They're the same picture: a software-verification flow adapted for hardware verification

Andreas Lööw
Imperial College London
London, UK

Magnus O. Myreen
Chalmers University of Technology
Gothenburg, Sweden

ABSTRACT

Software verification and hardware verification are traditionally considered separate enterprises, but our experience suggests that activities in software verification have natural counter parts in hardware verification and vice versa. We believe that there ought to be more technology transfer between the two. In this text, we describe one such instance of technology transfer from the world of interactive theorem proving: we outline how we have adapted the functional-programming-based software-verification flow of the CakeML project to a hardware-verification flow for Verilog. The work described here has been carried out in the HOL4 interactive theorem prover.

CCS CONCEPTS

• **Hardware** → **Hardware description languages and compilation**; **Theorem proving and SAT solving**; • **Software and its engineering** → **Software verification**; **Compilers**.

KEYWORDS

software verification, hardware verification, compiler verification

1 INTRODUCTION

The combination of software development inside of, and with the help of, interactive theorem provers (ITPs) and verified compilers, developed inside the same ITPs, forms the basis of a development-and-verification flow that addresses the following fundamental tension in software development: (a) we want to reason about software at the same abstraction level as we are used to writing software, that is, at the source level of a programming language such as Rust, C, or SML, and, simultaneously, (b) have the correctness guarantees we establish carry over to the compiler-generated machine code that actually runs.

The flow based on ITPs and verified compilers addresses this tension by decoupling the problem into three separate subproblems: (1) artefact development and verification at the source level, (2) transfer of the source-level artefact to the compiler target level, and (3) transfer of source-level verification results to the compiler target level.

From the perspective of a developer following the development flow: the complexity of subproblem (1) depends on the complexity of the software artefact under interest; subproblem (2) is automated away by the verified compiler; and subproblem (3) boils down to a trivial composition of the artefact-correctness theorem proved in subproblem (1) and the correctness theorem of the verified compiler.

The same fundamental tension is found in hardware development: if we in the above replace "Rust, C, or SML" with, e.g., "Verilog, VHDL, or Bluespec" and "machine code" with "netlists", the above now applies to hardware development. We believe a viable solution to this hardware tension is the same as the solution to the software tension: ITPs and verified compilers; or, in the vocabulary of hardware development: ITPs and verified synthesis tools.

In this short text, we describe how we have exploited the similarities between software and hardware development to adapt an existing realisation of the software-verification flow described above, designed for CakeML, to a hardware-verification flow, for Verilog.

2 THE SOFTWARE-VERIFICATION FLOW

The CakeML project [3, 9] is a realisation of the above-described software development-and-verification flow inside the HOL4 interactive theorem prover, with which some nontrivial software artefacts have been produced, e.g., a verified clone of the HOL Light interactive theorem prover [2].

The CakeML project is based on an SML dialect called CakeML. Two pieces of the project are relevant for our discussion here: its proof-producing translator-based verification infrastructure and its verified CakeML compiler.[1] The CakeML language comes in two variants: one shallowly embedded variant, enabling embedding CakeML programs as standard monadic functional programs, which is convenient for verification, and a deeply embedded variant, which can be used as input for the CakeML compiler. The job of the CakeML translator is to automatically translate shallowly embedded CakeML programs to their equivalent deeply embedded variant and produce a correspondence theorem for each translation.

In short, the subproblems (1)–(3) of the development flow are addressed as follows in the CakeML project:

- (1a) Implement your program in the shallow embedding variant of the CakeML language;
- (1b) Prove that your shallowly embedded program is correct, i.e., prove the source-level correctness of your program;
- (1c) Translate your shallow embedded program into the deeply embedded CakeML language using the CakeML translator;
- (2) Run the CakeML compiler inside the logic to produce machine code implementing the given CakeML code and a theorem certifying that the machine code is the result of evaluating the CakeML compiler function;
- (3) Simply compose the theorems from the above steps and the CakeML compiler correctness theorem so that the correctness property proved at the source level is carried down to the compiler-generated machine code.

3 THE HARDWARE-VERIFICATION FLOW

We now describe how we have adapted the CakeML software development flow to a hardware development flow. In short, for our adaptation [4, 5, 8] we have introduced the following new semantics

[1] There is also a verification flow based on separation logic, which we do not cover here.

and tools, and made the following adaptations to the flow:

the CakeML programming language	→	the Verilog hardware-description language
a mechanised semantics for the CakeML language	→	a new mechanised semantics for Verilog
a shallow embedding of the CakeML language	→	a new shallow embedding of Verilog
machine code and its mechanised semantics	→	netlists and their mechanised semantics
the proof-producing CakeML translator	→	a new proof-producing Verilog translator
the verified CakeML compiler	→	a new verified Verilog synthesis tool called Lutsig

With the above in place, the CakeML flow described in the previous section is now applicable to hardware development: (1a) implement your circuit as a shallowly embedded Verilog module; (1b) verify that the module is correct; (1c) translate the module to a deeply embedded Verilog module using the proof-producing Verilog translator; (2) run Lutsig to synthesise the deeply embedded Verilog module; (3) transport your correctness theorem about the shallowly embedded Verilog module to the netlist level by composing the module-correctness theorem with the correspondence theorem produced by the proof-producing Verilog translator and the correctness theorem of Lutsig.

The CakeML flow and the adapted hardware-development flow are similar but not identical. We highlight some differences next.

Inspired by the CakeML project, we shallowly embed Verilog module as functional programs. Our hardware-development flow handles synchronous hardware and models such hardware as next-state functions, describing the cycle-by-cycle behaviour of circuit. Both CakeML and Verilog contain features not found in the purely functional language of HOL4; to support such features they must be modelled indirectly in the respective shallow embedding (e.g., both CakeML and Verilog feature mutable state). Among the features of Verilog that require extra care to model are so-called blocking and nonblocking assignments and concurrently executing processes (blocking assignments correspond to traditional imperative-language assignments and nonblocking assignments are a Verilog construct allowing race-free communication between processes).

One problem we did not have to address in our CakeML work is language standard interpretation and correspondence. Although a dialect of SML, the CakeML language is an invention by the CakeML project. In contrast, the Verilog language is a language defined by a standard document [1] written in English prose. The standard is in places ambiguous (e.g., see Lööw [6] for one example of a potential problem with the standard), making mechanisation of the standard difficult. In ongoing work, we are exploring visual formalisation as a way to validate our interpretation of the standard.

The problems associated with Verilog do not stop at its standard. Although the most popular hardware-description language today, the Verilog language itself is known for its many idiosyncrasies and peculiarities. Of particular interest for verification and without a clear analogue in software development are so-called simulation-and-synthesis mismatches. This type of mismatch refers to when the behaviour of a Verilog circuit differs before and after synthesis. Such mismatches do not arise from synthesis-tool bugs but from the fact that Verilog has two semantics: Verilog's so-called simulation semantics and synthesis semantics. In our work on Lutsig, we address simulation-and-synthesis mismatches and suggest what roles the two semantics should play in verified-hardware development.

As for the CakeML compiler and Lutsig, we find both similarities and dissimilarities. Among the similarities we find, e.g., that the correctness theorems of the CakeML compiler and Lutsig are largely the same: both are variants of semantics preservation. There are also natural counter parts between various software-compilation steps and hardware-synthesis steps: e.g., in compilation we do instruction selection to translate a program to a particular machine architecture and in synthesis we do technology mapping to translate a hardware design to a particular technology (such as FPGAs). Dissimilarities exist as well, such as the types of optimisations applicable. E.g., the X-value construct in Verilog, which allows hardware designers to signal to their synthesis tool that the tool can freely fill in any value in the X-value's place, opens up for optimisation opportunities in hardware synthesis. Exploiting this opportunity, Lutsig includes a simple optimisation pass for X-values. The pass required us to wrestle with the semantics of X-values, which is a (yet another) tricky part of the Verilog language.

4 SOFTWARE-HARDWARE CO-VERIFICATION

After embedding both software and hardware development inside the same ITP, a next logical step is to consider the development of artefacts consisting of both software and hardware, such as computer systems. In computer systems, software and hardware meet at the so-called instruction set architecture (ISA) boundary. Examples of ISAs include x86, ARM, and RISC-V. In Lööw et al. [7] we report on our experience on developing a small verified processor and connecting it at the ISA boundary to verified software developed using the CakeML software development flow, all inside the HOL4 ITP, resulting in a computer system with a remarkably small trusted computing base.

REFERENCES

[1] 2018. IEEE Standard for SystemVerilog–Unified Hardware Design, Specification, and Verification Language. *IEEE Std 1800-2017* (2018). https://doi.org/10.1109/IEEESTD.2018.8299595

[2] Oskar Abrahamsson, Magnus O. Myreen, Ramana Kumar, and Thomas Sewell. 2022. Candle: A Verified Implementation of HOL Light. In *ITP*. https://doi.org/10.4230/LIPIcs.ITP.2022.3

[3] Ramana Kumar, Magnus O. Myreen, Michael Norrish, and Scott Owens. 2014. CakeML: A Verified Implementation of ML. In *POPL*. https://doi.org/10.1145/2535838.2535841

[4] Andreas Lööw. 2022. Reconciling Verified-Circuit Development and Verilog Development. In *FMCAD*. https://doi.org/10.34727/2022/ISBN.978-3-85448-053-2_15

[5] Andreas Lööw. 2021. Lutsig: A Verified Verilog Compiler for Verified Circuit Development. In *CPP*. https://doi.org/10.1145/3437992.3439916

[6] Andreas Lööw. 2022. A small, but important, concurrency problem in Verilog's semantics?. In *MEMOCODE*. https://doi.org/10.1109/MEMOCODE57689.2022.9954591

[7] Andreas Lööw, Ramana Kumar, Yong Kiam Tan, Magnus O. Myreen, Michael Norrish, Oskar Abrahamsson, and Anthony Fox. 2019. Verified Compilation on a Verified Processor. In *PLDI*. https://doi.org/10.1145/3314221.3314622

[8] Andreas Lööw and Magnus O. Myreen. 2019. A proof-producing translator for Verilog development in HOL. In *FormaliSE*. https://doi.org/10.1109/FormaliSE.2019.00020

[9] Magnus O. Myreen. 2021. The CakeML Project's Quest for Ever Stronger Correctness Theorems. In *ITP*. https://doi.org/10.4230/LIPIcs.ITP.2021.1