Lifting Hardware Models from Implementations for Verification

Jonathan Shi

Electrical Engineering and Computer Sciences
University of California, Berkeley

Technical Report No. UCB/EECS-2022-149
http://www2.eecs.berkeley.edu/Pubs/TechRpts/2022/EECS-2022-149.html

May 19, 2022

Acknowledgement

I would first like to thank my advisor, Professor Sanjit A. Seshia, for his support over the course of this project. I also would like to thank Professor Yatin Manerkar, Kevin Laeufer, and Dr. Elizabeth Polgreen for their guidance and feedback. Finally, this work and I owe a great deal to Adwait Godbole, as our joint work on EECS 219C and CS 263 course projects provides the foundation for this thesis.

Lifting Hardware Models from Implementations for Verification

by

Jonathan Shi

A thesis submitted in partial satisfaction of the

requirements for the degree of

Master of Science

in

Electrical Engineering and Computer Sciences

in the

Graduate Division

of the

University of California, Berkeley

Spring 2022

Lifting Hardware Models from Implementations for Verification

by Jonathan Shi

Research Project

Submitted to the Department of Electrical Engineering and Computer Sciences, University of California at Berkeley, in partial satisfaction of the requirements for the degree of **Master of Science, Plan II**.

Approval for the Report and Comprehensive Examination:

Committee:

[illegible]

Professor Sanjit A. Seshia
Research Advisor

5/19/2022

(Date)

* * * * * * *

[illegible]

Professor Alvin Cheung
Second Reader

May 19, 2022

(Date)

Acknowledgments

I would first like to thank my advisor, Professor Sanjit A. Seshia, for his support over the course of this project. I also would like to thank Professor Yatin Manerkar, Kevin Laeufer, and Dr. Elizabeth Polgreen for their guidance and feedback. Finally, this work and I owe a great deal to Adwait Godbole, as our joint work on EECS 219C and CS 263 course projects provides the foundation for this thesis.

Abstract

Lifting Hardware Models from Implementations for Verification

by

Jonathan Shi

Master of Science in Electrical Engineering and Computer Sciences

University of California, Berkeley

Professor Sanjit A. Seshia, Chair

We introduce RTL2MODEL, a compositional Python framework for modeling hardware designs. RTL2MODEL models can be generated from RTL with various degrees of micro-architectural granularity, and can be composed with other models that are either manually constructed or algorithmically produced. We combine cone-of-influence algorithms with syntax-guided synthesis techniques to produce simpler models than those that are translated directly from RTL, thus reducing the model-to-implementation gap and facilitating more efficient verification.

To my friends and family, without whom I would not be here today.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

As hardware designs become more complex and more specialized, the need for scalable methods of verifying their functional correctness continues to grow. Moreover, the relatively recent discoveries of side-channel attacks like Spectre [19] and Meltdown [21] have demonstrated the need for methods that encompass micro-architectural behaviors, in additional the the architecture-level behaviors tested by traditional verification techniques. The task of manually constructing such formal models of RTL designs is tedious, time-consuming, and error-prone; to remedy this, this paper provides a framework for “lifting,” or automatically synthesizing abstract models, directly from concrete RTL implementations with minimal need for human input.

1.1 Motivation

The process of creating a formal model for a hardware design can be split into three phases: (1) creation of an abstract model, (2) proving the correctness of the abstraction with respect to the implementation, and (3) verifying properties of interest on the abstract model. A fundamental trade-off exists between the complexity of the abstract model and the difficulty of verification: the abstract model must still be detailed enough to capture the relevant behaviors of the underlying RTL, but too much detail may make verification efforts computationally expensive. At one extreme, a verification engineer could directly verify assertions on RTL signals using tools like JasperGold [9] or SymbiYosys [12], in which case the complexity of the abstraction remains comparable to that of the original hardware implementation. All stages of this process (abstraction, proof of correctness, and verification) have historically required significant manual labor on the part of designers and verification engineers, and all three stages generally scale poorly with the complexity of the hardware design. The key to reducing this complexity is *composition*: just as software programs and hardware designs are split into smaller, reusable units that are easier to understand and test, decomposing abstract models into smaller components is the key to enabling more efficient formal verification. This work provides a framework that allows composition of these models, and also presents techniques to automatically lift them from RTL designs.

To facilitate easier hardware composition, this work seeks to make the following improvement existing RTL modeling techniques:

- **Increased automation:** Though users must still define the programs under which they wish to test a processor, RTL2MODEL automatically generates and synthesizes large portions of hardware models from RTL.
- **Easy composition of models:** Larger hardware designs are typically split into smaller sub-modules that perform some specialized functionality; for example, a typical CPU design might have separate modules for the datapath and control logic, and the datapath module itself could have dedicated sub-modules for the ALU and register file. As with the use of classes in object-oriented software design, decomposing RTL into black-boxed modules allow hardware designers to provide different implementations of the same functionality, as long as these implementations share the same input-output interface. Our modeling framework allows users to create sub-models analogous to RTL sub-modules, thus enabling independent verification of parts of designs, and replacing more complex models with simpler ones that might allow faster verification.
- **Varying granularity of micro-architectural properties:** Using cone-of-influence algorithms, users of the RTL2MODEL framework may specify different levels of granularity for RTL state that is preserved in the lifted model.

1.2 Related Work

Traditional hardware verification techniques for pipelined processors are centered around the so-called “flushing abstraction” introduced by Burch and Dill [8], where the equivalence of executing an instruction between abstract and implementation state is verified by “flushing” the implementation state to complete the currently executing instruction. The ATLAS approach by Brady et al. [5, 6] uses random simulation and static analysis to identify portions of designs that can be partially abstracted automatically, and then performs Burch-Dill style equivalence checks to verify the correctness of their abstractions.

Programmable processors typically have a predefined specification in the form of an *Instruction Set Architecture* (ISA), which describes the architecture-level effects of executing an instruction on the processor. Subramanyan et al. [33] extend this to co-processors by introducing the notion of an *Instruction-Level Abstraction* (ILA), a generalization of ISAs that can describe the software-visible state changes for instructions executing on any manner of hardware accelerator. Further work by Huang et al. [14] explores hierarchical composition and synthesis of ILA instructions by using “sub-instructions” to describe portions of instruction behavior, and more recent work by others explores techniques to automatically generate environment invariants [37] and detect architectural state variables [35]. These previous ILA-based works focus on generating the high-level architectural model for a processor design; to our knowledge, ours is the first work that attempts to include microarchitectural details in such automatically generated operational models.

Our work applies syntax-guided synthesis (SyGuS) [1, 16, 28] techniques to automatically generate components of hardware models; we use a counterexample-guided inductive synthesis (CEGIS) algorithm to generate implementations for combinatorial functions in hardware that are simpler than that in the original design. Jha et al. [16] define the oracle-guided inductive synthesis approach, where external "oracles" are invoked to validate and provide constraints on synthesized candidate functions. In particular, we use a hardware simulator as an "I/O oracle" to provide input-output examples for combinatorial functions, and a model checker as a "validation oracle" to determine whether a synthesized function correctly abstracts the hardware.

Finally, the Check tools and μspec take an axiomatic approach to modeling and verifying hardware properties, especially memory consistency models (MCMs). PipeCheck [23] uses "microarchitectural happens before" graphs to verify that a given processor follows a desired MCM, and CCICheck [25] applies similar techniques to verify memory coherence and consistency properties. More recently, rtl2μspec [13] allows for automated generation of the axioms needed to perform these proofs directly from RTL designs.

1.3 Contributions and Overview

To enable easier and more efficient hardware verification, we introduce RTL2MODEL, a Python framework that allows designers to generate and compose lifted models of hardware designs. Designers can construct models by hand, synthesize portions of them with SyGuS techniques, or translate them directly from RTL; all these different sorts of models can be composed together in a hierarchical fashion.

In Chapter 2, we discuss in further detail the techniques underlying RTL2MODEL, and its differences from previous works. In Chapter 3, we describe the semantics, implementation, and evaluation of the framework on example hardware designs.

Chapter 2

Background

2.1 Processor Modeling and Verification

2.1.1 The Hardware Lifting Problem

Formal verification of hardware designs usually involves three components:

- **RTL implementation:** This is the original design, usually described in a language like Chisel or Verilog.
- **Verification model:** This is an intermediate model, which may still contain micro-architectural details that were present in the original RTL. Its software-visible behavior still matches that of the RTL.
- **Architectural specification:** This defines the effect of instructions on the software-visible state of the hardware, with micro-architectural details abstracted away. For general processors, this is typically the instruction-set architecture (ISA). Instruction-level abstractions (ILAs) [14, 33] further generalize the notion of an ISA to encompass software-visible state of accelerators as well. The ISA/ILA for a design is often formally described by a "golden model" that defines these behaviors.

An important goal in hardware verification is to determine whether a concrete hardware implementation properly implements its architectural specification. Directly applying verification techniques to the transition system described by RTL is possible, but often very computationally expensive. Thus, many verification approaches operate on an intermediate model from the RTL design that captures the desired behavior, but contains less complexity and can be verified more efficiently.

Once a verification model is generated, we must then prove that it is refined by (i.e. abstracts) the original implementation. This is usually accomplished by a Burch-Dill style check, where the same instruction is run on both the RTL and the lifted model; a "flush" is then performed on the RTL, and the architectural states of the RTL and abstract model are checked for equivalence.

After refinement has been proved, we must now show that the lifted model satisfies the architectural specification. This can take the form of another equivalence check to ensure the lifted model refines the ISA, but in some cases it may be sufficient (and computationally cheaper) to demonstrate that the ISA simulates the lifted model.

Instruction-Level Abstractions

In addition to providing a generalized mechanism for describing accelerator and processor behaviors, ILAs also allow modeling of design behaviors at different levels of granularity. In the work of Huang et al. [14], the ILA of a RISC-V processor is very similar to the predefined RISC-V ISA, but with additional microarchitectural state to properly model interrupt and TLB behaviors. By contrast, their model for the the `START_ENCRYPT` ILA instruction of an AES accelerator is split into constituent child instructions: `LOAD_BLOCK` to load a block to encrypt, `ENCRYPT` to encrypt a block of data, and `STORE_RESULT` to either move on to the next block of data or end the encryption algorithm. Though it is possible to apply program synthesis techniques to automatically generate portions of hierarchical ILAs, the bulk of an ILA must still be be manually specified by an engineer.

Our work applies techniques similar to those of prior ILA works [33, 14], as we require users to specify refinement relations between the verification model and RTL, in addition to a program sketch that serves as a "template" for the behaviors we wish to verify. ILAs are designed primarily to capture the behavior of the processor's hardware-software interface; by contrast, RTL2MODEL models seek to operate at a lower level of abstraction, and more deliberately capture microarchitectural details.

2.1.2 UCLID5 Models

UCLID5 [29] is a formal modeling system that supports a variety of specification, verification, and synthesis techniques; its applications include verifying security properties of speculative processors [10], formalizing execution time models of processor pipelines [15], and verifying trusted execution environments [11]. It is evolved from the older UCLID system for formal modeling and verification [7].

Verilog designs that operate within a single clock domain can be modeled as transition systems: for the set of boolean/bit-vector/array state variables S and set of input variables W, there exists a set of functions $N = \{N_i : (S \times W) \to S\}$ that describes the values of S on the next clock cycle. UCLID5 readily supports modeling transition systems, and thus naturally lends itself to modeling hardware designs.

Figure 2.1 contains an abridged version of an example for verifying a processor design from the UCLID5 tutorial [32], which verifies that the processor's behavior is deterministic by proving that two copies of the model will always have the same behavior when the same program is run. UCLID5 modules may contain inputs, outputs, state variables, and synthesis functions; variable assignments in the `next` block of a module correspond to register updates

```
module cpu {
  type op_t   = enum { op_mov, op_add, op_sub };
  input imem : [bv8]bv8;    // Program memory
  var regs   : [bv8]bv8;    // Register state
  var pc     : bv8;         // Program counter
  var inst   : bv8;
  // Decoding functions
  function inst2op   (i : bv8) : op_t;
  function inst2reg0 (i : bv8) : bv8;
  function inst2reg1 (i : bv8) : bv8;
  function inst2imm  (i : bv8) : bv8;

  procedure exec_inst(inst : bv8, pc : bv8)
    returns (pc_next : bv8)
    modifies regs;
  {
    var op           : op_t;
    var r0ind, r1ind : bv8;
    var r0, r1       : bv8;
    var result       : bv8;

    op = inst2op(inst);
    r0ind, r1ind = inst2reg0(inst), inst2reg1(inst);
    r0, r1 = regs[r0ind], regs[r1ind];
    case
      (op == op_mov) : { result = inst2imm(inst); }
      (op == op_add) : { result = r0 + r1; }
      (op == op_sub) : { result = r0 - r1; }
    esac
    pc_next = pc + 1bv8;
    regs[r0ind] = result;
  }

  init {
    assume (forall (r : bv8) :: regs[r] == 0bv8);
    pc, inst = 0bv8, 0bv8;
  }

  next {
    inst' = imem[pc];
    call (pc') = exec_inst(inst, pc);
  }
}
```

(a) Transition system for CPU.

```
module main {
  var imem : [bv8]bv8;
  instance cpu1 : cpu(imem : (imem));
  instance cpu2 : cpu(imem : (imem));

  init { }

  next {
    // Perform state updates for each copy
    next (cpu1);
    next (cpu2);
  }

  // Properties to verify
  invariant eq_regs :
    (forall (ri : bv8)
      :: cpu1.regs[ri] == cpu2.regs[ri]);
  invariant eq_pc   : (cpu1.pc == cpu2.pc);
  invariant eq_inst : (cpu1.inst == cpu2.inst);

  // Performs 3 cycle bounded model check
  control {
    bmc(3);
    check;
    print_results;
  }
}
```

(b) Main module with proof script.

Figure 2.1: UCLID5 model of a simple processor, with bounded proof of its determinism.

in RTL. In the example, registers and the program counter are left as concrete state, while decoding operations are left as uninterpreted functions.

RTL2MODEL can translate Verilog models directly to the equivalent UCLID5 transition systems to facilitate further analysis and verification. Other efforts to translate HDL code directly to such models include the ChiselUCL project [24], which converts models from Chisel HDL to UCLID5; and ATLAS [5, 6], which uses random simulation and static analysis techniques to generate models in UCLID (UCLID5's predecessor). The techniques discussed in this thesis can be applied to any transitional modeling system that supports uninterpreted functions. We choose to build a custom framework, rather than use an existing tool, in order to make applying transformations to models, interfacing with the synthesis engine, and invoking oracles easier.

2.2 Synthesis Techniques

2.2.1 Syntax-Guided Synthesis and Synthesis Modulo Oracles

Syntax-guided synthesis (SyGuS) has gained popularity in recent years as a means to generate program implementations that satisfy certain constraints. Formally, the classic SyGuS problem is formulated as $\exists \vec{f}. \forall \vec{x}. \phi$, where $\vec{f}$ is a set of functions to be synthesized, $\vec{x}$ is a set of variables, and ϕ is a formula in some logical theory. Synthesis terminates when a set of function implementations $\vec{f^*}$ is found that satisfies the formula $\forall \vec{x}. \phi$, with all $\vec{f}$ replaced by $\vec{f^*}$ in ϕ [27]. A common approach to solving SyGuS problems is that of *counterexample-guided inductive synthesis* (CEGIS), where the behavior of synthesis candidate functions that do not satisfy ϕ is used to add further constraints to block invalid candidates. A typical CEGIS loop, as outlined in [31], is shown in Figure 2.2.

Seshia [28] describes the combination of inductive reasoning that learns from examples (like CEGIS algorithms) with deductive reasoning to answer validation queries (like SMT solvers used to check properties). Jha and Seshia [17, 16] generalize the use of CEGIS in this context to the notion of *oracle-guided inductive synthesis* (OGIS), where an inductive *learner* queries black-boxed *oracles* to provide positive examples and counterexamples. In OGIS, the learner synthesizes a candidate function that satisfies a provided set of input-output examples from a larger class of functions; in the case of CEGIS with SyGuS, this class is the set of functions producible by the given grammar. Queries to oracles then determine the validity of a candidate, and provide positive examples or negative counterexamples.

In RTL2MODEL, we rely on external hardware simulators and model checking tools to verify the correctness of a generated model. In the context of the OGIS paradigm, the hardware simulator serves as an input-output (I/O) oracle, where the resulting simulation trace adds positive examples that describe the desired input-output behavior of candidate functions. The model checker serves as a validation/correctness oracle with counterexamples: it verifies the validity of candidate functions, and produces a concrete counterexample trace if the synthesis candidates are invalid.

2.2.2 Program Sketches

Similar to previous ILA-based works like [36], the RTL2MODEL framework relies on user-provided *program sketches*–partially-specified templates of programs with "holes" that a synthesis algorithm fills with concrete values. Traditionally, program sketches are used as pure input-output relations, with holes as inputs and the corresponding outputs used as

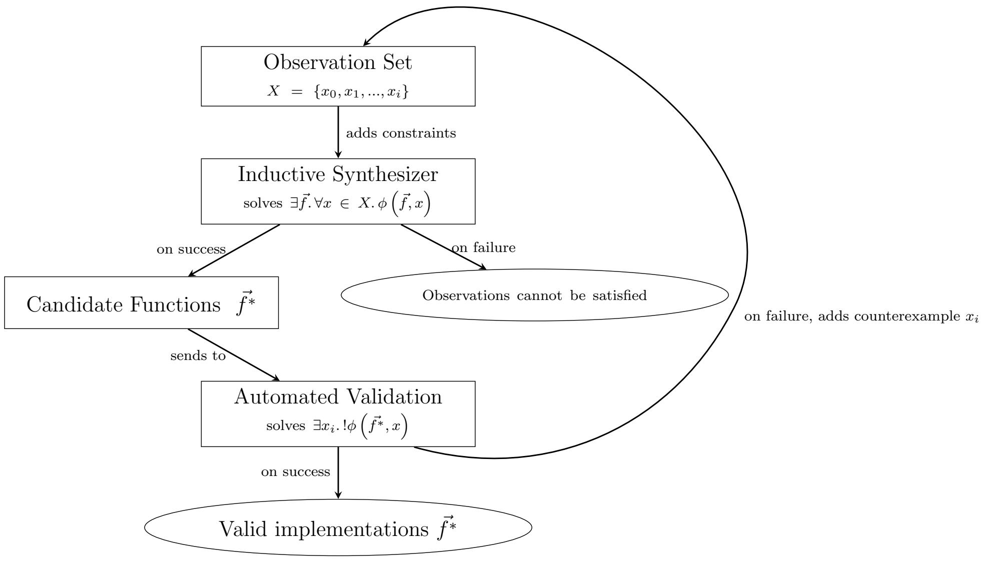

Observation Set
$X = \{x_0, x_1, ..., x_i\}$
adds constraints
Inductive Synthesizer
solves $\exists \vec{f}.\forall x \in X. \phi\left(\vec{f}, x\right)$
on success
on failure
Candidate Functions $\vec{f^*}$
Observations cannot be satisfied
on failure, adds counterexample x_i
sends to
Automated Validation
solves $\exists x_i. !\phi\left(\vec{f^*}, x\right)$
on success
Valid implementations $\vec{f^*}$

Figure 2.2: A typical CEGIS loop for synthesizing a set of functions $\vec{f}$ that satisfies a set of observations X.

examples for a CEGIS loop [31]. We use sketches somewhat differently: holes are still treated as inputs to our synthesis functions, but outputs used for I/O examples are instead derived from RTL simulation or model checking counterexamples. This process is explained in further detail in Section 3.2.2.

Chapter 3

Implementation and Semantics

This section discusses the semantics of RTL2MODEL models, and the algorithms involved in their construction. Figure 3.1 outlines the lifting procedure in its entirety; each constituent component is explained in the sections below.

3.1 Model Semantics

3.1.1 Expression Grammar

Expressions in RTL2MODEL models describe transition relations and assertions, and correspond closely to terms in SMT-LIB [4]. Users construct expressions and sorts through the RTL2MODEL Python API, which in turn wraps the Python API of the CVC5 SMT solver [3]. We construct our own API rather than directly invoking CVC5 both to allow users to write more ergonomic code, and to more easily translate equivalent expressions to other modeling or simulation languages like Verilog and UCLID5 [29].

Sorts

As in SMT-LIB, all expressions have a *sort*, which represents the possible values it can take on. RTL2MODEL has four sorts:

- Booleans, constructed in Python code with `BoolSort()`.
- n-bit bit-vectors, constructed with `BVSort(`n`)`.
- Fixed-size arrays, constructed with `ArraySort(idx, elem)`.
- Functions, constructed with `FunctionSort(*inputs, codomain)`.

Terms

Expressions are made up of terms, which can be any of the following:

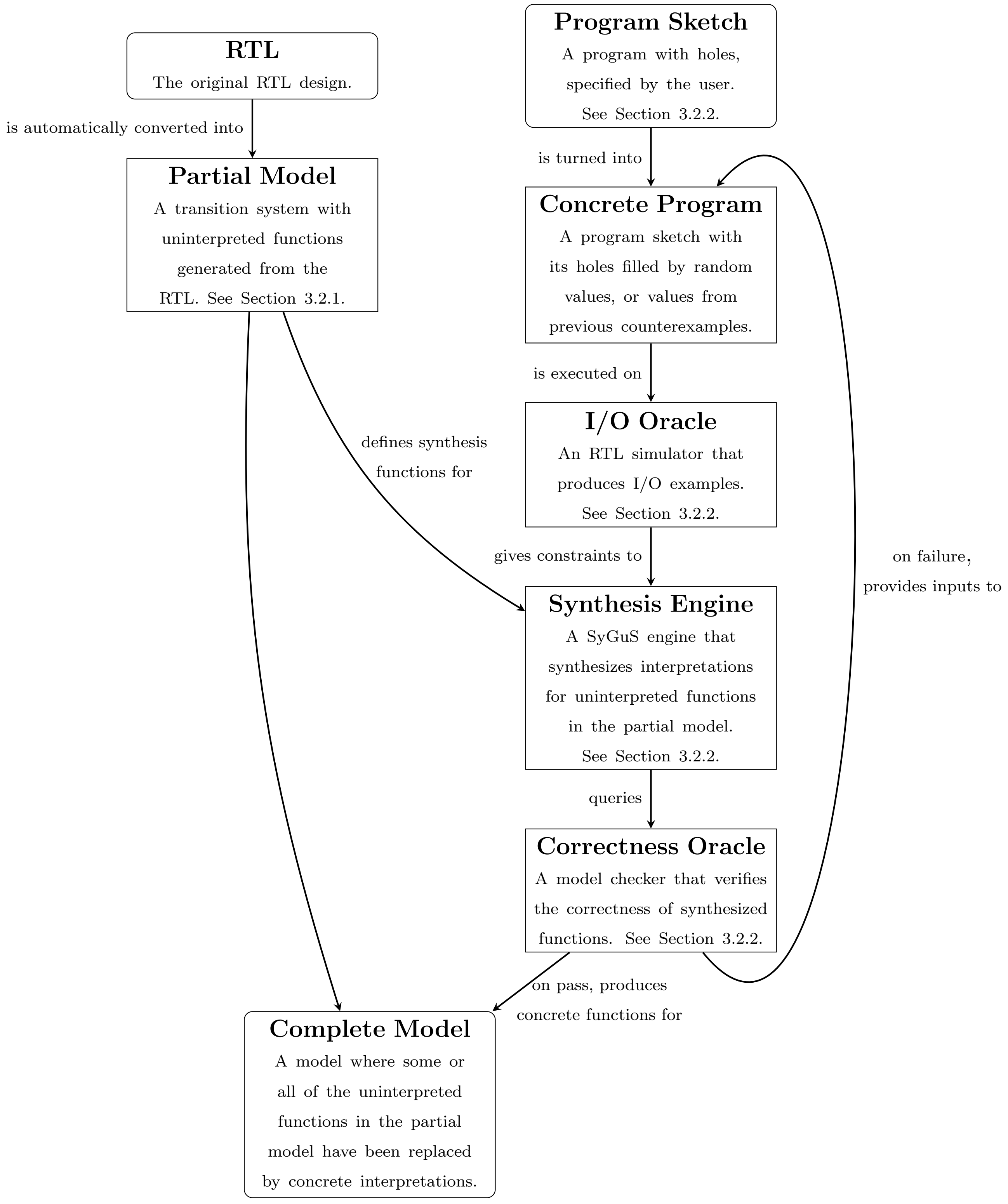

RTL
The original RTL design.
is automatically converted into
Partial Model
A transition system with uninterpreted functions generated from the RTL. See Section 3.2.1.
Program Sketch
A program with holes, specified by the user. See Section 3.2.2.
is turned into
Concrete Program
A program sketch with its holes filled by random values, or values from previous counterexamples.
is executed on
I/O Oracle
An RTL simulator that produces I/O examples. See Section 3.2.2.
defines synthesis functions for
gives constraints to
Synthesis Engine
A SyGuS engine that synthesizes interpretations for uninterpreted functions in the partial model. See Section 3.2.2.
queries
Correctness Oracle
A model checker that verifies the correctness of synthesized functions. See Section 3.2.2.
on failure, provides inputs to
on pass, produces concrete functions for
Complete Model
A model where some or all of the uninterpreted functions in the partial model have been replaced by concrete interpretations.

Figure 3.1: Flowchart outlining the entirety of the lifting procedure.

- `BoolConst`, `BVConst`: These represent boolean constants (true/false) and bit-vector constants (e.g. `BVConst(3, 3)` represents the 3-bit bit-vector `011`).
- `Variable`: Variables have a name and a sort; `Variable("x", BVSort(3))` represents a reference to a 3-bit bit-vector variable named `x`.
- `OpTerm`: `OpTerm`s represent more complex expressions like bit-vector manipulations and comparisons. Each `OpTerm` has a `Kind` representing the sort of operation to perform to its arguments, which are other terms. For example, adding together two bit-vector variables is expressed by `OpTerm(Kind.BVAdd, (x, y))`, or the syntactic sugar `x + y`. The sort of an `OpTerm` can be inferred from the sorts of its arguments; the previous bit-vector addition term would have the same sort as `x` and `y`.
- `UFTerm` and `ApplyUF`: `UFTerm` represents an uninterpreted function, which has a name, a codomain sort, and a vector of parameters. `ApplyUF` represents the application of an uninterpreted function to an appropriate vector of argument terms.

All expressions are checked for valid argument arity and sorts by the Python framework as they are constructed; for example, an `OpTerm` representing bit-vector addition would ensure that there are exactly two arguments, and that both are bit-vectors of the same width.

3.1.2 Model Definition

Let E be the set of all possible terms, as defined above. A RTL2MODEL model is a tuple of the following form: $M = (W, O, S, U, U_{next}, P, L, N)$ where

W	is a vector of input variables,
O	... output variables,
S	... state variables,
U	... uninterpreted functions representing combinatorial expressions,
U_{next}	... uninterpreted functions representing transition relations (see Section 3.2.1),
$P : string \to (\{W_M \to E\} \times M)$	is a map of instance names to submodules with input bindings,
$L = (V \subseteq (S \cup O)) \to E$	represents combinatorial expressions that determine the values of state variables and outputs,
$N = (V \subseteq (S \cup O)) \to E$	represents transition relations for register variables that determine their value on the next clock cycle.

The lists of inputs and outputs for a RTL2MODEL model align with the input/output interface of the RTL module being modeled. In previous ILA works [14], the state variables in S are restricted to “architectural”, or software-visible state that persists across instructions.

By contrast, state variables in RTL2MODEL models can represent any level of microarchitectural granularity, and includes signals whose values are determined by combinatorial logic.

A user may choose to replace some signals by uninterpreted functions of varying arity, and which may take on state variables and other uninterpreted functions as arguments (this is discussed in further detail in 3.2.1). We refer to RTL2MODEL models with such uninterpreted functions as *partial models*, and those without as *complete models*.

3.1.3 Translation to Other Representations

Expressions in RTL2MODEL can be translated into SyGuS or CVC5 Python binding expressions to perform synthesis; our framework uses CVC5 as a synthesis backend. Partial models can be hierarchically translated into equivalent UCLID5 models (as UCLID5 supports uninterpreted functions), and complete models can be translated to either UCLID5 or Verilog.

3.2 Model Construction

3.2.1 Generation From Verilog

Users may manually construct models as outlined above, or can call the `verilog_to_model` function to generate them from the dataflow graph of an RTL design. We obtain a dataflow graph from a modified fork of Pyverilog[1] [34], a Python library for parsing Verilog circuits with built-in dataflow and control flow analysis functions. We formally represent the dataflow graph as a pair of edgelists $(E_{logic}, E_{transition})$. Each entry in both edgelists maps a signal to a RTL2MODEL expression that determines its value; E_{logic} represents same-cycle combinatorial relations, whereas $E_{transition}$ represents next-cycle state updates. For example, the below Verilog snippet induces a graph where $E_{logic} = \{x : \{b, c\}\}$, $E_{transition} = \{y : \{a, b\}\}$.

```verilog
module top(input clk, input a, input b, input c);
    wire x;
    reg y;
    always @(posedge clk) begin
        y <= a + b; // Clocked register assignment
    end
    assign x = b + c; // Combinatorial assignment
endmodule
```

Every non-input signal in the design has a corresponding entry in either E_{logic} or $E_{transition}$, but not both. Figure 3.2 depicts an RTL circuit and its corresponding dataflow diagram.

Once a dataflow graph is obtained, it can readily be translated into a RTL2MODEL model: RTL inputs correspond to model inputs, RTL state variables correspond to model state

[1] `https://github.com/noloerino/Pyverilog/tree/term-memoization`

```
module top(input clk,
           input rst,
           input [1:0] a,
           output [1:0] o);
    wire [1:0] cl_0;
    reg [1:0] state_0;
    reg [1:0] state_1;
    always @(posedge clk) begin
        if (rst) begin
            state_0 <= 2'b00;
            state_1 <= 2'b00;
        end else begin
            state_0 <= cl_0;
            state_1 <= cl_1;
        end
    end
    assign cl_0 = a ^ 2'b11;
    assign o = state_1;
endmodule
```

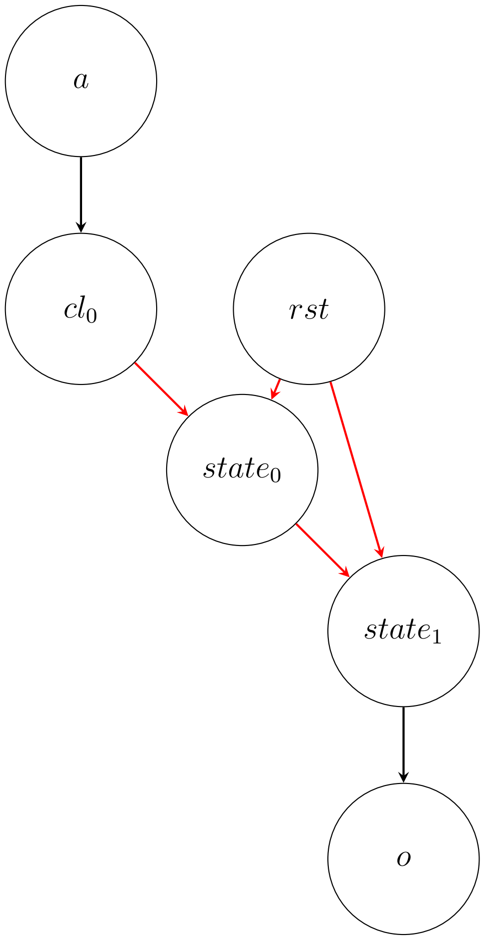


Figure 3.2: Verilog circuit and its corresponding dataflow graph. Same-cycle edges from E_{logic} are drawn in black, while cross-cycle edges from $E_{transition}$ are drawn in red.

variables, and so on. Variables described by combinatorial logic in E_{logic} will have entries in L, and those with transition relations in $E_{transition}$ will have entries in N. The RTL2MODEL expression for each assignment is produced by using Pyverilog to parse and traverse the Verilog AST. Simple expressions like addition and bit manipulation operators have very straightforward correspondences; for example, the Verilog expression `a & b` is translated to RTL2MODEL expression `OpTerm(Kind.BVAnd, (a, b))`. The following Verilog constructs have non-trivial translations:

- `if/else` statements: RTL2MODEL cannot directly model `if/else` statements where the `else` case is missing, and instead converts them to chains of ternary expressions with inferred latches. For example, the Verilog code

```
module top(input clk, input [1:0] c, output [1:0] o);
    always @(posedge clk) begin
        if (c == 0) o <= 2'h3;
        else if (c == 1) o <= 2'h2;
        // Missing else branch
    end
endmodule
```

results in the following pseudocode expression tree for `o`'s transition relation:

```
c = Variable("c", BVSort(2))
o = Variable("o", BVSort(2))
T[o] = OpTerm(Kind.Ite, (
    OpTerm(Kind.Eq, (c, BVConst(0, 2))),      # if condition
    BVConst(3, 2),                            # if true branch
    OpTerm(Kind.Ite, (                        # if false branch
        OpTerm(Kind.Eq, (c, BVConst(1, 2))), # else if condition
        BVConst(2, 2),                        # else if true branch
        o,                                    # else branch
                                              #      (inferred latch)
    )),
))
```

In all designs we examine, this assumed latch is either intended, or explicitly prevented by the structure of the RTL.

- `case` statements: Due to limitations in Pyverilog's dataflow analysis, Verilog `case` statements are translated into `if/else` chains with appropriate conditions, instead of a simpler "match" construct like that in SMT-LIB. For example, the above pseudocode expression tree for `o` could also be produced by this RTL module:

```
module top(input clk, input [1:0] c, output [1:0] o);
    always @(posedge clk) begin
        case (c)
            2'h0 : o <= 2'h3;
            2'h1 : o <= 2'h2;
        endcase
    end
end
```

- Variable bit-vector indexing: Since RTL2MODEL operators more or less mirror operators available in SMT-LIB, indexing a bit-vector by another bit-vector variable is prohibited, as indices of a bit-vector extract must be constant. However, this is a valid operation in Verilog, and appears in one of our case studies in a snippet similar to the following:

```verilog
// All signals are 8 bits wide
always @* begin
    wr_bit_byte = data0;
    if (/*<condition>*/)
        wr_bit_byte[cmd1[2:0]] = 1'b0;
    else if (/*<other condition>*/)
        wr_bit_byte[cmd1[2:0]] = ~wr_bit_byte[cmd1[2:0]];
end
```

  Here, the 8-bit `wr_bit_byte` signal is indexed by a 3-bit signal (`cmd1[2:0]`) to change the value of a particular bit. We model this assignment as a combination of bit-shifts and masks, which are allowed in SMT-LIB and RTL2MODEL. In particular, the Verilog code `x[idx] = y` is equivalent to `x = (x & ~(1 << idx)) | (y << idx)`.

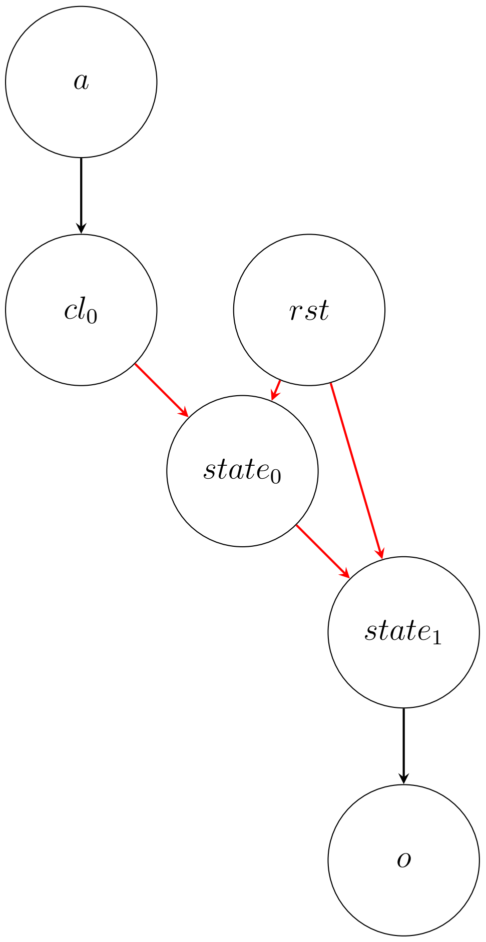
Figure 3.3 demonstrates the entire process of translating RTL to dataflow graph to RTL2MODEL model on a simple circuit. All instances of submodules encountered in the top-level design are recursively converted into RTL2MODEL models by the above procedure; all instances of the same RTL module will share the same RTL2MODEL model. If the user wishes to supply a pre-generated model (either manually constructed or lifted from RTL prior), `verilog_to_rtl` takes as optional argument a list `defined_modules` that will be used instead.

It is worth noting that Hsiao et al. [13] use Verific and Yosys to generate a dataflow graph directly from a netlist. We choose Pyverilog for easier integration with the rest of the RTL2MODEL Python codebase, but the alternative implementation is also viable.

Cone-of-Influence Analysis

In order to reduce the state space and complexity of a generated model, the user may choose to enable a *cone-of-influence* (COI) analysis that automatically eliminates irrelevant state variables. Traditionally, the term "cone-of-influence" is used to refer to the set of all variables that a particular property or formula is influenced by; we use it here to refer to the set of all ancestors of a particular signal in the dataflow graph of a circuit, including itself. The user specifies a list of "important" signals in the top-level module they wish to preserve, and one of three configuration options to model all non-important signals:

1. `NO_COI` (default): No COI analysis is performed; any non-important signals referenced in the expression for an important signal is left as a 1-argument uninterpreted function



```python
bv2 = BVSort(2)
rst = Variable("rst", BoolSort())
a = Variable("a", bv2)
o = Variable("o", bv2)
c1_0 = Variable("c1_0", bv2)
state_0 = Variable("state_0", bv2)
state_1 = Variable("state_1", bv2)
Model(
    "top",
    inputs=[rst, a],
    outputs=[o],
    state=[c1_0, state_0, state_1],
    logic={
        c1_0: a ^ 0b11,
        o: state_1,
    },
    transition={
        state_0: rst.ite(0, cl_0),
        state_1: rst.ite(0, state_0),
    },
)
```

Figure 3.3: Conversion of the dataflow graph in Figure 3.2 to a model. Expression operations are parsed from the RTL, and convenience methods are used for simplicity (e.g. `x ^ y` is equivalent to `OpTerm(Kind.BVXor, (x, y))`. When applicable, Python integers are automatically converted into a bit-vector constant of appropriate size.

of equivalent width. This single argument is necessary because uninterpreted functions are deterministic; that is, for an uninterpreted function f, then $x = y \implies f(x) = f(y)$. We therefore introduce a single argument variable that does not correspond to any hardware signal in the design in order to ensure the uninterpreted function is not assigned a constant value by the solver.

2. `KEEP_COI`: Any signal found in the COI of an important signal is explicitly preserved as a state variable.

3. `UF_ARGS_COI`: Any signal x found in the COI of an important signal is replaced as an uninterpreted function. Unlike with `NO_COI`, the uninterpreted function that models x takes as argument any important signals in the COI of x in the original design. If the COI of x contains more signals than x's immediate parents, then the function also takes a free variable argument as described earlier.

   In order to account for the presence of cross-cycle dependencies in the cone-of-influence, any uninterpreted functions, or parents of uninterpreted functions, that are assigned by a transition relation in the RTL (that is, not combinatorially) are themselves preserved as an uninterpreted function. These functions form the U_{next} set, and implicitly define an additional register state variable; a synthesized interpretation for a member of U_{next} is the transition relation for this new state variable.

Figure 3.4 illustrates these different COI behaviors on a simple circuit.

The COI computation procedure roughly follows the "fast cone-of-influence" algorithm described by Locaino et al. [22]. The pseudocode is described in Algorithm 1, where DG represents the dependency graph computed from the RTL (with E_{logic} and $E_{transition}$ combined together), and $S_{important}$ is the vector of important signals specified by the user. $DG[s]$ returns the direct parents of s in the dependency graph.

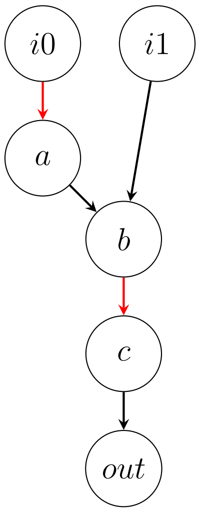
Note that we do not perform inter-module dependency analysis. When constructing the dependency graph for a parent module, we effectively treat each child instance as a single signal: if there exists an assignment of the form `parent.x = parent.child.output_value`, then `parent.x` is assumed to depend on all input signals to instance `parent.child`. Furthermore, we only perform COI analysis on signals declared within the top-level module: if a user wishes to apply COI analysis to submodules, they can lift those from RTL and pass the resulting models with the `defined_modules` argument.

3.2.2 Synthesis Algorithm

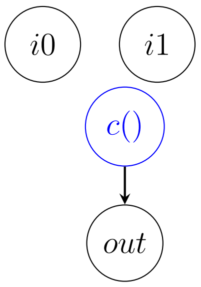
Since the purpose of the lifted model is to demonstrate that certain properties hold on the original RTL design, our model must be *consistent* with the RTL; intuitively, any behavior possible in an RTL simulation should be possible in the lifted model as well. We do not necessarily require soundness, meaning it is acceptable (though undesirable) to have a behavior that is possible in the model but not under the original hardware design.

```
module top(input clk,
           input [1:0] i0,
           input [1:0] i1,
           output [1:0] out);
    reg [1:0] a;
    reg [1:0] b;
    reg [1:0] c;
    always @(posedge clk) begin
        a <= i0 + 1;
        c <= b;
    end
    assign b = a & i1;
    assign out = c;
endmodule
```

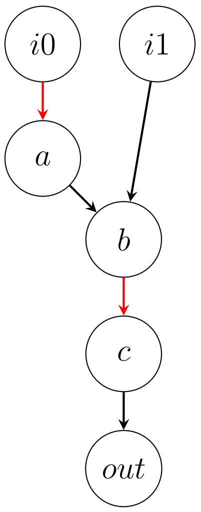
(a) Sample circuit RTL.



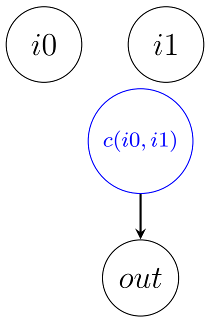
(b) Dataflow graph for circuit.



(c) Graph representing partial model with no COI checks; all variables that are immediate parents of designated "important" variables are left as UFs.



(d) Graph for partial model where referenced variables in the COI are preserved. Because every signal is int he COI of *out*, every signal is preserved, and the model effectively matches the original dataflow graph.



(e) Partial model where referenced important variables in the COI are used as arguments to UFs. Because $i0$ and $i1$ are in the COI of c, they are passed as arguments to c.

Figure 3.4: Demonstration of various cone-of-influence behaviors for a simple circuit. In each case, the specified list of "important signals" is $\{out, i0, i1\}$. State left as uninterpreted functions are drawn in blue, with arguments as applicable.

Algorithm 1 Pseudocode for cone-of-influence computation.

```
1: procedure ConeOfInfluence($DG$, $S_{important}$)
2:     $C \leftarrow \{\}$                                   ▷ Maps signal to its COI set
3:     $V \leftarrow \emptyset$                                   ▷ Set of visited signals
4:     for $s \in S_{important}$ do
5:         if $s \notin V$ then
6:             Visit($C$, $DG$, $V$, $s$)
7:         end if
8:     end for
9:     return $C$
10: end procedure

11: procedure Visit($C$, $DG$, $V$, $s$)
12:     $V \leftarrow V \cup \{s\}$
13:     $parents \leftarrow DG[s]$
14:     if $s \notin C$ then
15:         $C[s] \leftarrow \{s\}$
16:     end if
17:     for $p \in parents$ do
18:         if $p \notin V$ then
19:             if $p \notin C$ then
20:                 $C[p] \leftarrow \{p\}$
21:             end if
22:             $C[p] \leftarrow C[p] \cup$ Visit($C$, $DG$, $V$, $p$)
23:         end if
24:         $C[s] \leftarrow C[s] \cup C[p]$
25:     end for
26: end procedure
```

Program Sketch and Refinement Relations

We require the user to provide a program sketch that is used as input to the simulator I/O oracle. Holes in the sketch correspond to the inputs of some synthesis function in the partial model.

We also require the user to define a refinement relation between model state variables and the RTL, which is specified through annotations. Each annotation is associated with a predicate p over RTL state. There are four types of annotations possible for some RTL signal s when p holds:

1. **Don't Care**(p): When p holds, the value of s is irrelevant to the model, and can be left symbolic (this is the default).

2. **Assumed**(p): When performing the correctness check, the value of s when p holds is assigned the constant value that was encountered on that cycle during simulation. This is used for signals that are fixed by the program sketch, such as values for instructions to initialize state.

3. **Parameter**(p, v): When p holds, the value of this signal is that of the corresponding hole v in the program sketch, and an input to some synthesis function in the model. In the correctness check, this will be left as a solver-chosen variable.

4. **Output**(p, f): When p holds, the value of this signal represents the output of some synthesis function f in the model. In the correctness check, this will be replaced by the appropriate synthesized candidate function.

The refinement relations can be difficult to specify correctly, and an error may easily cause spurious synthesis results. However, we are not aware of any simpler methods of specifying correspondence without forcing the model's state variables to have a 1-to-1 mapping with RTL signals.

This aspect of our work is very similar to previous synthesis efforts in ILAng [14, 33]. ILAng users must hand-craft templates for synthesis; our users must still provide manual program sketches and refinement mappings, but the templates themselves are derived from the RTL.

I/O and Correctness Oracles

We make use of a hardware simulator I/O oracle and a model checker correctness oracle to produce inputs for synthesis. In all our examples, Verilator [30] serves as the simulator, and SymbiYosys [12] as the model checker.

The I/O oracle simulates the behavior of a concrete program (a program sketch with all holes filled) on the RTL. Using the refinement relation, we mine the resulting simulator trace to obtain the values of our annotated output signals. Combined with the inputs used to fill the holes in the program sketches, this constitutes a single input/output pair that is used to constrain the synthesizer.

The correctness oracle checks that the synthesized functions are valid under the specified program sketch. The assumed values, parameters, and outputs from the refinement relation are encoded into SystemVerilog assertions, and a bounded model check is performed on the result.[2] If a counterexample is produced, its inputs will be used as the next set of inputs to the I/O oracle. If the outputs of the I/O oracle contradict the outputs identified in the counterexample by the correctness oracle, then this likely indicates either a bug in the refinement relation or simulation code.

[2]For simplicity, all our examples run the model checker for a fixed number of cycles until the last instruction in the concrete program commits. We assume the user is able to identify an upper bound on this cycle count in advance. In principle, an unbounded inductive check can also be used in the correctness oracle.

CEGIS Loop

We use CVC5 [3] as a synthesis engine, and translate RTL2MODEL expressions to CVC5's Python bindings to add constraints. Putting all the components together gives us the following CEGIS loop, as outlined in Figure 3.1.

1. The user supplies the **original RTL** designs, a **program sketch**, a cone-of-influence setting, and a set of refinement annotations.
    a) By calling `verilog_to_model` with a list of important signals, the user transforms the RTL design into a **partial rtl2model model**.
    b) The user identifies a list of uninterpreted functions in this partial model to synthesize.
2. Synthesis loop:
    a) Counterexamples from previous calls to the correctness oracle determine values for certain RTL signals (on the first iteration, these are random values). These values are then matched against the user-specified **refinement annotations** to fill the program sketch's holes. The program sketch becomes a **concrete program**.
    b) The execution of the concrete program is simulated on the RTL by the **I/O oracle**, and input/output pairs for synthesis functions are generated. These constraints are then fed to the **synthesis engine**.
    c) The synthesis engine generates a set of **candidate functions** that satisfy the I/O constraints.
        - If synthesis fails, then there is likely a bug with one of the oracles, the program sketch, or refinement relations. No valid model can be generated.
    d) If synthesis succeeds, these candidate functions are passed to the **correctness oracle**, which verifies their consistency with the program sketch and design.
        - If the correctness check fails, the resulting counterexample is saved as a constraint, and its inputs are used as the next set of inputs to the program sketch. Synthesis continues with step 2(a).
    e) If the correctness check passes, then the candidate functions are used to replace the uninterpreted functions in the partial model.

3.3 The rtl2model Python Framework

We implement the aforementioned cone-of-influence and synthesis techniques in the RTL2MODEL Python framework. Some of its features are described here.

Oracle Interactions

Users must specify a set of shell commands to compile the I/O oracle simulation binary, in addition to code to read simulation traces from CSV files and write new concrete programs for the I/O oracle to execute. The framework automatically generates C++ code that can be linked with a Verilator simulation file to specify what RTL signals should appear in the resulting trace.

Once the commands to invoke the I/O and correctness oracles are specified, the framework automatically handles all relevant interactions with them throughout the synthesis loop: new concrete programs are generated automatically from counterexamples, which are parsed from traces output by SymbiYosys, our model checker.

Verilog Parsing and COI Algorithms

RTL2MODEL transition system models can be constructed manually with the expression DSL described in Section 3.1.1, or parsed directly from RTL. Consider this example code for parsing the `top` module from a file named `full.v`:

```
alu = ... # Defined in some previous code
top = verilog_to_model(
    verilog_path="full.v",
    top_name="top",
    defined_modules=[alu],
    important_signals=["i1", "i2", "s1", "s2", "out"],
    coi_conf=COIConf.UF_ARGS_COI,
)
```

The `verilog_path` is self-explanatory. `top_name` specifies the name of the top-level module to parse from the file.

`defined_modules` provides a list of sub-models already defined either by manual construction or a previous call to `verilog_to_model`. This allows for compositional applications of our synthesis techniques, where a simpler version of a sub-model can be constructed and verified, and then reused in verification of the top-level model.

`important_signals` represents the list of "important" state variables to preserve when constructing the model, and `coi_conf` determines how that list is used in pruning other state. Refer back to Section 3.2.1 for a discussion on these techniques.

Program Sketch and Refinement Specification

Users construct `ProgramSketch` objects as combinations of variable holes and concrete binary values. As an example, the following program sketch represents the RISC-V program `addi x11, x0, ??; addi x12, x0, ??; ?? a3, a1, a2` (the immediates of the first two instructions and the operation of the last are holes). `short_a` and `short_b` are defined as 12-bit bit-vector variables, and `f3` as a 3-bit bit-vector.

```
ProgramSketch(
    # First instruction: addi a1, x0, ??
    # The immediate value is encoded in the upper 12 bits of the
    # instruction, which are left as a hole. The lower 20 bits are
    # fixed to specify registers and the addition operation.
    Inst(short_a, SketchValue(0b10110010011, 20)),
    # Second instruction: addi a2, x0, ??
    Inst(short_b, SketchValue(0b11000010011, 20)),
    # Third instruction: ?? a3, a1, a2
    # The most significant 17 bits, as well as the least significant 12
    # bits, contain register and operation information. The middle 3
    # bits determine the arithmetic operation being performed, and are
    # left as a hole.
    Inst(SketchValue(0x18b, 17), f3, SketchValue(0x6b3, 12)),
    # 20 occurrences of the NOP instruction to flush the pipeline
    inst_word(0x13) * 20,
)
```

The burden remains on the user to understand the relationship between ISA/ILA-level instructions and their binary encoding, but we find this to be a reasonable restriction.

The relationship between sketch holes and RTL signals in counterexample traces is defined by refinement annotations. Annotations for an RTL signal are a mapping of SMT expressions to an annotation type (Assumed, Parameter, or Output); the annotation type is applied on cycles in the simulation when the predicate holds. For example, the following annotation expresses when the value of `f3` should be sampled from the RTL:

```
guidance.annotate("lft_tile_fe_inst", {
    # pc is a variable representing the program counter
    # This predicate is true when the PC is 0x208
    pc.op_eq(0x208): [
        # Bits 31-15 and 11-0 of the instruction are fixed in the program
        # sketch, and are therefore fixed in simulation as well
        AnnoType.AssumeIndexed((31, 15), (11, 0)),
        # Bits 14-12 of the instruction on this cycle correspond to the
        # f3 input variable
        AnnoType.ParamIndexed((14, 12), f3)
    ],
})
```

SyGuS Grammars

Users construct gramamrs for synthesis functions in a declarative manner closely resembling BNF; the framework converts them automatically into the appropriate calls to the SMT solver backend. All terms in the grammar are specified in the RTL2MODEL expression language, and users can use Python operator overrides as syntactic sugar for many operations. The following is an example of a grammar for the function `s0(i1 : bv8, i2 : bv8) : bv8`. Note that the Python `range` operator allows for ergonomic specification of a range of bit-vector constants.

```python
bv8 = smt.BVSort(8)
i1 = smt.Variable("i1", bv8)
i2 = smt.Variable("i2", bv8)
start = smt.Variable("start", bv8)
substart = smt.Variable("substart", bv8)
grammar = smt.Grammar(
    bound_vars=(i1, i2),
    nonterminals=(start, substart),
    # Each entry in this dictionary represents a nonterminal
    # and a list of its expansion rules
    terms={
        start: (
            start.ite(start, start),
            substart,
        ),
        substart: (
            substart + substart,
            substart - substart,
            substart | substart,
            i1, i2,
            *list(smt.BVConst(n, 8) for n in range(0xFF)),
        ),
    }
)
```

3.3.1 Implementation Limitations

To automate the lifting process as much as possible, we would like to minimize the amount of input and complexity of inputs required by the user. Currently, the refinement annotations, program sketch, and SyGuS grammar specifications require the user to possess a modest amount of knowledge of the design under test, and remain easy to specify incorrectly. We found the construction of grammars to be especially impactful, as leaving the grammar for a function unspecified (which allows the solver to choose any valid term of the correct sort) result in synthesis candidates that were long `if/else` chains conditioned on the input/output examples observed, and did not produce a correct function in a reasonable amount of iterations. However, as with most applications of SyGuS, we find the requirement for users to specify grammars with common operations to be a reasonable one.

Control flow instructions, which may cause a different sequence of instructions to execute on the processor, are also difficult to verify under our framework because the fetch and commit of instructions are identified based on the program counter of the appropriate pipeline stage. Since control flow instructions may affect the program counter in unexpected ways due to the extra cycles needed to kill instructions currently in the pipeline when a jump is taken, we chose to avoid verifying any control flow instructions in this work.

Non-Combinatorial Functions

The most important limitation at present is that RTL2MODEL can only synthesize interpretations for purely combinatorial functions; that is, any uninterpreted functions for transition relations cannot be synthesized under our framework, as allowing functions to span multiple cycles would significantly increase the complexity of constraints provided by the I/O oracle. A workaround for this is to augment the RTL design with extra registers to store values of the signal from previous cycles, though this is inelegant and requires some amount of manual labor.

More fundamentally, refinement annotations currently must explicitly refer to the inputs and outputs of the functions being synthesized. If users were instead allowed to specify constraints over any signal in a trace, this would allow synthesis of multi-cycle functions, at the aforementioned cost of increased complexity in solver constraints. For example, if a user specifies as output some signal $x = f(a, b)$, where f is an uninterpreted function and a and b are either uninterpreted functions or state variables, the expression for the asserted I/O constraint must now contain the expressions/interpretations for a and b as well. This is feasible if f is a purely combinatorial relation, but if f is a *transitional* relation, things are significantly complicated: our constraint now depends on previous values of a and b, which would potentially require encoding the entirety of the transition system outlined by the model into SyGuS constraints and assumptions. While this is also possible, it may significantly increase the burden on the solver, and additional investigation is required to test its scalability.

This limitation also restricts the applicability of cone-of-influence techniques to model generation: currently, these algorithms are only used to identify dataflow dependencies that are then used as the arguments of functions to synthesize. Allowing functions to model relations that span multiple clock cycles would allow these COI techniques to automate away larger portions of the lifting process by possibly making refinement relations simpler, as discussed above.

Name	Verilog source LoC	Type
R8051	1064	general-purpose processor
riscv-mini	2381[4]	general-purpose processor

Table 3.1: Summary of designs modeled.

3.4 Evaluation

We evaluate RTL2MODEL by constructing models for a subset of the behaviors of two designs:

- **R8051** [20]: a general-purpose processor implementing the 8051 ISA
- **riscv-mini** [18]: a single-core, in-order, 32-bit RISC-V processor with three pipeline stages

The RTL designs for each are slightly modified to make exposing signals to the RTL simulator easier; the behavior of the processors is not fundamentally changed. Partial models were lifted from RTL using our Python framework, with various parameters for synthesis and COI analysis manually specified as described below.[3]

All experiments are run on a 4.8GHz AMD Ryzen 7 5800X processor on Fedora Linux 35. All code in the framework is single-threaded.

3.4.1 R8051

The R8051 processor is the simplest of the designs we examine; the design is contained in a single RTL module, and has a three-stage pipeline, where each stage processes one instruction byte at a time. We synthesize a function to model the behavior of the `psw_c` signal, which is an architectural status flag set to indicate the presence of a carry bit, when a register-accumulator addition instruction is executed. The specific signature of the function we synthesize is `psw_c(data1 : bv8, acc : bv8) : bool`, where `data1` is a register value and `acc` is the accumulator value.

We choose to synthesize this property because it is specified by a moderately complex expression in the RTL. An abridged version of the Verilog code to set the next value of `psw_c` is as follows:

```
always @(posedge clk) begin
    if (rst) psw_c <= 1'b0;
    else begin
        // Large if/else tree determining behavior for different ops
```

[3]All artifacts can be found on GitHub: `https://github.com/uclid-org/rtl2model`.

[4]riscv-mini is originally written in Chisel; we examined the emitted equivalent Verilog.

```
        if (/*<operation is addition or subtrraction>*/)
            // `add_a`, `add_b` are the arguments being added/subtracted,
            // `high` contains the upper bits of the addition
            psw_c <= is_subtract ? (add_a[7] - add_b[7] - high[3])
                                 : (add_a[7] + add_b[7] + high[3]);
        else if (/*<operation is clear>*/)
        else if (/*<operation is multiplication>*/)
        // etc.
        // ...
    end
end
```

Our goal is to demonstrate that synthesis can be used to lift models simpler than the original RTL; in this case, we wish to model the behavior of `psw_c` when only an addition instruction is executed.

Program Sketch

The 8051 ISA specifies variable-length instruction encodings, but the processor is pipelined to only ever process a single byte at a time. This means we need to add further `nops` after our instructions of interest to simulate a pipeline flush. This implementation detail does not otherwise affect the refinement annotations or program sketch.

```
NOP                   # PC=1
MOV R0, ??            # PC=2 2-byte instruction; sets data1
MOV A, ??             # PC=4 2-byte instruction; sets acc
ADD A, R0             # PC=6 1-byte instruction; performs addition
NOP                   # 20 more nops afterwards
```

Refinement Annotations

Recall that values “assumed” on a particular cycle are set to the corresponding value during simulation, “parameter” values are constants chosen by the solver, and “outputs” are the output of the synthesis function when called on those inputs.

Because of the variable-byte nature of the 8051 instruction encoding, the architectural program counter does not correspond to the number of executed instructions; rather, it corresponds to the byte index of the executed instructions. Both instructions with holes in our sketch are 2 bytes long, while all others are a single byte.

`cmd0`, `cmd1`, and `cmd2` represent the instruction byte being processed in each pipeline stage. `pc` is the program counter of the first (0th) stage.

- **Assumed**:
  - On every cycle: `rst`, `pc`
- **Parameters**:
  - Register `data1` when `pc == 3` (since the first byte of `MOV R0, ??` is at `PC=2`, the immediate enters the pipeline on the next byte when `PC=3`)
  - Register `acc` when `pc == 5` (since `MOV A, ??` starts at `PC=4`, the immediate enters the pipeline when `PC=5`
- **Output**: Flag `psw_c` when `pc == 8`

Synthesis Grammar

Below is the pseudocode grammar used in synthesizing `psw_c(data1, acc) : bool`. Zero-padding of the 8-bit arguments is introduced into the grammar to avoid possible overflows, as are the magic numbers 0 and $2^8 - 1$ (`0xFF`). The need for this sort of user-provided guidance is one of the weaknesses of SyGuS (as discussed in Section 3.3.1).

$\langle start \rangle$ ::= $\langle bv9 \rangle$ == $\langle bv9 \rangle$
  | $\langle bv9 \rangle < \langle bv9 \rangle$ [unsigned]
  | $\neg\langle start \rangle$
  | $\langle start \rangle \vee \langle start \rangle$s
  | $\langle start \rangle \wedge \langle start \rangle$
  | $\langle start \rangle \oplus \langle start \rangle$
  | true

$\langle bv9 \rangle$ ::= $\langle bv9 \rangle + \langle bv9 \rangle$
  | $\langle bv9 \rangle$ - $\langle bv9 \rangle$
  | $\langle bv9 \rangle \mid \langle bv9 \rangle$
  | $\langle bv9 \rangle$ & $\langle bv9 \rangle$
  | $\langle bv9 \rangle \oplus \langle bv9 \rangle$
  | $\langle bv8 \rangle$.zero_pad(1)
  | $\langle bv8 \rangle$.sign_extend(1)

$\langle bv8 \rangle$ ::= x
  | y
  | 0bv8
  | 0xFFbv8

Synthesized Function

The framework generates the following SyGuS function:

```
(define-fun psw_c
    ((lft_data1 (_ BitVec 8)) (lft_acc (_ BitVec 8))) Bool
    (bvult ((_ zero_extend 1) #xff) (bvadd ((_ zero_extend 1) lft_data1) ((_ zero_extend 1) lft_acc))))
```

As expected, it returns true if the result of adding the two zero-padded operands would exceed the maximum value of one byte (`0xFF`). The extraneous decoding logic and addition optimizations seen in the RTL implementation have been elided.

Performance

Table 3.2 presents a breakdown of the runtimes of each component of the lifting algorithm. The bulk of the runtime is taken up by invocations to the correctness oracle, but it still completes in a very reasonable amount of time.

Operation	Count	Total (s)	Average (s)
Dataflow parsing from Verilog	1	1.437	1.437
Partial model construction from dataflow	1	0.120	0.120
I/O oracle invocation	7	0.020	0.003
Correctness oracle invocation	7	21.857	3.122
Solver synthesis invocation	7	0.060	0.009

Table 3.2: Runtimes for various synthesis algorithm components for R8051. The “count” column for the oracle and solver invocations refers to the number of iterations in the synthesis loop outlined in Figure 3.1.

3.4.2 riscv-mini

The riscv-mini processor is a three-stage RISC-V pipeline. We synthesize an interpretation for its ALU function under a set of 4 register-register instructions: `add`, `xor`, `or`, and `and`. The specific signature of the function is `alu_area(a : bv12, b : bv12, f3 : bv3)`, where `a` and `b` are 12-bit immediates loaded into registers, and `f3` is a field in instruction encoding that determines which arithmetic operation is performed. Although the processor and ALU operate on 32-bit values, we limit the inputs to the program sketch to 12-bit values, since RISC-V instruction encoding restricts immediates to that size. 32-bit immediates require 2 instructions to load; we could have changed the program sketch to accommodate this, but chose not to for simplicity

In the original RTL, the ALU result is expressed as a large case expression. We seek to lift a model with a subset of these cases.

We evaluate the effectiveness of our lifting framework on this function in three different scenarios:

①: All components (refinement annotations, sketch, grammar) are specified correctly.

②: The refinement annotations and program sketch are correct, but a few operations are missing from the grammar.

③: The refinement annotation is specified incorrectly.

Program Sketch

Since riscv-mini initializes the program counter to byte address `0x200` (512 in decimal) on reset, our program sketch needs to contain padding bytes to fill those out. We also include `nops` at the end to function as a "flushing" mechanism.

```
unimp                  # 496 bytes of 0s
# Due to some quirks in how the pipeline is initialized,
# we need to include a few nops
nop; nop; nop; nop
addi x11, x0, ??       # PC=0x200 Hole for short_A
addi x12, x0, ??       # PC=0x204 Hole for short_B
??   x13, x11, x12     # PC=0x208 Hole for f3
nop                    # 20 more nops afterwards
```

`f3` is finally constrained to 3 possible values: 000 (addition), 100 (bitwise XOR), 110 (bitwise OR), and 111 (bitwise AND).

Refinement Annotations

`fe_pc` is the program counter of the instruction in the second pipeline stage, and `ew_pc` is the program counter of that in the third and final stage. `fe_inst` represents the binary encoding of the instruction currently in the second stage.

- **Assumed**:
  - On every cycle: `reset`, `pc`, `fe_pc`, `ew_pc`
  - Partially assumed on certain cycles:
    * `fe_inst[19:0]` when `fe_pc == 0x200` or `0x204` (everything except the immediate)
    * `fe_inst[31:15]` and `fe_inst[11:0]` when `fe_pc == 0x208` (everything except `f3`)
    * The entirety of `fe_inst` otherwise
- **Parameters**:
  - Register `x11` when `ew_pc == 0x204`
  - Register `x12` when `ew_pc == 0x208`

  - Fetched instruction `fe_inst[14:12]` when `fe_pc == 0x208`

- **Output**: Register `x13` when `ew_pc == 0x20C`

The above refinement relation is used for scenarios ① and ②. In the incorrect refinement relation for scenario ③, the `ew_pc` predicates for `x11` and `x12` are erroneously swapped, leading the correctness oracle to sample the incorrect registers.

Synthesis Grammar

Below is the pseudocode grammar used in scenario ① (the fully correct one) for synthesizing the function `alu(short_A : bv12, short_B : bv12, f3 : bv3) : bv32`. Once again, we provide a set of common bit-vector operations, but also need additional specifications for "magic number" constants like the possible values of `f3`, and decoding-related behavior to sign-extend the 12-bit function inputs to 32 bits. We also must specify the basic structure of the case split by conditioning on the value of `f3`.

```
⟨start⟩       ::= (f3 == ⟨bv3⟩).ite(⟨start⟩, ⟨start⟩)
                |  ⟨substart⟩

⟨substart⟩    ::= ⟨substart⟩ + ⟨substart⟩
                |  ⟨substart⟩ - ⟨substart⟩
                |  ⟨substart⟩ | ⟨substart⟩
                |  ⟨substart⟩ & ⟨substart⟩
                |  ⟨substart⟩ ⊕ ⟨substart⟩
                |  ⟨boolterm⟩.ite(1bv32, 0bv32)
                |  ⟨boolterm⟩.ite(0xFFFFF000bv32, 0bv32)
                |  short_A.sign_extend(20)
                |  short_B.sign_extend(20)

⟨boolterm⟩    ::= short_A.extract(11)
                |  short_B.extract(11)

⟨bv3⟩         ::= 0b000bv3
                |  0b100bv3
                |  0b110bv3
                |  0b111bv3
```

In scenario ②, we underspecify the grammar by removing the expansion rules for *⟨substart⟩*&*⟨substart⟩* and *⟨substart⟩* ⊕ *⟨substart⟩*. A valid function is still synthesized, but the bitwise XOR and AND operators are omitted from the grammar, leading to a more complex expression in the result.

Synthesized Function

Scenario ①. In scenario ①, the following SyGuS function is produced (we provide Python pseudocode below it for readability):

```
(define-fun alu_result
    ((short_A (_ BitVec 12)) (short_B (_ BitVec 12)) (f3 (_ BitVec 3))) (_ BitVec 32)
    (ite (= f3 #b110)
         (bvor ((_ sign_extend 20) short_A) ((_ sign_extend 20) short_B))
         (ite (= f3 #b100)
              (bvxor ((_ sign_extend 20) short_A) ((_ sign_extend 20) short_B))
              (ite (= f3 #b111)
                   (bvand ((_ sign_extend 20) short_A) ((_ sign_extend 20) short_B))
                   (bvadd ((_ sign_extend 20) short_A) ((_ sign_extend 20) short_B))))))
```

Python pseudocode for the scenario ① SyGuS function:

```
def alu(short_A : bv12, short_B : bv12, f3 : bv3):
    a = short_A.sign_extend(20) # Sign extend to 32 bits
    b = short_B.sign_extend(20) # Sign extend to 32 bits
    if f3 == 0b000:
        return a + b
    elif f3 == 0b110:
        return a | b
    elif f3 == 0b111:
        return a & b
    else:
        return a ^ b
```

Scenario ②. In scenario ②, where the grammar does not include every single used bit-vector operation, the resulting expression for the ALU function is somewhat complicated; further rounds of synthesis with a different grammar could be applied to simplify it further.

The SyGuS function from scenario ② is as follows (Python pseudocode is again given afterwards for readability):

```
(define-fun alu
    ((short_A (_ BitVec 12)) (short_B (_ BitVec 12)) (f3 (_ BitVec 3))) (_ BitVec 32)
    (ite (= f3 #b000)
         (bvadd ((_ sign_extend 20) short_A) ((_ sign_extend 20) short_B))
         (ite (= f3 #b110)
              (bvor ((_ sign_extend 20) short_A) ((_ sign_extend 20) short_B))
              (ite (= f3 #b111)
                   (bvadd ((_ sign_extend 20) short_A)
                          (bvsub ((_ sign_extend 20) short_B)
                                 (bvor ((_ sign_extend 20) short_A) ((_ sign_extend 20) short_B))))
                   (bvsub (bvor ((_ sign_extend 20) short_A) ((_ sign_extend 20) short_B))
                          (bvadd ((_ sign_extend 20) short_A)
                                 (bvsub ((_ sign_extend 20) short_B)
                                        (bvor ((_ sign_extend 20) short_A)
                                              ((_ sign_extend 20) short_B)))))))))
```

Python pseudocode for the scenario ② SyGuS function:

```
def alu(short_A : bv12, short_B : bv12, f3 : bv3):
    a = short_A.sign_extend(20) # Sign extend to 32 bits
    b = short_B.sign_extend(20) # Sign extend to 32 bits
    if f3 == 0b000:
        return a + b
    elif f3 == 0b110:
        return a | b
    elif f3 == 0b111:
        return a + (b - (a | b)) # Equivalent to AND
    else:
        return (a | b) - (a + (b - (a | b))) # Equivalent to XOR
```

Scenario ③. In scenario ③, a BMC counterexample that contradicts the behavior of the I/O oracle is found after 2 iterations. Since this results in contradictory constraints on the solver, no synthesis candidate can be produced, and failure is recognized very quickly. We found the process of refinement specification to be the most difficult task involved in setting up the framework, and this quick failure mode allows users to easily detect and iterate on mistakes.

Performance

As before, runtime breakdowns for scenarios ①, ②, and ③ are given in Tables 3.3, 3.4, and 3.5, respectively. In all scenarios, the correctness oracle takes up an even heftier portion of the overall runtime compared to the 8051 design, likely due to the increased complexity of generated synthesis candidates and the design as a whole; however, it still remains tolerable for the user experience.

Operation	**Count**	**Total (s)**	**Average (s)**
Dataflow parsing from Verilog	2	2.839	1.420
Partial model construction from dataflow	2	1.455	0.727
I/O oracle invocation	4	0.022	0.006
Correctness oracle invocation	4	89.886	22.471
Solver synthesis invocation	4	0.017	0.004

Table 3.3: Runtimes for various synthesis algorithm components for scenario ① on riscv-mini (all components are correctly specified).

Operation	**Count**	**Total (s)**	**Average (s)**
Dataflow graph parsing from Verilog	2	2.877	1.438
Partial model construction from dataflow	2	1.533	0.766
I/O oracle invocation	5	0.014	0.006
Correctness oracle invocation	5	140.617	28.123
Solver synthesis call	5	1.221	0.244

Table 3.4: Runtimes for various synthesis algorithm components for scenario ② on riscv-mini (the grammar is slightly underspecified).

Operation	**Count**	**Total (s)**	**Average (s)**
Dataflow graph parsing from Verilog	2	2.851	2.851
Partial model construction from dataflow	2	1.517	1.517
I/O oracle invocation	2	0.012	0.006
Correctness oracle invocation	1	15.873	15.873
Solver synthesis call	2	0.007	0.004

Table 3.5: Runtimes for various synthesis algorithm components for scenario ③ on riscv-mini (the refinement relation contains an error).

Chapter 4

Conclusion

In this thesis, we present RTL2MODEL, a compositional modeling framework for RTL designs, as well as methods for automatically lifting them from RTL with varying granularity. We demonstrate the effectiveness of synthesis techniques in lifting simpler models of complex RTL combinatorial expressions for a pair of example processors; the resulting models are correct by construction with respect to the supplied program sketch and refinement annotations, and substantially easier to understand and verify. The RTL2MODEL Python framework is flexible, and can be augmented further in future work to automate even more significant portions of the hardware lifting process.

4.1 Future Work

In order to decrease the amount of work required of the user in generating a model, the RTL2MODEL framework can further be adapted to synthesize definitions for non-combinatorial functions, and generalize I/O traces to apply constraints to more than just the immediate inputs and outputs of synthesis functions. We discuss a potential avenue to implementing this in further detail in Section 3.3.1.

For processors which have formalized instruction sets (as with the RISC-V and 8051 designs we examine), the full end-to-end lifting process should also include a mechanism for verifying that the lifted model refines the ISA specification. RISC-V has an operational specification written in the SAIL language [26], and we also produced a version of the user-level specification in UCLID5.[5] Since RTL2MODEL is capable of converting native models to UCLID5 models, a next step is to apply this capability to perform refinement checks against one such golden model specification.

RTL2MODEL follows a recent trend towards increased automation in the hardware lifting process [35, 37]. An area of substantial interest is to test the scalability of our algorithms in verifying processors with more complex features and micro-architectural designs, such as the RISC-V Rocket chip [2] and Berkeley Out-of-Order Machine (BOOM) [38] designs. These

[5]`https://github.com/uclid-org/uclid-riscv`

processors are significantly more complicated than the designs we examine, and compositionally verifying individual modules such as the TLB and BTB may be of more practical use than the ISA-based examples we present in this work.

In addition to scaling up the size of processor designs examined, we also wish to further explore modeling of security properties on processors, as one of the primary strengths of RTL2MODEL compared to previous ILA works is the ability to capture micro-architectural state. Verifying common security properties like non-interference is possible under our framework, but requires the user to provide somewhat detailed program sketches, which may not always be available. Other properties of interest, especially timing-based ones, are also currently impossible to express in RTL2MODEL.

A last potential avenue of exploration is the application of RTL2MODEL to verifying processors' memory consistency models. Though the models we construct are essentially operational in nature, it is possible to simultaneously infer microarchitectural happens-before invariants from RTL, as done in [13]. Mixing together elements of both operational and axiomatic modeling has the potential to enable verification of a wider range of micro-architectural properties.

Bibliography

[1] Rajeev Alur, Rastislav Bodik, Garvit Juniwal, Milo M. K. Martin, Mukund Raghothaman, Sanjit A. Seshia, Rishabh Singh, Armando Solar-Lezama, Emina Torlak, and Abhishek Udupa. "Syntax-guided synthesis". In: *2013 Formal Methods in Computer-Aided Design*. 2013, pp. 1–8. DOI: `10.1109/FMCAD.2013.6679385`.

[2] Krste Asanovic, Rimas Avizienis, Jonathan Bachrach, Scott Beamer, David Biancolin, Christopher Celio, Henry Cook, Daniel Dabbelt, John Hauser, Adam Izraelevitz, et al. "The rocket chip generator". In: *EECS Department, University of California, Berkeley, Tech. Rep. UCB/EECS-2016-17* 4 (2016).

[3] Haniel Barbosa et al. "cvc5: A Versatile and Industrial-Strength SMT Solver". In: *Tools and Algorithms for the Construction and Analysis of Systems - 28th International Conference, TACAS 2022, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2022, Munich, Germany, April 2-7, 2022, Proceedings, Part I*. Ed. by Dana Fisman and Grigore Rosu. Vol. 13243. Lecture Notes in Computer Science. Springer, 2022, pp. 415–442. DOI: `10.1007/978-3-030-99524-9\_24`. URL: `https://doi.org/10.1007/978-3-030-99524-9%5C_24`.

[4] Clark Barrett, Pascal Fontaine, and Cesare Tinelli. *The Satisfiability Modulo Theories Library (SMT-LIB)*. `www.SMT-LIB.org`. 2016.

[5] Bryan A Brady, Randal E Bryant, and Sanjit A. Seshia. "Learning conditional abstractions". In: *2011 Formal Methods in Computer-Aided Design (FMCAD)*. IEEE. 2011, pp. 116–124.

[6] Bryan A Brady, Randal E Bryant, Sanjit A. Seshia, and John W O'Leary. "ATLAS: automatic term-level abstraction of RTL designs". In: *Eighth ACM/IEEE International Conference on Formal Methods and Models for Codesign (MEMOCODE 2010)*. IEEE. 2010, pp. 31–40.

[7] Randal E. Bryant, Shuvendu K. Lahiri, and Sanjit A. Seshia. "Modeling and Verifying Systems using a Logic of Counter Arithmetic with Lambda Expressions and Uninterpreted Functions". In: *CAV02*. Ed. by E. Brinksma and K. G. Larsen. LNCS 2404. July 2002, pp. 78–92.

[8] Jerry R. Burch and David L. Dill. “Automatic Verification of Pipelined Microprocessor Control”. In: *Proceedings of the 6th International Conference on Computer Aided Verification*. CAV ’94. Berlin, Heidelberg: Springer-Verlag, 1994, pp. 68–80. ISBN: 3540581790.

[9] Cadence Design Systems, Inc. *Jasper RTL Apps*. `https://www.cadence.com/en_US/home/tools/system-design-and-verification/formal-and-static-verification/jasper-gold-verification-platform.html`. Accessed: 2022-04-26. 2022.

[10] Kevin Cheang, Cameron Rasmussen, Sanjit Seshia, and Pramod Subramanyan. “A formal approach to secure speculation”. In: *2019 IEEE 32nd Computer Security Foundations Symposium (CSF)*. IEEE. 2019, pp. 288–28815.

[11] Pranav Gaddamadugu. “Formally Verifying Trusted Execution Environments with UCLID5”. MA thesis. EECS Department, University of California, Berkeley, Aug. 2021. URL: `http://www2.eecs.berkeley.edu/Pubs/TechRpts/2021/EECS-2021-200.html`.

[12] Yosys HQ. *SymbiYosys (sby) Documentation*. `https://yosyshq.readthedocs.io/projects/sby/en/latest/`. Accessed: 2022-04-26. 2022.

[13] Yao Hsiao, Dominic P. Mulligan, Nikos Nikoleris, Gustavo Petri, and Caroline Trippel. “Synthesizing Formal Models of Hardware from RTL for Efficient Verification of Memory Model Implementations”. In: *MICRO-54: 54th Annual IEEE/ACM International Symposium on Microarchitecture*. New York, NY, USA: Association for Computing Machinery, 2021, pp. 679–694. ISBN: 9781450385572. URL: `https://doi.org/10.1145/3466752.3480087`.

[14] Bo Yuan Huang, Hongce Zhang, Pramod Subramanyan, Yakir Vizel, Aarti Gupta, and Sharad Malik. “Instruction-level abstraction (ILA): A uniform specification for system-on-chip (SOC) verification”. English (US). In: *ACM Transactions on Design Automation of Electronic Systems* 24.1 (Jan. 2019). ISSN: 1084-4309. DOI: `10.1145/3282444`.

[15] Mathieu Jan, Mihail Asavoae, Martin Schoeberl, and Edward A Lee. “Formal semantics of predictable pipelines: a comparative study”. In: *2020 25th Asia and South Pacific Design Automation Conference (ASP-DAC)*. IEEE. 2020, pp. 103–108.

[16] Susmit Jha, Sumit Gulwani, S. Seshia, and A. Tiwari. “Oracle-guided component-based program synthesis”. In: *2010 ACM/IEEE 32nd International Conference on Software Engineering* 1 (2010), pp. 215–224.

[17] Susmit Jha and Sanjit A. Seshia. “A Theory of Formal Synthesis via Inductive Learning”. In: *Acta Informatica* 54.7 (2017), pp. 693–726.

[18] Donggyu Kim. *riscv-mini*. `https://github.com/ucb-bar/riscv-mini`. Accessed: 2022-05-04. 2017.

[19] Paul Kocher, Jann Horn, Anders Fogh, Daniel Genkin, Daniel Gruss, Werner Haas, Mike Hamburg, Moritz Lipp, Stefan Mangard, Thomas Prescher, et al. "Spectre attacks: Exploiting speculative execution". In: *2019 IEEE Symposium on Security and Privacy (SP)*. IEEE. 2019, pp. 1–19.

[20] Xinbing Li. *R8051*. `https://github.com/risclite/R8051`. Accessed: 2022-05-04. 2019.

[21] Moritz Lipp et al. "Meltdown: Reading Kernel Memory from User Space". In: *27th USENIX Security Symposium (USENIX Security 18)*. 2018.

[22] Carmelo Loiacono, Marco Palena, Paolo Pasini, Denis Patti, Stefano Quer, S Ricossa, Danilo Vendraminetto, and J Baumgartner. "Fast cone-of-influence computation and estimation in problems with multiple properties". In: *2013 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. IEEE. 2013, pp. 803–806.

[23] Daniel Lustig, Michael Pellauer, and Margaret Martonosi. "PipeCheck: Specifying and verifying microarchitectural enforcement of memory consistency models". In: *2014 47th Annual IEEE/ACM International Symposium on Microarchitecture*. IEEE. 2014, pp. 635–646.

[24] Albert Magyar and Pranav Gaddamadugu. *ChiselUCL*. `https://github.com/uclid-org/chiselucl`. Accessed: 2022-04-26. 2021.

[25] Yatin A. Manerkar, Daniel Lustig, Michael Pellauer, and Margaret Martonosi. "CCI-Check: Using μhb graphs to verify the coherence-consistency interface". In: *2015 48th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. 2015, pp. 26–37. DOI: `10.1145/2830772.2830782`.

[26] Prashanth Mundkur et al. *RISCV Sail Model*. `https://github.com/riscv/sail-riscv`. Accessed: 2022-05-04. 2018.

[27] Elizabeth Polgreen, Andrew Reynolds, and Sanjit A. Seshia. "Satisfiability and Synthesis Modulo Oracles". In: *International Conference on Verification, Model Checking, and Abstract Interpretation*. Springer. 2022, pp. 263–284.

[28] Sanjit A. Seshia. "Combining Induction, Deduction, and Structure for Verification and Synthesis". In: *Proceedings of the IEEE* 103.11 (2015), pp. 2036–2051.

[29] Sanjit A. Seshia and Pramod Subramanyan. "UCLID5: Integrating modeling, verification, synthesis and learning". In: *2018 16th ACM/IEEE International Conference on Formal Methods and Models for System Design (MEMOCODE)*. IEEE. 2018, pp. 1–10.

[30] Wilson Snyder. *Welcome to Verilator*. `https://veripool.org/verilator/`. Accessed: 2022-05-07. 2003.

[31] Armando Solar-Lezama. "The sketching approach to program synthesis". In: *Asian Symposium on Programming Languages and Systems*. Springer. 2009, pp. 4–13.

[32] Pramod Subramanyan and Sanjit A. Seshia. *Getting Started with Uclid5*. `https://raw.githubusercontent.com/uclid-org/uclid/master/tutorial/tutorial.pdf`. Accessed: 2022-04-26. 2021.

[33] Pramod Subramanyan, Yakir Vizel, Sayak Ray, and Sharad Malik. “Template-based synthesis of instruction-level abstractions for SoC verification”. In: *2015 Formal Methods in Computer-Aided Design (FMCAD)*. 2015, pp. 160–167. DOI: `10.1109/FMCAD.2015.7542266`.

[34] Shinya Takamaeda-Yamazaki. “Pyverilog: A Python-Based Hardware Design Processing Toolkit for Verilog HDL”. In: *Applied Reconfigurable Computing*. Vol. 9040. Lecture Notes in Computer Science. Springer International Publishing, Apr. 2015, pp. 451–460. DOI: `10.1007/978-3-319-16214-0_42`. URL: `http://dx.doi.org/10.1007/978-3-319-16214-0_42`.

[35] Yu Zeng, Bo-Yuan Huang, Hongce Zhang, Aarti Gupta, and Sharad Malik. “Generating Architecture-Level Abstractions from RTL Designs for Processors and Accelerators Part I: Determining Architectural State Variables”. In: *2021 IEEE/ACM International Conference On Computer Aided Design (ICCAD)*. 2021, pp. 1–9. DOI: `10.1109/ICCAD51958.2021.9643584`.

[36] Hongce Zhang, Caroline Trippel, Yatin A Manerkar, Aarti Gupta, Margaret Martonosi, and Sharad Malik. “ILA-MCM: integrating memory consistency models with instruction-level abstractions for heterogeneous system-on-chip verification”. In: *2018 Formal Methods in Computer Aided Design (FMCAD)*. IEEE. 2018, pp. 1–10.

[37] Hongce Zhang, Weikun Yang, Grigory Fedyukovich, Aarti Gupta, and Sharad Malik. “Synthesizing environment invariants for modular hardware verification”. In: *International Conference on Verification, Model Checking, and Abstract Interpretation*. Springer. 2020, pp. 202–225.

[38] Jerry Zhao, Ben Korpan, Abraham Gonzalez, and Krste Asanovic. “Sonicboom: The 3rd generation berkeley out-of-order machine”. In: *Fourth Workshop on Computer Architecture Research with RISC-V*. Vol. 5. 2020.