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. 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 and allows us to more easily reason about the language and its compilation process as a whole.

In this work, we describe the architecture of Critic, an abstract interpreter we have been designing for Chisel. As well, we discuss some challenges that we have faced in implementing Critic and how we have addressed these.

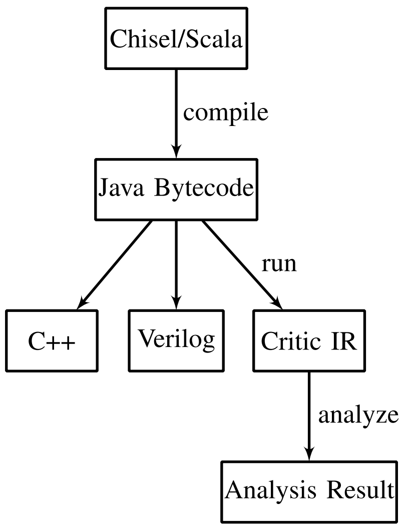

Fig. 1. Chisel/Critic workflow.

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 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

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

$$\begin{array}{rcl}
p \in \mathit{Prog} & ::= & \vec{N}\ \vec{e}\ \overrightarrow{\mathit{mem}} \\
N \in \mathit{Node} & ::= & \ell^e_1 \oplus \ell^e_2 \Rightarrow \ell^e_3 \ \mid\ \odot\, \ell^e_1 \Rightarrow \ell^e_2 \ \mid\ \ell^e_1\ \textsf{\textbf{mux}}\ \ell^e_2\ \ell^e_3 \Rightarrow \ell^e_4 \\
& & \mid\ \textsf{\textbf{memR}}\ \ell^m\ \ell^e_1 \Rightarrow \ell^e_2 \ \mid\ \textsf{\textbf{memW}}\ \ell^m\ \ell^e_1\ \ell^e_2\ \ell^e_3 \Rightarrow \ell^e_4 \\
& & \mid\ \textsf{\textbf{reg}}\ \ell^r\ \ell^e_1 \Rightarrow \ell^e_2 \ \mid\ \textsf{\textbf{in}}\ \ell^e \ \mid\ \textsf{\textbf{out}}\ \ell^e \ \mid\ \textsf{\textbf{lit}}\ \vec{b} \Rightarrow \ell^e \\
\oplus \in \mathit{BinOp} & ::= & \textsf{\textbf{and}} \mid \textsf{\textbf{or}} \mid \veebar \mid \wedge \mid \vee \mid \equiv \mid \not\equiv \mid \gg \mid \ll \\
& & \mid\ \mathbin{+\!\!+} \mid + \mid - \mid \times \mid < \mid \leq \\
\odot \in \mathit{UnOp} & ::= & - \mid \sim \mid \textsf{\textbf{and}}_{\downarrow} \mid \textsf{\textbf{or}}_{\downarrow} \mid \veebar_{\downarrow} \mid \neg \mid n_1..n_2
\end{array}$$

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

a cleaner version of the Chisel DAG which we have designed to be suitable for program analysis.

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 numebr 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 $\textsf{\textbf{and}}_{\downarrow}$ are to perform a boolean AND over all the bits in the input vector.

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 things like numbers, strings, or addresses. 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. Abstract interpretation requires that these functions define an *overapproximation*, which is formalized as follows.

Definition 1. *Let $\mathcal{L}, \mathcal{L}^\sharp$ be partially ordered sets, and let $\alpha\colon \mathcal{L} \to \mathcal{L}^\sharp, \gamma\colon \mathcal{L}^\sharp \to \mathcal{L}$. The pair α, γ is an* overapproximation *if for all $x \in \mathcal{L}$, we have $x \in \gamma(\alpha(x))$ and similarly for*

References

[1] H. Kopka and P. W. Daly, *A Guide to LaTeX*, 3rd ed. Harlow, England: Addison-Wesley, 1999.