Incremental Conversion of SVA Properties to Synthesizable Hardware

Amelia Dobis
Princeton University
USA

Fabian Schuiki
SiFive
USA

Mae Milano
Princeton University
USA

Abstract

There is a mismatch between specification and verification tooling for hardware. The modal logic required to correctly specify sequential designs is difficult to interpret using the first-order logic (FOL) supported by SMT-based verification tools. In this work, we propose to close this gap via the design of an Intermediate Representation (IR) and an incremental lowering that would allow for temporal expressions to be encoded as synthesizable hardware without the need for expensive formulae monitoring automata. By lowering these expressions to synthesizable hardware inside of a generalized compiler such as CIRCT, we create a single point of truth for these expressions, allowing them to be understood and handled by any verification tool that supports synthesizable Verilog. This will allow any of CIRCT's frontends, including SystemVerilog, to utilize temporal expressions in the specification of their designs and verify them without the need for commercial verification tools. This contributes towards creating a fully open-source hardware verification ecosystem.

1 Introduction

While hardware verification tooling has improved [7, 8, 11, 20], writing *specifications* for these tools remains an arduous task. A fundamental disconnect is present between common sequential designs specified in modal logics (like LTL [16]), and the logics understood by underlying SMT-based verification tools (usually FOL) [3, 6, 14]. Supporting LTL specifications requires an expensive and complex lowering to synthesizable hardware through the creation of formulae-monitoring Büchi automata [4, 5, 9, 12, 19]. Existing tools such as Yosys [20] have supported this conversion through a series of hard-coded special cases, focusing mainly on the most common properties expressed using SystemVerilog Assertions (SVA) [1] – a sub-language of SystemVerilog that can express temporal specifications using sequences and properties. As a result, modern high-level hardware languages have been forced to rely on commercial tools for SystemVerilog to be able to support temporal expressions.

We need a way to guarantee support for temporal specifications in all open-source verification tools without requiring this repeated complex lowering to take place in every solver. We call for a generic design for expressing and lowering temporal expressions via a new intermediate representation (IR), enabling expressive specifications and verification directly in the language's compiler. CIRCT [10] is an MLIR [13]-based compiler infrastructure for hardware description and verification that was proposed as a solution to unify the disjointed set of domain-specific hardware compilers into a single generic tool. In this work, we redesign CIRCT's `ltl` dialect [17], and propose a set of lowerings to incrementally convert these expressions into synthesizable hardware. Crucially, choosing CIRCT enables this specification expressivity in all of CIRCT's frontends such as Chisel [2], Magma [18], Kanagawa [15], or SystemVerilog [1]. With this proposed IR, all of CIRCT's frontends will be able to express temporal specifications in a manner that is supported by all standard open-source verification tools.

2 High-Level Overview

To demonstrate the efficacy of our IR at encoding temporal expressions, consider the statement "if `a` followed by `b` after one cycle followed by `c` after another cycle, then `d` will hold followed by `e` after one cycle", written using SVA [1]:

```
((a ##1 b) ##1 c) |-> (d ##1 e)
```

We start by identifying the smallest sub-expressions, i.e. expressions comprised of a single operator and two operands. In the case of our example these are `(a ##1 b)`, `(ab ##1 c)`, `(d ##1 e)`, and `abc |-> de`. We can then iteratively *fold* these separate blocks until we are left with a single large block. This final block can then be implemented as synthesizable hardware, where delays become shift-registers, as it no longer contains any modal logic elements. For our example, this process is illustrated in Figure 1.

3 IR Design

The goal of the IR design is to allow for a centralized representation of an arbitrary set of temporal expressions loosely following the syntax of SVA properties. These typically describe sequences of events that are compared to each other using high-level logical operators. In practice, expressions like the one in our example represent an automata in which each node is a logical comparison performed at a given clock cycle. Our IR explicitly encodes this underlying structure, while following three design principles: *expressivity* (should

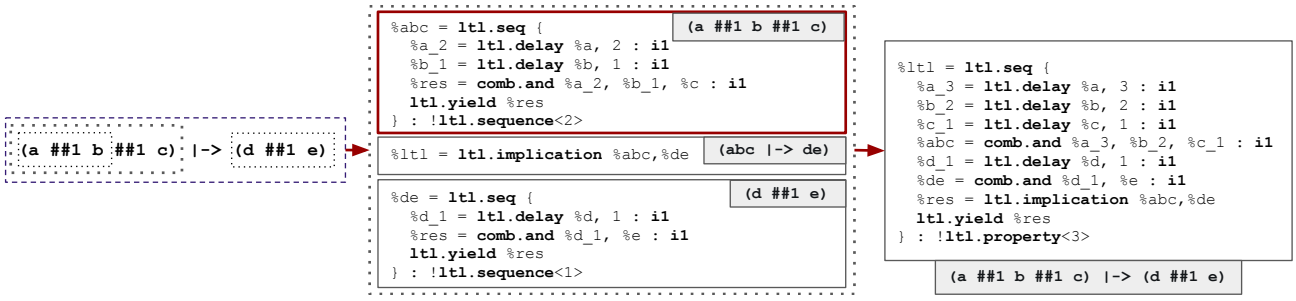

Figure 1. Overview of our proposed incremental lowering from an SVA property to a synthesizable expression.

support a wide variety of temporal expressions), *compositionality* (nested sequences should lower correctly without additional efforts), and *synthesizability* (properties should be lowered to synthesizable hardware in a straightforward manner).

The core unit of our IR is the `ltl.seq` operation which encodes a single state in our LTL automaton with the following signature: `ltl.seq (<arguments>) <body> : <type>` The optional arguments it can take represent this operation's predecessor states and are used to encapsulate them into Single Static Assignment (SSA) values that can be reasoned about as a block. The body contains the logical comparisons that compute the result of the sequence, expressed using the `ltl.yield` operation. Finally, the type of the operation can be an `!ltl.sequence` or `!ltl.property` and is annotated with an integer parameter that encodes the end-to-end latency of this block (in cycles). This type can then be exposed to frontend languages as a *latency hint*, allowing frontend programmers to reason directly about delays (or any other timing properties). With this core operation, we have satisfied our expressivity goal: we support arbitrary logical relations, and can reason about them across time.

4 Lowering to Synthesizable Hardware

We now focus on our two remaining goals: compositionality and synthesizability. Compositionality is achieved through the use of incremental folding, which allows us to simply merge two `ltl.seq` operations, as illustrated in Figure 2 – where `%ab` and `%abc` are folded into a larger sequence. The implementation of this folding pass occurs in three steps. First, any delay applied to an argument of a sequence is propagated to that argument's inputs, e.g. `%a`, and `%b` in Figure 2, which are discovered through a DFS exploration of the argument's `yield` operand. Next, we *fold* any consecutive delays applied to the same input, yielding a single delay per input. Finally, we inline the body of the argument in its place and update the resulting type parameter to reflect the new end-to-end latency, as shown in Figure 2. This pass can be applied incrementally to reduce a complex expression into a single `ltl.seq` block. Finally, synthesizability is achieved by applying our lowerings until a fixed point is reached –

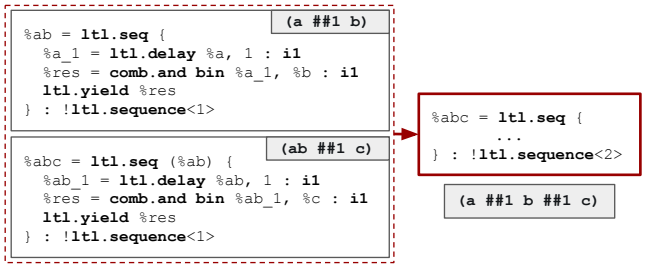

Figure 2. Folding pass resulting in `%abc` in Figure 1.

which happens when all of the individual sequence operations have been reduced to a single operation, as illustrated in Figure 1. This form can then be lowered to CIRCT's core dialects by encoding all delays as shift-registers. To ensure an expression is only enabled when its minimum valid cycle count is reached, we additionally introduce a global enable register; this minimum cycle count is encoded as the type parameter to the final, fully-folded sequence. Integrating this simple IR design into CIRCT enables the use of SVA property-style expressions in any verification backend that supports synthesizable Verilog, thus improving the overall viability of open-source verification tools.

5 Conclusion

In this work, we propose an IR design and incremental lowering that allows for temporal expressions to be encoded as synthesizable hardware without the need for expensive formulae-monitoring automata to be created. By lowering these expressions to synthesizable hardware inside of a generalized compiler such as CIRCT, we create a single point of truth for these expressions, allowing them to be understood and handled by any verification tool that supports synthesizable Verilog. This will allow any of CIRCT's frontends to utilize temporal expressions in the specification of their designs, and verify them without having to rely on commercial verification tools. The hope is that this will contribute to enabling a fully open-source hardware verification ecosystem.

References

[1] 2024. IEEE Standard for SystemVerilog–Unified Hardware Design, Specification, and Verification Language. *IEEE Std 1800-2023 (Revision of IEEE Std 1800-2017)* (2024), 1–1354. https://doi.org/10.1109/IEEESTD.2024.10458102

[2] Jonathan Bachrach, Huy Vo, Brian Richards, Yunsup Lee, Andrew Waterman, Rimas Avižienis, John Wawrzynek, and Krste Asanović. 2012. Chisel: constructing hardware in a Scala embedded language. In *Proceedings of the 49th Annual Design Automation Conference* (San Francisco, California) *(DAC '12)*. Association for Computing Machinery, New York, NY, USA, 1216–1225. https://doi.org/10.1145/2228360.2228584

[3] Clark W. Barrett, Aaron Stump, and Cesare Tinelli. 2010. The SMT-LIB Standard Version 2.0. https://api.semanticscholar.org/CorpusID:7943149

[4] Armin Biere, Keijo Heljanko, Tommi Junttila, Timo Latvala, and Viktor Schuppan. 2006. Linear Encodings of Bounded LTL Model Checking. *Logical Methods in Computer Science* Volume 2, Issue 5 (Nov. 2006). https://doi.org/10.2168/LMCS-2(5:5)2006

[5] Jacek Cichon, Adam Czubak, and Andrzej Jasinski. 2009. Minimal Büchi Automata for Certain Classes of LTL Formulas. In *2009 Fourth International Conference on Dependability of Computer Systems*. 17–24. https://doi.org/10.1109/DepCoS-RELCOMEX.2009.31

[6] 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.

[7] Amelia Dobis. 2024. *Formal Verification of Hardware using MLIR*. Master Thesis. ETH Zurich, Zurich. https://doi.org/10.3929/ethz-b-000668906

[8] Amelia Dobis, Kevin Laeufer, Hans Jakob Damsgaard, Tjark Petersen, Kasper Juul Hesse Rasmussen, Enrico Tolotto, Simon Thye Andersen, Richard Lin, and Martin Schoeberl. 2023. Verification of Chisel Hardware Designs with ChiselVerify. *Microprocessors and Microsystems* 96 (2023), 104737. https://doi.org/10.1016/j.micpro.2022.104737

[9] A. Duret-Lutz. 2011. LTL translation improvements in spot. In *Proceedings of the Fifth International Conference on Verification and Evaluation of Computer and Communication Systems* (Tunis, Tunisia) *(VECoS'11)*. BCS Learning & Development Ltd., Swindon, GBR, 72–83.

[10] Schuyler Eldridge, Prithayan Barua, Aliaksei Chapyzhenka, Adam Izraelevitz, Jack Koenig, Chris Lattner, Andrew Lenharth, George Leontiev, Fabian Schuiki, Ram Sunder, et al. [n. d.]. MLIR as hardware compiler infrastructure.

[11] Martin Erhart, Fabian Schuiki, Zachary Yedidia, Bea Healy, and Tobias Grosser. [n. d.]. Arcilator: Fast and cycle-accurate hardware simulation in CIRCT. https://llvm.org/devmtg/2023-10/slides/techtalks/Erhart-Arcilator-FastAndCycleAccurateHardwareSimulationInCIRCT.pdf.

[12] R. Gerth, D. Peled, M. Y. Vardi, and P. Wolper. 1996. *Simple On-the-fly Automatic Verification of Linear Temporal Logic*. Springer US, Boston, MA, 3–18. https://doi.org/10.1007/978-0-387-34892-6_1

[13] Chris Lattner, Mehdi Amini, Uday Bondhugula, Albert Cohen, Andy Davis, Jacques Pienaar, River Riddle, Tatiana Shpeisman, Nicolas Vasilache, and Oleksandr Zinenko. 2021. MLIR: scaling compiler infrastructure for domain specific computation. In *Proceedings of the 2021 IEEE/ACM International Symposium on Code Generation and Optimization* (Virtual Event, Republic of Korea) *(CGO '21)*. IEEE Press, 2–14. https://doi.org/10.1109/CGO51591.2021.9370308

[14] Aina Niemetz, Mathias Preiner, Claire Wolf, and Armin Biere. 2018. Btor2 , BtorMC and Boolector 3.0. In *International Conference on Computer Aided Verification*. https://api.semanticscholar.org/CorpusID:51868414

[15] Blake Pelton, Adam Sapek, Ken Eguro, Daniel Lo, Alessandro Forin, Matt Humphrey, Jinwen Xi, David Cox, Rajas Karandikar, Johannes de Fine Licht, Evgeny Babin, Adrian Caulfield, and Doug Burger. 2024. Wavefront Threading Enables Effective High-Level Synthesis. *Proc. ACM Program. Lang.* 8, PLDI, Article 190 (June 2024), 25 pages. https://doi.org/10.1145/3656420

[16] Amir Pnueli. 1977. The temporal logic of programs. In *18th Annual Symposium on Foundations of Computer Science (sfcs 1977)*. 46–57. https://doi.org/10.1109/SFCS.1977.32

[17] Fabian Schuiki and Amelia Dobis. 2024. CIRCT ltl dialect. https://circt.llvm.org/docs/Dialects/LTL.

[18] Lenny Truong and Pat Hanrahan. 2019. A Golden Age of Hardware Description Languages: Applying Programming Language Techniques to Improve Design Productivity. In *3rd Summit on Advances in Programming Languages (SNAPL 2019)*.

[19] Moshe Y. Vardi. 1996. *An automata-theoretic approach to linear temporal logic*. Springer Berlin Heidelberg, Berlin, Heidelberg, 238–266. https://doi.org/10.1007/3-540-60915-6_6

[20] Claire Wolf. [n. d.]. Yosys Open SYnthesis Suite. https://yosyshq.net/yosys/.