Strong Formal Verification Across a Hardware-Software Stack with RISC-V

Adam Chlipala
SIGARCH Visioning Workshop 2019

Joint work with: Thomas Bourgeat, Joonwon Choi, Ian Clester, Andres Erbsen, Jason Gross, Sam Gruetter, Mirai Ikebuchi, Jade Philipoom, Robert Sloan, Andy Wright
The Promise of Formal Verification

- Run lots of tests
- Prove invariants for all executions
- Prove adherence to ISA spec

End-to-End Verification

Full Formal Verification:

High-Level SW
Compilers
Low-Level SW
Compilers
RTL
EDA tools
Silicon
Mechanized Proofs

Real System (source code)

Specification (source code)

Proof (source code)

(e.g., Coq proof assistant, which we use)

Proof Checker (algorithm)
Proofs of Functional Correctness

Real System (source code)

Specification

no segfaults

outputs right answer

✓
End-to-End Proofs

Layer 1

Layer 2

Layer 3

Whole-System Specification

Layers Proved Modularly

Certificate

Certificate

Certificate
Introducing the Verified IoT Lightbulb!

SiFive FE310 board (to be replaced by FPGA running our verified processor)

Ethernet connection

Power strip with GPIO interface (lightbulb gets plugged in)
Consider all **traces** the system could generate: 00100, 11000, 00100, ...  

Recording pin values each cycle  

**Output pins:** we as spec-writers may mandate what they are allowed to be!  

**Input pins:** the environment may choose any values each cycle  

“Output pin controlling lightbulb is only on if the last valid Ethernet packet said so.”
Key Layers of End-to-End Proof

- Controller Spec (Trace Predicate)
- Controller SW
- Programming Language Semantics
- Verified Compiler
- ISA Family Semantics
- Verified Hardware
- RTL Semantics
What is a Spec?

• It is a “tasteful” implementation of an interface.
• It is written to convey behavior and not complicated by optimizations.
  – Anti-example: the canonical RISC-V simulator Spike does not count because it is written in C, a language that is quite complicated to reason about.
• It is flexible to cover dimensions of design discretion.
  – Anti-example: Spike hardcodes most of these choices.
execute (Lw rd rs1 oimm12) = do
  a <- getRegister rs1
  addr <- translate Load 4 (a + fromImm oimm12)
  x <- loadWord addr
  setRegister rd (int32ToReg x)

execute Sret = do
  priv <- getPrivMode
  when (priv < Supervisor) (raiseException 0 2)
  tsr <- getCSRField Field.TSR
  when (tsr == 1) (raiseException 0 2)
  spp <- getCSRField Field.SPP
  setCSRField Field.SPP (encodePrivMode User)
  setPrivMode (decodePrivMode spp)
  spie <- getCSRField Field.SPIE
  setCSRField Field.SPIE 1
  setCSRField Field.SIE spie
  sepc <- getCSRField Field.SEPC
  setPC ((fromIntegral:: MachineInt -> t) sepc)

Underlined operations are parameters of the spec: instruction definitions don’t commit to what they mean or what state they touch.
Many Interpretations of Parameter Functions

1. **Simple single-core**
   - Registers
   - Core
   - Memory
   - getR → setR
   - setM → getM

2. **Multicore, weak mem.**
   - Registers
   - Core
   - Memory
   - getR → setR → getR
   - setM → getM

3. **MMIO**
   - Registers
   - Core
   - Memory
   - getR → setR
   - setM → getM
   - or log of IO ops.
Program modules are objects with mutable private state, accessed via methods.
Every method call appears to execute **atomically**. Any step is summarized by a *trace* of calls. Object *refinement* is inclusion of possible traces.
Bluespec-Style Modularity

Composing objects hides internal method calls.
End-to-End Correctness for SoCs?

OoO processor → Cache system → Reference proc. Naive memory

≈

Optimized SW ≈ Reference SW
Decompose Modularly!

OoO processor \(\approx\) Reference proc. \(\approx\) Cache system \(\approx\) Naive memory

Optimized SW \(\approx\) Reference SW
Example SW Routine: recvEthernet

Write formal **precondition**: what we assume about machine state at start.

(* Read RX_FIFO_INF *)
io! info = lan9250_readword(constr:(Ox"7C"));
rxunused = ((info >> constr:(16))
   & ((constr:(1) << constr:(8)) - constr:(1)));
require (rxunused - constr:(0)) else { r = (constr:(-1)) }

(* Read Status FIFO Port *)
io! rx_status = lan9250_readword(constr:(Ox"40"));

(* Pad num_bytes to next word *)
num_bytes = (rx_status >> constr:(16)
   & ((constr:(1) << constr:(14)) - constr:(1)));
um_words = ((num_bytes + constr:(4) - constr:(1)) >> constr:(2));
num_bytes = (num_words * constr:(4));

Write formal **postcondition**: what we guarantee about machine state at end

(* num_bytes <= MAXEthernet *)
require (num_bytes < constr:(1520 + 1))
else { r = (constr:(-1)) }

c = (constr:(0));
value = (constr:(0));
while (c < num_bytes) {
   io! value = lan9250_readword(constr:(0));
   store4(rx_packet + c, value);
   c = (c + constr:(4))
};
r = (num_bytes)

Write **loop invariant**
Verified Compilers

- SW program
- SW semantics
- set of possible IO traces
- Machine code
- ISA semantics
- set of possible IO traces
- compiler proof

Diagram showing the process of verifying compilers with SW and ISA semantics, leading to a set of possible IO traces.
Disappearing Specs

Controller Spec (Trace Predicate)

Must get this spec right.
Everything this box hides is no longer trusted!

System as a Proved Black Box

Must get this one right, too.

RTL Semantics
Potential to Reduce Verification Effort

Verification team, stressing out about coverage

Time-consuming simulation

Instead...

Proof checker

Proof

Verification team, stressing out about spec (only for “top”/”bottom” layers)
Spec & Proof Marketplaces

Processors -> System -> Compilers

System -> SW Libraries

System -> Accelerators

Proof

Proof

Proof

Proof

Proof

Proof

Proof