Towards mechanized verification of Verilog equivalence checking

Michalis Pardalos
Imperial College London
United Kingdom

Laura Pozzi
Università della Svizzera italiana (USI)
Lugano
Switzerland

John Wickerson
Imperial College London
United Kingdom

Abstract

Equivalence checking is an essential tool in the verification engineer's toolbox, providing a high-assurance method to verify hardware designs. These tools are trusted, which means that equivalence checker bugs can have very high impact. We believe that the way to place this trust on a solid foundation is *formal proof*.

To this end, we are developing Vera, an equivalence checker for Verilog, that is accompanied by a computer checked proof of correctness in the Coq proof assistant. While we are still in the early stages of development, benchmarks show that our tool is usable on real-world designs.

We believe that this approach — proving automated verifiers correct with computer-checked proofs — can also be more generally applicable, allowing hardware designers to benefit from the assurances of computer-checked proofs for no additional effort.

1 Introduction

Among hardware verification techniques, equivalence checking stands out for delivering stronger assurance than simulation-based methods while handling designs that may not suit property-based verification approaches. This power, however, means that equivalence checking tools are highly trusted — a positive result from the equivalence checker is often taken as close to definitive proof of the correctness of the design.

However, this trust is not necessarily well placed. Equivalence checkers, like all software, are not perfect, and bugs in the equivalence checker can manifest themselves as missed bugs in real designs. Recent work [7] has discovered a number of such bugs in even commercial equivalence checkers.

We propose that the way to place this trust on a solid foundation is *computer-checked proof*. Having access to an equivalence checker that has been proven correct in a proof assistant would allow designers to use the guarantees of this proof, while reaping all the productivity benefits of automated verification.

With this goal in mind, we are developing Vera, an equivalence checker for Verilog designs, which is accompanied by a proof of correctness in the Coq [4] proof assistant. Although automated hardware verification tools have existed for decades, they typically lack formal guarantees of their own correctness. Our approach combines the convenience of push-button automation with the strong guarantees of mechanised proof.

LATTE '25, March 30, 2025, Rotterdam, Netherlands

Our vision for this tool is as a drop-in replacement for a tool like symbiyosys/eqy [11, 12] — but accompanied by a formal proof of correctness. It should be possible for users to replace their current choice of equivalence checker with our tool with minimal effort. This includes both combinational and sequential equivalence checking use-cases, while also covering enough of the Verilog language to verify real-world designs.

2 Architecture

We are using Coq as both the implementation and verification for Vera. The equivalence checker consists of a "frontend", which can be thought of as a compiler from Verilog to SMTLIB [3] and a "backend" which uses SMTCoq [2] to get a verified result for the SMTLIB query — either a model in the SAT case or a proof of unsatisfiability if UNSAT. This is illustrated in Figure 1. The frontend consists of an external (trusted) elaborator, and a series of three passes — typechecking, canonicalization and SMT encoding — which gradually lower the elaborated Verilog into SMTLIB.

The elaborator is primarily needed to resolve the implicit bitwidths and signedness of Verilog expressions and simplify some syntax into simpler constructs. It also functions as the parser. We have used Slang [8] for this task. Once we have the elaborated Verilog, in which every intermediate expression is annotated with its type (width and signedness), we apply a series of passes that gradually lower it into SMTLIB queries:

Typechecking First, we check that the types assigned by the elaborator are consistent (e.g. that operands to arithmetic operators have been assigned the same type). This eliminates some possible errors from the unverified elaborator, and the guarantees it establishes can be used by the correctness proofs of later passes.

Canonicalization The next step is to eliminate any procedural logic (i.e. "if"-statements) from the Verilog. The goal is to get a module with a single, static assignment to each wire/register, in either a combinational (`always_comb`) or sequential (`always_ff @(posedge clk)`) block. This simplifies the next pass.

SMT Encoding Having a single expression to determine the value of each variable, we can produce a single assertion statement for each of those assignments in SMTLIB, using the QF_BV (quantifier-free bitvector) logic to encode the expressions.

The combination of these three passes gives us an encoding of the source Verilog in SMTLIB. The encodings of the two modules to be compared can then be combined in a straightforward way to yield a larger query which asks for evidence of a difference between the two modules.

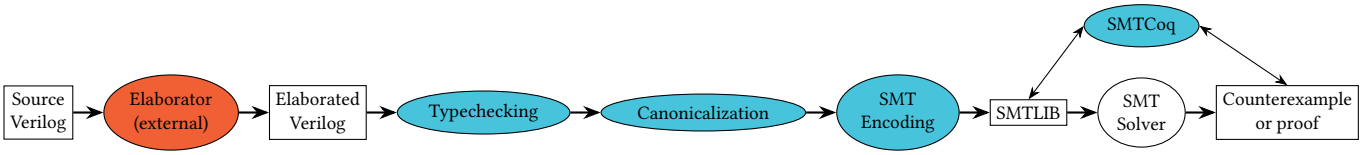

Figure 1: Equivalence checker structure.
Passes drawn as ellipses, intermediate languages drawn as rectangles.

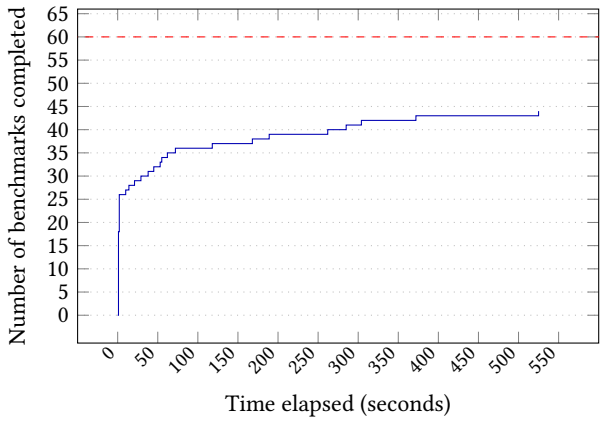

Figure 2: Runtimes of successful benchmarks

3 Formal verification approach

Intuitively, the correctness theorem for the equivalence checker should state that if it gives a positive result, then, for identical inputs, both modules produce identical outputs for all cycles. If it gives a negative result (which needs to come with a counter-example) then that counter-example should indeed indicate a difference. It should be possible to replicate it by running the module on the inputs specified by the counterexample. Since it is also possible to produce an error, we make no guarantees about that case.

We plan to establish this high-level correctness theorem by chaining together the correctness of the intermediate passes. This is trivially true for the typechecking pass, since it only performs a check. For the canonicalization pass, we can re-use the high-level equivalence statement, showing that the design before and after canonicalization are equivalent. Finally, for the SMT encoding pass, this equivalence can be expressed as the generated formula being satisfiable for all possible valuations of the input variables if and only if the output variables are given the values produced by running the Verilog module on those inputs.

The Verilog semantics that we are using for verification is a modified version of those used in the Lutsig Verilog synthesiser [6], with some key changes. Namely, we more closely follow the Verilog standard in regards to Verilog 'X' values.

4 Preliminary Evaluation

We ran a preliminary evaluation of Vera on the EPFL benchmark suite [1]. For each benchmark, we used Vera to compare Verilog version against its BLIF [10] (Berkeley Logic Interchange Format) version (converting the BLIF to verilog using yosys), as well as against its depth- and size-optimised versions. Out of a total of 60 tests, 44 were successful (produced a "positive" result), 10 produced negative results or errors (indicating a bug in the current, unverified, version of Vera), and 6 timed out (after a time-out of one hour). The runtimes of successful benchmarks are summarised in Figure 2.

While we are still investigating the incorrect results, we find the low number of timeouts (and reasonable average checking time) encouraging. Our current design does not perform any optimisations, relying entirely on the SMT solver, which for these benchmarks was Z3 [5]. Yet, it appears to be sufficient to successfully verify these small-to-moderate size benchmarks. Adding optimisation or simplification passes to the equivalence checker, as well as experimenting with different SMT solvers could improve these results further.

5 Research Direction

The immediate next goal of this project is to complete the correctness proof. Parts of this proof will be based on the correctness proof of Lutsig [6]. Particularly, the typechecking pass is very similar to that of Lutsig, and the canonicalization pass shares some similarities with the synthesis process — which in Lutsig is a compilation pass from Verilog to a netlist.

A usual feature of equivalence checking tools which has not been discussed is sequential equivalence checking. This is usually achieved using a method such as k-induction [9]. Implementation of such an algorithm is simple, however proving it correct is an entirely different matter, as it will require changes to the correctness statement of the checker. Nevertheless, we expect it to be an incremental extension, as correctness of combinational checking is a necessary prerequisite for sequential checking.

The design we have described contains a trusted component — the elaborator. This is following with previous work on formally verified Verilog tooling (namely, Lutsig) which also uses a trusted external elaborator. We consider this to be an interesting verification task, which is, however, outside the scope of this work.

6 Conclusion

We have presented our preliminary work on Vera, an automated equivalence checker for Verilog with a mechanized proof of correctness. Our performance evaluation shows promising initial results, which we expect to improve further with additional optimisations and experimentation with alternative SMT solvers.

We believe that this category of hardware design tooling — automated verifiers — are a prime target for mechanized verification. Establishing a formal proof of correctness for these tools allows users to benefit from this high assurance standard *for free*, establishing the correctness of their designs with no additional effort over an "unverified" verifier.

References

[1] Luca Amarù, Pierre-Emmanuel Gaillardon, and Giovanni De Micheli. 2015. The EPFL Combinational Benchmark Suite. *Proceedings of the 24th International Workshop on Logic and Synthesis (IWLS)*. https://infoscience.epfl.ch/handle/20.500.14299/113476

[2] Michael Armand, Germain Faure, Benjamin Grégoire, Chantal Keller, Laurent Théry, and Benjamin Werner. 2011. *A Modular Integration of SAT/SMT Solvers to Coq through Proof Witnesses*. Springer Berlin Heidelberg, 135–150. https://doi.org/10.1007/978-3-642-25379-9_12

[3] Clark Barrett, Pascal Fontaine, and Cesare Tinelli. 2016. The Satisfiability Modulo Theories Library (SMT-LIB). https://www.SMT-LIB.org.

[4] Yves Bertot and Pierre Castéran. 2004. *Interactive Theorem Proving and Program Development*. Springer Berlin Heidelberg. nil pages. https://doi.org/10.1007/978-3-662-07964-5

[5] Leonardo De Moura and Nikolaj Bjørner. 2008. Z3: an efficient SMT solver. In *Proceedings of the Theory and Practice of Software, 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems* (Budapest, Hungary) *(TACAS'08/ETAPS'08)*. Springer-Verlag, Berlin, Heidelberg, 337–340.

[6] Andreas Lööw. 2021. Lutsig: a verified Verilog compiler for verified circuit development. In *Proceedings of the 10th ACM SIGPLAN International Conference on Certified Programs and Proofs (CPP '21)*. ACM. https://doi.org/10.1145/3437992.3439916

[7] Michalis Pardalos, Alastair F. Donaldson, and Emiliano Morini. 2024. Who checks the checkers? Automatically finding bugs in C-to-RTL Formal Equivalence Checkers. (2024). https://doi.org/10.30420/566438006

[8] Michael Popoloski. 2015–2025. slang. https://github.com/MikePopoloski/slang. MIT License.

[9] Mary Sheeran, Satnam Singh, and Gunnar Stålmarck. 2000. Checking Safety Properties Using Induction and a SAT-Solver. In *Formal Methods in Computer-Aided Design*. Springer Berlin Heidelberg, 127–144. https://doi.org/10.1007/3-540-40922-x_8

[10] Berkeley University of California. 1992. *Berkeley Logic Interchange Format (BLIF)*. Technical Report. University of California, Berkeley. https://course.ece.cmu.edu/~ee760/760docs/blif.pdf

[11] Claire Xenia Wolf, N. Engelhardt, National Technology, and LLC Engineering Solutions of Sandia. 2020. Equivalence Checking with Yosys (EQY). https://yosyshq.readthedocs.io/projects/eqy/en/latest/. Front-end driver program for Yosys-based formal hardware verification flows. Licensed under the ISC license..

[12] YosysHQ. 2020. SymbiYosys: A Formal Verification Framework for Yosys. https://yosyshq.readthedocs.io/projects/yosys/en/latest/. A formal verification front-end for Yosys-based hardware designs. Licensed under the ISC license..