

Critic: Designing a Sound Static Analyzer for Chisel

Ethan A. Kuefner, Timothy Sherwood, Ben Hardekopf
University of California, Santa Barbara
{eakuefner, sherwood, benh}@cs.ucsb.edu

John Sarracino
Harvey Mudd College
jsarracino@g.hmc.edu

I. INTRODUCTION

For hardware designers, testing is commonplace as a way to ensure that hardware designs meet their specifications. However, in high assurance settings such as flight systems, medicine, and defense, strong guarantees on the behavior of a system may be required. To achieve such guarantees using testing can be tedious for sufficiently large designs, since achieving suitable test coverage can require a potentially exponential number of test inputs. Formal static analysis provides an alternative to testing in which mathematical models are reasoned about in order to provide provable guarantees about the operation of a system. Additionally, we can design analyses to be *sound*, which means that they will only return answers which are guaranteed to be true. Thus, a static analysis-based approach to ensuring conformance of designs can remove overhead associated with exhaustive testing as well as potentially provide stronger guarantees than non-exhaustive testing.

While static analysis seems attractive, it is also difficult; in fact, Rice’s theorem states that any non-trivial property of programs is undecidable. Many approaches exist to deal with this problem, but in particular, we appeal to the framework of abstract interpretation [?], which was originally designed as a technique to perform sound analyses of programs. In abstract interpretation, a formal concrete semantics for a programming language is given a mathematical correspondence to an abstract semantics that soundly approximates the behavior of programs. Abstract interpretation is well-known in the programming languages community and has even been used commercially in tools like ASTRÉE, which has successfully verified C code for a variety of avionics software.

Abstract interpretation can be applied to a hardware design setting through the observation that hardware description languages are essentially a particular class of domain-specific languages (DSLs), that is, programming languages designed to target a specific application. Chisel [?] is a new environment for designing and testing hardware that has been designed as an embedded DSL on top of the Scala programming language. For hardware designers, it provides advantages over existing HDLs by bringing advanced features of Scala, such as higher-order functions and object oriented programming, to the hardware design landscape. As well, it presents a good opportunity for analysis designers, since Chisel has been explicitly designed from the ground up for synthesis, unlike either VHDL or Verilog. This reins in the size of the language and allows us to more easily reason about the language and its compilation process as a whole.

This work is the latest in a series of collaborations between the Programming Languages Lab and the ArchLab towards applying techniques in programming languages to provably

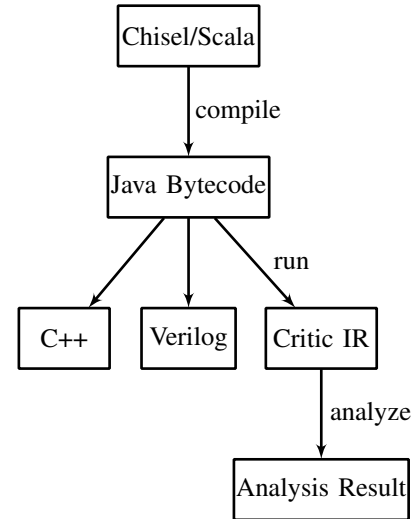


Fig. 1. Chisel/Critic workflow.

correct hardware design. In this work, we describe the architecture of Critic, an abstract interpreter we have been designing for Chisel, and our plans for applying it to proving particular invariants of hardware designs.

II. DESIGN

In this section we outline first the process of extracting a hardware description amenable to analysis from Chisel, and second, the method we use to perform analysis on it.

A. Chisel

Chisel itself is merely a set of data structures which hardware designers instantiate in a particular way to build components. These data structures together represent a directed acyclic graph (DAG) of nodes such as binary operations, multiplexers, and memory reads and writes. Chisel allows extraction of C++ or Verilog from this DAG, for use in simulation or synthesis. When a designer “programs” in Chisel, she is actually programming in Scala. Figure [?] gives an overview of the Chisel/Critic workflow.

To extract a design from a Chisel program, a designer must first compile the program using the Scala compiler, producing Java bytecode. At runtime, the designer selects a “backend”, either C++ or Verilog, and runs the Java bytecode, passing a flag for their selected backend. Chisel then elaborates the design, leaving a flat Chisel data structure that can be traversed for compilation. The specific backend then traverses

$$n \in \mathbb{N} \quad b \in \mathbb{B} = \{0, 1\} \quad \ell^e \in \text{EdgeLabel} \\ \ell^r \in \text{RegLabel} \quad \ell^m \in \text{MemLabel}$$

$$p \in \text{Prog} ::= \vec{N} \vec{e} \overrightarrow{\text{mem}} \\ N \in \text{Node} ::= \ell_1^e \oplus \ell_2^e \Rightarrow \ell_3^e \mid \odot \ell_1^e \Rightarrow \ell_2^e \mid \ell_1^e \text{ mux } \ell_2^e \ell_3^e \Rightarrow \ell_4^e \\ \mid \text{memR } \ell^m \ell_1^e \Rightarrow \ell_2^e \mid \text{memW } \ell^m \ell_1^e \ell_2^e \ell_3^e \Rightarrow \ell_4^e \\ \mid \text{reg } \ell^r \ell_1^e \Rightarrow \ell_2^e \mid \text{in } \ell^e \mid \text{out } \ell^e \mid \text{lit } \vec{b} \Rightarrow \ell^e \\ \oplus \in \text{BinOp} ::= \text{and} \mid \text{or} \mid \vee \mid \wedge \mid \vee \mid \equiv \mid \neq \mid \gg \mid \ll \\ \mid ++ \mid + \mid - \mid \times \mid < \mid \leq \\ \odot \in \text{UnOp} ::= - \mid \sim \mid \text{and}_{\downarrow} \mid \text{or}_{\downarrow} \mid \vee_{\downarrow} \mid \neg \mid n_1..n_2$$

Fig. 2. Critic IR Abstract Syntax. [Need to add reg, mem, edge definitions.]

the data structure and outputs corresponding C++ or Verilog. Fortunately, the backend architecture is extensible, so that users can write their own backends. We take advantage of this by defining our own backend, which converts the Chisel DAG into the Critic intermediate representation (IR), which is essentially a cleaner version of the Chisel DAG which we have designed to be suitable for program analysis. The process of translating Chisel to Critic is more difficult than it may seem, since Chisel is still a research project under active development. As well, although both projects are implemented in Scala, the developers of Chisel have used an object-oriented, imperative style, whereas we prefer a more functional approach. We have had to rectify these issues by reading prose descriptions of Chisel and frequently by deferring to the actual source code for Chisel.

The abstract syntax of our IR is shown in Figure 2. The IR mostly mirrors the Chisel IR, providing all of the same major nodes, with a number of optimizations. It describes a program in the Critic IR as a product of sequences of nodes, edges, memory definitions, and register definitions. Memories have positive edge-triggered writes and combinational reads, registers delay their inputs by a clock cycle, and the \downarrow -subscripted unary operators denote componentwise operation over bit vectors. For example, the semantics of the unary operator and_{\downarrow} are to perform a boolean AND over all the bits in the input vector.

B. Abstract interpretation

To perform abstract interpretation, we begin by formalizing the concrete semantics of the language. We use operational semantics, which specify the behavior of the language as a state transition system, potentially infinite, which operates on a collection of concrete domains representing values and components of the state. In the particular case of Critic, our state consists of a static representation of the DAG, together with a description of the current values of memories and registers, as well as the values held by the wires in the circuit and a program counter. The DAG is organized in a topological order based on data dependencies, and the program counter represents the interpreter's current node of interest in the topological order. We then formalize the state transitions in a function $\mathcal{F}: \text{State} \rightarrow \text{State}$ which matches deterministically on the components of the state to decide what to do. For

example, if the node at the current program counter is a mux, the semantics says to read in its inputs and select one or the other based on the value of the select bit.

After we have fixed the concrete semantics, we derive an abstract semantics which finitizes the state transition system by defining a series of so-called abstract domains, together with a pair of functions called abstraction and concretization that connect the concrete semantics to the abstract semantics. The main difference between the abstract semantics and the concrete semantics is that the state transition system specified by the abstract semantics is nondeterministic; that is, it can be viewed as a function $\mathcal{F}^\sharp: \text{State}^\sharp \rightarrow \mathcal{P}(\text{State}^\sharp)$. The precision of the analysis is based on the design of the abstract domains, where a key decision that has to be made regarding how much information about these sets of states, which could grow exponentially in the worst case, to discard.

The advantage of specifying the semantics in the form described above, which we refer to as the *small-step abstract machine* style, is principally in the derivation of the abstract semantics. David Van Horn and Matt Might show [?] that it is straightforward to derive a computable abstract semantics from an arbitrary small-step abstract machine semantics. We apply their techniques in constructing the abstract semantics to be applied in this work.

III. FUTURE DIRECTIONS

We are currently hard at work on our Scala implementation of the concrete and abstract toolchains for Critic, which we hope to apply to analyzing notions such as a new kind of deadlock freedom for network-on-chip designs. As well, it may be interesting to use our analysis to target programs where previous attempts at applying formal methods to analyze properties such as secure information flow fell short.

IV. CONCLUSION

In this extended abstract we have detailed Critic, a design for an abstract interpreter based on Chisel, a new hardware construction language from UC Berkeley. We described our process for working with Chisel, as well as aspects of the design of our analyzer.

REFERENCES

- [1] [Bibliography goes here.]