 3×3 topology requires a modified acceptor specification to include the additional proposers. No changes to the proposer specification are required, because proposers only communicate with acceptors. Hardware with more I/O pins could enable larger topologies to be specified, but due to the deterministic nature of the current UNITY subset supported by COMET, such specifications become rather unwieldy as they get larger.

Role	Clauses	Synthesis (sec)	Verification (sec)
Arduino acceptor	14	985	86
Arduino proposer	34	4109	1928
Verilog acceptor	14	4974	3668
Verilog proposer	34	7357	816

Table 7.1: Synthesis and verification time for Paxos components

7.3 Synthesis of the roles

COMET is able to successfully synthesize Arduino and Verilog implementations of both roles. Arduino and Verilog implementations are listed in Appendix A. The time to synthesize and verify each implementation is summarized in Table 7.1. Recall that COMET assembles synthesized program segments into a whole program which is verified against the specification. This verification time represents a significant portion of computation time and is presented separately.

7.4 Discussion

The resulting code was actually readable: recursive expression synthesis results in composites of simple expressions, as opposed to shallower *seemingly magic* expressions. This was especially the case for bitwise buffer expressions.

The differences in synthesis time were surprising. Arduino synthesis was significantly faster than Verilog synthesis: this difference is almost entirely attributed to time spent in partial trace synthesis. Recall that partial trace values are synthesized into native Rosette expressions. For the Arduino model, COMET synthesizes bitvector expressions, and for the Verilog model, COMET synthesizes a mix of boolean and bitvector expressions. This heterogeneity may be a cause of the slowdown.

The differences in verification time were also surprising. Verification of the Arduino programs was far faster than Verilog programs. To add puzzlement, the simpler Verilog acceptor took far longer to verify than the more complex Verilog proposer. Unlike synthesis that occurs in individually timed phases, verification happens in one timed action, so it is difficult to pinpoint a proba-

ble cause, although some pathological behaviour in the Verilog interpreter may be to blame. In both synthesis and verification cases, further exploration with Rosette's symbolic profiler is warranted [5].

CHAPTER 8

Conclusion

It is a perilous task to design and implement reactive software in a concurrent setting, where components operate independently and where the overall system exhibits a large degree of non-determinism. Specification languages such as UNITY allow designers to take advantage of analysis tools such as model checkers or theorem provers to show their reactive system designs are correct. Unfortunately, to use these designs on real-world systems, implementations in low-level languages have to be written, and showing their fidelity to the specification requires yet more analysis.

Program synthesis relieves the designer from both of these burdens by searching for an implementation that satisfies a correctness constraint. Solver-aided languages such as Rosette allow designers to model their languages for a SVM and to use symbolic execution to verify and synthesize programs for all states.

Symbolic approaches are not without cost, however. Synthesizing whole programs in one step against all but the most trivial specifications is intractable due to the massive state explosion in both the execution of the specification and the search space of low-level programs.

COMET, the system presented here, minimizes this state explosion and enables task decomposition. It synthesizes non-trivial programs using three techniques: symbolic execution of the specification into guarded traces, partial trace synthesis against trace values, and recursive expression synthesis. Guarded traces are motivated by the observation that a UNITY execution has a single

branching point from a pre-state, and that each branch and its guard can be synthesized separately and recombined into a correct whole program. Partial trace synthesis is motivated by the observation that UNITY's atomic multiple assignment yields specification states that allow for the separate synthesis of trace components and their recombination into a correct low-level trace. Recursive expression synthesis is motivated by the observation that synthesis of a deep expression can happen *bottom-up* by passing synthesized subexpressions as hints for synthesizing higher-up expressions. Each of these techniques is designed to scalably synthesize low-level language components such that the final program can be verified to satisfy a refinement simulation relation with the specification.

To validate COMET, we synthesize Arduino and Verilog implementations from UNITY specifications of the Paxos proposer and acceptor roles. These contributions are not without their limitations, which are covered in the rest of this chapter, along with avenues for future work.

8.1 Limitations and future work

8.1.1 Refinement mapping

The current refinement mapping and context translation algorithm make it possible to relate UNITY low-level model executions. It provides a rich source of information used by COMET to constrain the search space when synthesizing traces, expressions, and statements. Both the refinement mapping and the context translation algorithms are targets for synthesis because they are currently derived by hand.

8.1.2 Cyclic data dependencies

As mentioned in Chapter 6, COMET does not support UNITY specifications that induce cyclic dependencies when synthesizing traces for sequential models. For example, this occurs when two variables swap values. A general way to support such behaviours is by *shadow copying* pre-state values to temporary variables. This emulates the UNITY semantics of computing post-values from

pre-state values. Optimizations such as biasing against using shadowed values and pruning unused shadowed values from the synthesized trace can reduce unnecessary copying.

8.1.3 Deterministic specifications

One of UNITY's strengths as a specification language lies in the nondeterministic selection of enabled assignments. The evaluation of guards and assignments in a UNITY specification is subject to a *weak fairness* constraint: a persistently enabled action will eventually happen. This property allows for the proof of liveness properties: that a system makes progress. The deterministic implementations synthesized evaluate their guards in a fixed order, so one channel is always checked before another, for example. Support for bounded nondeterminism and the synthesis of implementations that satisfy fairness constraints would be very useful for many applications, including operating system schedulers and distributed algorithms like Paxos.

8.1.4 Sequential Verilog timing model

UNITY's execution model makes atomic transitions between the pre- and post-states. Verilog is currently modelled in the same way: we model the ideal behaviour of a synchronous clocked system. For asynchronous or locally-synchronous systems without a global clock, events occur sequentially. Support for annotating Verilog programs with timing constraints could allow for the synthesis of asynchronous circuits.

8.2 Conclusion

In closing, COMET achieves tractable reactive program synthesis by formalizing specification and implementation languages, characterizing specification behaviour as a set of guarded traces, and decomposing each guarded trace's components into tractable subproblems. This approach makes it possible to synthesize interoperable hardware and software components from a single specification. Beyond distributed algorithms such as Paxos, COMET may be applicable

to other reactive systems, especially those that operate at the liminal space between hardware and software.

Bibliography

[1] Arduino. Arduino language reference, 2020. URL https://www.arduino.cc/reference/en/. Accessed: 2020-09-02. → pages 3, 10, 12

[2] Arduino. Arduino MKR vidor 4000, 2020. URL https://store.arduino.cc/usa/mkr-vidor-4000. Accessed: 2021-01-06. → page 73

[3] H. Barragán. Wiring: Prototyping physical interaction design. Master's thesis, Interaction Design Institute Ivrea, 2004. URL http://people.interactionivrea.org/h.barragan/thesis/thesis_low_res.pdf. → page 12

[4] A. W. Biermann. On the inference of turing machines from sample computations. *Artif. Intell.*, 3(1):181–198, Jan. 1972. ISSN 0004-3702. doi:10.1016/0004-3702(72)90048-3. URL https://doi.org/10.1016/0004-3702(72)90048-3. → page 5

[5] J. Bornholt and E. Torlak. Finding code that explodes under symbolic evaluation. *Proc. ACM Program. Lang.*, 2(OOPSLA), Oct. 2018. doi:10.1145/3276519. URL https://doi.org/10.1145/3276519. → page 75

[6] E. M. Clarke, E. A. Emerson, and J. Sifakis. Model checking: Algorithmic verification and debugging. *Commun. ACM*, 52(11):74–84, Nov. 2009. ISSN 0001-0782. doi:10.1145/1592761.1592781. URL https://doi.org/10.1145/1592761.1592781. → page 1

[7] T. Coquand. Metamathematical investigations of a calculus of constructions. Technical Report RR-1088, INRIA, Sept. 1989. URL https://hal.inria.fr/inria-00075471. → page 1

[8] R. M. Costa. Compiling distributed system specifications into implementations. Master's thesis, University of British Columbia, 2019.

URL
https://open.library.ubc.ca/collections/ubctheses/24/items/1.0378287. → page 5

[9] P. Coussy, D. D. Gajski, M. Meredith, and A. Takach. An introduction to high-level synthesis. *IEEE Design Test of Computers*, 26(4):8–17, 2009. → page 6

[10] M. N. Do. Corresponding formal specifications with distributed systems. Master's thesis, University of British Columbia, 2019. URL https://open.library.ubc.ca/collections/ubctheses/24/items/1.0378335. → page 5

[11] B. Finkbeiner, F. Klein, R. Piskac, and M. Santolucito. Synthesizing functional reactive programs. In *Proceedings of the 12th ACM SIGPLAN International Symposium on Haskell*, Haskell 2019, page 162–175, New York, NY, USA, 2019. Association for Computing Machinery. ISBN 9781450368131. doi:10.1145/3331545.3342601. URL https://doi.org/10.1145/3331545.3342601. → page 7

[12] B. Finkbeiner, F. Klein, R. Piskac, and M. Santolucito. Temporal stream logic: Synthesis beyond the bools. In I. Dillig and S. Tasiran, editors, *Computer Aided Verification - 31st International Conference, CAV 2019, New York City, NY, USA, July 15-18, 2019, Proceedings, Part I*, volume 11561 of *Lecture Notes in Computer Science*, pages 609–629. Springer, 2019. doi:10.1007/978-3-030-25540-4_35. URL https://doi.org/10.1007/978-3-030-25540-4_35. → page 7

[13] A. Gal, B. Eich, M. Shaver, D. Anderson, D. Mandelin, M. R. Haghighat, B. Kaplan, G. Hoare, B. Zbarsky, J. Orendorff, J. Ruderman, E. W. Smith, R. Reitmaier, M. Bebenita, M. Chang, and M. Franz. Trace-based just-in-time type specialization for dynamic languages. In *Proceedings of the 30th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '09, page 465–478, New York, NY, USA, 2009. Association for Computing Machinery. ISBN 9781605583921. doi:10.1145/1542476.1542528. URL https://doi.org/10.1145/1542476.1542528. → page 6

[14] C. A. R. Hoare. Communicating sequential processes. *Commun. ACM*, 21 (8):666–677, Aug. 1978. ISSN 0001-0782. doi:10.1145/359576.359585. URL https://doi-org.ezproxy.library.ubc.ca/10.1145/359576.359585. → page 1

[15] W. A. Hunt. Microprocessor design verification. *J. Autom. Reason.*, 5(4): 429–460, Nov. 1989. ISSN 0168-7433. → page 5

[16] IEEE Standards Association et al. IEEE standard for verilog hardware description language. *Design Automation Standards Committee, IEEE Std 1364TM-2005*, 2, 2005. → pages 3, 6, 10, 13

[17] M. Johnson, A. Sigsoft, and A. S. I. G. on Programming Languages. *Proceedings of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on High-Level Debugging: Pacific Grove, California, March 20-23, 1983*. ACM Digital Library. Association for Computing Machinery, 1983. ISBN 9780897911115. URL https://books.google.ca/books?id=FQEpAQAAMAAJ. → page 1

[18] M. Kaufmann, P. Manolios, and J. Moore. *Computer-Aided Reasoning: An Approach*. Advances in Formal Methods. Springer US, 2000. ISBN 9780792377443. URL https://books.google.ca/books?id=76zOp-lSpukC. → page 1

[19] J. C. King. Symbolic execution and program testing. *Commun. ACM*, 19(7): 385–394, July 1976. ISSN 0001-0782. doi:10.1145/360248.360252. URL https://doi.org/10.1145/360248.360252. → page 8

[20] V. Kuznetsov, J. Kinder, S. Bucur, and G. Candea. Efficient state merging in symbolic execution. *SIGPLAN Not.*, 47(6):193–204, June 2012. ISSN 0362-1340. doi:10.1145/2345156.2254088. URL https://doi.org/10.1145/2345156.2254088. → page 9

[21] L. Lamport. The part-time parliament. *ACM Trans. Comput. Syst.*, 16(2): 133–169, May 1998. ISSN 0734-2071. doi:10.1145/279227.279229. URL https://doi.org/10.1145/279227.279229. → page 72

[22] L. Lamport. Specifying concurrent systems with TLA+. *Calculational System Design*, pages 183–247, April 1999. URL https://www.microsoft.com/en-us/research/publication/specifying-concurrent-systems-tla/. → page 1

[23] L. Lamport. The PlusCal algorithm language. In *Proceedings of the 6th International Colloquium on Theoretical Aspects of Computing*, ICTAC '09, page 36–60, Berlin, Heidelberg, 2009. Springer-Verlag. ISBN 9783642034657. doi:10.1007/978-3-642-03466-4_2. URL https://doi.org/10.1007/978-3-642-03466-4_2. → page 5

[24] T. A. Lau and D. S. Weld. Programming by demonstration: An inductive learning formulation. In *Proceedings of the 4th International Conference on Intelligent User Interfaces*, IUI '99, page 145–152, New York, NY, USA, 1998. Association for Computing Machinery. ISBN 1581130988. doi:10.1145/291080.291104. URL https://doi.org/10.1145/291080.291104. → page 6

[25] X. Leroy. Formal certification of a compiler back-end or: Programming a compiler with a proof assistant. In *Conference Record of the 33rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '06, page 42–54, New York, NY, USA, 2006. Association for Computing Machinery. ISBN 1595930272. doi:10.1145/1111037.1111042. URL https://doi.org/10.1145/1111037.1111042. → page 5

[26] N. Lynch and F. Vaandrager. Forward and backward simulations part i: Untimed systems (replaces tm-486). Technical report, USA, 1994. URL https://groups.csail.mit.edu/tds/papers/Lynch/TM-486.pdf. → page 50

[27] P. Madhusudan. Synthesizing Reactive Programs. In M. Bezem, editor, *Computer Science Logic (CSL'11) - 25th International Workshop/20th Annual Conference of the EACSL*, volume 12 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 428–442, Dagstuhl, Germany, 2011. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik. ISBN 978-3-939897-32-3. doi:10.4230/LIPIcs.CSL.2011.428. URL http://drops.dagstuhl.de/opus/volltexte/2011/3247. → page 7

[28] K. Mani Chandy and J. Misra. Parallel program design: a foundation, 1988. → pages 2, 11, 21

[29] R. Milner. The polyadic pi calculus: a tutorial. ecs-lfcs-91-180. *Computer Science Dept., University of Edinburgh*, 1991. → page 1

[30] J. S. Moore. *Piton: A Mechanically Verified Assembly-Level Language*. Kluwer Academic Publishers, USA, 1996. ISBN 0792339207. → page 4

[31] L. C. Paulson. Natural deduction as higher-order resolution. *The Journal of Logic Programming*, 3(3):237 – 258, 1986. ISSN 0743-1066. doi:https://doi.org/10.1016/0743-1066(86)90015-4. URL http://www.sciencedirect.com/science/article/pii/0743106686900154. → page 1

[32] S. Porncharoenwase, J. Bornholt, and E. Torlak. Fixing code that explodes under symbolic evaluation. *Verification, Model Checking, and Abstract Interpretation (VMCAI'20)*, Jan 2020. URL http://par.nsf.gov/biblio/10127172. → page 9

[33] C. Reas and B. Fry. *Processing: A Programming Handbook for Visual Designers and Artists*. The MIT Press, 2014. ISBN 026202828X. → page 12

[34] L. Ryzhyk, P. Chubb, I. Kuz, E. Le Sueur, and G. Heiser. Automatic device driver synthesis with termite. In *Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles*, SOSP '09, page 73–86, New York, NY, USA, 2009. Association for Computing Machinery. ISBN 9781605587523. doi:10.1145/1629575.1629583. URL https://doi.org/10.1145/1629575.1629583. → page 8

[35] L. Ryzhyk, A. Walker, J. Keys, A. Legg, A. Raghunath, M. Stumm, and M. Vij. User-guided device driver synthesis. In *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation*, OSDI'14, page 661–676, USA, 2014. USENIX Association. ISBN 9781931971164. → page 8

[36] A. Solar-Lezama, R. Rabbah, R. Bodík, and K. Ebcioundefinedlu. Programming by sketching for bit-streaming programs. In *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '05, page 281–294, New York, NY, USA, 2005. Association for Computing Machinery. ISBN 1595930566. doi:10.1145/1065010.1065045. URL https://doi.org/10.1145/1065010.1065045. → page 7

[37] E. Torlak and R. Bodik. A lightweight symbolic virtual machine for solver-aided host languages. *SIGPLAN Not.*, 49(6):530–541, June 2014. ISSN 0362-1340. doi:10.1145/2666356.2594340. URL https://doi.org/10.1145/2666356.2594340. → pages 3, 9, 10, 15

[38] B. Zhang. PGo: Corresponding a high-level formal specification with its implementation. *SOSP SRC*, page 3, 2016. → page 5

APPENDIX A

Supporting Materials

A.1 UNITY specifications of Paxos

A.1.1 Acceptor

```
;; Model a Paxos acceptor
(define acceptor
  (unity*
   (declare*
    (list (cons 'ballot 'natural)
          (cons 'value 'natural)
          (cons 'phase 'natural)
          (cons 'prom_bal 'natural)
          (cons 'prop_mbal 'natural)
          (cons 'prop_mval 'natural)
          (cons 'out_prop 'send-channel)
          (cons 'in_prop 'recv-channel)
          (cons 'out_prop_bal 'send-buf)
          (cons 'out_prop_val 'send-buf)
          (cons 'in_prop_bal 'recv-buf)
          (cons 'in_prop_val 'recv-buf)))
   (initially*
    (list
     (:=* (list 'ballot
                'value
                'phase
                'prom_bal
                'out_prop
                'in_prop_bal)
          (list 0
                0
                1
                0
                'empty
                (empty-recv-buf* 8)))))
   (assign*
    (list
     (list
```

```
        ;; Phase 0: Accepting state
        ;; Phase 255: Failure state
        ;; Phase 1->2: prepare receive buffers
        (:=* (list 'in_prop_bal
                   'phase)
             (case*
              (list
               (cons (list (empty-recv-buf* 8)
                           2)
                     (=* 'phase 1)))))
        ;; Phase 2: receive proposal ballot from proposer
        (:=* (list 'in_prop
                   'in_prop_bal)
             (case*
              (list
               (cons (list 'empty
                           (recv-buf-put* 'in_prop_bal (value* 'in_prop)))
                     (and* (full?* 'in_prop)
                           (and* (not* (recv-buf-full?* 'in_prop_bal))
                                 (=* 'phase 2)))))))
        ;; Phase 2->3: read proposal
        (:=* (list 'prop_mbal
                   'phase)
             (case*
              (list
               (cons (list (recv-buf->nat* 'in_prop_bal)
                           3)
                     (and* (recv-buf-full?* 'in_prop_bal)
                           (=* 'phase 2))))))
        ;; Phase 3->4: prepare promise response
        ;; Proposed ballot # > 'ballot and > 'prom_bal
        (:=* (list 'out_prop_bal
                   'out_prop_val
                   'prom_bal
                   'phase)
             (case*
              (list
               (cons (list (nat->send-buf* 8 'ballot)
                           (nat->send-buf* 8 'value)
                           'prop_mbal
                           4)
                     (and* (<* 'ballot
                               'prop_mbal)
                           (and* (<* 'prom_bal
                                     'prop_mbal)
                                 (=* 'phase 3)))))))
        ;; Phase 4: send promise message
        (:=* (list 'out_prop
                   'out_prop_bal
                   'out_prop_val)
             (case*
              (list
               (cons (list (message* (send-buf-get* 'out_prop_bal))
                           (send-buf-next* 'out_prop_bal)
                           'out_prop_val)
                     (and* (empty?* 'out_prop)
                           (and* (not* (send-buf-empty?* 'out_prop_bal))
                                 (=* 'phase 4))))
               (cons (list (message* (send-buf-get* 'out_prop_val))
                           'out_prop_bal
                           (send-buf-next* 'out_prop_val))
                     (and* (empty?* 'out_prop)
                           (and* (send-buf-empty?* 'out_prop_bal)
```

```
                              (and* (not* (send-buf-empty?* 'out_prop_val))
                                    (=* 'phase 4))))))))
      ;; Phase 4->5: prepare to receive accept message
      (:=* (list 'in_prop_bal
                 'in_prop_val
                 'phase)
           (case*
            (list
             (cons (list (empty-recv-buf* 8)
                         (empty-recv-buf* 8)
                         5)
                   (and* (send-buf-empty?* 'out_prop_val)
                         (=* 'phase 4))))))
      ;; Phase 5: receive accept ballot/value from proposer
      (:=* (list 'in_prop
                 'in_prop_bal
                 'in_prop_val)
           (case*
            (list
             (cons (list 'empty
                         (recv-buf-put* 'in_prop_bal (value* 'in_prop))
                         'in_prop_val)
                   (and* (full?* 'in_prop)
                         (and* (not* (recv-buf-full?* 'in_prop_bal))
                               (=* 'phase 5))))
             (cons (list 'empty
                         'in_prop_bal
                         (recv-buf-put* 'in_prop_val (value* 'in_prop)))
                   (and* (full?* 'in_prop)
                         (and* (recv-buf-full?* 'in_prop_bal)
                               (and* (not* (recv-buf-full?* 'in_prop_val))
                                     (=* 'phase 5))))))))
      ;; Phase 5->6: read accept
      (:=* (list 'prop_mbal
                 'prop_mval
                 'phase)
           (case*
            (list
             (cons (list (recv-buf->nat* 'in_prop_bal)
                         (recv-buf->nat* 'in_prop_val)
                         6)
                   (and* (recv-buf-full?* 'in_prop_val)
                         (=* 'phase 5))))))
      ;; Phase 6->7: check validity of accept
      (:=* (list 'ballot
                 'value
                 'out_prop_bal
                 'out_prop_val
                 'phase)
           (case*
            (list
             (cons (list 'prop_mbal
                         'prop_mval
                         (nat->send-buf* 8 'prop_mbal)
                         (nat->send-buf* 8 'prop_mval)
                         7)
                   (and* (=* 'prop_mbal 'prom_bal)
                         (=* 'phase 6))))))
      ;; Phase 7: send accept acknowledged message
      (:=* (list 'out_prop
                 'out_prop_bal
                 'out_prop_val)
           (case*
```

```
             (list
              (cons (list (message* (send-buf-get* 'out_prop_bal))
                          (send-buf-next* 'out_prop_bal)
                          'out_prop_val)
                    (and* (empty?* 'out_prop)
                          (and* (not* (send-buf-empty?* 'out_prop_bal))
                                (=* 'phase 7))))
              (cons (list (message* (send-buf-get* 'out_prop_val))
                          'out_prop_bal
                          (send-buf-next* 'out_prop_val))
                    (and* (empty?* 'out_prop)
                          (and* (send-buf-empty?* 'out_prop_bal)
                                (and* (not* (send-buf-empty?* 'out_prop_val))
                                      (=* 'phase 7))))))))
       ;; Phase 7->0: complete
       (:=* (list 'phase)
            (case*
             (list
              (cons (list 0)
                    (and* (send-buf-empty?* 'out_prop_val)
                          (=* 'phase 7)))))))))
```

A.1.2 Proposer

```
;; Model a Paxos proposer
;; A proposer sends phase 1a and 2a and receives phase 1b and 2b messages
;; Internal vars:
;; Ballot
;; Value
;; Phase
;; Sent#
;; Rcvd#
;; Send/Recv Buffers for each Acceptor
;; Send/Recv Channels for each Acceptor
(define proposer
  (unity*
   (declare*
    (list (cons 'ballot 'natural)
          (cons 'value 'natural)
          (cons 'phase 'natural)
          (cons 'a_mbal 'natural)
          (cons 'b_mbal 'natural)
          (cons 'c_mbal 'natural)
          (cons 'a_mval 'natural)
          (cons 'b_mval 'natural)
          (cons 'c_mval 'natural)
          (cons 'out_a 'send-channel)
          (cons 'out_b 'send-channel)
          (cons 'out_c 'send-channel)
          (cons 'in_a 'recv-channel)
          (cons 'in_b 'recv-channel)
          (cons 'in_c 'recv-channel)
          (cons 'out_a_bal 'send-buf)
          (cons 'out_b_bal 'send-buf)
          (cons 'out_c_bal 'send-buf)
          (cons 'out_a_val 'send-buf)
          (cons 'out_b_val 'send-buf)
          (cons 'out_c_val 'send-buf)
          (cons 'in_a_bal 'recv-buf)
          (cons 'in_b_bal 'recv-buf)
```

```
           (cons 'in_c_bal 'recv-buf)
           (cons 'in_a_val 'recv-buf)
           (cons 'in_b_val 'recv-buf)
           (cons 'in_c_val 'recv-buf)))
     (initially*
      (list
       (:=* (list 'ballot
                  'value
                  'phase
                  'out_a
                  'out_b
                  'out_c)
            (list 1
                  32
                  1
                  'empty
                  'empty
                  'empty))))
     (assign*
      (list
       (list
        ;; Phase 0: Accepting state
        ;; Phase 255: Failure state
        ;; 01 Failure mode: max ballot value
        (:=* (list 'phase)
             (case*
              (list (cons (list 255)
                          (=* 'ballot 255)))))
        ;; 02 Phase 1->2: load buffers for sending prepare messages
        (:=* (list 'out_a_bal
                   'out_b_bal
                   'out_c_bal
                   'phase)
             (case*
              (list (cons (list (nat->send-buf* 8 'ballot)
                                (nat->send-buf* 8 'ballot)
                                (nat->send-buf* 8 'ballot)
                                2)
                          (=* 'phase 1)))))
        ;; 03 Phase 2: send promise message to acceptors
        (:=* (list 'out_a
                   'out_a_bal)
             (case*
              (list
               (cons (list (message* (send-buf-get* 'out_a_bal))
                           (send-buf-next* 'out_a_bal))
                     (and* (empty?* 'out_a)
                           (and* (not* (send-buf-empty?* 'out_a_bal))
                                 (=* 'phase 2)))))))
        ;; 04
        (:=* (list 'out_b
                   'out_b_bal)
             (case*
              (list
               (cons (list (message* (send-buf-get* 'out_b_bal))
                           (send-buf-next* 'out_b_bal))
                     (and* (empty?* 'out_b)
                           (and* (not* (send-buf-empty?* 'out_b_bal))
                                 (=* 'phase 2)))))))
        ;; 05
        (:=* (list 'out_c
                   'out_c_bal)
             (case*
```

```
             (list
              (cons (list (message* (send-buf-get* 'out_c_bal))
                          (send-buf-next* 'out_c_bal))
                    (and* (empty?* 'out_c)
                          (and* (not* (send-buf-empty?* 'out_c_bal))
                                (=* 'phase 2)))))))
       ;; 06 Phase 2->3: prepare receive buffers
       (:=* (list 'in_a_bal
                  'in_b_bal
                  'in_c_bal
                  'in_a_val
                  'in_b_val
                  'in_c_val
                  'phase)
            (case*
             (list (cons (list (empty-recv-buf* 8)
                               (empty-recv-buf* 8)
                               (empty-recv-buf* 8)
                               (empty-recv-buf* 8)
                               (empty-recv-buf* 8)
                               (empty-recv-buf* 8)
                               3)
                         (and* (send-buf-empty?* 'out_a_bal)
                               (and* (send-buf-empty?* 'out_b_bal)
                                     (and* (send-buf-empty?* 'out_c_bal)
                                           (=* 'phase 2)))))))
       ;; Phase 3: receive promise replies from acceptors
       (:=* (list 'in_a
                  'in_a_bal
                  'in_a_val)
            (case*
             (list
              ;; 07
              (cons (list 'empty
                          (recv-buf-put* 'in_a_bal (value* 'in_a))
                          'in_a_val)
                    (and* (full?* 'in_a)
                          (and* (not* (recv-buf-full?* 'in_a_bal))
                                (=* 'phase 3))))
              ;; 08
              (cons (list 'empty
                          'in_a_bal
                          (recv-buf-put* 'in_a_val (value* 'in_a)))
                    (and* (full?* 'in_a)
                          (and* (recv-buf-full?* 'in_a_bal)
                                (and* (not* (recv-buf-full?* 'in_a_val))
                                      (=* 'phase 3))))))))
       (:=* (list 'in_b
                  'in_b_bal
                  'in_b_val)
            (case*
             (list
              ;; 09
              (cons (list 'empty
                          (recv-buf-put* 'in_b_bal (value* 'in_b))
                          'in_b_val)
                    (and* (full?* 'in_b)
                          (and* (not* (recv-buf-full?* 'in_b_bal))
                                (=* 'phase 3))))
              ;; 10
              (cons (list 'empty
                          'in_b_bal
                          (recv-buf-put* 'in_b_val (value* 'in_b)))
```

```
                  (and* (full?* 'in_b)
                        (and* (recv-buf-full?* 'in_b_bal)
                              (and* (not* (recv-buf-full?* 'in_b_val))
                                    (=* 'phase 3))))))))
      (:=* (list 'in_c
                 'in_c_bal
                 'in_c_val)
           (case*
            (list
             ;; 11
             (cons (list 'empty
                         (recv-buf-put* 'in_c_bal (value* 'in_c))
                         'in_c_val)
                   (and* (full?* 'in_c)
                         (and* (not* (recv-buf-full?* 'in_c_bal))
                               (=* 'phase 3))))
             ;; 12
             (cons (list 'empty
                         'in_c_bal
                         (recv-buf-put* 'in_c_val (value* 'in_c)))
                   (and* (full?* 'in_c)
                         (and* (recv-buf-full?* 'in_c_bal)
                               (and* (not* (recv-buf-full?* 'in_c_val))
                                     (=* 'phase 3))))))))
      ;; 13 Phase 3->4
      (:=* (list 'a_mbal
                 'b_mbal
                 'c_mbal
                 'a_mval
                 'b_mval
                 'c_mval
                 'phase)
           (case*
            (list (cons (list (recv-buf->nat* 'in_a_bal)
                              (recv-buf->nat* 'in_b_bal)
                              (recv-buf->nat* 'in_c_bal)
                              (recv-buf->nat* 'in_a_val)
                              (recv-buf->nat* 'in_b_val)
                              (recv-buf->nat* 'in_c_val)
                              4)
                        (and* (recv-buf-full?* 'in_a_val)
                              (and* (recv-buf-full?* 'in_b_val)
                                    (and* (recv-buf-full?* 'in_c_val)
                                          (=* 'phase 3))))))))
      ;; Phase 4->5: select value
      (:=* (list 'value
                 'phase)
           (case*
            (list
             ;; 14
             (cons (list 'value
                         5)
                   (and* (=* 0 'a_mbal)
                         (and* (=* 0 'b_mbal)
                               (and* (=* 0 'c_mbal)
                                     (=* 'phase 4)))))
             ;; 15
             (cons (list 'a_mval
                         5)
                   (and* (or* (=* 'b_mbal 'a_mbal)
                              (<* 'b_mbal 'a_mbal))
                         (and* (or* (=* 'c_mbal 'a_mbal)
                                    (<* 'c_mbal 'a_mbal))
```

```
                              (=* 'phase 4))))
             ;; 16
             (cons (list 'b_mval
                         5)
                   (and* (or* (=* 'a_mbal 'b_mbal)
                              (<* 'a_mbal 'b_mbal))
                         (and* (or* (=* 'c_mbal 'b_mbal)
                                    (<* 'c_mbal 'b_mbal))
                               (=* 'phase 4))))
             ;; 17
             (cons (list 'c_mval
                         5)
                   (and* (or* (=* 'a_mbal 'c_mbal)
                              (<* 'a_mbal 'c_mbal))
                         (and* (or* (=* 'b_mbal 'c_mbal)
                                    (<* 'b_mbal 'c_mbal))
                               (=* 'phase 4)))))))
       ;; 18 Phase 5->6: load buffers for sending vote messages
       (:=* (list 'out_a_bal
                  'out_b_bal
                  'out_c_bal
                  'out_a_val
                  'out_b_val
                  'out_c_val
                  'phase)
            (case*
             (list (cons (list (nat->send-buf* 8 'ballot)
                               (nat->send-buf* 8 'ballot)
                               (nat->send-buf* 8 'ballot)
                               (nat->send-buf* 8 'value)
                               (nat->send-buf* 8 'value)
                               (nat->send-buf* 8 'value)
                               6)
                         (=* 'phase 5)))))
       ;; Phase 6: send vote message to acceptors
       (:=* (list 'out_a
                  'out_a_bal
                  'out_a_val)
            (case*
             (list
              ;; 19
              (cons (list (message* (send-buf-get* 'out_a_bal))
                          (send-buf-next* 'out_a_bal)
                          'out_a_val)
                    (and* (empty?* 'out_a)
                          (and* (not* (send-buf-empty?* 'out_a_bal))
                                (=* 'phase 6))))
              ;; 20
              (cons (list (message* (send-buf-get* 'out_a_val))
                          'out_a_bal
                          (send-buf-next* 'out_a_val))
                    (and* (empty?* 'out_a)
                          (and* (send-buf-empty?* 'out_a_bal)
                                (and* (not* (send-buf-empty?* 'out_a_val))
                                      (=* 'phase 6))))))))
       (:=* (list 'out_b
                  'out_b_bal
                  'out_b_val)
            (case*
             (list
              ;; 21
              (cons (list (message* (send-buf-get* 'out_b_bal))
                          (send-buf-next* 'out_b_bal)
```

```
                        'out_b_val)
                  (and* (empty?* 'out_b)
                        (and* (not* (send-buf-empty?* 'out_b_bal))
                              (=* 'phase 6))))
            ;; 22
            (cons (list (message* (send-buf-get* 'out_b_val))
                        'out_b_bal
                        (send-buf-next* 'out_b_val))
                  (and* (empty?* 'out_b)
                        (and* (send-buf-empty?* 'out_b_bal)
                              (and* (not* (send-buf-empty?* 'out_b_val))
                                    (=* 'phase 6))))))))
      (:=* (list 'out_c
                 'out_c_bal
                 'out_c_val)
           (case*
            (list
             ;; 23
             (cons (list (message* (send-buf-get* 'out_c_bal))
                         (send-buf-next* 'out_c_bal)
                         'out_c_val)
                   (and* (empty?* 'out_c)
                         (and* (not* (send-buf-empty?* 'out_c_bal))
                               (=* 'phase 6))))
             ;; 24
             (cons (list (message* (send-buf-get* 'out_c_val))
                         'out_c_bal
                         (send-buf-next* 'out_c_val))
                   (and* (empty?* 'out_c)
                         (and* (send-buf-empty?* 'out_c_bal)
                               (and* (not* (send-buf-empty?* 'out_c_val))
                                     (=* 'phase 6))))))))
      ;; 25 Phase 6->7
      (:=* (list 'in_a_bal
                 'in_b_bal
                 'in_c_bal
                 'in_a_val
                 'in_b_val
                 'in_c_val
                 'phase)
           (case*
            (list (cons (list (empty-recv-buf* 8)
                              (empty-recv-buf* 8)
                              (empty-recv-buf* 8)
                              (empty-recv-buf* 8)
                              (empty-recv-buf* 8)
                              (empty-recv-buf* 8)
                              7)
                        (and* (send-buf-empty?* 'out_a_val)
                              (and* (send-buf-empty?* 'out_b_val)
                                    (and* (send-buf-empty?* 'out_c_val)
                                          (=* 'phase 6))))))))
      ;; Phase 7: receive vote replies from acceptors
      (:=* (list 'in_a
                 'in_a_bal
                 'in_a_val)
           (case*
            (list
             ;; 26
             (cons (list 'empty
                         (recv-buf-put* 'in_a_bal (value* 'in_a))
                         'in_a_val)
                   (and* (full?* 'in_a)
```

```
                        (and* (not* (recv-buf-full?* 'in_a_bal))
                              (=* 'phase 7))))
              ;; 27
              (cons (list 'empty
                          'in_a_bal
                          (recv-buf-put* 'in_a_val (value* 'in_a)))
                    (and* (full?* 'in_a)
                          (and* (recv-buf-full?* 'in_a_bal)
                                (and* (not* (recv-buf-full?* 'in_a_val))
                                      (=* 'phase 7))))))))
        (:=* (list 'in_b
                   'in_b_bal
                   'in_b_val)
             (case*
              (list
               ;; 28
               (cons (list 'empty
                           (recv-buf-put* 'in_b_bal (value* 'in_b))
                           'in_b_val)
                     (and* (full?* 'in_b)
                           (and* (not* (recv-buf-full?* 'in_b_bal))
                                 (=* 'phase 7))))
               ;; 29
               (cons (list 'empty
                           'in_b_bal
                           (recv-buf-put* 'in_b_val (value* 'in_b)))
                     (and* (full?* 'in_b)
                           (and* (recv-buf-full?* 'in_b_bal)
                                 (and* (not* (recv-buf-full?* 'in_b_val))
                                       (=* 'phase 7))))))))
        (:=* (list 'in_c
                   'in_c_bal
                   'in_c_val)
             (case*
              (list
               ;; 30
               (cons (list 'empty
                           (recv-buf-put* 'in_c_bal (value* 'in_c))
                           'in_c_val)
                     (and* (full?* 'in_c)
                           (and* (not* (recv-buf-full?* 'in_c_bal))
                                 (=* 'phase 7))))
               ;; 31
               (cons (list 'empty
                           'in_c_bal
                           (recv-buf-put* 'in_c_val (value* 'in_c)))
                     (and* (full?* 'in_c)
                           (and* (recv-buf-full?* 'in_c_bal)
                                 (and* (not* (recv-buf-full?* 'in_c_val))
                                       (=* 'phase 7))))))))
        ;; 32 Phase 7->8
        (:=* (list 'a_mbal
                   'b_mbal
                   'c_mbal
                   'a_mval
                   'b_mval
                   'c_mval
                   'phase)
             (case*
              (list (cons (list (recv-buf->nat* 'in_a_bal)
                                (recv-buf->nat* 'in_b_bal)
                                (recv-buf->nat* 'in_c_bal)
                                (recv-buf->nat* 'in_a_val)
```

```
                              (recv-buf->nat* 'in_b_val)
                              (recv-buf->nat* 'in_c_val)
                              8)
                        (and* (recv-buf-full?* 'in_a_val)
                              (and* (recv-buf-full?* 'in_b_val)
                                    (and* (recv-buf-full?* 'in_c_val)
                                          (=* 'phase 7))))))))
      ;; Phase 8: check value
      (:=* (list 'phase)
           (case*
            (list
             ;; 33
             (cons (list 0)
                   (and* (=* 'value 'a_mval)
                         (and* (=* 'value 'b_mval)
                               (and* (=* 'value 'c_mval)
                                     (=* 'phase 8)))))
             ;; 34
             (cons (list 255)
                   (and* (not* (and* (=* 'value 'a_mval)
                                     (and* (=* 'value 'b_mval)
                                           (=* 'value 'c_mval))))
                         (=* 'phase 8)))))))))))
```

A.2 Arduino implementations of Paxos

A.2.1 Acceptor

```
(arduino*
 (setup*
  (list
   (byte* 'ballot)
   (byte* 'value)
   (byte* 'phase)
   (byte* 'prom_bal)
   (byte* 'prop_mbal)
   (byte* 'prop_mval)
   (pin-mode* 'd2 'OUTPUT)
   (pin-mode* 'd1 'INPUT)
   (pin-mode* 'd0 'OUTPUT)
   (pin-mode* 'd5 'INPUT)
   (pin-mode* 'd4 'OUTPUT)
   (pin-mode* 'd3 'INPUT)
   (byte* 'out_prop_bal_vals)
   (byte* 'out_prop_bal_sent)
   (byte* 'out_prop_val_vals)
   (byte* 'out_prop_val_sent)
   (byte* 'in_prop_bal_vals)
   (byte* 'in_prop_bal_rcvd)
   (byte* 'in_prop_val_vals)
   (byte* 'in_prop_val_rcvd)
   (:=* 'ballot (bv #x00 8))
   (:=* 'value (bv #x00 8))
   (:=* 'in_prop_bal_vals (bv #x00 8))
   (:=* 'in_prop_bal_rcvd (bv #x00 8))
   (:=* 'prom_bal (bv #x00 8))
   (:=* 'phase (bv #x01 8))
   (write* 'd0 (read* 'd1))))
```

```
(loop*
 (list
  (if*
   (eq* (bv #x01 8) 'phase)
   (list
    (:=* 'in_prop_bal_vals (bv #x00 8))
    (:=* 'in_prop_bal_rcvd (bv #x00 8))
    (:=* 'phase (bv #x02 8)))
   (list
    (if*
     (and*
      (bwxor* (read* 'd4) (read* 'd3))
      (shl* (eq* 'phase (bv #x02 8)) 'in_prop_bal_rcvd))
     (list
      (:=*
       'in_prop_bal_vals
       (bwxor*
        (shl* (eq* (bv #x00 8) (read* 'd5)) 'in_prop_bal_rcvd)
        (bwor* (shl* (bv #x01 8) 'in_prop_bal_rcvd) 'in_prop_bal_vals)))
      (:=* 'in_prop_bal_rcvd (add* (bv #x01 8) 'in_prop_bal_rcvd))
      (write* 'd4 (read* 'd3)))
     (list
      (if*
       (and* (eq* 'phase (bv #x02 8)) (shr* 'in_prop_bal_rcvd (bv #x03 8)))
       (list (:=* 'prop_mbal 'in_prop_bal_vals) (:=* 'phase (bv #x03 8)))
       (list
        (if*
         (and*
          (lt* 'ballot 'prop_mbal)
          (and* (eq* (bv #x03 8) 'phase) (lt* 'prom_bal 'prop_mbal)))
         (list
          (:=* 'phase (bv #x04 8))
          (:=* 'prom_bal 'prop_mbal)
          (:=* 'out_prop_bal_vals 'ballot)
          (:=* 'out_prop_bal_sent (bv #x00 8))
          (:=* 'out_prop_val_vals 'value)
          (:=* 'out_prop_val_sent (bv #x00 8)))
         (list
          (if*
           (and*
            (shl* (eq* (bv #x04 8) 'phase) 'out_prop_bal_sent)
            (eq* (read* 'd1) (read* 'd0)))
           (list
            (write*
             'd2
             (shr*
              (bv #x01 8)
              (eq*
               (bwand*
                (shr* 'out_prop_bal_vals 'out_prop_bal_sent)
                (bv #x01 8))
               (bv #x00 8))))
            (write* 'd0 (eq* (read* 'd1) (bv #x00 8)))
            (:=* 'out_prop_bal_sent (add* (bv #x01 8) 'out_prop_bal_sent)))
           (list
            (if*
             (bwand*
              (eq* (read* 'd1) (read* 'd0))
              (and*
               (shl* (eq* (bv #x04 8) 'phase) 'out_prop_val_sent)
               (shr* 'out_prop_bal_sent (bv #x03 8))))
             (list
              (write*
```

```
                'd2
                (bwxor*
                 (bv #x01 8)
                 (eq*
                  (bv #x00 8)
                  (bwand*
                   'out_prop_val_vals
                   (shl* (bv #x01 8) 'out_prop_val_sent)))))
               (write* 'd0 (eq* (read* 'd1) (bv #x00 8)))
               (:=* 'out_prop_val_sent (add* (bv #x01 8) 'out_prop_val_sent)))
              (list
               (if*
                (bwand*
                 (eq* (bv #x04 8) 'phase)
                 (lt* (bv #x07 8) 'out_prop_val_sent))
                (list
                 (:=* 'in_prop_bal_vals (bv #x00 8))
                 (:=* 'in_prop_bal_rcvd (bv #x00 8))
                 (:=* 'phase (bv #x05 8))
                 (:=* 'in_prop_val_vals (bv #x00 8))
                 (:=* 'in_prop_val_rcvd (bv #x00 8)))
                (list
                 (if*
                  (and*
                   (add*
                    (shr* (read* 'd3) (read* 'd4))
                    (bwand* (eq* (read* 'd3) (bv #x00 8)) (read* 'd4)))
                   (shl* (eq* 'phase (bv #x05 8)) 'in_prop_bal_rcvd))
                  (list
                   (:=*
                    'in_prop_bal_vals
                    (bwxor*
                     (bwxor* (shl* (read* 'd5) 'in_prop_bal_rcvd) (bv #xff 8))
                     (bwor*
                      (shl* (bv #x01 8) 'in_prop_bal_rcvd)
                      (bwxor* 'in_prop_bal_vals (bv #xff 8)))))
                   (:=* 'in_prop_bal_rcvd (add* 'in_prop_bal_rcvd (bv #x01 8)))
                   (write* 'd4 (read* 'd3)))
                  (list
                   (if*
                    (bwand*
                     (bwor*
                      (and* (read* 'd4) (shl* (bv #x80 8) (read* 'd3)))
                      (lt* (read* 'd4) (read* 'd3)))
                     (and*
                      (bwand* (bv #xf8 8) 'in_prop_bal_rcvd)
                      (bwand*
                       (eq* (bv #x05 8) 'phase)
                       (eq*
                        (bv #x00 8)
                        (bwand* (bv #xf8 8) 'in_prop_val_rcvd)))))
                    (list
                     (write* 'd4 (read* 'd3))
                     (:=*
                      'in_prop_val_vals
                      (bwxor*
                       (bwor*
                        (shl* (bv #x01 8) 'in_prop_val_rcvd)
                        'in_prop_val_vals)
                       (shl*
                        (shr* (bv #x01 8) (read* 'd5))
                        (shr* 'in_prop_val_rcvd (read* 'd5)))))
                     (:=*
```

```
                    'in_prop_val_rcvd
                    (add* (bv #x01 8) 'in_prop_val_rcvd)))
                  (list
                   (if*
                    (and*
                     (shr* 'in_prop_val_rcvd (bv #x03 8))
                     (eq* (bv #x05 8) 'phase))
                    (list
                     (:=* 'prop_mbal 'in_prop_bal_vals)
                     (:=* 'prop_mval 'in_prop_val_vals)
                     (:=* 'phase (bv #x06 8)))
                    (list
                     (if*
                      (bwand*
                       (eq* 'prop_mbal 'prom_bal)
                       (eq* 'phase (bv #x06 8)))
                      (list
                       (:=* 'out_prop_val_vals 'prop_mval)
                       (:=* 'out_prop_val_sent (bv #x00 8))
                       (:=* 'value 'prop_mval)
                       (:=* 'ballot 'prop_mbal)
                       (:=* 'phase (bv #x07 8))
                       (:=* 'out_prop_bal_vals 'prop_mbal)
                       (:=* 'out_prop_bal_sent (bv #x00 8)))
                      (list
                       (if*
                        (and*
                         (and*
                          (add*
                           (lt* (bv #x07 8) 'out_prop_bal_sent)
                           (bv #xff 8))
                          (eq* 'phase (bv #x07 8)))
                         (eq* (read* 'd1) (read* 'd0)))
                        (list
                         (write*
                          'd2
                          (shr*
                           (bv #x01 8)
                           (shr*
                            (bv #x01 8)
                            (bwand*
                             'out_prop_bal_vals
                             (shl* (bv #x01 8) 'out_prop_bal_sent)))))
                         (write* 'd0 (eq* (read* 'd0) (bv #x00 8)))
                         (:=*
                          'out_prop_bal_sent
                          (add* (bv #x01 8) 'out_prop_bal_sent)))
                        (list
                         (if*
                          (bwand*
                           (and*
                            (bwand*
                             (eq* 'phase (bv #x07 8))
                             (lt*
                              (bwand* (bv #xf8 8) 'out_prop_val_sent)
                              (bv #x02 8)))
                            (bwand* 'out_prop_bal_sent (bv #xf8 8)))
                           (eq* (read* 'd1) (read* 'd0)))
                          (list
                           (write*
                            'd2
                            (bwxor*
                             (bv #x01 8)
```

```
                              (lt*
                               (bwand*
                                'out_prop_val_vals
                                (shl* (bv #x01 8) 'out_prop_val_sent))
                               (bv #x01 8))))
                             (write* 'd0 (shl* (bv #x80 8) (read* 'd0)))
                             (:=*
                              'out_prop_val_sent
                              (add* 'out_prop_val_sent (bv #x01 8))))
                            (list
                             (if*
                              (and*
                               (shr* 'out_prop_val_sent (bv #x03 8))
                               (eq* 'phase (bv #x07 8)))
                              (list (:=* 'phase (bv #x00 8)))
                              '())))))))))))))))))))))))))))))))
```

A.2.2 Proposer

```
(arduino*
 (setup*
  (list
   (byte* 'ballot)
   (byte* 'value)
   (byte* 'phase)
   (byte* 'a_mbal)
   (byte* 'b_mbal)
   (byte* 'c_mbal)
   (byte* 'a_mval)
   (byte* 'b_mval)
   (byte* 'c_mval)
   (pin-mode* 'd2 'OUTPUT)
   (pin-mode* 'd1 'INPUT)
   (pin-mode* 'd0 'OUTPUT)
   (pin-mode* 'd5 'OUTPUT)
   (pin-mode* 'd4 'INPUT)
   (pin-mode* 'd3 'OUTPUT)
   (pin-mode* 'd8 'OUTPUT)
   (pin-mode* 'd7 'INPUT)
   (pin-mode* 'd6 'OUTPUT)
   (pin-mode* 'd11 'INPUT)
   (pin-mode* 'd10 'OUTPUT)
   (pin-mode* 'd9 'INPUT)
   (pin-mode* 'd14 'INPUT)
   (pin-mode* 'd13 'OUTPUT)
   (pin-mode* 'd12 'INPUT)
   (pin-mode* 'd17 'INPUT)
   (pin-mode* 'd16 'OUTPUT)
   (pin-mode* 'd15 'INPUT)
   (byte* 'out_a_bal_vals)
   (byte* 'out_a_bal_sent)
   (byte* 'out_b_bal_vals)
   (byte* 'out_b_bal_sent)
   (byte* 'out_c_bal_vals)
   (byte* 'out_c_bal_sent)
   (byte* 'out_a_val_vals)
   (byte* 'out_a_val_sent)
   (byte* 'out_b_val_vals)
   (byte* 'out_b_val_sent)
   (byte* 'out_c_val_vals)
```

```
    (byte* 'out_c_val_sent)
    (byte* 'in_a_bal_vals)
    (byte* 'in_a_bal_rcvd)
    (byte* 'in_b_bal_vals)
    (byte* 'in_b_bal_rcvd)
    (byte* 'in_c_bal_vals)
    (byte* 'in_c_bal_rcvd)
    (byte* 'in_a_val_vals)
    (byte* 'in_a_val_rcvd)
    (byte* 'in_b_val_vals)
    (byte* 'in_b_val_rcvd)
    (byte* 'in_c_val_vals)
    (byte* 'in_c_val_rcvd)
    (write* 'd3 (read* 'd4))
    (write* 'd0 (read* 'd1))
    (:=* 'phase (bv #x01 8))
    (:=* 'ballot (bv #x01 8))
    (:=* 'value (bv #x20 8))
    (write* 'd6 (read* 'd7))))
  (loop*
   (list
    (if*
     (lt* (bv #xfe 8) 'ballot)
     (list (:=* 'phase (bv #xff 8)))
     (list
      (if*
       (eq* (bv #x01 8) 'phase)
       (list
        (:=* 'out_a_bal_vals 'ballot)
        (:=* 'out_a_bal_sent (bv #x00 8))
        (:=* 'out_c_bal_vals 'ballot)
        (:=* 'out_c_bal_sent (bv #x00 8))
        (:=* 'out_b_bal_vals 'ballot)
        (:=* 'out_b_bal_sent (bv #x00 8))
        (:=* 'phase (bv #x02 8)))
       (list
        (if*
         (and*
          (eq*
           (bv #x00 8)
           (or*
            (bwand* (eq* (read* 'd0) (bv #x00 8)) (read* 'd1))
            (shr* (read* 'd0) (read* 'd1))))
          (shl* (eq* 'phase (bv #x02 8)) 'out_a_bal_sent))
         (list
          (write*
           'd2
           (eq*
            (bv #x00 8)
            (shr*
             (bv #x01 8)
             (bwand* (bv #x01 8) (shr* 'out_a_bal_vals 'out_a_bal_sent)))))
          (write* 'd0 (lt* (read* 'd1) (bv #x01 8)))
          (:=* 'out_a_bal_sent (add* (bv #x01 8) 'out_a_bal_sent)))
         (list
          (if*
           (bwand*
            (eq* (read* 'd4) (read* 'd3))
            (and*
             (shr* (bv #x20 8) (bwand* 'out_b_bal_sent (bv #xf8 8)))
             (eq* (bv #x02 8) 'phase)))
           (list
            (write*
```

```
              'd5
              (eq*
               (bv #x00 8)
               (lt*
                (bwand* 'out_b_bal_vals (shl* (bv #x01 8) 'out_b_bal_sent))
                (bv #x01 8))))
             (write* 'd3 (eq* (read* 'd3) (bv #x00 8)))
             (:=* 'out_b_bal_sent (add* (bv #x01 8) 'out_b_bal_sent)))
            (list
             (if*
              (and*
               (eq* (read* 'd7) (read* 'd6))
               (and*
                (shl* (bv #xf7 8) 'out_c_bal_sent)
                (eq* (bv #x02 8) 'phase)))
              (list
               (write*
                'd8
                (shr*
                 (bv #x01 8)
                 (eq*
                  (bv #x00 8)
                  (bwand* (shl* (bv #x01 8) 'out_c_bal_sent) 'out_c_bal_vals))))
               (write* 'd6 (shl* (bv #x80 8) (read* 'd6)))
               (:=* 'out_c_bal_sent (add* 'out_c_bal_sent (bv #x01 8))))
              (list
               (if*
                (and*
                 (and*
                  (bwand* (bv #xf8 8) 'out_b_bal_sent)
                  (and*
                   (bwand* (bv #xf8 8) 'out_c_bal_sent)
                   (eq* (bv #x02 8) 'phase)))
                 (lt* (bv #x07 8) 'out_a_bal_sent))
                (list
                 (:=* 'in_b_val_vals (bv #x00 8))
                 (:=* 'in_b_val_rcvd (bv #x00 8))
                 (:=* 'in_a_bal_vals (bv #x00 8))
                 (:=* 'in_a_bal_rcvd (bv #x00 8))
                 (:=* 'in_c_bal_vals (bv #x00 8))
                 (:=* 'in_c_bal_rcvd (bv #x00 8))
                 (:=* 'phase (bv #x03 8))
                 (:=* 'in_b_bal_vals (bv #x00 8))
                 (:=* 'in_b_bal_rcvd (bv #x00 8))
                 (:=* 'in_a_val_vals (bv #x00 8))
                 (:=* 'in_a_val_rcvd (bv #x00 8))
                 (:=* 'in_c_val_vals (bv #x00 8))
                 (:=* 'in_c_val_rcvd (bv #x00 8)))
                (list
                 (if*
                  (and*
                   (shl* (eq* (bv #x03 8) 'phase) 'in_a_bal_rcvd)
                   (bwxor* (read* 'd9) (read* 'd10)))
                  (list
                   (:=*
                    'in_a_bal_vals
                    (bwxor*
                     (bwxor* (shl* (read* 'd11) 'in_a_bal_rcvd) (bv #xff 8))
                     (bwor*
                      (shl* (bv #x01 8) 'in_a_bal_rcvd)
                      (bwxor* 'in_a_bal_vals (bv #xff 8)))))
                   (:=* 'in_a_bal_rcvd (add* (bv #x01 8) 'in_a_bal_rcvd))
                   (write* 'd10 (read* 'd9)))
```

```
                (list
                 (if*
                  (bwand*
                   (bwor*
                    (bwand* (eq* (read* 'd9) (bv #x00 8)) (read* 'd10))
                    (and* (read* 'd9) (shl* (bv #x80 8) (read* 'd10))))
                   (and*
                    (bwand* 'in_a_bal_rcvd (bv #xf8 8))
                    (and*
                     (eq* (bv #x03 8) 'phase)
                     (lt* (bwand* (bv #xf8 8) 'in_a_val_rcvd) (bv #x02 8)))))
                  (list
                   (:=*
                    'in_a_val_vals
                    (bwxor*
                     (bwor*
                      (bwxor* (bv #xff 8) 'in_a_val_vals)
                      (shl* (bv #x01 8) 'in_a_val_rcvd))
                     (bwxor* (bv #xff 8) (shl* (read* 'd11) 'in_a_val_rcvd))))
                   (:=* 'in_a_val_rcvd (add* 'in_a_val_rcvd (bv #x01 8)))
                   (write* 'd10 (read* 'd9)))
                  (list
                   (if*
                    (and*
                     (bwand*
                      (eq* (bv #x00 8) (bwand* (bv #xf8 8) 'in_b_bal_rcvd))
                      (eq* (bv #x03 8) 'phase))
                     (bwor*
                      (shr* (read* 'd12) (read* 'd13))
                      (shr* (read* 'd13) (read* 'd12))))
                    (list
                     (:=*
                      'in_b_bal_vals
                      (bwxor*
                       (bwxor* (shl* (read* 'd14) 'in_b_bal_rcvd) (bv #xff 8))
                       (bwor*
                        (shl* (bv #x01 8) 'in_b_bal_rcvd)
                        (bwxor* 'in_b_bal_vals (bv #xff 8)))))
                     (:=* 'in_b_bal_rcvd (add* (bv #x01 8) 'in_b_bal_rcvd))
                     (write* 'd13 (read* 'd12)))
                    (list
                     (if*
                      (and*
                       (and*
                        (bwand* (bv #xf8 8) 'in_b_bal_rcvd)
                        (shl* (eq* 'phase (bv #x03 8)) 'in_b_val_rcvd))
                       (bwxor* (read* 'd13) (read* 'd12)))
                      (list
                       (write* 'd13 (read* 'd12))
                       (:=*
                        'in_b_val_vals
                        (bwxor*
                         (bwor*
                          (shl* (bv #x01 8) 'in_b_val_rcvd)
                          (bwxor* 'in_b_val_vals (bv #xff 8)))
                         (bwxor*
                          (bv #xff 8)
                          (shl* (read* 'd14) 'in_b_val_rcvd))))
                       (:=* 'in_b_val_rcvd (add* (bv #x01 8) 'in_b_val_rcvd)))
                      (list
                       (if*
                        (bwand*
                         (bwxor* (read* 'd15) (read* 'd16))
```

```
                      (bwand*
                       (lt* (bwand* (bv #xf8 8) 'in_c_bal_rcvd) (bv #x02 8))
                       (eq* (bv #x03 8) 'phase)))
                     (list
                      (:=*
                       'in_c_bal_vals
                       (bwxor*
                        (bwor*
                         (shl* (bv #x01 8) 'in_c_bal_rcvd)
                         (bwxor* 'in_c_bal_vals (bv #xff 8)))
                        (bwxor*
                         (bv #xff 8)
                         (shl* (read* 'd17) 'in_c_bal_rcvd))))
                      (:=* 'in_c_bal_rcvd (add* 'in_c_bal_rcvd (bv #x01 8)))
                      (write* 'd16 (read* 'd15)))
                     (list
                      (if*
                       (bwand*
                        (and*
                         (bwand*
                          (lt*
                           (bwand* (bv #xf8 8) 'in_c_val_rcvd)
                           (bv #x02 8))
                          (eq* (bv #x03 8) 'phase))
                         (bwand* 'in_c_bal_rcvd (bv #xf8 8)))
                        (bwor*
                         (bwand*
                          (eq* (read* 'd15) (bv #x00 8))
                          (read* 'd16))
                         (and*
                          (eq* (read* 'd16) (bv #x00 8))
                          (read* 'd15))))
                       (list
                        (write* 'd16 (read* 'd15))
                        (:=*
                         'in_c_val_vals
                         (bwxor*
                          (bwor*
                           (bwxor* (bv #xff 8) 'in_c_val_vals)
                           (shl* (bv #x01 8) 'in_c_val_rcvd))
                          (bwxor*
                           (shl* (read* 'd17) 'in_c_val_rcvd)
                           (bv #xff 8))))
                        (:=*
                         'in_c_val_rcvd
                         (add* (bv #x01 8) 'in_c_val_rcvd)))
                       (list
                        (if*
                         (and*
                          (bwand* (bv #xf8 8) 'in_a_val_rcvd)
                          (and*
                           (and*
                            (eq* (bv #x03 8) 'phase)
                            (shr* 'in_c_val_rcvd (bv #x03 8)))
                           (shr* 'in_b_val_rcvd (bv #x03 8))))
                         (list
                          (:=* 'b_mbal 'in_b_bal_vals)
                          (:=* 'a_mval 'in_a_val_vals)
                          (:=* 'c_mval 'in_c_val_vals)
                          (:=* 'c_mbal 'in_c_bal_vals)
                          (:=* 'phase (bv #x04 8))
                          (:=* 'a_mbal 'in_a_bal_vals)
                          (:=* 'b_mval 'in_b_val_vals))
```

```
                        (list
                         (if*
                          (shr*
                           (shr*
                            (lt* 'c_mbal (eq* (bv #x04 8) 'phase))
                            'b_mbal)
                           'a_mbal)
                          (list (:=* 'phase (bv #x05 8)))
                          (list
                           (if*
                            (bwand*
                             (bwor*
                              (lt* 'b_mbal 'a_mbal)
                              (eq* 'a_mbal 'b_mbal))
                             (and*
                              (eq* (bv #x04 8) 'phase)
                              (bwxor*
                               (lt* 'c_mbal 'a_mbal)
                               (eq* 'c_mbal 'a_mbal))))
                            (list
                             (:=* 'value 'a_mval)
                             (:=* 'phase (bv #x05 8)))
                            (list
                             (if*
                              (and*
                               (bwxor*
                                (eq* 'c_mbal 'b_mbal)
                                (lt* 'c_mbal 'b_mbal))
                               (eq* (bv #x04 8) 'phase))
                              (list
                               (:=* 'value 'b_mval)
                               (:=* 'phase (bv #x05 8)))
                              (list
                               (if*
                                (eq* (bv #x04 8) 'phase)
                                (list
                                 (:=* 'value 'c_mval)
                                 (:=* 'phase (bv #x05 8)))
                                (list
                                 (if*
                                  (eq* 'phase (bv #x05 8))
                                  (list
                                   (:=* 'out_c_bal_vals 'ballot)
                                   (:=* 'out_c_bal_sent (bv #x00 8))
                                   (:=* 'out_c_val_vals 'value)
                                   (:=* 'out_c_val_sent (bv #x00 8))
                                   (:=* 'out_a_bal_vals 'ballot)
                                   (:=* 'out_a_bal_sent (bv #x00 8))
                                   (:=* 'out_a_val_vals 'value)
                                   (:=* 'out_a_val_sent (bv #x00 8))
                                   (:=* 'out_b_val_vals 'value)
                                   (:=* 'out_b_val_sent (bv #x00 8))
                                   (:=* 'out_b_bal_vals 'ballot)
                                   (:=* 'out_b_bal_sent (bv #x00 8))
                                   (:=* 'phase (bv #x06 8)))
                                  (list
                                   (if*
                                    (bwand*
                                     (and*
                                      (shl* (bv #x01 8) 'out_a_bal_sent)
                                      (eq* (bv #x06 8) 'phase))
                                     (eq* (read* 'd1) (read* 'd0)))
                                    (list
```

```
                                (write*
                                 'd2
                                 (bwxor*
                                  (bv #x01 8)
                                  (shr*
                                   (bv #x01 8)
                                   (bwand*
                                    (shl* (bv #x01 8) 'out_a_bal_sent)
                                    'out_a_bal_vals))))
                                (write*
                                 'd0
                                 (eq* (read* 'd0) (bv #x00 8)))
                                (:=*
                                 'out_a_bal_sent
                                 (add* 'out_a_bal_sent (bv #x01 8))))
                               (list
                                (if*
                                 (bwand*
                                  (and*
                                   (and*
                                    (shr* (bv #xb6 8) 'out_a_val_sent)
                                    (eq* 'phase (bv #x06 8)))
                                   (bwand*
                                    (bv #xf8 8)
                                    'out_a_bal_sent))
                                  (eq* (read* 'd1) (read* 'd0)))
                                 (list
                                  (write*
                                   'd2
                                   (bwxor*
                                    (shr*
                                     (bv #x01 8)
                                     (bwand*
                                      (bv #x01 8)
                                      (shr*
                                       'out_a_val_vals
                                       'out_a_val_sent)))
                                    (bv #x01 8)))
                                  (write*
                                   'd0
                                   (eq* (read* 'd0) (bv #x00 8)))
                                  (:=*
                                   'out_a_val_sent
                                   (add*
                                    (bv #x01 8)
                                    'out_a_val_sent)))
                                 (list
                                  (if*
                                   (and*
                                    (bwand*
                                     (lt*
                                      (bwand*
                                       (bv #xf8 8)
                                       'out_b_bal_sent)
                                      (bv #x02 8))
                                     (eq* (bv #x06 8) 'phase))
                                    (eq* (read* 'd3) (read* 'd4)))
                                   (list
                                    (write*
                                     'd5
                                     (eq*
                                      (bv #x00 8)
                                      (eq*
```

```
                    (bv #x00 8)
                    (bwand*
                     'out_b_bal_vals
                     (shl*
                      (bv #x01 8)
                      'out_b_bal_sent)))))
                  (write*
                   'd3
                   (eq* (read* 'd3) (bv #x00 8)))
                  (:=*
                   'out_b_bal_sent
                   (add*
                    (bv #x01 8)
                    'out_b_bal_sent)))
                 (list
                  (if*
                   (and*
                    (and*
                     (lt*
                      (bv #x07 8)
                      'out_b_bal_sent)
                     (and*
                      (eq* 'phase (bv #x06 8))
                      (lt*
                       (bwand*
                        (bv #xf8 8)
                        'out_b_val_sent)
                       (bv #x02 8))))
                    (eq* (read* 'd4) (read* 'd3)))
                   (list
                    (write*
                     'd5
                     (eq*
                      (bv #x00 8)
                      (shr*
                       (bv #x01 8)
                       (bwand*
                        (shr*
                         'out_b_val_vals
                         'out_b_val_sent)
                        (bv #x01 8)))))
                    (write*
                     'd3
                     (eq* (read* 'd3) (bv #x00 8)))
                    (:=*
                     'out_b_val_sent
                     (add*
                      (bv #x01 8)
                      'out_b_val_sent)))
                   (list
                    (if*
                     (bwand*
                      (eq* (read* 'd6) (read* 'd7))
                      (and*
                       (shl*
                        (bv #x25 8)
                        'out_c_bal_sent)
                       (eq* (bv #x06 8) 'phase)))
                     (list
                      (write*
                       'd8
                       (bwxor*
                        (shr*
```

```
                (bv #x01 8)
                (bwand*
                 (bv #x01 8)
                 (shr*
                  'out_c_bal_vals
                  'out_c_bal_sent)))
                (bv #x01 8)))
              (write*
               'd6
               (eq*
                (read* 'd7)
                (bv #x00 8)))
              (:=*
               'out_c_bal_sent
               (add*
                (bv #x01 8)
                'out_c_bal_sent)))
             (list
              (if*
               (bwand*
                (eq*
                 (bv #x00 8)
                 (bwxor*
                  (read* 'd7)
                  (read* 'd6)))
                (and*
                 (bwand*
                  (lt*
                   (bwand*
                    (bv #xf8 8)
                    'out_c_val_sent)
                   (bv #x02 8))
                  (eq* 'phase (bv #x06 8)))
                 (shr*
                  'out_c_bal_sent
                  (bv #x03 8))))
               (list
                (write*
                 'd8
                 (lt*
                  (shr*
                   (bv #x01 8)
                   (bwand*
                    (shl*
                     (bv #x01 8)
                     'out_c_val_sent)
                    'out_c_val_vals))
                  (bv #x01 8)))
                (write*
                 'd6
                 (eq*
                  (read* 'd6)
                  (bv #x00 8)))
                (:=*
                 'out_c_val_sent
                 (add*
                  (bv #x01 8)
                  'out_c_val_sent)))
               (list
                (if*
                 (and*
                  (and*
                   (bwand*
```

```
    (bv #xf8 8)
    'out_b_val_sent)
   (and*
    (eq* (bv #x06 8) 'phase)
    (bwand*
     (bv #xf8 8)
     'out_c_val_sent)))
  (bwand*
   (bv #xf8 8)
   'out_a_val_sent))
 (list
  (:=*
   'in_c_bal_vals
   (bv #x00 8))
  (:=*
   'in_c_bal_rcvd
   (bv #x00 8))
  (:=*
   'in_b_bal_vals
   (bv #x00 8))
  (:=*
   'in_b_bal_rcvd
   (bv #x00 8))
  (:=*
   'in_b_val_vals
   (bv #x00 8))
  (:=*
   'in_b_val_rcvd
   (bv #x00 8))
  (:=*
   'in_a_bal_vals
   (bv #x00 8))
  (:=*
   'in_a_bal_rcvd
   (bv #x00 8))
  (:=* 'phase (bv #x07 8))
  (:=*
   'in_a_val_vals
   (bv #x00 8))
  (:=*
   'in_a_val_rcvd
   (bv #x00 8))
  (:=*
   'in_c_val_vals
   (bv #x00 8))
  (:=*
   'in_c_val_rcvd
   (bv #x00 8)))
 (list
  (if*
   (and*
    (shl*
     (eq*
      'phase
      (bv #x07 8))
     'in_a_bal_rcvd)
    (bwor*
     (bwand*
      (read* 'd10)
      (bwxor*
       (bv #x01 8)
       (read* 'd9)))
     (lt*
```

```
    (read* 'd10)
    (read* 'd9))))
 (list
  (:=*
   'in_a_bal_vals
   (bwxor*
    (bwor*
     (bwxor*
      'in_a_bal_vals
      (bv #xff 8))
     (shl*
      (bv #x01 8)
      'in_a_bal_rcvd))
    (bwxor*
     (shl*
      (read* 'd11)
      'in_a_bal_rcvd)
     (bv #xff 8))))
  (:=*
   'in_a_bal_rcvd
   (add*
    (bv #x01 8)
    'in_a_bal_rcvd))
  (write*
   'd10
   (read* 'd9)))
 (list
  (if*
   (and*
    (bwxor*
     (read* 'd10)
     (read* 'd9))
    (and*
     (shl*
      (eq*
       (bv #x07 8)
       'phase)
      'in_a_val_rcvd)
     (bwand*
      (bv #xf8 8)
      'in_a_bal_rcvd)))
   (list
    (write*
     'd10
     (read* 'd9))
    (:=*
     'in_a_val_vals
     (bwxor*
      (bwor*
       (shl*
        (bv #x01 8)
        'in_a_val_rcvd)
       (bwxor*
        (bv #xff 8)
        'in_a_val_vals))
      (bwxor*
       (shl*
        (read* 'd11)
        'in_a_val_rcvd)
       (bv #xff 8))))
    (:=*
     'in_a_val_rcvd
     (bwxor*
```

```
                (bv #xff 8)
                (add*
                 (bwxor*
                  'in_a_val_rcvd
                  (bv #xff 8))
                 (bv #xff 8)))))
            (list
             (if*
              (and*
               (bwxor*
                (read* 'd12)
                (read* 'd13))
               (shl*
                (eq*
                 'phase
                 (bv #x07 8))
                'in_b_bal_rcvd))
              (list
               (write*
                'd13
                (read* 'd12))
               (:=*
                'in_b_bal_vals
                (bwxor*
                 (bwor*
                  (shl*
                   (bv #x01 8)
                   'in_b_bal_rcvd)
                  (bwxor*
                   (bv #xff 8)
                   'in_b_bal_vals))
                 (bwxor*
                  (shl*
                   (read* 'd14)
                   'in_b_bal_rcvd)
                  (bv #xff 8))))
               (:=*
                'in_b_bal_rcvd
                (add*
                 'in_b_bal_rcvd
                 (bv #x01 8))))
              (list
               (if*
                (bwand*
                 (bwxor*
                  (read* 'd12)
                  (read* 'd13))
                 (and*
                  (bwand*
                   (bv #xf8 8)
                   'in_b_bal_rcvd)
                  (shl*
                   (eq*
                    'phase
                    (bv #x07 8))
                   'in_b_val_rcvd)))
                (list
                 (write*
                  'd13
                  (read* 'd12))
                 (:=*
                  'in_b_val_vals
                  (bwxor*
```

```
                                                  (bwxor*
                                                   (shl*
                                                    (read* 'd14)
                                                    'in_b_val_rcvd)
                                                   (bv #xff 8))
                                                  (bwor*
                                                   (shl*
                                                    (bv #x01 8)
                                                    'in_b_val_rcvd)
                                                   (bwxor*
                                                    'in_b_val_vals
                                                    (bv #xff 8)))))
                                                 (:=*
                                                  'in_b_val_rcvd
                                                  (add*
                                                   (bv #x01 8)
                                                   'in_b_val_rcvd)))
                                                (list
                                                 (if*
                                                  (bwand*
                                                   (bwand*
                                                    (eq*
                                                     (bv #x07 8)
                                                     'phase)
                                                    (eq*
                                                     (bv #x00 8)
                                                     (bwand*
                                                      'in_c_bal_rcvd
                                                      (bv #xf8 8))))
                                                   (bwxor*
                                                    (read* 'd15)
                                                    (read* 'd16)))
                                                  (list
                                                   (:=*
                                                    'in_c_bal_vals
                                                    (bwxor*
                                                     (bwor*
                                                      (shl*
                                                       (bv #x01 8)
                                                       'in_c_bal_rcvd)
                                                      'in_c_bal_vals)
                                                     (shl*
                                                      (shr*
                                                       (bv #x01 8)
                                                       (read*
                                                        'd17))
                                                      'in_c_bal_rcvd)))
                                                   (:=*
                                                    'in_c_bal_rcvd
                                                    (add*
                                                     'in_c_bal_rcvd
                                                     (bv #x01 8)))
                                                   (write*
                                                    'd16
                                                    (read* 'd15)))
                                                  (list
                                                   (if*
                                                    (bwand*
                                                     (and*
                                                      (bwand*
                                                       (bv #xf8 8)
                                                       'in_c_bal_rcvd)
                                                      (and*
```

```
                                    (shl*
                                     (bv #x4d 8)
                                     'in_c_val_rcvd)
                                    (eq*
                                     (bv #x07 8)
                                     'phase)))
                                  (bwxor*
                                   (read* 'd16)
                                   (read*
                                    'd15)))
                                 (list
                                  (:=*
                                   'in_c_val_vals
                                   (bwxor*
                                    (shl*
                                     (shr*
                                      (bv #x01 8)
                                      (read*
                                       'd17))
                                     'in_c_val_rcvd)
                                    (bwor*
                                     'in_c_val_vals
                                     (shl*
                                      (bv #x01 8)
                                      'in_c_val_rcvd))))
                                  (:=*
                                   'in_c_val_rcvd
                                   (add*
                                    (bv #x01 8)
                                    'in_c_val_rcvd))
                                  (write*
                                   'd16
                                   (read*
                                    'd15)))
                                 (list
                                  (if*
                                   (and*
                                    (bwand*
                                     (bv #xf8 8)
                                     'in_a_val_rcvd)
                                    (and*
                                     (and*
                                      (bwand*
                                       (bv #xf8 8)
                                       'in_c_val_rcvd)
                                      (eq*
                                       (bv #x07 8)
                                       'phase))
                                     (bwand*
                                      (bv #xf8 8)
                                      'in_b_val_rcvd)))
                                   (list
                                    (:=*
                                     'b_mbal
                                     'in_b_bal_vals)
                                    (:=*
                                     'b_mval
                                     'in_b_val_vals)
                                    (:=*
                                     'c_mbal
                                     'in_c_bal_vals)
                                    (:=*
                                     'a_mbal
```

```
                                                                  'in_a_bal_vals)
                                                                 (:=*
                                                                  'phase
                                                                  (bv #x08 8))
                                                                 (:=*
                                                                  'a_mval
                                                                  'in_a_val_vals)
                                                                 (:=*
                                                                  'c_mval
                                                                  'in_c_val_vals))
                                                                (list
                                                                 (if*
                                                                  (and*
                                                                   (bwand*
                                                                    (bwand*
                                                                     (eq*
                                                                      'phase
                                                                      (bv #x08 8))
                                                                     (eq*
                                                                      'value
                                                                      'c_mval))
                                                                    (eq*
                                                                     'value
                                                                     'b_mval))
                                                                   (eq*
                                                                    'value
                                                                    'a_mval))
                                                                  (list
                                                                   (:=*
                                                                    'phase
                                                                    (bv #x00 8)))
                                                                  (list
                                                                   (if*
                                                                    (eq*
                                                                     (bv #x08 8)
                                                                     'phase)
                                                                    (list
                                                                     (:=*
                                                                      'phase
                                                                      (bv #xff 8)))
                                                                    '())))))))))))))))))))))))))))))))))))))))))))))
                                                                          ↪ ))))))))))))))))))))))))))))))))))
                                                                          ↪ ))
```

A.3 Verilog implementations of Paxos

A.3.1 Acceptor

```
(verilog-module*
 'acceptor
 '(d3 d4 d5 d0 d1 d2 clock reset)
 (list
  (reg* 8 'in_prop_val_rcvd)
  (reg* 8 'in_prop_val_vals)
  (reg* 8 'in_prop_bal_rcvd)
  (reg* 8 'in_prop_bal_vals)
  (reg* 8 'out_prop_val_sent)
  (reg* 8 'out_prop_val_vals)
```

```
   (reg* 8 'out_prop_bal_sent)
   (reg* 8 'out_prop_bal_vals)
   (input* (wire* 1 'd3))
   (output* (reg* 1 'd4))
   (input* (wire* 1 'd5))
   (output* (reg* 1 'd0))
   (input* (wire* 1 'd1))
   (output* (reg* 1 'd2))
   (reg* 8 'prop_mval)
   (reg* 8 'prop_mbal)
   (reg* 8 'prom_bal)
   (reg* 8 'phase)
   (reg* 8 'value)
   (reg* 8 'ballot)
   (input* (wire* 1 'clock))
   (input* (wire* 1 'reset)))
  (list
   (always*
    (or* (posedge* 'clock) (posedge* 'reset))
    (list
     (if*
      'reset
      (list
       (<=* 'in_prop_bal_vals (bv #x00 8))
       (<=* 'in_prop_bal_rcvd (bv #x00 8))
       (<=* 'prom_bal (bv #x00 8))
       (<=* 'phase (bv #x01 8))
       (<=* 'value (bv #x00 8))
       (<=* 'ballot (bv #x00 8)))
      (list
       (if*
        (bweq* (bv #x01 8) 'phase)
        (list
         (<=* 'phase (bv #x02 8))
         (<=* 'in_prop_bal_vals (bv #x00 8))
         (<=* 'in_prop_bal_rcvd (bv #x00 8)))
        (list
         (if*
          (and*
           (or* (and* (eq* #f 'd4) 'd3) (and* (not* 'd3) 'd4))
           (and*
            (eq* #f (lt* (bv #x07 8) 'in_prop_bal_rcvd))
            (bweq* (bv #x02 8) 'phase)))
          (list
           (<=*
            'in_prop_bal_vals
            (bwxor*
             (bwor*
              (bwnot* 'in_prop_bal_vals)
              (shl* (bv #x01 8) 'in_prop_bal_rcvd))
             (bwnot* (shl* (bool->vect* 'd5) 'in_prop_bal_rcvd))))
           (<=*
            'in_prop_bal_rcvd
            (bwnot* (add* (bwnot* 'in_prop_bal_rcvd) (bv #xff 8))))
           (<=* 'd4 'd3))
          (list
           (if*
            (and*
             (lt* (bv #x07 8) 'in_prop_bal_rcvd)
             (bweq* (bv #x02 8) 'phase))
            (list (<=* 'phase (bv #x03 8)) (<=* 'prop_mbal 'in_prop_bal_vals))
            (list
             (if*
```

```
               (and*
                (lt* 'ballot 'prop_mbal)
                (and* (bweq* 'phase (bv #x03 8)) (lt* 'prom_bal 'prop_mbal)))
               (list
                (<=* 'phase (bv #x04 8))
                (<=* 'prom_bal 'prop_mbal)
                (<=* 'out_prop_val_vals 'value)
                (<=* 'out_prop_val_sent (bv #x00 8))
                (<=* 'out_prop_bal_vals 'ballot)
                (<=* 'out_prop_bal_sent (bv #x00 8)))
               (list
                (if*
                 (and*
                  (and*
                   (bweq* (bv #x04 8) 'phase)
                   (eq* #f (lt* (bv #x07 8) 'out_prop_bal_sent)))
                  (not* (or* (and* 'd0 (eq* 'd1 #f)) (and* 'd1 (eq* #f 'd0)))))
                 (list
                  (<=* 'out_prop_val_vals 'out_prop_val_vals)
                  (<=* 'out_prop_val_sent 'out_prop_val_sent)
                  (<=* 'out_prop_bal_vals 'out_prop_bal_vals)
                  (<=* 'out_prop_bal_sent (add* 'out_prop_bal_sent (bv #x01 8)))
                  (<=*
                   'd2
                   (eq*
                    (bwand*
                     'out_prop_bal_vals
                     (shl* (bv #x01 8) 'out_prop_bal_sent))
                    (shl* (bv #x01 8) 'out_prop_bal_sent)))
                  (<=* 'd0 (not* 'd1)))
                 (list
                  (if*
                   (and*
                    (not* (or* (and* (eq* #f 'd0) 'd1) (and* 'd0 (eq* #f 'd1))))
                    (and*
                     (lt* (bv #x07 8) 'out_prop_bal_sent)
                     (and*
                      (eq* #f (lt* (bv #x07 8) 'out_prop_val_sent))
                      (bweq* (bv #x04 8) 'phase))))
                   (list
                    (<=* 'out_prop_val_vals 'out_prop_val_vals)
                    (<=*
                     'out_prop_val_sent
                     (add* 'out_prop_val_sent (bv #x01 8)))
                    (<=* 'out_prop_bal_vals 'out_prop_bal_vals)
                    (<=* 'out_prop_bal_sent 'out_prop_bal_sent)
                    (<=*
                     'd2
                     (eq*
                      (bv #x00 8)
                      (bwand*
                       (shl* (bv #x01 8) 'out_prop_val_sent)
                       (bwnot* 'out_prop_val_vals))))
                    (<=* 'd0 (not* 'd1)))
                   (list
                    (if*
                     (and*
                      (bweq* (bv #x04 8) 'phase)
                      (lt* (bv #x07 8) 'out_prop_val_sent))
                     (list
                      (<=* 'phase (bv #x05 8))
                      (<=* 'in_prop_val_vals (bv #x00 8))
                      (<=* 'in_prop_val_rcvd (bv #x00 8))
```

```
                  (<=* 'in_prop_bal_vals (bv #x00 8))
                  (<=* 'in_prop_bal_rcvd (bv #x00 8)))
                 (list
                  (if*
                   (and*
                    (and*
                     (eq* #f (lt* (bv #x07 8) 'in_prop_bal_rcvd))
                     (bweq* 'phase (bv #x05 8)))
                    (or* (and* (eq* 'd3 #f) 'd4) (and* 'd3 (eq* 'd4 #f))))
                   (list
                    (<=* 'in_prop_val_vals 'in_prop_val_vals)
                    (<=* 'in_prop_val_rcvd 'in_prop_val_rcvd)
                    (<=*
                     'in_prop_bal_vals
                     (bwxor*
                      (bwnot* (shl* (bool->vect* 'd5) 'in_prop_bal_rcvd))
                      (bwor*
                       (shl* (bv #x01 8) 'in_prop_bal_rcvd)
                       (bwnot* 'in_prop_bal_vals))))
                    (<=*
                     'in_prop_bal_rcvd
                     (add* 'in_prop_bal_rcvd (bv #x01 8)))
                    (<=* 'd4 'd3))
                   (list
                    (if*
                     (and*
                      (or* (and* 'd4 (eq* 'd3 #f)) (and* 'd3 (eq* 'd4 #f)))
                      (and*
                       (and*
                        (eq* #f (lt* (bv #x07 8) 'in_prop_val_rcvd))
                        (bweq* 'phase (bv #x05 8)))
                       (lt* (bv #x07 8) 'in_prop_bal_rcvd)))
                     (list
                      (<=*
                       'in_prop_val_vals
                       (bwxor*
                        (bwor*
                         (bwxor* (bv #xff 8) 'in_prop_val_vals)
                         (shl* (bv #x01 8) 'in_prop_val_rcvd))
                        (bwxor*
                         (bv #xff 8)
                         (shl* (bool->vect* 'd5) 'in_prop_val_rcvd))))
                      (<=*
                       'in_prop_val_rcvd
                       (add* 'in_prop_val_rcvd (bv #x01 8)))
                      (<=* 'in_prop_bal_vals 'in_prop_bal_vals)
                      (<=* 'in_prop_bal_rcvd 'in_prop_bal_rcvd)
                      (<=* 'd4 'd3))
                     (list
                      (if*
                       (and*
                        (bweq* (bv #x05 8) 'phase)
                        (lt* (bv #x07 8) 'in_prop_val_rcvd))
                       (list
                        (<=* 'phase (bv #x06 8))
                        (<=* 'prop_mval 'in_prop_val_vals)
                        (<=* 'prop_mbal 'in_prop_bal_vals))
                       (list
                        (if*
                         (and*
                          (bweq* 'phase (bv #x06 8))
                          (bweq* 'prom_bal 'prop_mbal))
                         (list
```

```
                        (<=* 'phase (bv #x07 8))
                        (<=* 'out_prop_val_vals 'prop_mval)
                        (<=* 'out_prop_val_sent (bv #x00 8))
                        (<=* 'out_prop_bal_vals 'prom_bal)
                        (<=* 'out_prop_bal_sent (bv #x00 8))
                        (<=* 'value 'prop_mval)
                        (<=* 'ballot 'prom_bal))
                       (list
                        (if*
                         (and*
                          (and*
                           (bweq* 'phase (bv #x07 8))
                           (eq* #f (lt* (bv #x07 8) 'out_prop_bal_sent)))
                          (eq* 'd0 'd1))
                         (list
                          (<=* 'out_prop_val_vals 'out_prop_val_vals)
                          (<=* 'out_prop_val_sent 'out_prop_val_sent)
                          (<=* 'out_prop_bal_vals 'out_prop_bal_vals)
                          (<=*
                           'out_prop_bal_sent
                           (add* 'out_prop_bal_sent (bv #x01 8)))
                          (<=*
                           'd2
                           (eq*
                            #f
                            (eq*
                             (bwand*
                              (bv #x01 8)
                              (shr* 'out_prop_bal_vals 'out_prop_bal_sent))
                             (bv #x00 8))))
                          (<=* 'd0 (eq* 'd1 #f)))
                         (list
                          (if*
                           (and*
                            (and*
                             (and*
                              (bweq* 'phase (bv #x07 8))
                              (eq*
                               #f
                               (lt* (bv #x07 8) 'out_prop_val_sent)))
                             (lt* (bv #x07 8) 'out_prop_bal_sent))
                            (eq* 'd1 'd0))
                           (list
                            (<=* 'out_prop_val_vals 'out_prop_val_vals)
                            (<=*
                             'out_prop_val_sent
                             (add* (bv #x01 8) 'out_prop_val_sent))
                            (<=* 'out_prop_bal_vals 'out_prop_bal_vals)
                            (<=* 'out_prop_bal_sent 'out_prop_bal_sent)
                            (<=*
                             'd2
                             (eq*
                              (bv #x00 8)
                              (bwand*
                               (shl* (bv #x01 8) 'out_prop_val_sent)
                               (bwxor* 'out_prop_val_vals (bv #xff 8)))))
                            (<=* 'd0 (eq* 'd1 #f)))
                           (list
                            (if*
                             (and*
                              (lt* (bv #x07 8) 'out_prop_val_sent)
                              (bweq* 'phase (bv #x07 8)))
                             (list (<=* 'phase (bv #x00 8)))
```

```
                                  '()))))))))))))))))))))))))))))))))))))
```

A.3.2 Proposer

```
(verilog-module*
 'proposer
 '(d15 d16 d17 d12 d13 d14 d9 d10 d11 d6 d7 d8 d3 d4 d5 d0 d1 d2 clock reset)
 (list
  (reg* 8 'in_c_val_rcvd)
  (reg* 8 'in_c_val_vals)
  (reg* 8 'in_b_val_rcvd)
  (reg* 8 'in_b_val_vals)
  (reg* 8 'in_a_val_rcvd)
  (reg* 8 'in_a_val_vals)
  (reg* 8 'in_c_bal_rcvd)
  (reg* 8 'in_c_bal_vals)
  (reg* 8 'in_b_bal_rcvd)
  (reg* 8 'in_b_bal_vals)
  (reg* 8 'in_a_bal_rcvd)
  (reg* 8 'in_a_bal_vals)
  (reg* 8 'out_c_val_sent)
  (reg* 8 'out_c_val_vals)
  (reg* 8 'out_b_val_sent)
  (reg* 8 'out_b_val_vals)
  (reg* 8 'out_a_val_sent)
  (reg* 8 'out_a_val_vals)
  (reg* 8 'out_c_bal_sent)
  (reg* 8 'out_c_bal_vals)
  (reg* 8 'out_b_bal_sent)
  (reg* 8 'out_b_bal_vals)
  (reg* 8 'out_a_bal_sent)
  (reg* 8 'out_a_bal_vals)
  (input* (wire* 1 'd15))
  (output* (reg* 1 'd16))
  (input* (wire* 1 'd17))
  (input* (wire* 1 'd12))
  (output* (reg* 1 'd13))
  (input* (wire* 1 'd14))
  (input* (wire* 1 'd9))
  (output* (reg* 1 'd10))
  (input* (wire* 1 'd11))
  (output* (reg* 1 'd6))
  (input* (wire* 1 'd7))
  (output* (reg* 1 'd8))
  (output* (reg* 1 'd3))
  (input* (wire* 1 'd4))
  (output* (reg* 1 'd5))
  (output* (reg* 1 'd0))
  (input* (wire* 1 'd1))
  (output* (reg* 1 'd2))
  (reg* 8 'c_mval)
  (reg* 8 'b_mval)
  (reg* 8 'a_mval)
  (reg* 8 'c_mbal)
  (reg* 8 'b_mbal)
  (reg* 8 'a_mbal)
  (reg* 8 'phase)
  (reg* 8 'value)
  (reg* 8 'ballot)
  (input* (wire* 1 'clock))
```

```
  (input* (wire* 1 'reset)))
 (list
  (always*
   (or* (posedge* 'clock) (posedge* 'reset))
   (list
    (if*
     'reset
     (list
      (<=* 'phase (bv #x01 8))
      (<=* 'value (bv #x20 8))
      (<=* 'ballot (bv #x01 8)))
     (list
      (if*
       (bweq* 'ballot (bv #xff 8))
       (list (<=* 'phase (bv #xff 8)))
       (list
        (if*
         (bweq* (bv #x01 8) 'phase)
         (list
          (<=* 'phase (bv #x02 8))
          (<=* 'out_c_bal_vals 'ballot)
          (<=* 'out_c_bal_sent (bv #x00 8))
          (<=* 'out_b_bal_vals 'ballot)
          (<=* 'out_b_bal_sent (bv #x00 8))
          (<=* 'out_a_bal_vals 'ballot)
          (<=* 'out_a_bal_sent (bv #x00 8)))
         (list
          (if*
           (and*
            (and*
             (bweq* 'phase (bv #x02 8))
             (eq* #f (lt* (bv #x07 8) 'out_a_bal_sent)))
            (not* (or* (and* 'd0 (eq* #f 'd1)) (and* (eq* #f 'd0) 'd1))))
           (list
            (<=* 'out_a_bal_vals 'out_a_bal_vals)
            (<=* 'out_a_bal_sent (add* 'out_a_bal_sent (bv #x01 8)))
            (<=*
             'd2
             (eq*
              (eq*
               (bv #x00 8)
               (bwand* 'out_a_bal_vals (shl* (bv #x01 8) 'out_a_bal_sent)))
              #f))
            (<=* 'd0 (eq* 'd1 #f)))
           (list
            (if*
             (and*
              (and*
               (eq* #f (lt* (bv #x07 8) 'out_b_bal_sent))
               (bweq* 'phase (bv #x02 8)))
              (eq* 'd4 'd3))
             (list
              (<=* 'out_b_bal_vals 'out_b_bal_vals)
              (<=* 'out_b_bal_sent (add* 'out_b_bal_sent (bv #x01 8)))
              (<=*
               'd5
               (eq*
                (bv #xff 8)
                (bwor* (bv #xfe 8) (shr* 'out_b_bal_vals 'out_b_bal_sent))))
              (<=* 'd3 (eq* 'd4 #f)))
             (list
              (if*
               (and*
```

```
                (and*
                 (bweq* (bv #x02 8) 'phase)
                 (eq* #f (lt* (bv #x07 8) 'out_c_bal_sent)))
                (not* (or* (and* 'd6 (not* 'd7)) (and* (eq* #f 'd6) 'd7))))
               (list
                (<=* 'out_c_bal_vals 'out_c_bal_vals)
                (<=* 'out_c_bal_sent (add* 'out_c_bal_sent (bv #x01 8)))
                (<=*
                 'd8
                 (eq*
                  (bwand*
                   (bwnot* 'out_c_bal_vals)
                   (shl* (bv #x01 8) 'out_c_bal_sent))
                  (bv #x00 8)))
                (<=* 'd6 (not* 'd7)))
               (list
                (if*
                 (and*
                  (and*
                   (lt* (bv #x07 8) 'out_b_bal_sent)
                   (and*
                    (bweq* 'phase (bv #x02 8))
                    (lt* (bv #x07 8) 'out_c_bal_sent)))
                  (lt* (bv #x07 8) 'out_a_bal_sent))
                 (list
                  (<=* 'phase (bv #x03 8))
                  (<=* 'in_c_val_vals (bv #x00 8))
                  (<=* 'in_c_val_rcvd (bv #x00 8))
                  (<=* 'in_b_val_vals (bv #x00 8))
                  (<=* 'in_b_val_rcvd (bv #x00 8))
                  (<=* 'in_a_val_vals (bv #x00 8))
                  (<=* 'in_a_val_rcvd (bv #x00 8))
                  (<=* 'in_c_bal_vals (bv #x00 8))
                  (<=* 'in_c_bal_rcvd (bv #x00 8))
                  (<=* 'in_b_bal_vals (bv #x00 8))
                  (<=* 'in_b_bal_rcvd (bv #x00 8))
                  (<=* 'in_a_bal_vals (bv #x00 8))
                  (<=* 'in_a_bal_rcvd (bv #x00 8)))
                 (list
                  (if*
                   (and*
                    (and*
                     (bweq* 'phase (bv #x03 8))
                     (eq* #f (lt* (bv #x07 8) 'in_a_bal_rcvd)))
                    (or* (and* 'd9 (not* 'd10)) (and* 'd10 (eq* 'd9 #f))))
                   (list
                    (<=* 'in_a_val_vals 'in_a_val_vals)
                    (<=* 'in_a_val_rcvd 'in_a_val_rcvd)
                    (<=*
                     'in_a_bal_vals
                     (bwxor*
                      (shl*
                       (bwxor* (bool->vect* 'd11) (bv #x01 8))
                       'in_a_bal_rcvd)
                      (bwor*
                       'in_a_bal_vals
                       (shl* (bv #x01 8) 'in_a_bal_rcvd))))
                    (<=* 'in_a_bal_rcvd (add* (bv #x01 8) 'in_a_bal_rcvd))
                    (<=* 'd10 'd9))
                   (list
                    (if*
                     (and*
                      (and*
```

```
                    (lt* (bv #x07 8) 'in_a_bal_rcvd)
                    (and*
                     (lt* 'in_a_val_rcvd (bv #x08 8))
                     (bweq* (bv #x03 8) 'phase)))
                   (or* (and* 'd9 (eq* #f 'd10)) (and* (eq* 'd9 #f) 'd10)))
                  (list
                   (<=*
                    'in_a_val_vals
                    (bwxor*
                     (bwxor*
                      (shl* (bool->vect* 'd11) 'in_a_val_rcvd)
                      (bv #xff 8))
                     (bwor*
                      (shl* (bv #x01 8) 'in_a_val_rcvd)
                      (bwnot* 'in_a_val_vals))))
                   (<=* 'in_a_val_rcvd (add* 'in_a_val_rcvd (bv #x01 8)))
                   (<=* 'in_a_bal_vals 'in_a_bal_vals)
                   (<=* 'in_a_bal_rcvd 'in_a_bal_rcvd)
                   (<=* 'd10 'd9))
                  (list
                   (if*
                    (and*
                     (or* (and* (not* 'd12) 'd13) (and* (not* 'd13) 'd12))
                     (and*
                      (bweq* (bv #x03 8) 'phase)
                      (eq* #f (lt* (bv #x07 8) 'in_b_bal_rcvd))))
                    (list
                     (<=* 'in_b_val_vals 'in_b_val_vals)
                     (<=* 'in_b_val_rcvd 'in_b_val_rcvd)
                     (<=*
                      'in_b_bal_vals
                      (bwxor*
                       (bwor*
                        (shl* (bv #x01 8) 'in_b_bal_rcvd)
                        (bwnot* 'in_b_bal_vals))
                       (bwxor*
                        (shl* (bool->vect* 'd14) 'in_b_bal_rcvd)
                        (bv #xff 8))))
                     (<=* 'in_b_bal_rcvd (add* 'in_b_bal_rcvd (bv #x01 8)))
                     (<=* 'd13 'd12))
                    (list
                     (if*
                      (and*
                       (and*
                        (and*
                         (eq* #f (lt* (bv #x07 8) 'in_b_val_rcvd))
                         (bweq* (bv #x03 8) 'phase))
                        (lt* (bv #x07 8) 'in_b_bal_rcvd))
                       (or*
                        (and* 'd13 (eq* 'd12 #f))
                        (and* 'd12 (eq* 'd13 #f))))
                      (list
                       (<=*
                        'in_b_val_vals
                        (bwxor*
                         (bwnot* (shl* (bool->vect* 'd14) 'in_b_val_rcvd))
                         (bwor*
                          (shl* (bv #x01 8) 'in_b_val_rcvd)
                          (bwnot* 'in_b_val_vals))))
                       (<=*
                        'in_b_val_rcvd
                        (add* (bv #x01 8) 'in_b_val_rcvd))
                       (<=* 'in_b_bal_vals 'in_b_bal_vals)
```

```
                        (<=* 'in_b_bal_rcvd 'in_b_bal_rcvd)
                        (<=* 'd13 'd12))
                       (list
                        (if*
                         (and*
                          (or*
                           (and* 'd16 (not* 'd15))
                           (and* (eq* 'd16 #f) 'd15))
                          (and*
                           (bweq* (bv #x03 8) 'phase)
                           (eq* #f (lt* (bv #x07 8) 'in_c_bal_rcvd))))
                         (list
                          (<=* 'in_c_val_vals 'in_c_val_vals)
                          (<=* 'in_c_val_rcvd 'in_c_val_rcvd)
                          (<=*
                           'in_c_bal_vals
                           (bwxor*
                            (bwor*
                             (shl* (bv #x01 8) 'in_c_bal_rcvd)
                             'in_c_bal_vals)
                            (shl*
                             (bwxor* (bv #x01 8) (bool->vect* 'd17))
                             'in_c_bal_rcvd)))
                          (<=*
                           'in_c_bal_rcvd
                           (add* (bv #x01 8) 'in_c_bal_rcvd))
                          (<=* 'd16 'd15))
                         (list
                          (if*
                           (and*
                            (or*
                             (and* 'd15 (not* 'd16))
                             (and* 'd16 (not* 'd15)))
                            (and*
                             (lt* (bv #x07 8) 'in_c_bal_rcvd)
                             (and*
                              (eq* #f (lt* (bv #x07 8) 'in_c_val_rcvd))
                              (bweq* 'phase (bv #x03 8)))))
                           (list
                            (<=*
                             'in_c_val_vals
                             (bwxor*
                              (bwand*
                               'in_c_val_vals
                               (shl* (bv #x01 8) 'in_c_val_rcvd))
                              (bwxor*
                               'in_c_val_vals
                               (shl* (bool->vect* 'd17) 'in_c_val_rcvd))))
                            (<=*
                             'in_c_val_rcvd
                             (add* (bv #x01 8) 'in_c_val_rcvd))
                            (<=* 'in_c_bal_vals 'in_c_bal_vals)
                            (<=* 'in_c_bal_rcvd 'in_c_bal_rcvd)
                            (<=* 'd16 'd15))
                           (list
                            (if*
                             (and*
                              (lt* (bv #x07 8) 'in_a_val_rcvd)
                              (and*
                               (lt* (bv #x07 8) 'in_b_val_rcvd)
                               (and*
                                (bweq* (bv #x03 8) 'phase)
                                (lt* (bv #x07 8) 'in_c_val_rcvd))))
```

```
                        (list
                         (<=* 'phase (bv #x04 8))
                         (<=* 'c_mval 'in_c_val_vals)
                         (<=* 'b_mval 'in_b_val_vals)
                         (<=* 'a_mval 'in_a_val_vals)
                         (<=* 'c_mbal 'in_c_bal_vals)
                         (<=* 'b_mbal 'in_b_bal_vals)
                         (<=* 'a_mbal 'in_a_bal_vals))
                        (list
                         (if*
                          (and*
                           (and*
                            (and*
                             (and*
                              (bweq* 'phase (bv #x04 8))
                              (bweq* (bv #x00 8) 'c_mbal))
                             (bweq* 'b_mbal (bv #x00 8)))
                            (lt* 'a_mbal (bv #x01 8)))
                           (list
                            (<=* 'phase (bv #x05 8))
                            (<=* 'value 'value))
                           (list
                            (if*
                             (and*
                              (or*
                               (lt* 'b_mbal 'a_mbal)
                               (bweq* 'b_mbal 'a_mbal))
                              (and*
                               (or*
                                (bweq* 'a_mbal 'c_mbal)
                                (lt* 'c_mbal 'a_mbal))
                               (bweq* (bv #x04 8) 'phase)))
                             (list
                              (<=* 'phase (bv #x05 8))
                              (<=* 'value 'a_mval))
                             (list
                              (if*
                               (and*
                                (bweq* (bv #x04 8) 'phase)
                                (or*
                                 (bweq* 'b_mbal 'c_mbal)
                                 (lt* 'c_mbal 'b_mbal)))
                               (list
                                (<=* 'phase (bv #x05 8))
                                (<=* 'value 'b_mval))
                               (list
                                (if*
                                 (bweq* (bv #x04 8) 'phase)
                                 (list
                                  (<=* 'phase (bv #x05 8))
                                  (<=* 'value 'c_mval))
                                 (list
                                  (if*
                                   (bweq* (bv #x05 8) 'phase)
                                   (list
                                    (<=* 'phase (bv #x06 8))
                                    (<=* 'out_c_val_vals 'value)
                                    (<=* 'out_c_val_sent (bv #x00 8))
                                    (<=* 'out_b_val_vals 'value)
                                    (<=* 'out_b_val_sent (bv #x00 8))
                                    (<=* 'out_a_val_vals 'value)
                                    (<=* 'out_a_val_sent (bv #x00 8))
                                    (<=* 'out_c_bal_vals 'ballot)
                                    (<=* 'out_c_bal_sent (bv #x00 8))
```

```
                                (<=* 'out_b_bal_vals 'ballot)
                                (<=* 'out_b_bal_sent (bv #x00 8))
                                (<=* 'out_a_bal_vals 'ballot)
                                (<=* 'out_a_bal_sent (bv #x00 8)))
                               (list
                                (if*
                                 (and*
                                  (eq* 'd1 'd0)
                                  (and*
                                   (bweq* 'phase (bv #x06 8))
                                   (not*
                                    (lt*
                                     (bv #x07 8)
                                     'out_a_bal_sent))))
                                 (list
                                  (<=*
                                   'out_a_val_vals
                                   'out_a_val_vals)
                                  (<=*
                                   'out_a_val_sent
                                   'out_a_val_sent)
                                  (<=*
                                   'out_a_bal_vals
                                   'out_a_bal_vals)
                                  (<=*
                                   'out_a_bal_sent
                                   (add*
                                    'out_a_bal_sent
                                    (bv #x01 8)))
                                  (<=*
                                   'd2
                                   (lt*
                                    (bwand*
                                     (shl*
                                      (bv #x01 8)
                                      'out_a_bal_sent)
                                     (bwnot* 'out_a_bal_vals))
                                    (bv #x01 8)))
                                  (<=* 'd0 (eq* 'd1 #f)))
                                 (list
                                  (if*
                                   (and*
                                    (and*
                                     (lt*
                                      (bv #x07 8)
                                      'out_a_bal_sent)
                                     (and*
                                      (bweq* 'phase (bv #x06 8))
                                      (not*
                                       (lt*
                                        (bv #x07 8)
                                        'out_a_val_sent))))
                                    (eq* 'd1 'd0))
                                   (list
                                    (<=*
                                     'out_a_val_vals
                                     'out_a_val_vals)
                                    (<=*
                                     'out_a_val_sent
                                     (add*
                                      'out_a_val_sent
                                      (bv #x01 8)))
                                    (<=*
```

```
                                        'out_a_bal_vals
                                        'out_a_bal_vals)
                                       (<=*
                                        'out_a_bal_sent
                                        'out_a_bal_sent)
                                       (<=*
                                        'd2
                                        (bweq*
                                         (bwand*
                                          (shr*
                                           'out_a_val_vals
                                           'out_a_val_sent)
                                          (bv #x01 8))
                                         (bv #x01 8)))
                                       (<=* 'd0 (eq* 'd1 #f)))
                                      (list
                                       (if*
                                        (and*
                                         (not*
                                          (or*
                                           (and* (eq* #f 'd4) 'd3)
                                           (and* 'd4 (eq* 'd3 #f))))
                                         (and*
                                          (bweq* 'phase (bv #x06 8))
                                          (eq*
                                           (lt*
                                            (bv #x07 8)
                                            'out_b_bal_sent)
                                           #f)))
                                        (list
                                         (<=*
                                          'out_b_val_vals
                                          'out_b_val_vals)
                                         (<=*
                                          'out_b_val_sent
                                          'out_b_val_sent)
                                         (<=*
                                          'out_b_bal_vals
                                          'out_b_bal_vals)
                                         (<=*
                                          'out_b_bal_sent
                                          (add*
                                           'out_b_bal_sent
                                           (bv #x01 8)))
                                         (<=*
                                          'd5
                                          (eq*
                                           (bwand*
                                            (bwnot* 'out_b_bal_vals)
                                            (shl*
                                             (bv #x01 8)
                                             'out_b_bal_sent))
                                           (bv #x00 8)))
                                         (<=* 'd3 (not* 'd3)))
                                        (list
                                         (if*
                                          (and*
                                           (and*
                                            (lt*
                                             (bv #x07 8)
                                             'out_b_bal_sent)
                                            (and*
                                             (lt*
```

```
                    'out_b_val_sent
                    (bv #x08 8))
                   (bweq*
                    'phase
                    (bv #x06 8))))
                 (not*
                  (or*
                   (and* (not* 'd3) 'd4)
                   (and* 'd3 (eq* 'd4 #f)))))
                (list
                 (<=*
                  'out_b_val_vals
                  'out_b_val_vals)
                 (<=*
                  'out_b_val_sent
                  (add*
                   'out_b_val_sent
                   (bv #x01 8)))
                 (<=*
                  'out_b_bal_vals
                  'out_b_bal_vals)
                 (<=*
                  'out_b_bal_sent
                  'out_b_bal_sent)
                 (<=*
                  'd5
                  (not*
                   (bweq*
                    (bv #x00 8)
                    (bwand*
                     (shr*
                      'out_b_val_vals
                      'out_b_val_sent)
                     (bv #x01 8)))))
                 (<=* 'd3 (not* 'd4)))
                (list
                 (if*
                  (and*
                   (and*
                    (bweq* 'phase (bv #x06 8))
                    (not*
                     (lt*
                      (bv #x07 8)
                      'out_c_bal_sent)))
                   (eq* 'd6 'd7))
                  (list
                   (<=*
                    'out_c_val_vals
                    'out_c_val_vals)
                   (<=*
                    'out_c_val_sent
                    'out_c_val_sent)
                   (<=*
                    'out_c_bal_vals
                    'out_c_bal_vals)
                   (<=*
                    'out_c_bal_sent
                    (add*
                     (bv #x01 8)
                     'out_c_bal_sent))
                   (<=*
                    'd8
                    (bweq*
```

```
                                    (bwand*
                                     (shl*
                                      (bv #x01 8)
                                      'out_c_bal_sent)
                                     (bwxor*
                                      (bv #xff 8)
                                      'out_c_bal_vals))
                                    (bv #x00 8)))
                                  (<=* 'd6 (not* 'd6)))
                                 (list
                                  (if*
                                   (and*
                                    (and*
                                     (and*
                                      (bweq*
                                       (bv #x06 8)
                                       'phase)
                                      (eq*
                                       #f
                                       (lt*
                                        (bv #x07 8)
                                        'out_c_val_sent)))
                                      (lt*
                                       (bv #x07 8)
                                       'out_c_bal_sent))
                                     (eq* 'd6 'd7))
                                    (list
                                     (<=*
                                      'out_c_val_vals
                                      'out_c_val_vals)
                                     (<=*
                                      'out_c_val_sent
                                      (add*
                                       (bv #x01 8)
                                       'out_c_val_sent))
                                     (<=*
                                      'out_c_bal_vals
                                      'out_c_bal_vals)
                                     (<=*
                                      'out_c_bal_sent
                                      'out_c_bal_sent)
                                     (<=*
                                      'd8
                                      (eq*
                                       (bwor*
                                        (shr*
                                         'out_c_val_vals
                                         'out_c_val_sent)
                                        (bv #x01 8))
                                       (shr*
                                        'out_c_val_vals
                                        'out_c_val_sent)))
                                     (<=* 'd6 (eq* 'd7 #f)))
                                    (list
                                     (if*
                                      (and*
                                       (and*
                                        (and*
                                         (bweq*
                                          (bv #x06 8)
                                          'phase)
                                         (lt*
                                          (bv #x07 8)
```

```
                'out_c_val_sent))
              (lt*
               (bv #x07 8)
               'out_b_val_sent))
             (lt*
              (bv #x07 8)
              'out_a_val_sent))
            (list
             (<=*
              'phase
              (bv #x07 8))
             (<=*
              'in_c_val_vals
              (bv #x00 8))
             (<=*
              'in_c_val_rcvd
              (bv #x00 8))
             (<=*
              'in_b_val_vals
              (bv #x00 8))
             (<=*
              'in_b_val_rcvd
              (bv #x00 8))
             (<=*
              'in_a_val_vals
              (bv #x00 8))
             (<=*
              'in_a_val_rcvd
              (bv #x00 8))
             (<=*
              'in_c_bal_vals
              (bv #x00 8))
             (<=*
              'in_c_bal_rcvd
              (bv #x00 8))
             (<=*
              'in_b_bal_vals
              (bv #x00 8))
             (<=*
              'in_b_bal_rcvd
              (bv #x00 8))
             (<=*
              'in_a_bal_vals
              (bv #x00 8))
             (<=*
              'in_a_bal_rcvd
              (bv #x00 8)))
            (list
             (if*
              (and*
               (or*
                (and*
                 (eq* #f 'd9)
                 'd10)
                (and*
                 (eq* 'd10 #f)
                 'd9))
               (and*
                (bweq*
                 (bv #x07 8)
                 'phase)
                (eq*
                 #f
```

```
                (lt*
                 (bv #x07 8)
                 'in_a_bal_rcvd))))
              (list
               (<=*
                'in_a_val_vals
                'in_a_val_vals)
               (<=*
                'in_a_val_rcvd
                'in_a_val_rcvd)
               (<=*
                'in_a_bal_vals
                (bwxor*
                 (shl*
                  (shr*
                   (bv #x01 8)
                   (bool->vect*
                    'd11))
                  'in_a_bal_rcvd)
                 (bwor*
                  (shl*
                   (bv #x01 8)
                   'in_a_bal_rcvd)
                  'in_a_bal_vals)))
               (<=*
                'in_a_bal_rcvd
                (add*
                 'in_a_bal_rcvd
                 (bv #x01 8)))
               (<=* 'd10 'd9))
              (list
               (if*
                (and*
                 (and*
                  (and*
                   (not*
                    (lt*
                     (bv #x07 8)
                     'in_a_val_rcvd))
                   (bweq*
                    'phase
                    (bv #x07 8)))
                  (lt*
                   (bv #x07 8)
                   'in_a_bal_rcvd))
                 (or*
                  (and*
                   'd10
                   (not* 'd9))
                  (and*
                   (not* 'd10)
                   'd9)))
                (list
                 (<=*
                  'in_a_val_vals
                  (bwxor*
                   (bwnot*
                    (shl*
                     (bool->vect*
                      'd11)
                     'in_a_val_rcvd))
                   (bwor*
                    (shl*
```

```
                (bv #x01 8)
                'in_a_val_rcvd)
               (bwnot*
                'in_a_val_vals))))
             (<=*
              'in_a_val_rcvd
              (add*
               (bv #x01 8)
               'in_a_val_rcvd))
             (<=*
              'in_a_bal_vals
              'in_a_bal_vals)
             (<=*
              'in_a_bal_rcvd
              'in_a_bal_rcvd)
             (<=* 'd10 'd9))
            (list
             (if*
              (and*
               (and*
                (eq*
                 #f
                 (lt*
                  (bv #x07 8)
                  'in_b_bal_rcvd))
                (bweq*
                 (bv #x07 8)
                 'phase))
               (or*
                (and*
                 'd12
                 (eq* 'd13 #f))
                (and*
                 'd13
                 (eq*
                  'd12
                  #f))))
              (list
               (<=*
                'in_b_val_vals
                'in_b_val_vals)
               (<=*
                'in_b_val_rcvd
                'in_b_val_rcvd)
               (<=*
                'in_b_bal_vals
                (bwxor*
                 (bwxor*
                  (bv #xff 8)
                  (shl*
                   (bool->vect*
                    'd14)
                   'in_b_bal_rcvd))
                 (bwor*
                  (shl*
                   (bv #x01 8)
                   'in_b_bal_rcvd)
                  (bwnot*
                   'in_b_bal_vals))))
               (<=*
                'in_b_bal_rcvd
                (add*
                 (bv #x01 8)
```

```
                'in_b_bal_rcvd))
              (<=* 'd13 'd12))
             (list
              (if*
               (and*
                (and*
                 (and*
                  (bweq*
                   (bv #x07 8)
                   'phase)
                  (eq*
                   #f
                   (lt*
                    (bv #x07 8)
                    'in_b_val_rcvd)))
                 (lt*
                  (bv #x07 8)
                  'in_b_bal_rcvd))
                (or*
                 (and*
                  'd12
                  (eq*
                   #f
                   'd13))
                 (and*
                  (eq* #f 'd12)
                  'd13)))
               (list
                (<=*
                 'in_b_val_vals
                 (bwxor*
                  (bwnot*
                   (shl*
                    (bool->vect*
                     'd14)
                    'in_b_val_rcvd))
                  (bwor*
                   (shl*
                    (bv #x01 8)
                    'in_b_val_rcvd)
                   (bwxor*
                    'in_b_val_vals
                    (bv #xff 8)))))
                (<=*
                 'in_b_val_rcvd
                 (add*
                  (bv #x01 8)
                  'in_b_val_rcvd))
                (<=*
                 'in_b_bal_vals
                 'in_b_bal_vals)
                (<=*
                 'in_b_bal_rcvd
                 'in_b_bal_rcvd)
                (<=*
                 'd13
                 'd12))
               (list
                (if*
                 (and*
                  (or*
                   (and*
                    'd16
```

```
                (not*
                 'd15))
               (and*
                'd15
                (eq*
                 'd16
                 #f)))
              (and*
               (bweq*
                'phase
                (bv #x07 8))
               (eq*
                #f
                (lt*
                 (bv #x07 8)
                 'in_c_bal_rcvd))))
             (list
              (<=*
               'in_c_val_vals
               'in_c_val_vals)
              (<=*
               'in_c_val_rcvd
               'in_c_val_rcvd)
              (<=*
               'in_c_bal_vals
               (bwxor*
                (bwnot*
                 (shl*
                  (bool->vect*
                   'd17)
                  'in_c_bal_rcvd))
                (bwor*
                 (shl*
                  (bv #x01 8)
                  'in_c_bal_rcvd)
                 (bwnot*
                  'in_c_bal_vals))))
              (<=*
               'in_c_bal_rcvd
               (add*
                (bv #x01 8)
                'in_c_bal_rcvd))
              (<=*
               'd16
               'd15))
             (list
              (if*
               (and*
                (or*
                 (and*
                  (not*
                   'd16)
                  'd15)
                 (and*
                  'd16
                  (not*
                   'd15)))
                (and*
                 (lt*
                  (bv #x07 8)
                  'in_c_bal_rcvd)
                 (and*
                  (eq*
```

```
    #f
    (lt*
     (bv #x07 8)
     'in_c_val_rcvd))
   (bweq*
    'phase
    (bv #x07 8)))))
(list
 (<=*
  'in_c_val_vals
  (bwxor*
   (bwor*
    (bwnot*
     'in_c_val_vals)
    (shl*
     (bv #x01 8)
     'in_c_val_rcvd))
   (bwnot*
    (shl*
     (bool->vect*
      'd17)
     'in_c_val_rcvd))))
 (<=*
  'in_c_val_rcvd
  (add*
   'in_c_val_rcvd
   (bv #x01 8)))
 (<=*
  'in_c_bal_vals
  'in_c_bal_vals)
 (<=*
  'in_c_bal_rcvd
  'in_c_bal_rcvd)
 (<=*
  'd16
  'd15))
(list
 (if*
  (and*
   (lt*
    (bv #x07 8)
    'in_a_val_rcvd)
   (and*
    (and*
     (lt*
      (bv #x07 8)
      'in_c_val_rcvd)
     (bweq*
      (bv #x07 8)
      'phase))
    (lt*
     (bv #x07 8)
     'in_b_val_rcvd)))
  (list
   (<=*
    'phase
    (bv #x08 8))
   (<=*
    'c_mval
    'in_c_val_vals)
   (<=*
    'b_mval
    'in_b_val_vals)
```

```
 (<=*
  'a_mval
  'in_a_val_vals)
 (<=*
  'c_mbal
  'in_c_bal_vals)
 (<=*
  'b_mbal
  'in_b_bal_vals)
 (<=*
  'a_mbal
  'in_a_bal_vals))
(list
 (if*
  (and*
   (bweq*
    'a_mval
    'value)
   (and*
    (bweq*
     'value
     'b_mval)
    (and*
     (bweq*
      'value
      'c_mval)
     (bweq*
      'phase
      (bv #x08 8)))))
  (list
   (<=*
    'phase
    (bv #x00 8)))
  (list
   (if*
    (bweq*
     'phase
     (bv #x08 8))
    (list
     (<=*
      'phase
      (bv #xff 8)))
    '()))))))))))))))))))))))))))))))))))
          ↪ ))))))))))))))))))))))))))))
          ↪ )))))))))))
```