Leveraging Piecewise Composition to Infer Environment Constraints for Hardware Designs

Work-in-Progress Extended Abstract

Kaki Ryan and Cynthia Sturton
University of North Carolina at Chapel Hill
kakiryan@cs.unc.edu, csturton@cs.unc.edu

***Abstract*—We introduce a symbolic execution-based framework to generate the environment constraints needed to formally verify a hardware design. The core of the approach is a new search strategy that leverages piecewise composition, a divide-and-conquer algorithm introduced in prior work, to guide symbolic execution toward paths more likely to generate needed environment constraints. In our preliminary evaluation using the decoder module of the OpenTitan SoC, the framework finds the needed constraints without overconstraining the environment.**

***Index Terms*—hardware verification, formal models, symbolic execution**

I. Introduction

Hardware design companies like Intel and AMD rely on formal verification, typically model checking or symbolic simulation, to check functional and security properties of the designs early in the product development lifecycle . For large systems, verification is modular. Engineers may verify individual components such as a CPU core or accelerator destined for a system-on-chip (SoC).

To ensure sound and complete verification results, the design's inputs are constrained to reflect its intended operating environment. Underconstraining can lead to spurious counterexamples. Overconstraining can mask real bugs. Engineers often begin with minimal constraints and add more as needed. The key challenge is distinguishing real design bugs from artifacts of an underconstrained environment when a counterexample is found for a property ϕ in design D under environment e.

Prior work in assume-guarantee reasoning [1], abductive inference [2], and specification generation [3] has tackled related problems in software and hardware, but remains complex and difficult to scale. We take a practical approach based on symbolic execution, a path-based analysis technique .

II. Problem Statement and Approach

Let D be the design under verification, ϕ be the property to verify, and e be the environment, stated as a set of logical constraints over input values, under which the module will operate and under which the module should satisfy the property. The verification engineer must provide a set of constraints e' such that the property is true of the design when operating under input constraints e' if and only if the property is true of the design when operating in its environment e: $D \models_e \phi \Leftrightarrow D \models_{e'} \phi$.

979-8-3315-9379-7/25/$31.00 ©2025 IEEE

The framework starts by initializing the set of generated constraints, $e' := \top$, and then follows a two-step, iterative procedure similar to work in specification generation [4]:

1) Calculate a condition $\hat{e}$ such that $D \not\models_{\hat{e}} \phi$.
2) Search for evidence that $\neg\hat{e}$ can be violated by the true operating environment of the design under verification.

If such evidence is found, then the environment can produce conditions under which $D \not\models \phi$ and there is a bug in the design. If no evidence is found, then $\neg\hat{e}$ is added to the set of environment constraints for D: $e' := e' \wedge \neg\hat{e}$. The process then repeats at step 1, and a new condition $\hat{e}$ is found.

Importantly, we do not require that $\hat{e}$ be a necessary condition to violating ϕ; there may be a subset of $\hat{e}$ or a distinct set of conditions under which $D \not\models \phi$. As a consequence, it is not necessarily the case that the property holds under the generated constraint ($D \models_{\neg\hat{e}} \phi$ does not necessarily hold).

We prototype our framework using symbolic execution for Step 1 and dynamic simulation for Step 2. We aim to automate the manual, time-consuming work of the verification engineer, while ensuring that the outcome is at least as strong as what the human would produce. By weakening the requirements for $\hat{e}$, the framework can efficiently produce useful constraints, even for large designs in complex environments, over multiple clock cycles.

The framework's effectiveness relies on quickly identifying many candidate constraints $\hat{e}$ that are no more restrictive than necessary. We build on Sylvia, a symbolic execution engine that enables fine-grained modular path exploration [5]. This steers the analysis towards paths likely to violate property ϕ.

III. Background

Assume-Guarantee Reasoning: Assume-Guarantee reasoning simplifies verification by analyzing components individually and composing results [6]. Sound and complete assumptions are still required per component, limiting applicability despite automation efforts [7]. Our problem statement differs in three ways: we start with a single module and its environment (not a full, hard-to-decompose system [8]), focus on one component, and avoid full specification or guarantees of design behavior.

Abductive Inference: Abductive inference is an approach of logical reasoning for inferring the cause of a given result. The approach has been applied to generate invariants and preconditions in program and hardware verification [2]. In the context of hardware verification, one method uses SMT-based

reasoning to propose an environment predicate that, if assumed, would make an otherwise failing proof succeed [9].

Symbolic Execution with Piecewise Composition: Symbolic execution replaces inputs with symbolic values and tracks path constraints. In hardware, the number of paths grows exponentially with branches and cycles [10], [11]. Piecewise composition addresses this by separately executing independent blocks (e.g., `always` blocks) and reconstructing paths via SMT queries [5], [12]. We extend this with a search that prioritizes paths likely to produce $\hat{e}$.

IV. Leveraging Piecewise Composition to Efficiently Infer Environment Constraints

In Step 1, the framework uses symbolic execution to find violations of ϕ in the design. When a violation is found, the path condition describing the input constraints needed to take execution down that violating path becomes $\hat{e}$ such that $D \not\models_{\hat{e}} \phi$.

In general there may be many distinct paths that lead to a state in which the property is violated. If all such paths are found, then the conjunction of constraints $e' = \neg\hat{e}_0 \wedge \neg\hat{e}_1 \wedge \neg\hat{e}_2 \wedge \ldots$ forms a precise description of the constraints over inputs that must hold in order for ϕ to hold. If the true environment can be shown to violate e', then the design in its environment violates ϕ, otherwise it does not.

Since the number of paths is too large to explore exhaustively, we leverage piecewise composition to guide symbolic execution toward paths likely to violate the property, if any exist. Given a property stated as a temporal logic expression over variables, $\phi := f(v_1, v_2, \ldots, v_n)$, we prioritize 1) paths that write to a variable in $\{v_1, v_2, \ldots, v_n\}$, and 2) paths that write to variables that in turn affect writes to variables in $\{v_1, v_2, \ldots, v_n\}$.

With piecewise composition, we can do this efficiently. In SystemVerilog, only one `always` block writes to any given variable. The framework first identifies the `always` blocks $\{b_1, b_2, \ldots, b_n\}$ in which the variables $\{v_1, v_2, \ldots, v_n\}$ are written. During exploration, the framework will prioritize exploring new paths in the blocks $b_1, b_2, \ldots, b_n$. The dependency analysis may be extended. For each variable w_i that is *read* in blocks $b_1, b_2, \ldots, b_n$, the `always` block in which w_i is written is added to the set of blocks to prioritize for new paths.

V. Preliminary Results

We implement our strategy in Sylvia [5] and evaluate it on the Ibex decoder from the OpenTitan SoC with a property that holds in-core, but fails in an unconstrained environment. The property states that ALU instruction selectors are not unknown during Register-Immediate operations:

```
assert(opcode == OPCODE_OP_IMM) |->
   !$isunknown(instr[14:12]).
```

The decoder module has four `always` blocks and 1214 lines of code. All writes to the property's variables are in one `always` block. There are $O(2^{312})$ possible control-flow paths overall and $O(2^{107})$ possible in the `always` block.

The tool generates 12 environment constraints that prove the property. We validate the constraints against the true operating environment through simulation.

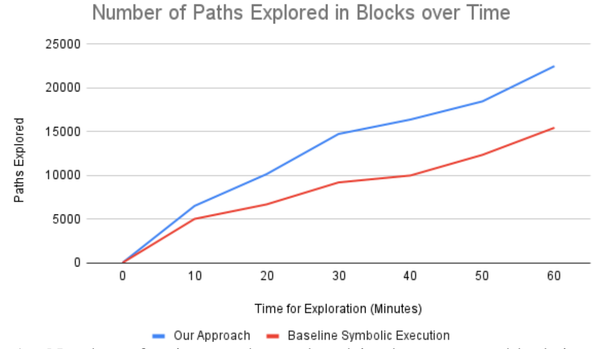

Fig. 1. Number of unique paths explored in the `always` block in which property variables are written with our strategy and with the baseline engine.

Each constraint was found in 4-6 seconds on average. As shown in Figure 1, our strategy uncovers more unique paths through the `always` block of interest than the baseline, leading to 8-11% faster constraint generation.

As a sanity check, we injected two bugs based on HACK@DAC 2018 Bug 13 and used properties to detect them [13]. Our tool generated constraints that, when validated in simulation, exposed the bugs without further tuning. This validates the constraints' correctness and the tool's effectiveness in uncovering true design flaws.

VI. Conclusion

We presented a symbolic execution framework that leverages piecewise composition to infer environment constraints of a hardware module. Our prototype finds the needed constraints for a module with high branching complexity without overconstraining the environment.

References

[1] C. Flanagan and S. Qadeer. Assume-guarantee model checking. In *Spin Symposium (SPIN)*. ACM, 2003.

[2] I. Dillig and T. Dillig. Explain: a tool for performing abductive inference. In *Computer Aided Verification (CAV)*. Springer, 2013.

[3] J. Choi, M. Vijayaraghavan, B. Sherman, A. Chlipala, and Arvind. Kami: A platform for high-level parametric HW specification and its modular verification. In *Intl Conf on Functional Prog (ICFP)*. ACM, 2017.

[4] H. Zhu, T. Dillig, and I. Dillig. Automated inference of library specifications for source-sink property verification. In *ESOP*. Springer, 2013.

[5] K. Ryan and C. Sturton. Sylvia: Countering the path explosion problem in the symbolic execution of hardware designs. In *Formal Methods in Computer-Aided Design (FMCAD)*. TU Wien Academic Press, 2023.

[6] C. Păsăreanu, M. Dwyer, and M. Huth. Assume-guarantee model checking of software: A comparative case study. In *SPIN*. Springer, 1999.

[7] J. Cobleigh, D. Giannakopoulou, and C. Păsăreanu. Learning assumptions for compositional verification. In *TACAS*. Springer, 2003.

[8] J.M. Cobleigh, G.S. Avrunin, and L.A. Clarke. Breaking up is hard to do: an investigation of decomposition for assume-guarantee reasoning. In *Intl Symp. on Software Testing and Analysis (ISSTA)*. ACM, 2006.

[9] S. Nedunuri and D. Smith. Inferring environment assumptions in model refinement. Technical report, Sandia National Lab, 2022.

[10] R. Zhang, C. Deutschbein, P. Huang, and C. Sturton. End-to-end automated exploit generation for validating the security of processor designs. In *Intl Symp on Microarchitecture (MICRO)*. IEEE/ACM, 2018.

[11] R. Zhang and C. Sturton. A recursive strategy for symbolic execution to find exploits in HW designs. In *Forml Methods & Security*. ACM, 2018.

[12] K. Ryan and C. Sturton. SylQ-SV: Scaling symbolic execution of hardware designs with query caching. In *ASPLOS*. ACM, 2025.

[13] J. Rogers, N. Shakeel, X. Tan, S. Espinosa, D. Mankani, C. Chabra, K. Ryan, and C. Sturton. HW security benchmarks for open-source SystemVerilog designs. In *Secure Devt Conf (SecDev)*. IEEE, 2025.