Scalable Formal Verification in High-Level Hardware Languages

Amelia Dobis [1] Fabian Schuiki [2] Bea Healy [3] Hideto Ueno [2] Martin Erhart [2] Lenny Truong [2] Mae Milano [1]

[1]Princeton University [2]SiFive [3]University of Cambridge

Background: Hardware Development

Idea: Write a description of a *Digital Circuit* in a high-level format (similar to a program) that is processed by EDA tools in one of 3 ways:

- **Production**: Create a physical layout of electronic components to produce a real chip.
- **Simulation**: Create a *software model* of the circuit to run it on a computer and test it.
- **Verification**: Create a *mathematical model* (logic formula) of the circuit and formally prove that it matches a given behavioral specification.

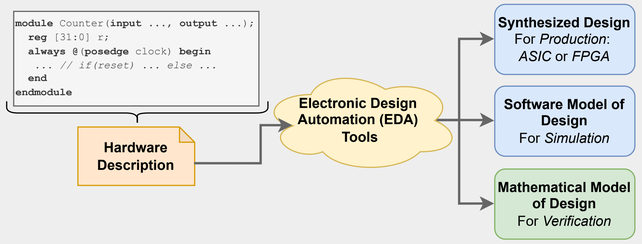

Figure 1. Overview of a typical hardware development flow.

Background: Formal Verification of Hardware

Problem: Producing a chip (tape-out) is an **expensive process** ($50Mio with modern technology). We need a way to **guarantee** that a design is correct before we produce it. Simulation is often not convincing enough.

Idea: Create a mathematical model of a given design, and formally prove that this model is equivalent to a given behavioral specification (assertions about the circuit's expected behavior).

How: Encode the circuit as a transition system that can be reasoned about mathematically.

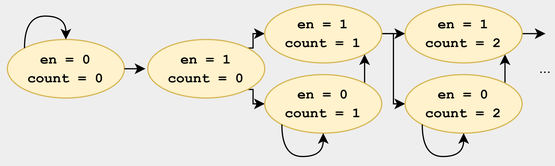

Figure 2. Illustration of the conversion of a hardware design to a state transition system.

This process is:

- **Difficult**: Formal verification tools are **slow** to find a solution.
- **Scales Poorly**: Modularity in a design is ignored, making larger designs much harder to verify.

Research Questions

Goal: Make formal verification *simple* and *scalable*. Can be done by answering the following:

- **Modularity**: How can we preserve modularity in a design during verification?
- **Generalizability**: Can we create a single solution that supports all hardware languages?
- **Expressiveness**: How can we faithfully express our design as a verification problem?

Key Idea: Retain Modularity during Verification

Current Approach: Handle modularity by **inlining** modules in place of instances before converting the design for formal verification. This results in **re-verifying modules** for every instance.

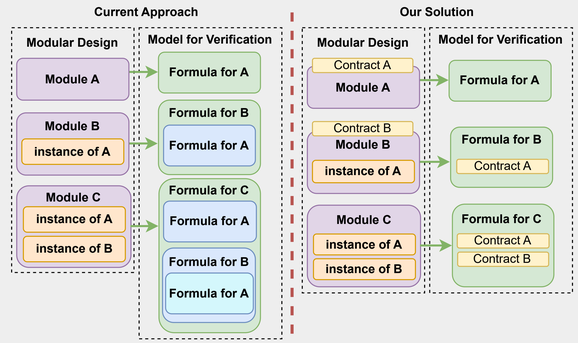

Figure 3. Comparison between the current formal verification approach (left) and our modular solution (right).

Our solution: Hardware Contracts – Verify all modules **exactly once**.

- **Annotate Modules** with *preconditions* (constraints on the module's inputs) and *postconditions* (guarantees on the module's outputs).
- **Abstract Module instances** using contracts to simplify verification while maintaining correctness.

Example

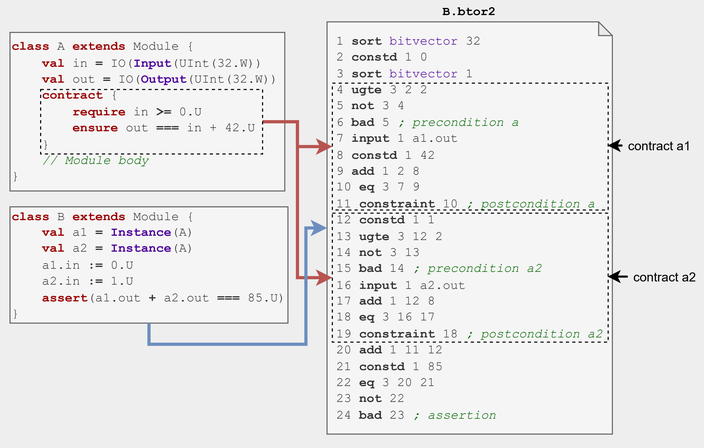

Figure 4. Example of a modular design, implemented in Chisel, converted into a logic formula, expressed in btor2.

Details: A Unified Representation for Modular Formal Verification

Goal: Create a **modular formal verification interface** that all hardware languages can *easily* target to unlock formal verification *for free*.

How: Extend CIRCT, a compiler and intermediate representation (IR) for hardware design, with:

- **Unified Interface** representing a formal verification problem in CIRCT's IR.
- **IR and Compiler Passes** to support hardware contracts.
- **Formal Backend** to convert CIRCT's IR into a format for formal verification.

This is implemented as the `verif` IR inside of CIRCT. With this **all languages** that use CIRCT have access to a scalable formal verification stack *for free*.

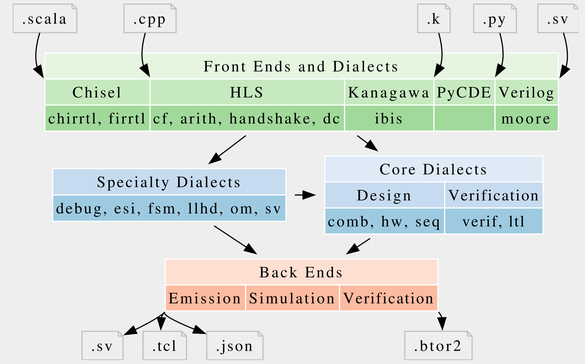

Figure 5. Overview of the CIRCT compiler. The compiler uses specialty dialects to support several front-ends regardless of their paradigms. The core dialects represent a generalized representation of hardware design and verification. These can then be lowered to target several targets.

Several problems had to be solved in order to enable this solution for hardware, including handling multi-clock designs, encoding various types of states, and expressing specifications about hardware. Please talk to me if you want to know more about how we solved these problems!

Initial Results

Initial results of verifying the small design from Figure 4 using **btormc**.

without contracts	with contracts	speedup
0.011s	0.007s	1.57x

Table 1. Wall-time average over 100 runs (in seconds) of verifying Figure 4 with the resulting speedup obtained from using hardware contracts.

These results are due to contracts **enabling solver parallelism** and **simplifying verification**. We believe that the speedup will scale with the size of the design.

References

[Dij75] Edsger W. Dijkstra. Guarded commands, nondeterminacy and formal derivation of programs. *Commun. ACM*, 18(8):453–457, August 1975.

[NPWB18] Aina Niemetz, Mathias Preiner, Clifford Wolf, and Armin Biere. Btor2 , btormc and boolector 3.0. In *Computer Aided Verification*. Springer, 2018.