Formal Methods in Computer-Aided Design 2025

Unlocking Hardware Verification with Oracle Guided Synthesis

Leiqi Ye [1], Yixuan Li[1], Guy Frankel[1], Jianyi Cheng[1], Elizabeth Polgreen[1]
[1]*University of Edinburgh, UK*
{leiqi.ye, yixuan.li.cs, G.Frankel-1, jianyi.cheng, elizabeth.polgreen}@ed.ac.uk

Artifact Reviewed
10.5281/zenodo.16896910

***Abstract*—Hardware verification is essential to ensure that hardware designs meet their design specifications and function as intended. However, use of formal verification requires extensive manual work in order to write formal specifications. Specification mining aims to alleviate this manual burden, with conventional techniques using statistical methods and pattern matching to generate likely specifications from sample execution traces from the hardware. A limitation of this is that the quality of the specifications is determined by the quality of the traces. In this paper, we present an approach that uses oracle-guided synthesis to generate specifications for hardware, using counterexamples and negative examples to refine specifications generated based on traces. We evaluate our approach on real-world Verilog benchmarks and demonstrate that specifications generated by our tool can detect high proportions of hardware mutations.**

I. Introduction

Hardware verification ensures that a hardware design meets its specification(s) and functions correctly by identifying design flaws before manufacturing. Traditional hardware verification can either be done through simulation or formal verification. Simulation tests a design on a set of predefined test cases, potentially missing corner cases that may cause errors; whereas formal verification captures all possible cases under a set of specifications given by the designer. Formal verification is widely used in hardware verification today due to its better coverage, because a single undetected bug can necessitate a redesign and re-fabrication, incurring millions of dollars in losses.

However, a major challenge in hardware formal verification is writing formal specifications for a given hardware design. Manually writing specifications is time-consuming, usually taking up to 60%-70% of the development time [40], and requires significant expertise due to the complexity of both hardware design and its requirements [18], [39]. As a result, there is substantial interest in automatically generating specifications from arbitrary hardware designs. This process is usually referred to as specification mining [4], [31], [32], and has been applied to software and hardware. Most specification mining algorithms are passive and based on inferring specifications using pattern matching on traces passively collected from a system. These approaches rely on the set of traces being sufficiently comprehensive to capture the behavior of the model, and need carefully designed property templates to avoid overfitting to the traces.

In this paper, we present **S**pecification **M**ining of **A**ssertions, **R**efined via **T**races (SMART): a specification mining method for hardware designs based on oracle-guided inductive synthesis. Our synthesis process is based on 3 types of examples: we collect simulation traces from the hardware design, which we convert into *positive examples*, we use a model checker to generate unreachable states, which are treated as *negative examples*, and once the synthesizer has proposed a candidate assertion, we use a model checker to generate *counterexamples* if the assertion does not fully capture the behavior of the hardware.

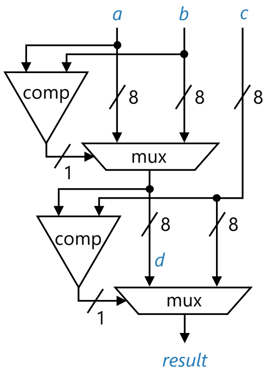
We evaluate our approach on Verilog benchmarks from the literature and assess the specifications generated by our approach by mutating the hardware benchmark under test. A specification that can differentiate between the original design and a high number of mutant hardware designs is likely to be a *meaningful* specification. Assertions generated by SMART can differentiate a high proportion of mutants, achieving a 100% mutation detection rate on several real-world benchmarks. In comparison to related work, our approach infers meaningful specifications without requiring custom templates or carefully provided trace sets and provides formal guarantees that the specifications hold.

II. Motivating Example

Figure 1a shows a digital circuit that determines the maximum value among three 8-bit inputs. The circuit compares the values between the first two inputs `a` and `b` and passes the larger value for the next comparison, annotated as `d`. The result is then compared with the third input, where the maximal value is determined and sent to the output `result`. The corresponding source that generates the digital circuit is shown in Figure 1b, described in Verilog, a hardware description language (HDL) for digital circuits [43]. The Verilog source contains three main components: IO declaration, hardware description, and property assertion. First, both the three inputs and the output of the digital circuit are specified in lines 4-7 in eight bits. Second, lines 9-13 describe the design of the circuit in Figure 1a using two conditional assignments. Finally, lines 15-19 describe the invariants of the digital circuit design, also known as design specifications.

The first two parts form a design description that can be directly mapped to a digital circuit. Descriptions of real-world

https://doi.org/10.34727/2025/isbn.978-3-85448-084-6_30



(a) A digital circuit that determines the maximal value among three 8-bit inputs. comp = comparator, and mux = multiplexer.

```
1  // Return the max value
2  // of three inputs.
3  module max3(
4    input  [7:0] a,
5    input  [7:0] b,
6    input  [7:0] c,
7    output [7:0] result
8  );
9    wire   [7:0] d;
10   // d = max(a, b)
11   assign d = (a>=b) ? a:b;
12   // result = max(d, c)
13   assign result = (d>=c) ?
       d:c;
14
15   // assertions
16   assert property
17     (a <= result);
18   assert property
19     (d <= result);
20 endmodule
```

(b) The Verilog and SVA source for the digital circuit.

Fig. 1: A motivating example. Existing work [21] only supports boolean circuits and cannot automatically generate SVAs for the example above. SMART automatically generates efficient SVAs for formal verification.

hardware designs are often more complex and finer-grained, down to the bit level, leading to a high risk of introducing design bugs. For example, the ">" character at line 13 may be mistakenly typed as "<", leading to a syntactically correct but functionally incorrect condition `a <= b`. Such design bugs could be captured by hardware verification using design invariants. The invariants for Verilog designs are described in a property specification language named SystemVerilog Assertions (SVA) [44], as defined in Section III-A. In verification, the circuit behavior is formally verified to match each SVA under any input values. The invariants on lines 15-19 of Figure 1b would fail if the circuit contained the previously described bug.

Still, existing approaches in hardware verification for a given Verilog design require human effort. A key challenge in automatically generating such specifications is the unique semantics of hardware description languages like Verilog compared to traditional software programs. Traditional static analysis techniques on software programs cannot be directly applied to digital circuit descriptions. For example, re-ordering statements in Verilog, such as swapping lines 11 and 13 in Figure 1b, does not affect the correctness of the output, while this usually causes errors in software programs that follow sequential execution. The analysis process could become more challenging when the digital design is implemented with complex obfuscation for security reasons [8].

Existing work uses customized static analysis to infer SVA [21]. However, it focuses on a small set of Verilog designs and only works under stringent constraints. For example, the hardware design must be described as a boolean circuit, while modern hardware designs contain wires of multiple bits. We seek a general approach for automated SVA generation.

This example also illustrates a key limitation of template-based assertion mining tools: they often rely on fixed syntactic forms or single-variable patterns. In contrast, the assertions shown in Figure 1(b), such as `assert property (a <= result);` and `assert property (d <= result);`, involve multiple variables and relational reasoning across signal dependencies. These kinds of compound assertions fall outside the expressiveness of common pattern-based templates. SMART, by framing assertion generation as a formal synthesis problem, is able to automatically infer such expressive multi-variable properties without relying on pre-defined templates.

The SVAs should precisely describe the hardware behaviors with minimal tolerance for design discrepancies. We formalize the hardware verification problem into a formal synthesis problem and leverage existing program synthesis tools for the automated generation of efficient SVAs. In the rest of the paper, we will describe our approach in detail.

III. Background

A. Verilog and SystemVerilog Assertions (SVAs)

Verilog is an HDL used to describe digital electronic systems. It has been widely used in digital system design and verification at register-transfer level (RTL). Compared to traditional software programming languages, such as C or Python, which describe the sequential execution of instructions, Verilog is tailored to describe parallel hardware behavior.

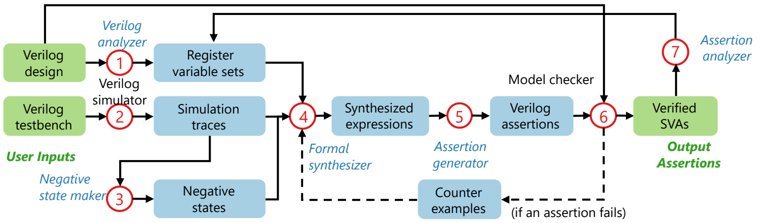
Most HDLs, including Verilog, provide two main programming styles for describing hardware designs: structural and behavioral descriptions. A structural description presents a circuit at a low level, specifying how individual components, such as gates, multiplexers, or registers, are interconnected. A behavioral description is used for high-level descriptions of hardware behaviors, focusing on what the circuit should do rather than how it is implemented. To maximize generality and avoid targeting particular use cases, our work supports both structural and behavioral Verilog inputs.

On the verification side, SVAs are used as an extension of the Verilog language for formal verification. Unlike traditional run-time assertions that are evaluated during simulation and are limited to runtime checking, SVAs specify properties that must always hold true, and can be checked with formal verification, to ensure that the implemented design meets its design specifications. In this work, we take Verilog and its SVA generation, as the example, to show how program synthesis can guide automated hardware verification. Our method is general and can be applied to other HDLs, for example VHDL [5] and its own Property Specification Logic [13].

B. Syntax-Guided Synthesis (SyGuS)

SyGuS is a formal framework for program synthesis, which automatically generates programs that satisfy a given specification. We frame the problem of generating assertions as a SyGuS problem.

A SyGuS problem has a closed formula of the form $\exists F.\forall \vec{x}.\phi$, where F is a function to be synthesized, $\vec{x}$ is a list of universally quantified variables, and ϕ is a first-order



(a) Overview of SMART. Our contributions are ***highlighted***.

```
(set-logic BV)
(synth-fun inv ((d (_ BitVec 8))
(result (_ BitVec 8))) Bool)
; Constraints from the simulation
    trace:
(constraint (= (inv #142 #148) true)
    )
(constraint (= (inv #172 #179) true)
    )
; Constraints from negative states:
(constraint (= (inv #111 #2) false))
(check-synth)
```

(b) A simplified example of a synthesis conjecture generated from simulation traces and negative states. It reasons about the variables d and $result$ in the circuit in Figure 1.

Fig. 2: The SMART tool workflow and SyGuS example for invariant synthesis.

formula in a background theory T which specifies the semantic correctness constraints for F. The synthesis function F has a corresponding grammar, G. A valid solution to a SyGuS formula is a function f such that $\forall x.\phi[F \mapsto f]$ is T-satisfiable and $f \in \mathcal{L}(G)$, where $\mathcal{L}(G)$ denotes the language defined by G.

C. Oracle Guided Inductive Synthesis

SyGuS problems are solved by Oracle-Guided Inductive Synthesis algorithms (OGIS) [25]. Given a SyGuS problem as described above, OGIS alternates between a learner, which guesses or enumerates through candidate programs in G, and an oracle that the learner may query to check if the candidate program satisfies the specification, or to obtain feedback if it does not. The most common variant of OGIS is CounterExample Guided Inductive Synthesis(CEGIS) [41], where the learner proposes a candidate program f^* to the oracle. If the candidate program fails to satisfy the specification, i.e., $\exists x.\neg\phi[F \mapsto f^*]$, the oracle returns a *counterexample*, which is an assignment to x [41]. The learner stores a set of counterexamples C, and then searches for a new candidate function such that $\forall x \in C.\phi[F \mapsto f]$ is T-satisfiable.

However, other oracles, which provide other forms of guidance, are sometimes used. Prior work in invariant synthesis [16] uses oracles that provide positive examples (on which the invariant should evaluate to true) and negative examples (on which the invariant should evaluate to false) to guide the search.

IV. Overview

A. Problem Statement

Given a hardware design, we wish to find a set of assertions that are valid (i.e., never violated by the hardware design). We frame this as a formal synthesis problem. First, let us define what we mean by hardware design.

Definition 1: A hardware design H is a tuple $\langle V_H, I, S, s_0, T\rangle$

- where V_H is a set of m state variables.
- $V_i \subseteq V_H$ is a set of input variables, i.e., variables that can be assigned to by an external source.
- S is a set of states, where each state corresponds to a full assignment to all variables in V_H, i.e., $s \in S$ and $s = \{v_1 \mapsto c_1, \dots v_m \mapsto c_m\}$ where $V_H = \{v_1, \dots v_m\}$ and $c_1, \dots, c_m$ are constant literals.
- $I \subseteq S$ is a set of initial states.
- $T \subseteq S \times S$ is a predicate that defines the set of possible transitions between states.

A valid assertion or specification is a predicate over the state variables that is true for all reachable states, i.e., it is never violated by any execution of H. We define this formally as follows:

Definition 2: A sequence π of states $s_0, \dots s_n$ is an execution in H if, for each $0 \leq i \leq n, \langle s_i, s_{i+1}\rangle \in T$. A state $s \in S$ is reachable if there is a trace $s_0, \dots s$, where $s_0 \in I$, i.e., if there is a path from an initial state to s.

Definition 3: An assertion $\alpha \subseteq S$ is a predicate over (a subset of) the variables V_H. We use $\alpha(s)$ to indicate the result of evaluating the predicate α over the assignment to the state variables given by s. An assertion thus defines a set of states $S_\alpha = \{s \,|\, \alpha(s) = \top\}$. An assertion α *holds* if and only if $S_\alpha \subseteq R$, where R is the set of all reachable states. If α holds, we say that $H \models \alpha$.

We note that the generated specifications should also be *meaningful*: they should not be vacuously true, and they should be compact and more readable than the hardware model itself. We evaluate our specifications against these criteria using mutation detection, described in Section VII-A2.

a) Formal Problem Statement: Formally, given a hardware design H (see Definition 1), we wish to generate a set of assertions $\alpha_1, \dots, \alpha_n$ such that $H \models (\alpha_1 \wedge \dots \wedge \alpha_n)$. Thus, the synthesis conjecture we wish to solve is $\exists \alpha_1, \dots \alpha_n.\forall V_H.\phi$. The terms are defined as follows.

- $\alpha_1 \dots \alpha_n$ are the assertions to be synthesized.
- V_H is the set of variables in H.
- ϕ is the specification, which should state that $H \models (\alpha_1 \wedge \dots \wedge \alpha_n)$.

We also provide a grammar for $\alpha_1, \dots, \alpha_n$, which serves to reduce overfitting as well as reduce the search space of possible assertions.

B. Proposed SMART Framework

We will explain how we use synthesis from positive and negative examples to overcome the above challenges. Specifically, Section V describes how we collect simulation traces from H and use these to construct the grammar, and Section VI describes the construction of our oracle-guided synthesizer. An overview of the SMART framework is illustrated in Figure 2a:

① A lightweight static analyzer parses the Verilog design to identify the key variables (register variable sets in the figure) used to construct grammars for the syntax-guided synthesis, described in Section V-B;

② For given user inputs, a hardware simulator generates a set of simulation traces as *positive examples*, described in Section V-A;

③-⑤ A random search is called with a hardware model checker to find unreachable states in the hardware design, known as *negative examples*, which are used to construct a synthesis conjecture, shown in Figure 2b together with the simulation traces and variables for an oracle-guided program synthesizer, leading to synthesized assertions that satisfy the synthesis conjecture and are later translated to syntactically correct SVAs, described in Section VI;

⑥ The assertions are checked against the hardware using an unbound model checker, resulting in either verification success (passing assertions to the output) or failure with *counterexamples* (refining the synthesis conjecture to produce more assertions), described in Section VI-C;

⑦ The verified assertions are analyzed to refine the variable set used to generate the grammar, accelerating the mining process, described in Section VI-D.

V. Trace Collection & Grammar Construction

The core idea behind SMART is to use formal synthesis to generate specifications for hardware designs. The inputs to our formal synthesis are a set of variables to be used in the specification, as well as a set of positive examples, negative examples, and counterexamples. In this section, we describe how we obtain the variable set, the positive examples, and the negative examples.

A. Verilog Simulation

Here, we explain how we obtain positive examples through simulation traces obtained from the Verilog simulator. We assume that a hardware design H comes with a corresponding hardware testbench. A hardware testbench is a simulation environment, used with a simulator to test the states of a hardware design by simulation. Different values are fed into the inputs V_i of the hardware design at different time steps, and the outputs are checked to detect potential discrepancies compared to given expectations. This leads to a simulation trace for each test run. A simulation trace contains a subset of internal wire variables V_H and records their states over a sequence of time stamps. For example, a trace may include an event where both `a` and `result` increase from 8 to 10 while `b` and `c` are both 5 at the time stamp 50 nanoseconds after the start of the simulation.

Each simulation trace is translated into an execution trace (see Definition 2).

B. Identifying the Variable Sets: Verilog Analysis

Formal synthesis in general relies on enumerating the space of programs, leading to a space growing exponentially with the number of variables. To reduce this problem, SMART uses heuristics to select only random subsets of variables for assertion synthesis. We use a lightweight analysis of the simulation trace and Verilog code to extract the register variables that only change values with clock edges. We experimented with heuristic-based selection strategies – variable dependency analysis, which attempts to select semantically related variables. However, empirical evaluation showed the method provides limited benefit in our setting.

Specifically, we identify the register variables from the hardware, denoted V_H. For a given trace π, we define a corresponding set of trace variables V_π to be the set of variables that change at some point during the trace. We use $s_t(v_i)$ to indicate the value assigned to variable v_i at time step s_t in one trace. That is $V_\pi = \{v \in V_H | \exists s_i, s_{i+1} \in \pi \wedge s_i(v) \neq s_{i+1}(v)\}$. Given a collection Π of traces, let $V_\Pi = \bigcup_{\pi \in \Pi} V_\pi$ be the union of all trace variables appearing in Π. This ensures we consider only variables that feature in the trace and are also in the Verilog design, i.e., variables introduced by the simulator and parameters defined but never used in the design are never considered.

SMART adopts a randomized variable subset selection strategy. We randomly sample a small subset of variables $V_i \subset V_\Pi$ These variable sets are updated at each iteration of SMART. Suppose α is synthesized from a subset $V_i \in \mathcal{V}$; Once α is produced, the assertion analyzer updates $V_i \leftarrow V_i \backslash \mathrm{var}(\alpha)$, where $\mathrm{var}(\alpha)$ returns the variables used in α. This avoids reusing those same variables in future assertions. SMART continues this process, reducing the size of the variable set V_i in each iteration until either $V_i = \emptyset$, or the synthesis problem is infeasible with the remaining variables, or a generated assertion is discovered to be invalid via formal verification. Once one of these conditions is satisfied, SMART moves on to considering V_{i+1}. The size of the initial subset is determined by the total number of variables in V_Π, that is:

$$|V_i| = \begin{cases} |V_\Pi| - 1, & \text{if } |V_\Pi| \leq 5 \\ 5, & \text{if } 5 < |V_\Pi| \leq 400, \\ 20, & \text{if } 400 < |V_\Pi|, \end{cases}$$

where $|V|$ indicates the number of variables in the set V. This ensures the extracted variables efficiently capture key hardware states, leading to more meaningful assertions.

VI. Oracle-Guided Synthesizer

In this section, we describe our Oracle Guided Inductive Synthesis approach to generating SVA assertions, as shown in Algorithm 1.

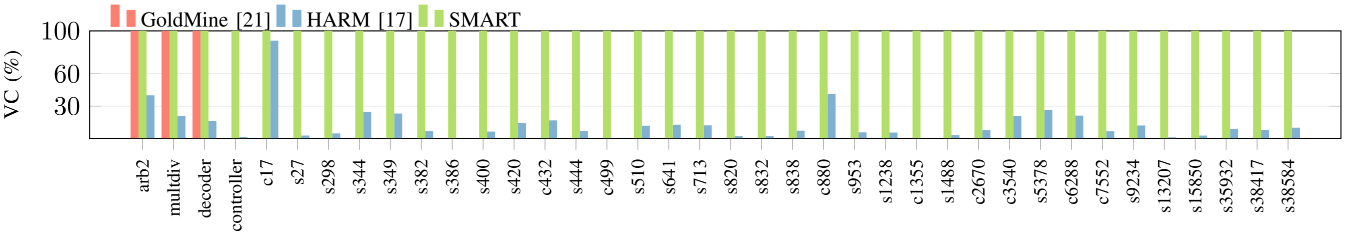


Fig. 3: Verification Correctness (VC) rates of different approaches over various benchmarks.

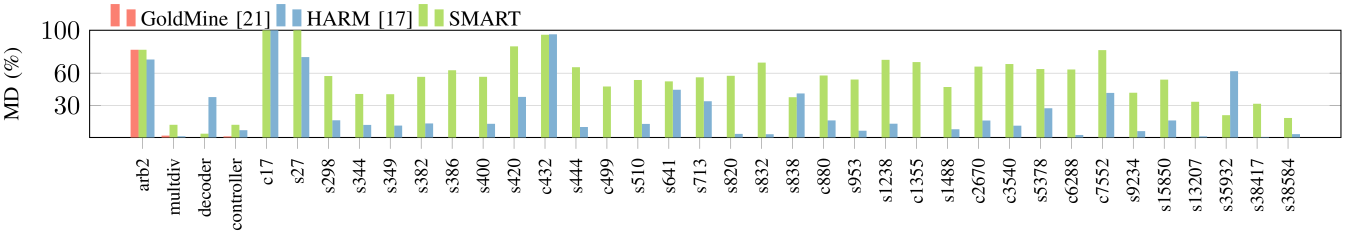


Fig. 4: Mutation detection (MD) rates of different approaches over various benchmarks.

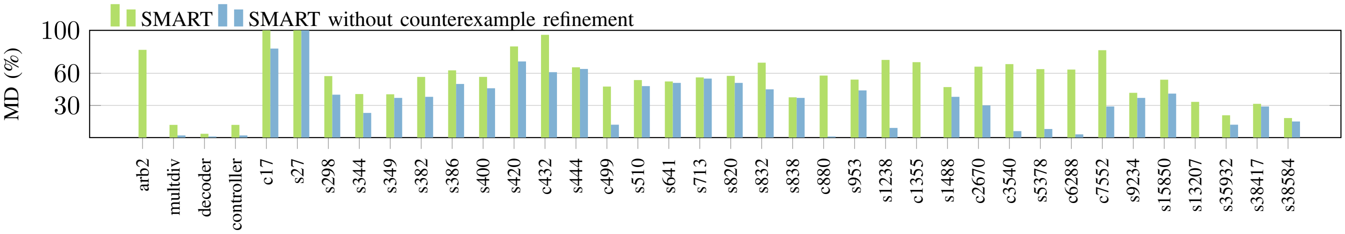


Fig. 5: Impact of Counterexamples on SMART's Mutation Detection Performance.

A. *Constructing the Grammar G*

The first step of the synthesis is to generate a grammar for the synthesis problem. Given a variable set V and a background theory, SMART constructs the following grammar:

$$\begin{aligned}
Bool ::= & Bool \simeq Bool \mid \neg Bool \mid Bool \wedge Bool \mid \\
& Bool \vee Bool \mid \text{any boolean } v \text{ in } V \\
& Bv \simeq Bv \mid Bv \geq Bv \mid Bv \leq Bv \mid & * \\
& Bv < Bv \mid Bv > Bv \mid Bv \geq_u Bv \mid & * \\
& Bv \leq_u Bv \mid Bv <_u Bv \mid Bv >_u Bv & * \\
Bv ::= & - Bv \mid \sim Bv \mid Bv \,\&\, Bv \mid Bv \circ Bv \mid Bv + Bv \mid & * \\
& Bv - Bv \mid Bv * Bv \mid \text{any bitvector } v \text{ in } V & *
\end{aligned}$$

$\circ$ denotes bitwise OR, and & denotes bitwise AND operation. X_u is the unsigned version of the comparator X, e.g., $<_u$ indicating an unsigned $<$. $\simeq$ denotes equality, and $=$ denotes assignment. SMART omits lines marked $*$ if there are no bitvector variables in V.

B. *Positive and Negative Examples*

Recall that our specification ϕ for our synthesis problem for synthesizing $A = \alpha_1 \ldots \alpha_n$ is that $H \models A$. It is infeasible to give this specification to an off-the-shelf SyGuS solver, which implements CEGIS, because encoding the formula $H \models A$ requires unwinding the transition system of H and suffers from the classic model-checking state-space explosion problem [11]. Instead, we construct an OGIS loop based on using positive and negative examples, combined with counterexamples.

A *positive example* s^+ is a reachable state in H. The hardware $H \models \alpha$ if and only if $\alpha(s^+) = \top$.

A *negative example* s^- is an unreachable state in H. We use negative examples to refine the assertions, and thus we require that $\alpha(s^-) = \bot$.

We generate a set of positive examples, $\mathcal{P}$ from the simulation traces. For every step in a trace $\pi = s_0, s_1 \ldots$, we generate a single positive example. In order to generate negative examples, we randomly generate assignments to the state variables. This gives us a random state s. We then check whether $s \in R$ using a model checker with k-induction. If s is not reachable, we add it to a set of negative examples $\mathcal{N}$.

Algorithm 1 Oracle Guided Inductive Synthesis

```
procedure OGIS($H, \mathcal{P}, \mathcal{N}, \mathcal{V}$)
    $A \leftarrow \emptyset$, $i \leftarrow 0$
    $V \leftarrow \mathcal{V}$, $G \leftarrow$ GENERATEGRAMMAR($V$)
    while $true$ do
        $\alpha \leftarrow$ ENUMERATE($G, \mathcal{P}, \mathcal{N}$)
        if VERIFY($\alpha, H$) then
            $A \leftarrow A \cup \alpha$
            $V \leftarrow V \setminus var(\alpha)$
            if $V = \emptyset$ then
                $i \leftarrow i + 1$
                $V \leftarrow \mathcal{V}$
                $G \leftarrow$ GENERATEGRAMMAR($V$)
            end if
        else
            $c \leftarrow$ VERIFY.GET_CEX
            $\mathcal{P} \leftarrow \mathcal{P} \cup \{c\}$
            $i \leftarrow i + 1$
            if $i = |\mathcal{V}|$ then return $A$                ▷ No more variable sets
            end if
            $V \leftarrow \mathcal{V}$
            $G \leftarrow$ GENERATEGRAMMAR($V$)
        end if
    end while
end procedure
```

Given a set of positive and negative examples, SMART uses program synthesis to synthesize an assertion that evaluates to true for the positive examples and false for the negative examples. That is, it solves the synthesis conjecture $\exists\alpha(\forall s \in \mathcal{P}. \alpha(s) = \top \wedge \forall s \in \mathcal{N}. \alpha(s) = \bot)$. We use an off-the-shelf enumerator built into cvc5 [6] to determine α. Given α, SMART applies simple syntactic translation rules to translate the assertions to syntactically correct SVAs for verification.

In our implementation, we generate one negative example per iteration. This is based on our observation that a single unreachable state is often sufficient to eliminate trivial or vacuously true assertions during synthesis. Using only one example keeps the synthesis problem small and improves solver performance. This design choice strikes a balance between counterexample strength and synthesis tractability, and we found in practice that increasing the number of negative examples per iteration did not significantly improve assertion quality or mutation detection rate.

C. Counterexamples

The generated SVAs are verified against H using the same model checker as the negative example generation, using k-induction [38]. We use $H \not\models \alpha$ to indicate that the hardware model H violated the assertion α, and $H \models \alpha$ to indicate that H did not violate the assertion.

If $H \not\models \alpha$, the verifier provides a counterexample trace $\pi_{cex} = s_0 s_1 \ldots s_k$ where $\alpha(s_k) = \bot$. All states in π_{cex} are, by definition, reachable, and so an α such that $H \models \alpha$ should evaluate to true on any state in π_{cex}. SMART adds all states in the counterexample to the set of positive examples.

In practice, counterexamples are frequently generated during the synthesis process, especially in early iterations where candidate assertions are often overly general. These counterexamples play a critical role in refining the synthesized assertions by eliminating vacuously true or weak predicates. We observe that incorporating counterexamples significantly improves both the quality and effectiveness of the final assertion set. The impact of counterexample refinement is further discussed in the section VII.

D. Assertion Analysis

If α is valid, SMART adds it to the set of assertions A, removes the variables in the assertion from the current variable set, and repeats the synthesis process. If the variable set is then empty, SMART moves on to the next variable set. The assertion analyzer also removes any syntactic duplicates. The iterative process terminates either when $|A|$ reaches a pre-defined limit, or we reach a pre-defined time-out, or all variable sets are exhausted.

VII. Evaluation

We implement SMART, using cvc5 [6] version 1.2.0 to enumerate assertions, and EBMC [1] version 5.5 to verify the correctness of the generated SVAs and as the model checker that provides counterexamples inside the synthesis process. We use Cocotb [37] as the Verilog simulator. We use benchmarks from the related work [21] and two standard low-level hardware benchmark sets, ISCAS'85 and ISCAS'89 [2]. The benchmark set published by GoldMine includes a set of realistic digital circuit designs in behavioral Verilog: `arb2`, `controller`, `decoder`, `ld_stage` and `multdiv`, taken from the existing Ibex RISC-V CPU implementation [36]. There are also two standard benchmarks to describe low-level digital circuit designs. The ISCAS'85 benchmark [19] contains a set of combinatorial circuits, which consist of logic gates only; and the ISCAS'89 benchmark [7] which contains a set of sequential circuits whose hardware behaviors depend on clock signals. We evaluated SMART, HARM [17] and GoldMine [21] over all the benchmarks that have available Verilog sources. Result format converters are added into the workflow of HARM and GoldMine to make their result into SVA to run our evaluator. This is provided in the artifacts.

A. Evaluation Metrics

We evaluate the quality of the generated specifications by SMART based on the following two metrics:

1. Correctness: the inferred specifications should correctly describe the hardware behavior;

2. Meaningfulness: the specifications should differentiate between the original design and a high number of mutant hardware designs.

These are described in detail in the following sections.

1) Correctness of Assertion Sets: All assertions generated by SMART are correct with respect to the hardware design, as verified by the model checker. As not all the tools provided in related works use formal verification, some assertions generated by these works do not hold on the original model, i.e., $H \not\models A$.

a) Formal Verification of SVAs: We verify all assertions generated by the related work using k-induction. For each assertion, we run it on the original hardware model; if it passes verification, we keep it; otherwise, we discard it. We then compute the verified correctness rate, VC-rate, which measures the proportion of assertions that correctly reflect the model. Suppose that A_{gen} is the set of generated assertions for one hardware design, and $A_{ver} = \{a \in A_{gen} \mid H \models a\}$ is the set of assertions that pass verification. We calculate the VC-rate as $\frac{|A_{\text{ver}}|}{|A_{\text{gen}}|}$.

2) Meaningfulness of Assertion Sets: In order to assess how *meaningful* our specifications are, we use mutation tests, as is standard in the literature [23]. Intuitively, a good specification should be one that can detect if we change the underlying hardware design. To enable a fair comparison with other tools when performing mutation testing, we only perform testing on the assertions that pass verification.

a) Mutant Generation: We use the mutation rules from Repinski *et al.* [22], as shown in Figure 6. The left-hand side indicates the operators to which a rule can be applied, and the right-hand side indicates the result, for example, rule r_{g1} replaces any gate operator with $\vee$. In general, rules $r_{g1} \dots r_{g8}$ are applicable to gate operators. Rules $r_{b1} \dots r_{b3}$ are applicable to bitwise operators. Rules $r_{a1}, \dots r_{a5}$ are applicable to arithmetic operators. Rules $r_{r1} \dots r_{r6}$ are applicable to relational operators, and we use $\simeq$ to indicate equality. The rule r_{neg} negates any boolean variable, and r_{ran} assigns a random value to any bitvector literal.

We iterate over the Verilog file, and for each operator, variable, and literal we encounter, we identify the set of applicable rules. We then randomly choose one rule from this set, and apply it to generate a new mutant. This generates a randomly chosen subset of all possible mutations that can be generated by applying one mutation rule to H. This is shown in Algorithm 2.

b) Mutant Detection: Given a hardware design H and a conjunction of synthesized assertions A, we mutate H using a pre-defined set of rewrite rules to generate a set of hardware designs $\mathcal{H}_{mut}$. A mutant H' can be detected by our set of assertions A if $H \models A$ and $H' \not\models A$. We use k-induction, as implemented in EBMC, to determine whether a mutant can be detected by a given assertion set.

We define the mutation detection rate to be:

$$\text{MD-rate} = \frac{|\{H' \in \mathcal{H} \mid H' \not\models \alpha\}|}{|\mathcal{H}|}$$

B. Baselines

We compare SMART to two related works: HARM [17] and GoldMine [21]. HARM is based on learning patterns from simulation traces, and we run it with 10,000-cycle random input traces. GoldMine combines static analysis with decision-tree-based data mining for SVA generation and uses the closed-source JasperGold tool to verify the correctness of the generated assertions. We replace JasperGold with EBMC as the formal verification tool, and the raw output from GoldMine is included in the benchmark for comparison. Unfortunately, GoldMine only supports behavioral Verilog. Both tools generate assertions in their own formats. For HARM, we provide a translation tool to convert its output into SVA. For GoldMine, we manually process its raw data into SVA. The resulting SVAs from both tools are evaluated using the metrics mentioned above.

$$(\wedge\ |\uparrow|\downarrow|\ \oplus \mid \odot \mid \neg \mid buf) \xrightarrow{r_{g1}} \vee$$
$$(\vee\ |\uparrow|\downarrow|\ \oplus \mid \odot \mid \neg \mid buf) \xrightarrow{r_{g2}} \wedge$$
$$(\vee \mid \wedge\ |\downarrow|\ \oplus \mid \odot \mid \neg \mid buf) \xrightarrow{r_{g3}} \uparrow$$
$$(\vee \mid \wedge\ |\uparrow|\ \oplus \mid \odot \mid \neg \mid buf) \xrightarrow{r_{g4}} \downarrow$$
$$(\vee \mid \wedge\ |\uparrow|\downarrow|\ \odot \mid \neg \mid buf) \xrightarrow{r_{g5}} \oplus$$
$$(\vee \mid \wedge\ |\uparrow|\downarrow|\ \oplus \mid \neg \mid buf) \xrightarrow{r_{g6}} \odot$$
$$(\vee \mid \wedge\ |\uparrow|\downarrow|\ \oplus \mid \odot \mid buf) \xrightarrow{r_{g7}} \neg$$
$$(\vee \mid \wedge\ |\uparrow|\downarrow|\ \oplus \mid \odot \mid \neg) \xrightarrow{r_{g8}} buf$$
$$(\circ\ |\hat{}\ ) \xrightarrow{r_{b1}} \&$$
$$(\&\ |\hat{}\ ) \xrightarrow{r_{b2}} \circ$$
$$(\& \mid \circ) \xrightarrow{r_{b3}} \hat{}$$
$$\text{bitvector literal} \xrightarrow{r_{ran}} \text{random value}$$

$$(- \mid * \mid / \mid \%) \xrightarrow{r_{a1}} +$$
$$(+ \mid * \mid / \mid \%) \xrightarrow{r_{a2}} -$$
$$(+ \mid - \mid / \mid \%) \xrightarrow{r_{a3}} *$$
$$(+ \mid - \mid * \mid \%) \xrightarrow{r_{a4}} /$$
$$(+ \mid - \mid * \mid /) \xrightarrow{r_{a5}} \%$$
$$(\neq|>|<|\geq|\leq) \xrightarrow{r_{r1}} \simeq$$
$$(\simeq|>|<|\geq|\leq) \xrightarrow{r_{r2}} \neq$$
$$(\simeq|\neq|<|\geq|\leq) \xrightarrow{r_{r3}} >$$
$$(\simeq|\neq|>|\geq|\leq) \xrightarrow{r_{r4}} <$$
$$(\simeq|\neq|>|<|\leq) \xrightarrow{r_{r5}} \geq$$
$$(\simeq|\neq|>|<|\geq) \xrightarrow{r_{r6}} \leq$$
$$\text{bool} \xrightarrow{r_{neg}} \neg\text{bool}$$

Fig. 6: Mutation rules. We use $\uparrow, \downarrow, \oplus, \odot$ to represent NAND, NOR, XOR, and XNOR respectively. $\&, \circ$, and $\hat{}$ represent bitwise AND, OR and XOR respectively. buf indicates the Verilog operator `buf`, which transfers an input to an output without changing polarity. This is thus equivalent to removing an operator.

Algorithm 2 Generation of Hardware Mutants

```
procedure GenerateMutants(Verilog file $F$, Rules $R$)
    $Mutants \leftarrow \emptyset$
    for $op$ in $F$ do                         ▷ Iterate over operators in file
        $R_{app} \leftarrow$ FindApplicableRules($R, op$)
        $r \leftarrow$ RandomSample($R$)
        $F' \leftarrow$ Mutate($F, op, r$)
        $Mutants \leftarrow F'$
    end for
    for boolean var $v$ in $F$ do
        $F' \leftarrow$ Mutate($F, op, r_{neg}$)
        $Mutants \leftarrow F'$
    end for
    for literals $l$ in $F$ do
        $F' \leftarrow$ MutateNumeral($F, op$)
        $Mutants \leftarrow F'$
    end for
    return $Mutants$
end procedure
```

C. Results

Table I shows a summary of the evaluation results.

a) VC-rate: We run formal verification of the generated assertions by various approaches and report the VC-rates in Figure 3. Overall, most of the assertions generated by HARM failed the verification in EBMC with returned counterexamples. This is because HARM infers the specifications of de-

TABLE I: Summary of results comparing SMART and related work. We report the average size of the generated assertion set A_{gen}, VC-rate and MD-rate across all benchmarks. The # column reports the number of benchmarks supported by each approach.

	Structural Verilog (34 benchmarks)				Behavioral Verilog (4 benchmarks)				All (38 benchmarks)			
	#	$\|A_{gen}\|$	VC-rate	MD-rate	#	$\|A_{gen}\|$	VC-rate	MD-rate	#	$\|A_{gen}\|$	VC-rate	MD-rate
GoldMine	0	0	n/a	0.0%	4	38.8	100%	21.16%	4	38.8	100%	4.3%
HARM	34	60676	8.41%	21.39%	4	3655	8.6%	29.6%	38	54674	8.4%	22.25%
SMART	34	1069	100%	55.41%	4	47	100%	25.17%	38	943.5	100%	52.23%

signs purely from traces and validates them using simulations, which could miss corner cases. The low rate persists even with long traces: in our experiments, HARM uses traces with a length of 10,000 cycles, which is significantly longer than the traces used for the other tools.

Both GoldMine and SMART always generate verified assertions for benchmarks, leading to 100% VC rates, but GoldMine only supports a fraction of the benchmarks (the behavioral Verilog benchmarks).

b) MD-rate: The mutation detection results are summarized in Figure 4. Overall, SMART achieves a higher MD-rate than any related work on 32/38 benchmarks. In the 4 behavioral Verilog benchmarks supported by GoldMine, SMART produces comparable results, with both achieving an average MD-rate around 25% using a comparable number of assertions. In comparison to HARM, across the whole benchmark set, SMART achieves higher MD-rates on 33 of the benchmarks, with an average MD-rate 2.62x higher than HARM.

The MD-rates achieved by SMART vary significantly across benchmarks, reflecting the complexity and structural differences of the circuits. For small, medium, and large benchmarks, we achieve a strong assertion-based detection rate. However, in the case of extremely large circuits, timeouts limit the number of generated assertions, leading to a lower mutation detection rate.

Some benchmarks do not achieve a high mutation detection rate. For SMART, this may be because these benchmarks require assertions that reason about multiple time steps to differentiate between some mutations. However, it is worth noting that some mutations may not change the behavior of the underlying circuit, so a mutation detection rate of 100% may not be possible on all benchmarks.

D. Readability

The readability of specifications is subjective. Here, we discuss quantitative metrics that we use to approximate how readable the assertion sets are.

First, we look at the size of the assertion sets produced. The average assertion set size produced by SMART is $> 50\times$ smaller than the assertion sets produced by HARM. The assertion sets produced by GoldMine are of comparable size to SMART, on the limited supported benchmarks.

TABLE II: Examples of boolean rewrite rules used in for equivalence rewriting. The rewriting system consisted of 105 rules for bitvectors and 10 for booleans.

Pattern	Transformation
$a = \neg a$	$\perp$
$a = b$	$b = a$
$a \mathbin{\&} b$	$b \mathbin{\&} a$
$a \mathbin{\&} (b \mathbin{\&} c)$	$(a \mathbin{\&} b) \mathbin{\&} c$
$a \mathbin{\|\|} b$	$b \mathbin{\|\|} a$
$a \mathbin{\|\|} (b \mathbin{\|\|} c)$	$(a \mathbin{\|\|} b) \mathbin{\|\|} c$

Second, we implement an approximate equivalence checker to estimate the amount of redundancy in the assertion sets. We use an approximation of equivalence as a complete equivalence check of all assertions is intractable. Specifically, we use a non-destructive rewrite system based on equivalence graphs [45] to rewrite the set of assertions returned by SMART and check whether any assertions are equivalent. An example of the rewrite rules used is in Appendix II. We find that, in all cases, the rewrite system reduces the size of the assertion sets produced by SMART by less than 3%, suggesting SMART assertion sets contain relatively little redundancy.

E. Rewrite system

The rewriting system used to evaluate assertion redundancy was done using an equivalence graph (e-graph). E-graphs provide a constructive method to apply rewrite rules to capture semantically equivalent terms. We use `egg`[45] to generate an e-graph for the assertion sets of each benchmark. We then test whether equivalences exist in the assertion sets, using rewrite rules from CVC5 [6], Bitwuzla [33], and E-syn [9].

VIII. Evaluation Details

A. Behavioral vs. Structural Verilog

We observe that SMART achieves comparable verification correctness (VC) rates on both behavioral and structural Verilog benchmarks, but mutation detection (MD) rates are generally higher for structural designs. This difference can be attributed to three main factors:

1) **Control/Data Path Complexity.** Behavioral Verilog often encapsulates complex control and data path logic using higher-level constructs (e.g., `if`, `case`, `always @(*)`), making the synthesized assertions less precise in targeting low-level signal interactions. In contrast, structural Verilog exposes finer-grained gates and flip-flops, enabling SMART to more effectively synthesize assertions that tightly capture actual hardware behavior.

2) **Grammar and Variable Set Size.** Structural designs tend to contain more low-level signals (e.g., internal nets, intermediate registers), resulting in larger variable sets and more expressive grammars. While this increases synthesis cost, it also gives the synthesizer more flexibility to discover strong invariants. In behavioral designs, the variable space is often smaller and more abstract, limiting synthesis diversity.
3) **Signal Variability.** We found that structural Verilog exhibits more frequent value changes in simulation traces due to its gate-level representation, leading to richer positive example sets. This makes it easier for SMART to distinguish between correct and mutated designs. Behavioral designs, by contrast, often abstract away low-level toggling, which may result in sparser and less informative traces.

These differences explain why SMART achieves stronger mutation detection on structural designs (average MD-rate 55.4%) compared to behavioral ones (25.17%)

B. Effect of Counterexample Refinement

We conducted an ablation study to quantify the effect of counterexample-based refinement in SMART. Specifically, we disabled the counterexample refinement mechanism described in Section VI-C, and re-ran the synthesis process using the same set of benchmarks and initial parameters.

The results are summarized in Figure 5. Across all benchmarks, removing refinement led to a consistent drop in mutation detection (MD) rate. The average MD-rate dropped from 63.1% to 32.1%, a 49% relative reduction. In some cases (e.g., `arb2`, `c1355`, `s13207`), MD-rate dropped to zero, indicating that the initial assertion was vacuously true and not filtered out without refinement. Only a few small circuits (e.g., `s27`, `s349`) showed minimal sensitivity to refinement.

These results confirm that counterexample refinement plays a critical role in eliminating weak or vacuous assertions. Without it, SMART often converges on trivial assertions that pass simulation but fail to distinguish between correct and mutated designs.

C. Limitations

Currently, SMART generates assertions that reason about a single time step, and is unable to capture behaviors that evolve over multiple cycles. Extending its synthesis capability to temporal properties would be feasible, but would increase the size of the variable sets that the synthesis procedure needs to consider. Specifically, we would need to introduce a copy of the original variable set for each time step we wish to consider in the temporal property, so a property reasoning over two time steps would duplicate the variable set. This is why related work requires the user to provide restrictive templates, which limit the temporal operators used in temporal logic properties. SMART may be complementary to this by providing predicates that can be used within such templates.

The runtime of SMART is limited by verification time. One possible optimization for SMART would be to initially use testing to discard assertions that can easily be falsified with testing. However, ultimately, verification is a necessary part of the SMART pipeline in order to allow us to generate *correct* assertions.

IX. Related Work

Automated assertion generation is a critical area of research for improving hardware verification. Developing static program analysis for SVA generation is actively being studied. GoldMine [21] combines static analysis and decision-tree-based data mining for SVA generation, supporting boolean circuits.

GoldMine's inability to support structural Verilog stems from its reliance on semantic analysis tailored to behavioral Verilog. The analysis framework assumes high-level constructs and control flow information typically found in behavioral code. However, structural Verilog presents a flat netlist-like representation, where connectivity is expressed via gate and wire instances rather than procedural code. Supporting structural Verilog would therefore require fundamentally different semantic analysis techniques, such as netlist-level reasoning, instance graph traversal, and low-level dependency tracking—none of which are addressed in GoldMine. As a result, extending GoldMine to handle structural designs is non-trivial and would require substantial redesign of its core algorithms. By contrast, SMART supports both behavioral and structural Verilog inputs uniformly, without requiring hand-crafted templates or specific HDL constructs.

HARM [17] heavily leverages user-defined hints, which are particularly challenging to define for structural Verilog. The resulting properties are validated against simulation traces. The closest piece of work to ours is ARTmine [23], which focuses on temporal assertion generation by mining patterns. GoldMine was extended by Liu *et al.* [30] to support word-level properties, but still requires pre-defined templates. SMART is the only tool that supports bitvectors, does not require templates, and verifies the generated assertions.

There has also been interest in relying on machine learning models to automatically generate SVAs. Exploiting natural language processing models for automated SVA generation has been widely studied [20], [28], [46], [29], [15], [27], [35], [3]. There has also been interest in using large language models (LLMs) for SVA mining [26], [34], [42], [14]. However, all these approaches require natural language specifications as inputs, while SMART automatically produces SVAs from traces.

Adjacent research in temporal logic mining and reactive synthesis also highlights the contrast with SMART. UNDINE [12] mines security-critical temporal logic (LTL) properties from RTL designs using typed event templates. While it supports expressive temporal operators, it relies on predefined templates and extensive trace preprocessing. SMART avoids templates altogether and synthesizes bit-precise safety assertions for both behavioral and structural Verilog.

Oracle-Guided Inductive Synthesis has been used for deobfuscating low-level bit-code code [24]. Here, the user must provide a library of components from which the deobfuscation is constructed, and the low-level code must be small enough that it can be reasoned about by a satisfiability solver. The fact that many of our mutation tests time-out demonstrates that the Verilog files we are able to handle are beyond the scope of these symbolic encoding-driven techniques.

The closest application of oracle-guided inductive synthesis to ours is inductive invariant synthesis [16]. Specification mining is related to but distinct from inductive invariant synthesis, where the user provides a target property, and an algorithm is used to infer an invariant that can be used to construct a proof-by-induction to show the target property holds. Unlike SMART, without a target property, the approaches in the literature for inductive invariant synthesis would not be able to infer an invariant.

A synthesis framework integrating reactive synthesis with SyGuS has been proposed [10], using Temporal Stream Logic modulo theories (TSL-MT) to generate control/data-reactive programs. While both approaches use SyGuS, SMART focuses on mining state-based assertions from execution traces, rather than synthesizing reactive strategies. As such, their synthesis goals and application domains are fundamentally different.

X. Conclusions

We have presented SMART: a novel approach for hardware specification mining based on oracle-guided synthesis. Our evaluation demonstrates that specifications generated by SMART can differentiate more mutants on real-world hardware verification benchmarks from the literature, using fewer assertions than the state of the art.

References

[1] Ebmc. https://github.com/diffblue/hw-cbmc/

[2] Circuit netlist benchmarks. https://sportlab.usc.edu/~msabrishami/benchmarks.html (2025)

[3] Aditi, F., Hsiao, M.S.: Hybrid rule-based and machine learning system for assertion generation from natural language specifications. In: 2022 IEEE 31st Asian Test Symposium (ATS). pp. 126–131. IEEE (2022)

[4] Ammons, G., Bodík, R., Larus, J.R.: Mining specifications. In: POPL. pp. 4–16. ACM (2002)

[5] Ashenden, P.J.: The designer's guide to VHDL. Morgan kaufmann (2010)

[6] Barbosa, H., Barrett, C.W., Brain, M., Kremer, G., Lachnitt, H., Mann, M., Mohamed, A., Mohamed, M., Niemetz, A., Nötzli, A., Ozdemir, A., Preiner, M., Reynolds, A., Sheng, Y., Tinelli, C., Zohar, Y.: cvc5: A versatile and industrial-strength SMT solver. In: Fisman, D., Rosu, G. (eds.) Tools and Algorithms for the Construction and Analysis of Systems - 28th International Conference, TACAS 2022, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2022, Munich, Germany, April 2-7, 2022, Proceedings, Part I. Lecture Notes in Computer Science, vol. 13243, pp. 415–442. Springer (2022). https://doi.org/10.1007/978-3-030-99524-9_24, https://doi.org/10.1007/978-3-030-99524-9_24

[7] Brglez, F., Bryan, D., Kozminski, K.: Combinational profiles of sequential benchmark circuits. In: 1989 IEEE International Symposium on Circuits and Systems (ISCAS). pp. 1929–1934. IEEE (1989)

[8] Chakraborty, R.S., Bhunia, S.: Hardware protection and authentication through netlist level obfuscation. In: 2008 IEEE/ACM International Conference on Computer-Aided Design. pp. 674–677. IEEE (2008)

[9] Chen, C., Hu, G., Zuo, D., Yu, C., Ma, Y., Zhang, H.: E-syn: E-graph rewriting with technology-aware cost functions for logic synthesis. In: Proceedings of the 61st ACM/IEEE Design Automation Conference. pp. 1–6 (2024)

[10] Choi, W., Finkbeiner, B., Piskac, R., Santolucito, M.: Can reactive synthesis and syntax-guided synthesis be friends? In: Proceedings of the 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation. pp. 556–571 (2022)

[11] Clarke, E.M.: Model checking - my 27-year quest to overcome the state explosion problem. In: LPAR. Lecture Notes in Computer Science, vol. 5330, p. 182. Springer (2008)

[12] Deutschbein, C., Sturton, C.: Mining security critical linear temporal logic specifications for processors. In: 2018 19th International Workshop on Microprocessor and SOC Test, Security and Verification (MTV). pp. 13–18. IEEE (2018)

[13] Eisner, C., Fisman, D.: A practical introduction to PSL. Springer Science & Business Media (2007)

[14] Fang, W., Li, M., Li, M., Yan, Z., Liu, S., Xie, Z., Zhang, H.: Assertllm: Generating and evaluating hardware verification assertions from design specifications via multi-llms (2024), https://arxiv.org/abs/2402.00386

[15] Frederiksen, S.J., Aromando, J., Hsiao, M.S.: Automated assertion generation from natural language specifications. In: 2020 IEEE International Test Conference (ITC). pp. 1–5. IEEE (2020)

[16] Garg, P., Löding, C., Madhusudan, P., Neider, D.: ICE: A robust framework for learning invariants. In: CAV. Lecture Notes in Computer Science, vol. 8559, pp. 69–87. Springer (2014)

[17] Germiniani, S., Pravadelli, G.: Harm: a hint-based assertion miner. IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems **41**(11), 4277–4288 (2022)

[18] Godbole, A., Ye, L., Manerkar, Y.A., Seshia, S.A.: Modelling and verification of security-oriented resource partitioning schemes. In: FMCAD. pp. 268–273 (2023)

[19] Hansen, M.C., Yalcin, H., Hayes, J.P.: Unveiling the iscas-85 benchmarks: A case study in reverse engineering. IEEE Design & Test of Computers **16**(3), 72–80 (1999)

[20] Harris, C.B., Harris, I.G.: Glast: Learning formal grammars to translate natural language specifications into hardware assertions. In: 2016 Design, Automation & Test in Europe Conference & Exhibition (DATE). pp. 966–971. IEEE (2016)

[21] Hertz, S., Sheridan, D., Vasudevan, S.: Mining hardware assertions with guidance from static analysis. IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems **32**(6), 952–965 (2013)

[22] IEEE: Combining dynamic slicing and mutation operators for ESL correction (2012)

[23] Iman, M.R.H., Jervan, G., Ghasempouri, T.: Artmine: Automatic association rule mining with temporal behavior for hardware verification. In: 2024 Design, Automation & Test in Europe Conference & Exhibition (DATE). pp. 1–6. IEEE (2024)

[24] Jha, S., Gulwani, S., Seshia, S.A., Tiwari, A.: Oracle-guided component-based program synthesis. In: ICSE (1). pp. 215–224. ACM (2010)

[25] Jha, S., Seshia, S.A.: A theory of formal synthesis via inductive learning. Acta Informatica **54**(7), 693–726 (2017)

[26] Kande, R., Pearce, H., Tan, B., Dolan-Gavitt, B., Thakur, S., Karri, R., Rajendran, J.: Llm-assisted generation of hardware assertions. arXiv preprint arXiv:2306.14027 (2023)

[27] Keszocze, O., Harris, I.G.: Chatbot-based assertion generation from natural language specifications. In: 2019 Forum for Specification and Design Languages (FDL). pp. 1–6. IEEE (2019)

[28] Krishnamurthy, R., Hsiao, M.S.: Controlled natural language framework for generating assertions from hardware specifications. In: 2019 IEEE 13th International Conference on Semantic Computing (ICSC). pp. 367–370. IEEE (2019)

[29] Krishnamurthy, R., Hsiao, M.S.: Ease: Enabling hardware assertion synthesis from english. In: Rules and Reasoning: Third International Joint Conference, RuleML+ RR 2019, Bolzano, Italy, September 16–19, 2019, Proceedings 3. pp. 82–96. Springer (2019)

[30] Liu, L., Lin, C.H., Vasudevan, S.: Word level feature discovery to enhance quality of assertion mining. In: Proceedings of the International Conference on Computer-Aided Design. pp. 210–217 (2012)

[31] Lo, D., Khoo, S.: Mining patterns and rules for software specification discovery. Proc. VLDB Endow. **1**(2), 1609–1616 (2008)

[32] Neider, D., Roy, R.: What is formal verification without specifications? A survey on mining LTL specifications. In: Principles of Verification (3). Lecture Notes in Computer Science, vol. 15262, pp. 109–125. Springer (2024)

[33] Niemetz, A., Preiner, M.: Bitwuzla. In: International Conference on Computer Aided Verification. pp. 3–17. Springer (2023)

[34] Orenes-Vera, M., Martonosi, M., Wentzlaff, D.: Using llms to facilitate formal verification of rtl. arXiv e-prints pp. arXiv–2309 (2023)

[35] Parthasarathy, G., Nanda, S., Choudhary, P., Patil, P.: Spectosva: Circuit specification document to systemverilog assertion translation. In: 2021 Second Document Intelligence Workshop at KDD (2021)

[36] Raveendran, R., Bhuinya, S.: Customization of ibex risc-v processor core. Customization of Ibex RISC-V Processor Core (2021)

[37] Rosser, B.J.: Cocotb: a python-based digital logic verification framework. In: Micro-electronics Section seminar. CERN, Geneva, Switzerland (2018)

[38] Sheeran, M., Singh, S., Stålmarck, G.: Checking safety properties using induction and a sat-solver. In: International conference on formal methods in computer-aided design. pp. 127–144. Springer (2000)

[39] Shin, H.: Data-centric machine learning pipeline for hardware verification. In: 2022 IEEE 35th International System-on-Chip Conference (SOCC). pp. 1–2. IEEE (2022)

[40] Shin, H.: Efficient bug discovery with machine learning for hardware verification. https://community.arm.com/arm-research/b/articles/posts/efficient-bug-discovery-with-machine-learning-for-hardware-verification (2025)

[41] Solar-Lezama, A., Tancau, L., Bodík, R., Seshia, S.A., Saraswat, V.A.: Combinatorial sketching for finite programs. In: ASPLOS. pp. 404–415. ACM (2006)

[42] Sun, C., Hahn, C., Trippel, C.: Towards improving verification productivity with circuit-aware translation of natural language to systemverilog assertions. In: First International Workshop on Deep Learning-aided Verification (2023)

[43] Thomas, D., Moorby, P.: The Verilog® hardware description language. Springer Science & Business Media (2008)

[44] Vijayaraghavan, S., Ramanathan, M.: A practical guide for SystemVerilog assertions. Springer Science & Business Media (2005)

[45] Willsey, M., Nandi, C., Wang, Y.R., Flatt, O., Tatlock, Z., Panchekha, P.: Egg: Fast and extensible equality saturation. Proceedings of the ACM on Programming Languages **5**(POPL), 1–29 (2021)

[46] Zhao, J., Harris, I.G.: Automatic assertion generation from natural language specifications using subtree analysis. In: 2019 Design, Automation & Test in Europe Conference & Exhibition (DATE). pp. 598–601. IEEE (2019)