VeriEQ: Finding Verilog Simulators and Synthesizers Bugs with Equivalence Circuit Transformation

ZHEN YAN, Tsinghua University, China
YUANLIANG CHEN*, Tsinghua University, China
FUCHEN MA, Tsinghua University, China
ZEHONG YU, Tsinghua University, China
DALONG SHI, AVIC International Digital Network Technology Co., Ltd., China
YU JIANG*, Tsinghua University, China

Verilog simulators and synthesizers play a critical role in chip design and verification. However, due to the complexity of simulation and synthesis processes, they easily introduce various types of bugs. Among them, Behavioral Deviation Bugs (BDBs) are particularly severe, as they can cause incorrect results by introducing subtle semantic deviations that make the chip behave differently from its intended design, potentially enabling hardware backdoors.

In this work, we propose VeriEQ, an automated framework based on the idea of metamorphic testing, which detects BDBs by generating semantically equivalent Verilog programs. First, to increase the likelihood of triggering BDB, we analyze the structural patterns of historical BDB and design a Verilog code template. Second, we generate semantically equivalent variants by applying equivalence circuit transformation rules. These rules include constraints on bit-width and signedness to ensure logical consistency before and after the transformation. Finally, we design an inlined deviation checking mechanism that embeds multiple equivalent modules within a single testbench to improve testing efficiency. We implement and evaluate VeriEQ on four mainstream Verilog simulators and synthesizer. Experimental results show that VeriEQ achieves a 138.1% to 4161.9% speedup over state-of-the-art tools. In total, VeriEQ successfully detects 33 previously unknown bugs, including 29 BDBs, along with 4 hang bugs as additional findings. All discovered bugs have been confirmed, with 27 already fixed. In contrast, the other tools are able to detect only 1 to 7 bugs.

CCS Concepts: • **Software and its engineering → Software testing and debugging**.

Additional Key Words and Phrases: Testing, Verilog Simulators and Synthesizers, Equivalence Transformation

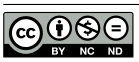
ACM Reference Format:
Zhen Yan, Yuanliang Chen, Fuchen Ma, Zehong Yu, Dalong Shi, and Yu Jiang. 2026. VeriEQ: Finding Verilog Simulators and Synthesizers Bugs with Equivalence Circuit Transformation. *Proc. ACM Program. Lang.* 10, OOPSLA1, Article 127 (April 2026), 29 pages. https://doi.org/10.1145/3798235

1 Introduction

In digital circuit design, Verilog synthesis tools are responsible for translating high-level behavioral Verilog descriptions into physically realizable circuit representations (e.g., gate-level netlists) [25, 42]. Verilog simulation tools are used in the early stages of design to verify the functional correctness

*Yuanliang Chen and Yu Jiang are the corresponding authors.

Authors' Contact Information: Zhen Yan, Tsinghua University, KLISS, BNRist, School of Software, China; Yuanliang Chen, Tsinghua University, KLISS, BNRist, School of Software, China; Fuchen Ma, Tsinghua University, KLISS, BNRist, School of Software, China; Zehong Yu, Tsinghua University, KLISS, BNRist, School of Software, China; Dalong Shi, AVIC International Digital Network Technology Co., Ltd., China; Yu Jiang, Tsinghua University, KLISS, BNRist, School of Software, China.



ACM 2475-1421/2026/4-ART127
https://doi.org/10.1145/3798235

of Verilog modules and ensure their behavior aligns with specifications [34, 38, 48, 51]. Working in tandem, they are widely employed throughout chip design, verification, and automated testing workflows, forming a critical part of modern electronic design automation (EDA) infrastructure [3].

However, both the synthesis process of Verilog synthesizers and the simulation process of Verilog simulators are highly complex [44]. They involve many stages, including lexical analysis, syntax parsing, intermediate representation construction, behavioral modeling, bit-width and signedness inference, and logic optimization [40]. Due to this complexity, Verilog simulators and synthesizers are prone to various bugs. One common type of bug is hang, where the synthesizer or simulator becomes unresponsive or enters an infinite loop while elaborating or optimizing Verilog code, causing delays in the hardware design workflow. Another type of bug causes the synthesizer or simulator to produce results that deviate behaviorally from the intended design during synthesis or simulation, leading to incorrect results. We refer to this kind of bug as a **B**ehavioral **D**eviation **B**ug (**BDB**). Compared to hang bugs, BDBs are more dangerous because they do not produce explicit error messages or runtime failures. Instead, they may silently introduce small but critical semantic deviations into the final simulation or synthesis results. Specifically, simulator BDBs manifest as inconsistencies between the simulator's runtime output and the expected behavior, whereas synthesizer BDBs manifest as behavioral deviations between the synthesized design and its original pre-synthesis version. Previous research has shown that some tools can exploit inconsistent simulation and synthesis results across Verilog simulators and synthesizers to successfully insert backdoors into CPU modules, without being detected by existing verification methods [49]. If these bugs are not detected during the design phase and proceed to tape-out, they can cause the chip's behavior to differ significantly from the intended design. In some cases, attackers may exploit these bugs to insert hardware backdoors, leading to severe economic losses and security risks [43].

To uncover BDBs in Verilog simulators and synthesizers, recent research has explored differential testing by randomly generating Verilog programs and executing them across multiple tools, then comparing their simulation or synthesis outputs to identify inconsistencies [23, 43, 49]. This approach has shown promising results. However, subtle differences in standard compliance among Verilog simulators and synthesizers, such as how unknown values are propagated or how event scheduling is handled, can cause differential testing across simulators and synthesizers to miss potential bugs [53, 55]. Moreover, existing tools like TransFuzz [49] and Verismith [23] fail to account for the syntactic characteristics of BDBs; they rely on random generation without considering critical syntactic constructs that are likely to trigger BDBs. For instance, constructs like the `case` statement, which is prevalent in state machine modeling [20], are overlooked in such tests. This situation underscores the urgent need for a new approach to complement differential testing.

To comprehensively detect BDBs in Verilog simulators and synthesizers, an intuitive strategy is to generate a set of semantically equivalent but syntactically diverse Verilog programs and perform metamorphic testing on a single simulator or synthesizer. Comparing the output behaviors of these equivalent programs can expose semantic deviations introduced during the simulation and synthesis process. However, applying metamorphic testing [2, 4, 13, 21, 65] to Verilog simulators and synthesizers presents three key challenges. **The first challenge** lies in generating structurally diverse initial Verilog programs. The effectiveness of semantic-equivalence transformations heavily depends on the structural characteristics of the initial Verilog code. If the generated code lacks the syntactic structures necessary to trigger the relevant simulator or synthesizer logic, it becomes difficult to expose potential BDBs. **The second challenge** is the difficulty of defining Verilog semantic equivalence rules. In Verilog, every signal has a discrete bit-width and signedness attribute. During expression evaluation, signals may undergo sign extension, zero extension, or truncation depending on the context, and the signedness of an expression may implicitly change based on that of the context operands. If these details are ignored and arithmetic identities such as `(a + 0 = a)`

or control-flow equivalences like transforming `if` statements into `case` statements are applied blindly, they may inadvertently change bit-width or signedness, causing behavioral mismatches and breaking semantic equivalence. **The third challenge** is how to efficiently detect BDBs. To detect BDBs, it is typically necessary to simulate or synthesize multiple semantically equivalent programs separately, and then compare their outputs one by one. This pairwise comparison is acceptable when the number of variants is small, but as the transformation scale increases, the testing cost grows linearly. In practice, to improve the likelihood of exposing bugs, multiple equivalent programs are often compared simultaneously, resulting in significant testing overhead.

To address these challenges, we propose VeriEQ, a Verilog simulator and synthesizer testing framework based on semantic-equivalence transformations, designed to effectively uncover BDBs. To address the first challenge, we conducted an in-depth analysis of existing BDB cases and systematically extracted code features highly correlated with bug triggering. These features include the use of `always` blocks, hierarchical statement structures, and specific syntactic combinations such as the signedness of signals in expressions and bit-select patterns. Based on this analysis, we designed a Verilog program generation template. This template preserves the flexibility of behavioral-level design while incorporating common triggering patterns, providing a more test-effective foundation for subsequent semantic transformations. To address the second challenge, we introduce semantic transformation rules tailored to Verilog, referred to as Equivalence Circuit Transformation Rules. During code generation, VeriEQ dynamically infers the bit-width and signedness of all signals. It then selects and applies transformation rules only when semantic equivalence constraints can be guaranteed. This ensures that the transformed code behaves exactly the same as the original. To address the third challenge, we introduce an inline deviation checking mechanism. Leveraging the semantic equivalence among all metamorphic variants, we instantiate multiple equivalent modules within the same testbench and drive them with shared stimuli. Their outputs are then compared inline within the testbench. This approach avoids the redundant overhead of simulating each variant separately, improving the efficiency of BDB detection.

We implemented and evaluated VeriEQ on three open-source simulators, Verilator [48], CXXRTL [58], and Icarus Verilog [61], as well as one open-source synthesis tool Yosys [62]. VeriEQ found a total of 33 bugs across these four Verilog simulators and synthesizers: 13 in Verilator, 2 in CXXRTL, 2 in Icarus Verilog, and 16 in Yosys. Among these, 29 bugs were BDBs. All 33 bugs were confirmed by the tool developers, and 27 of them have already been fixed. In comparison, Transfuzz was only able to find 7 bugs, while Verismith found none.

In summary, we make three key contributions:

- We propose a Verilog program generation template, semantic transformation rules, and an inline deviation checking mechanism, which together enable effective testing of BDBs.
- We implement these methods in VeriEQ, and apply it to three open-source simulators and one synthesis tool.
- Using VeriEQ, we discovered 33 previously unknown bugs. All of them are confirmed by developers, and 27 have been fixed. We open-source the VeriEQ [1] for broader practical use.

2 Background

2.1 Verilog Simulators and Synthesizers

Verilog simulators and synthesizers are critical tools in modern digital circuit design workflows. In practice, both simulators and synthesizers begin by performing front-end processing on Verilog code, which involves lexical and syntactic analysis, abstract syntax tree construction, and intermediate representation generation. To improve execution efficiency and reduce computational overhead,

[1] VeriEQ at: https://anonymous.4open.science/r/VeriEQ-5B2F

both types of tools apply a range of optimization techniques, such as constant folding, mux tree optimization, and register elimination and transformation [12, 24, 57, 69].

Although their front-end processing pipelines are similar, simulation and synthesis serve different purposes. The goal of a simulator is to build an executable model that accurately mimics the hardware behavior described by the Verilog code [56]. This model is then run under test stimuli to drive the circuit, capture outputs, and verify functional correctness [5, 54]. These test stimuli act as inputs to the program and typically involve module instantiation, input signal assignment, and output signal collection. In contrast, the goal of a synthesizer is to translate the design into an efficient gate-level implementation [6, 22, 52]. It achieves this by applying logic transformation, Boolean simplification, and technology mapping to generate the final netlist. Existing works [23, 78] often use formal verification to check whether the behavioral-level Verilog code before synthesis is functionally equivalent to the gate-level netlist after synthesis, in order to detect potential bugs. However, in practical testing, one can also compare the simulation outputs of the pre- and post-synthesis designs under the same input stimuli to detect bugs. This approach allows the testing of both synthesizers and simulators to be unified.

2.2 Metamorphic Testing

Metamorphic Testing (MT) is a widely adopted technique in scenarios where reliable test oracles are unavailable [8, 9, 33, 37, 46, 47, 50]. Instead of relying on the correctness of individual outputs, MT defines a set of metamorphic relations (MRs), which are expected logical relationships between multiple inputs and their corresponding outputs, to indirectly identify program errors. For instance, if a function $f(x)$ satisfies the identity $f(x) = f(x + 0)$, then the outputs of $f(10)$ and $f(10 + 0)$ should be identical; any discrepancy implies the presence of a potential bug.

```
1 module top;
2   input wire signed [2:0] e;
3   input wire [7:0] a, b;
4   output wire [7:0] c, d;
5   //a = b = 8'hFF -> c = 8'h00
6   assign c = (a + b) >> 1;
7   //e = 3'h7 -> d = 8'hFF
8   assign d = e;
9 endmodule
```

(a) Original Code

```
1 module top;
2   input wire signed [2:0] e;
3   input wire [7:0] a, b;
4   output wire [7:0] c, d;
5   //a = b = 8'hFF -> c = 8'h80
6   assign c = (a + b + 0) >> 1;
7   //e = 3'h7 -> d = 8'h07
8   assign d = e + 3'd0;
9 endmodule
```

(b) Transformed Code

Fig. 1. Code snippet showing how ignoring bit-width and signedness leads to an incorrect equivalence transformation in Verilog.

However, applying MT to Verilog simulators and synthesizers validation presents unique challenges, primarily due to Verilog's highly precise semantics regarding bit-width and signedness [1, 29]. As illustrated in Figure 1, some common arithmetic identities do not hold in Verilog. For example, the transformation of the expression on line 6 using the identity `(a + 0 = a)` results in a semantically different expression in Verilog. This is because both `a` and `b` are 8 bits wide, while the default width of `0` is 32 bits. As a result, the right-hand side expression causes `a` and `b` to be zero-extended to 32 bits, which may lead to mismatches in simulation results under certain input stimuli. The transformation on line 8 avoids zero-extension by preserving bit-widths, but still suffers from incorrect signedness handling. The signal `e` is signed, but adding the unsigned constant `3'd0` causes it to be cast to an unsigned value during computation, leading to incorrect results. Therefore, to apply MT correctly in Verilog simulators and synthesizers validation, it is essential

to incorporate bit-width- and signedness-sensitive transformation rules. This requires explicitly tracking bit-width and signedness in the transformation context to ensure semantic consistency before and after transformation, and to avoid false positives caused by subtle semantic deviations.

3 Motivating Example

Some BDBs hidden in Verilog simulators and synthesizers are difficult to detect yet can result in severe consequences. A real-world example is a BDB found in Verilator, a widely used Verilog simulator [45], which causes incorrect simulation results. Figure 2 shows the Verilog code snippet that triggers the BDB, while Figure 3 presents the core code responsible for this bug.

```
module top(
    input wire clock_0,
    input wire clock_1,
    input wire in1,
    output wire [7:0] out24
);
    reg [7:0] reg_10;
    initial begin
        reg_10 = 0;
    end
    // expected to get 8'd3, but get 8'd1
    assign out24 = reg_10;
    always @(negedge clock_1 or posedge clock_0) begin
        if (clock_0) begin
            reg_10 <= 0;
        end else begin
            // equal to reg_10 <= {1'b1, 1'b1}
            reg_10 <= {1'b1, ~((in1 & ~(in1)))};
        end
    end
endmodule
```

Fig. 2. A BDB in Verilator caused incorrect simulation results, posing risks to subsequent chip design.

This bug is caused by an incorrect optimization of the expression `~(in1 & ~(in1))`. Verilator first elaborates and optimizes the Verilog code and testbench into an executable simulation model, taking nearly 10 seconds. During elaboration, Verilator performs constant propagation by analyzing the bitwise operation tree. For example, when it encounters an AstAnd node like `(a & ~a)`, it applies the identity `(a & ~a) == 0` to fold the expression into a constant zero. Because `in1` has only one bit, Verilator also sets the width of the result to one bit. However, when Verilator performs the bitwise NOT operation on the optimized constant `1'b0`, it mistakenly applies zero-extension based on the width of the entire expression. As a result, in the expression `{1'b1, 1'b1}` (where the second `1'b1` comes from the negation), the lower bits overwrite the upper bits. This produces a wrong result of `1` instead of the expected value `3`, which breaks the circuit's functional logic. This bug was finally fixed by keeping the width of the optimized constant limited to its minimal required value during substitution.

Verilog simulators and synthesizers are widely used in chip design, verification, and automated testing workflows. However, their vulnerabilities may silently introduce functional defects through incorrect simulation or synthesis results, potentially compromising the correctness and reliability of the entire hardware system. We can draw **three important lessons** from this case: (1) Structural features play a critical role in triggering BDBs. This bug relies on specific patterns such as `always`

```
void replaceConst(AstNodeUniop* nodep) {
--    V3Number num{nodep, nodep->width()};
--    nodep->numberOperate(num, VN_AS(nodep->lhsp(), Const)->num());
++    V3Number numv{nodep, nodep->widthMinV()};
++    nodep->numberOperate(numv, constNumV(nodep->lhsp()));
++    const V3Number& num = toNumC(nodep, numv);
    UINFO(4, "UNICONST -> " << num << endl);
    VL_DO_DANGLING(replaceNum(nodep, num), nodep);
}
```

Fig. 3. The core code snippet of the example. It used incorrect bit-width information when replacing an optimized constant during a unary negation operation.

blocks, concatenation expressions, and mismatched widths. Existing work such as TransFuzz randomly generates netlist-level code interconnected through various operators but overlooks these structural patterns. If the randomly generated Verilog code lacks such patterns, it may fail to trigger similar issues even when the interconnection topology is diverse. To address this, VeriEQ introduces Verilog code generation templates derived from an empirical study of BDBs. These templates serve as a foundation for more effective transformations. (2) Semantic equivalence must account for precise bit-width and signedness semantics. The Verilator bug stems from overlooking the semantic change introduced by bit-width truncation during bitwise negation. When applying semantic-preserving transformations, neglecting such subtleties (e.g., `a & ~a = 0`) may easily break equivalence, posing a distinctive challenge for VeriEQ. For instance, in this case, replacing `~(in1 & ~in1)` with `1` (implicitly 32 bits) or `32'sd1` (signed) leads to a mismatch with the expected result `1'b1`, due to `in1` being 1-bit wide. To solve this, VeriEQ proposes equivalence circuit transformation rules, which preserve semantics by incorporating width and signedness information during transformations. (3) The cost of detection increases rapidly with the number of equivalence variants. Elaborating and simulating the code in this example takes nearly 10 seconds. Existing works also suffer from high verification overhead. For instance, TransFuzz performs differential testing across multiple simulators, and its overall speed is limited by the slowest backend. Verismith and VeriXmith rely on formal verification, which further slows down their testing process. In VeriEQ, if each transformed variant were elaborated, simulated, and compared separately, the total cost would increase linearly. To tackle this, VeriEQ employs an inline deviation checking mechanism, in which multiple semantically equivalent modules are instantiated in the same testbench. Their outputs are compared inline under shared stimuli to improve BDB detection efficiency.

4 Overview of Behavioral Deviation Bug

To effectively trigger BDBs, it is essential to understand the structural characteristics of Verilog code that tend to expose such bugs. To this end, we conducted an empirical study of historical BDB reports across major Verilog simulators and synthesizers, which guided the construction of structurally diverse initial programs for later equivalence transformations. In the following, we present our study methodology and findings.

4.1 Study Methodology

We conducted a rigorous empirical analysis of BDB using the open coding method. Our study consisted of the following steps:

Step 1. Data Collection: We collected BDBs from four widely used Verilog synthesis and simulation platforms, namely Yosys, Verilator, CXXRTL, and Icarus Verilog. Specifically, we first selected relevant issues from the issue trackers of these tools and filtered those whose titles or contents contained the keywords "incorrect result," "wrong runtime result," or "inconsistent". We then further narrowed the scope by retaining only those issues that had an assignee or were labeled with tags such as "bug" or "fix-pending". Next, we extracted key information from the description of each candidate issue to determine whether it indeed belonged to BDBs, in particular, whether it could lead to incorrect results after synthesis or simulation. In total, we confirmed 54 BDBs, including 32 in Verilator, 12 in Yosys, 7 in Icarus Verilog, and 3 in CXXRTL. Notably, due to the well-structured issue templates provided by developers, each case included information about the triggering conditions as well as its specific symptoms. This stage of the study enabled us to clearly characterize the fundamental properties of each BDB, laying a solid foundation for subsequent in-depth analysis.

Step 2. Initial Coding: We conducted an in-depth study of the issues corresponding to each collected BDB, gathering more detailed information. For BDBs that were confirmed but not fixed, we carefully examined the discussions between developers and issue reporters, as well as the attack vectors provided by the reporters and the abnormal behaviors they caused in the synthesizer or simulator. For fixed BDB cases, we further reviewed the patches developers submitted to resolve the bugs, in order to better summarize the root causes that triggered each BDB. For BDBs lacking sufficient information, we directly analyzed the source code and reproduced the attack vectors provided by the issue reporters to investigate their root causes and potential consequences.

Step 3. Categorization and Hypothesis Formation: We systematically categorized the BDBs based on the Verilog code characteristics that led to them. This process enabled us to identify patterns and commonalities across different BDBs, thereby laying the foundation for generating high-quality Verilog programs that capture these shared features and serve as initial seeds for equivalence transformations. Through statistical analysis, we further distilled three predominant code characteristics, which guide subsequent BDB detection.

4.2 Findings

To generate higher-quality initial Verilog programs as seeds for subsequent equivalence transformations, and thereby more effectively uncover BDBs, we distill the following three findings to guide the generation of initial Verilog code.

Finding 1. Small and shallow Verilog code snippets are sufficient to trigger BDBs: Among the 54 identified BDBs, we found that all can be triggered by shallow signal propagation chains, typically with a depth of no more than 6 levels. We assume that when a signal variable appears as an operand in an expression, it represents one propagation of the signal in the circuit diagram. Additionally, among the 10/54 BDBs involving sequential `always` blocks, 8 only require a single sequential `always` behavioral block, and the remaining 2 can be manually simplified to cases involving just one sequential `always` block as well. This finding suggests that, when generating Verilog code for BDB detection, we can limit the depth of signal propagation to favor a wide rather than deep circuit structure and restrict the design to a single sequential `always` block. Such a design improves debugging efficiency and makes the minimized code easier to analyze. Moreover, using only a single sequential `always` block can help reduce simulation overhead.

Finding 2. Some BDBs only manifest when signals are initialized to x: 6/54 BDBs can only be triggered when certain input signals take the value `x`, which in Verilog represents an unknown logic value. Although BDBs that require signals to take the `x` value are relatively limited in number among the 54 cases, they often lead to severe consequences. For synthesizers, for instance, Yosys bug #5014 [10] causes Yosys to incorrectly remove the entire module design when synthesizing

a circuit where only a single signal is left as x. For simulators, Icarus Verilog bug #1074 [15] produces incorrect simulation results when subtracting two signals whose bits are partially x-initialized, which could potentially be exploited by attackers to inject backdoors into hardware designs. In our empirical study of the 54 historical BDBs, we do not observe any cases related to z values, and thus z-value handling is not a focus of our work. However, existing work such as Verismith tends to overlook or underestimate the significance of x-valued inputs. In contrast, TransFuzz relies on differential testing across multiple simulators. Since different simulators adopt inconsistent strategies for handling x values, TransFuzz faces inherent difficulties in conducting reliable differential analysis. Consequently, both approaches fail to detect bugs that manifest only under these conditions. This finding highlights the need to not only generate Verilog code, but also to carefully design the corresponding input stimuli to include scenarios where signals are initialized with x. Doing so enables effective detection of BDBs that depend on x-value propagation.

Finding 3. BDBs are closely tied to some syntactic structures: Among the 54 BDBs, 15 involve bit-width mismatches during signal assignment. For example, Icarus Verilog bug #1099 [75] was caused by a mismatch between port and signal widths, which led to their concatenation being incorrectly optimized away. Another 13 involve shift operations. For instance, CXXRTL bug #3820 [70] occurred when the result of a left-shift operation exceeded the valid range without proper truncation, leading to incorrect simulation. 5 cases are related to the signedness of signals. For example, Icarus Verilog bug #1165 [26] resulted from treating the outcome of a partial signal selection as signed, which produced incorrect simulation results. In addition, 2 BDBs involve the use of `case` statements. For example, Yosys bug #4317 [31] was triggered by incorrect handling of the `default` branch, which led to wrong synthesis verification results. This construct has been overlooked in some prior works [23, 49]. Although VeriXmith [78] considers such structures, it failed to uncover this defect due to limitations in the effectiveness of its mutation algorithms and testing efficiency. In addition, 8 BDBs involve various forms of combinational logic, such as NOT, OR, and AND. These observations suggest that the initial program generation process should increase the occurrence probability of these syntax structures to better cover semantic boundaries where simulators and synthesizers are prone to errors. In particular, operators that are more likely to trigger BDBs should be assigned higher generation weights. Moreover, the design of equivalence transformation rules should also place particular emphasis on these syntactic structures that frequently lead to BDBs.

5 Design

5.1 VeriEQ Workflow

Figure 4 illustrates the workflow of VeriEQ, which consists of the following 7 steps: (1) We begin by analyzing historical BDB cases, summarizing common code characteristics and constructing a Verilog template. Based on the Extracted BDB Patterns in the template, VeriEQ generates an initial Verilog program that is more likely to trigger BDB. During generation, it tracks the bit-width and signedness of all variables in each expression. (2) Using the Test Stimulus Patterns defined in the template, VeriEQ generates appropriate stimulus inputs for each input signal in the Verilog design. (3) It then analyzes the syntactic structure of the initial Verilog code to select applicable equivalence transformation rules. (4) The selected equivalence rules are applied to transform the initial Verilog program, generating multiple semantically equivalent variants. (5) VeriEQ constructs an inlined deviation testbench that embeds all code variants and test stimuli into a single module, allowing their outputs to be compared simultaneously during simulation. (6) The generated testbench is sent to the target Verilog synthesizer or simulator. When targeting synthesizers, VeriEQ performs indirect testing by comparing simulation results before and after synthesis optimizations. For

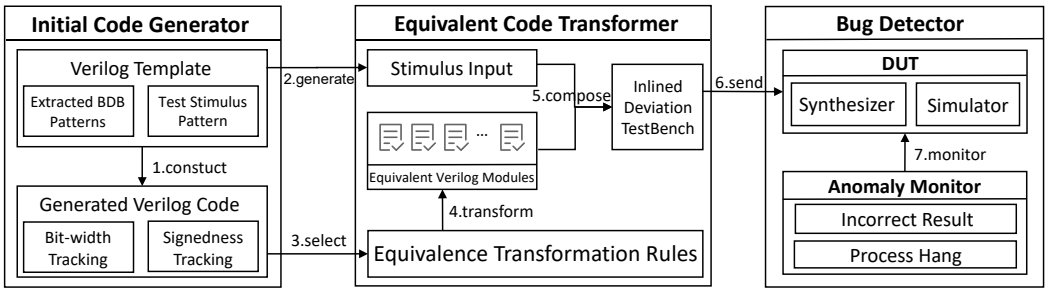


Fig. 4. The workflow of VeriEQ. It contains three main components: (1) Initial Code Generator for generating initial Verilog programs; (2) Equivalent Code Transformer for applying equivalence-preserving transformations to produce semantically equivalent variants; (3) Bug Detector for identifying potential bugs.

simulators, direct elaboration and simulation are performed. (7) Finally, the Anomaly Monitor monitors the entire testing process and outcomes. It identifies any behavioral mismatches or hangs and generates detailed error reports for further analysis.

5.2 Verilog Template Definition and Generation

Based on our findings of how BDBs are typically triggered, we design a Verilog generation template consisting of two main stages: generating an initial Verilog module based on BDB-triggering patterns, and generating input stimuli for simulation.

$$
\begin{aligned}
&\langle Module\rangle ::= \textbf{module}\ \langle CombBlock\rangle\ \langle SeqBlock\rangle\ \langle OutExpr\rangle\ \textbf{endmodule}\\
&\langle CombBlock\rangle ::= \langle Expr\rangle^{+}\\
&\langle SeqBlock\rangle ::= \textbf{always @(}\ \langle ClockStmt\rangle)\ \langle SeqExpr\rangle^{+}\ \textbf{) end}\\
&\langle Expr\rangle ::= \textbf{Operator(}\langle Expr\rangle^{+}) \mid \langle Signal\rangle \mid \langle Constant\rangle\\
&\langle SeqExpr\rangle ::= \langle ifStmt\rangle \mid \langle caseStmt\rangle \mid \langle Expr\rangle\\
&\langle ClockStmt\rangle ::= (\textbf{posedge} \mid \textbf{negedge})\ \langle Signal\rangle\\
&\langle ifStmt\rangle ::= \textbf{if}(\langle Expr\rangle)\ \langle SeqExpr\rangle\\
&\qquad\qquad \{\textbf{else if}(\langle Expr\rangle)\ \langle SeqExpr\rangle\}\\
&\qquad\qquad [\textbf{else}\ \langle SeqExpr\rangle]\\
&\langle caseStmt\rangle ::= \textbf{case (}\langle Expr\rangle\textbf{)}\ \{\langle caseItem\rangle\}\ \textbf{endcase}\\
&\langle caseItem\rangle ::= \langle Expr\rangle : \langle SeqExpr\rangle^{+}\\
&\langle OutExpr\rangle ::= \textbf{Concentrate}(\langle Signal\rangle^{+})
\end{aligned}
$$

Fig. 5. Syntax of generated Verilog modules. ⟨Module⟩ represents the entire Verilog module, which consists of combinational logic blocks ⟨CombBlock⟩ containing only combinational circuits, sequential logic blocks ⟨SeqBlock⟩ involving sequential circuits, and a final ⟨OutExpr⟩ that concatenates all outputs. **Operator()** denotes all operators used in combinational logic, such as "+", "«", "||", and "&" for arithmetic and Boolean operations, while **Concentrate()** represents the operation of aggregating all output signals using addition, where output signals refer to those not used as operands in any expression. ⟨Token⟩$^{+}$ indicates that the symbol ⟨Token⟩ may appear repeatedly.

Figure 5 illustrates the syntactic constructs present in the Verilog code generated by VeriEQ. The initial code includes input signal declarations, a combinational logic block, and a sequential *always* block (Finding 1), together forming a complete Verilog module. The combinational block contains

a variety of arithmetic and Boolean expressions *<Expr>*, with shift operations appearing more frequently to increase the likelihood of triggering BDBs (Finding 3). Each *<Expr>* may be a signal *<Signal>*, a constant *<Constant>*, or an operation applied to multiple sub-expressions. The sequential block *<SeqBlock>* incorporates behavioral constructs such as *if-else* and *case* statements, which are designed to expose BDBs related to high-level syntax (Finding 3). The triggering condition of *<SeqBlock>* is defined by *<ClockStmt>*. During our empirical study, we observed that BDBs are independent of clock signal selection. Therefore, for simplicity, we use a single external input signal as the clock.

`<Signal>` denotes a signal variable in Verilog, which can be either a register (`reg`) or a net type (`wire`). For simplicity, we abbreviate `<Signal>` as s in the following discussion. Each signal in a module is associated with five attributes: the declared bit-width $B_{\text{decl}}(s)$, declared signedness $S_{\text{decl}}(s)$, effective bit-width $B_{\text{eff}}(s)$, effective signedness $S_{\text{eff}}(s)$, and depth $D(s)$. The depth of a signal $D(s)$ is defined as the number of intermediate signals between it and the module's inputs. All input signals are assigned a depth of zero. The declared bit-width $B_{\text{decl}}(s)$ and declared signedness $S_{\text{decl}}(s)$ represent the bit-width and signedness explicitly specified when the signal is declared. In contrast, the effective bit-width $B_{\text{eff}}(s)$ and effective signedness $S_{\text{eff}}(s)$ reflect the bit-width and signedness that actually take effect when the signal s is used in a particular expression context. These effective attributes must be inferred according to context-sensitive rules and may vary across different expressions. Initially, all four attributes, namely $B_{\text{decl}}(s)$, $S_{\text{decl}}(s)$, $B_{\text{eff}}(s)$, and $S_{\text{eff}}(s)$, are unspecified when the signal is first created and are later determined through Algorithm 1.

Algorithm 1 Initial Verilog Module Generation

Input: N_{in}: Number of input signals; *target_size*: Target number of signals
Output: M: Verilog module

1: $S \leftarrow \{s_1, \ldots, s_{N_{\text{in}}}\}$
2: M.`setInput`(S)
3: M.`GenerateSeqBlock`(S)
4: **while** $|S| <$ `target_size` **do**
5: $E, s_{new} \leftarrow$ `GenerateExpr`(S)
6: s_{new}.`assign`(E)
7: $D(s_{\text{new}}) \leftarrow 1 + \max_{s \in E}(D(s))$
8: $S \leftarrow S \cup \{s\}$
9: M.`addCombOrSeq`(E, s_{new})
10: **end while**
11: **for** $s \in$ `SortByDepth`(S) **do**
12: `InferAttr`(s)
13: **end for**
14: $O \leftarrow$ `MergeUnused`(S)
15: M.`setOutput`(O)
16: **return** M

Algorithm 1 describes how the Verilog template generates the initial Verilog code following the syntax defined in Figure 5. The template maintains a pool of defined signals, which initially contains only the input signals (lines 1–2). It then constructs a sequential block following the *<SeqBlock>* grammar in Figure 5, selecting one signal from the pool to serve as the clock signal. The behavioral statements inside the block are initialized with placeholders, which will be filled in later once expressions are generated (line 3). In the expression generation phase (lines 4–10),

a subset of signals is randomly selected from the pool to form a new expression, with selection biased toward shallower signals. Specifically, the probability of selecting a signal s is weighted by the formula $1/(1 + D(s))$. This strategy helps limit the overall expression depth and localizes potential bugs (Finding 1). When forming a new expression, a signal can either appear as a variable or be expanded into its corresponding expression tree. A new signal is then created and assigned the generated expression, with its depth set to one plus the maximum depth among its operands. This expression is randomly inserted into either the combinational or the sequential block. For sequential blocks, insertion corresponds to replacing the previously generated placeholder. The newly defined signal is then added to the pool for subsequent use. After expression generation completes, the template performs attribute inference starting from the deepest signal (lines 11–13). It recursively traverses the sub-expressions of each signal to assign actual bit-width and signedness attributes. Specifically, for the deepest signals, we randomly initialize their bit-width and signedness. We then traverse their connected sub-signals, recording these properties for each one. Next, the signals are visited in descending order of depth. For each signal, if its bit-width has not yet been recorded, it is initialized with a new width; its signedness is chosen to balance the number of signed and unsigned sub-signals. These assigned attributes are propagated to the signal's sub-signals and recorded accordingly, continuing this process until the $B_{\text{decl}}(s)$ and $S_{\text{decl}}(s)$ have been initialized for all signals. This process ensures sufficient diversity in bit-widths and a balanced distribution of signed and unsigned types among signals within each expression (Finding 3). Finally, in lines 14–15, the template collects all signals that are not used on the right-hand side of any assignment and merges them via addition to construct the module's output signal.

In the stimulus generation stage, the template randomly selects a subset of input signals that are left unassigned during simulation, so their initial values remain `x`, enabling the detection of BDBs related to undefined inputs (Finding 2). For the remaining inputs, multiple rounds of simulation stimuli are generated for covering a wide range of sequential behaviors. In each round, every signal is assigned a random value within the range allowed by its bit-width. In practice, most bugs are triggered within a small number of stimulus cycles, and using more cycles brings little additional benefit while increasing simulation cost. Therefore, each stimulus round is limited to 20 cycles to balance bug coverage and simulation cost.

5.3 Equivalence Circuit Transformation Rules

In this step, VeriEQ applies a set of equivalence circuit transformation rules to the initial Verilog program to generate a collection of semantically equivalent but syntactically diverse code variants. We generate 10 variants by default, and this number is configurable. Applying equivalence rules to Verilog code is non-trivial. Bit-width and signedness properties of signals can significantly affect the semantics of expressions, potentially leading to incorrect behavior if ignored. Specifically, during evaluation, a Verilog expression computes its effective bit-width B_{eff} and effective signedness S_{eff} in a bottom-up manner, based on the declared attributes $B_{\text{decl}}(s)$ and $S_{\text{decl}}(s)$ of its sub-expressions (with the lowest-level sub-expressions being signals). The resulting B_{eff} and S_{eff} are then propagated to each signal s involved in the expression. Therefore, the validity of a constraint-based equivalence rule depends on the signals' effective attributes $B_{\text{eff}}(s)$ and $S_{\text{eff}}(s)$, rather than their declared attributes $B_{\text{decl}}(s)$ and $S_{\text{decl}}(s)$. Therefore, VeriEQ first infers $B_{\text{eff}}(s)$ and $S_{\text{eff}}(s)$ of each expression in the code before applying any transformation. We first describe this inference process, followed by an explanation of how the transformation rules are applied.

Bit-width and Signedness Inference. VeriEQ begins by inferring the signedness of each expression recursively, following the rules defined in IEEE Std 1364-2005 [1]. Specifically, if any subexpression is unsigned, the entire expression is treated as unsigned; only when all subexpressions are signed is the parent expression considered signed. Note that constants and signals are also

treated as expressions in this context. The effective signedness of a compound expression E is then recursively inferred from its immediate subexpressions as follows:

$$S_{\text{eff}}(E) = \begin{cases} \texttt{signed} & \text{if } \forall E_i \in \text{Sub}(E),\ S_{\text{eff}}(E_i) = \texttt{signed} \\ \texttt{unsigned} & \text{otherwise} \end{cases}$$

Next, based on the same standard, VeriEQ infers the effective bit-width $B_{\text{eff}}(E)$ of each expression recursively over the expression tree. Since Verilog supports a wide variety of expression types, including arithmetic, bitwise, logical, relational, and shift operators, the inference rules vary accordingly. As an illustrative example, for a binary arithmetic operation such as addition, the resulting bit-width is always determined by the maximum bit-width of the two operands, as specified in IEEE Std 1364-2005 [1]:

$$B_{\text{eff}}(E_1 + E_2) = \max\left(B_{\text{eff}}(E_1),\ B_{\text{eff}}(E_2)\right)$$

This is contrary to the intuitive assumption that, when the two operands have equal bit-widths, the result should have a bit-width increased by one. Once the effective bit-width and signedness of an expression are determined, these properties are propagated downward to its subexpressions, meaning that the top-level expression's $B_{\text{eff}}(E)$ and $S_{\text{eff}}(E)$ are assigned to all of its subexpressions as their context-driven attributes.

Equivalence Transformation Rules. To generate semantically equivalent variants of the original Verilog code, VeriEQ applies a set of carefully designed transformation rules, as listed in Table 1. To formally describe the applicability conditions of these rules, we introduce the following notations for conciseness: $B(E)$ denotes the effective bit-width of the expression E, $S(E)$ denotes its effective signedness, P_i represents a code block within a conditional statement, $\mathbb{Z}$ denotes the set of integers, `1'b0` is a one-bit constant with value zero, and MAX_V is the maximum allowed shift value, set to the maximum of `uint64`. These rules can be broadly categorized into two types.

The first category consists of *Arithmetic Equivalence Rules*, which encode general algebraic identities such as the commutativity of addition and multiplication. Most of these rules require constraints on the effective bit-width B_{eff} and signedness S_{eff} of expressions to ensure semantic equivalence after transformation. Among them, several rules (A1–A3, A7–A14) are inspired by the optimization patterns observed in the Yosys synthesis tool [73] and prior work [76, 77]. Other rules are derived from empirical observations of BDBs collected during our study. For instance, rules A4 and A5 are inspired by Verilator Bug #4857 [16], rule A6 by Verilator Bug #4864 [18], rule A15 by Verilator Bug #4709 [14], and rules A16–A19 by a combination of Yosys Bugs #4844, #4164, #3748 [17, 19, 71] and Icarus Verilog Bug #1165 [27]. These arithmetic rules primarily apply to expressions in combinational logic blocks and the dataflow components of sequential logic blocks.

The second category includes *Control Flow Equivalence Rules*, which target transformations of control-flow constructs within sequential logic blocks. Rules B1, B2, and B4 implement transformations between semantically equivalent `case` and `if` statements, inspired by Yosys Bug #4317 [32]. Rule B3 performs equivalence-preserving transformations by inserting syntactically valid but semantically unreachable branches into `case` statements. These dead branches do not affect the functional behavior of the design but enable the generation of semantically equivalent variants with altered control-flow structure. This transformation strategy is inspired by prior work on compiler testing that leverages dead code to expose optimization inconsistencies [30].

The rules are only applicable when their associated constraints are satisfied. Since applying one rule may yield a new expression that satisfies another rule, the transformation process may recurse. To avoid infinite transformation, VeriEQ limits the number of transformations per expression to a maximum of 6. This limit also aligns with our Finding 1 that most BDBs are triggered via shallow

Table 1. Equivalence circuit transformation rules, categorized into Arithmetic Equivalence Rules and Control Flow Equivalence Rules. Some rules are only valid under specific constraints. Here, E denotes an expression; $B(E)$ and $S(E)$ represent the effective bit-width and signedness of E, respectively; P_i denotes a code block within a conditional statement; $\mathbb{Z}$ denotes the set of integers; and MAX_V is the maximum allowed shift value, set to the maximum of `uint64`.

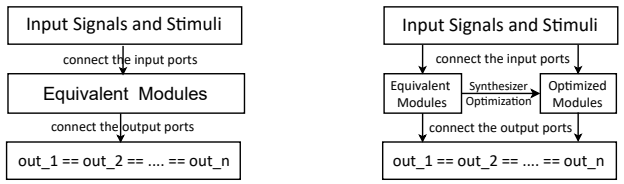
#	Equivalence Rules	Constraints
Category A: Arithmetic Equivalence Rules		
A1	$\text{div}(E, 1) \leftrightarrow E$	$S(1) \equiv S(E); B(1) \equiv B(E)$
A2	$\text{add}(E, \texttt{0}) \leftrightarrow E$	$S(\texttt{0}) \equiv S(E); B(\texttt{0}) \equiv B(E)$
A3	$\text{mul}(E, 1) \leftrightarrow E$	$S(1) \equiv S(E); B(1) \equiv B(E)$
A4	$\texttt{1'b1} \leftrightarrow \text{not}(E)$	$E \equiv 0; S(E) \equiv \texttt{unsigned}$
A5	$\texttt{1'b0} \leftrightarrow \text{not}(E)$	$E \neq 0; S(E) \equiv \texttt{unsigned}$
A6	$\text{and}(E, \text{not}(E)) \leftrightarrow \texttt{0}$	$S(\texttt{0}) \equiv S(E); B(\texttt{0}) \equiv B(E)$
A7	$\text{ge}(E_1, E_2) \leftrightarrow \text{le}(E_2, E_1)$	–
A8	$\text{or}(E_1, E_2) \leftrightarrow \text{or}(E_2, E_1)$	–
A9	$\text{or}(E, B(E)\{\texttt{1'b0}\}) \leftrightarrow E$	$S(E) \equiv \texttt{unsigned}$
A10	$\text{and}(E, B(E)\{\texttt{1'b1}\}) \leftrightarrow E$	$S(E) \equiv \texttt{unsigned}$
A11	$\text{xor}(E_1, E_2) \leftrightarrow \text{xor}(E_2, E_1)$	–
A12	$\text{and}(E_1, E_2) \leftrightarrow \text{and}(E_2, E_1)$	–
A13	$\text{add}(E_1, E_2) \leftrightarrow \text{add}(E_2, E_1)$	–
A14	$\text{mul}(E_1, E_2) \leftrightarrow \text{mul}(E_2, E_1)$	–
A15	$E \leftrightarrow \{E[i] \mid i = B(E) - 1 \ldots 0\}$	$E \in \mathbb{Z}$, $E[i]$: i-th bit
A16	$\texttt{0} \leftrightarrow \text{shl}(E, r),\ r \in [B(E), \text{MAX\_V}]$	$S(\texttt{0}) \equiv S(E); B(\texttt{0}) \equiv B(E)$
A17	$\texttt{0} \leftrightarrow \text{shr}(E, r),\ r \in [B(E), \text{MAX\_V}]$	$S(\texttt{0}) \equiv S(E); B(\texttt{0}) \equiv B(E)$
A18	$\texttt{0} \leftrightarrow \text{lshl}(E, r),\ r \in [B(E), \text{MAX\_V}]$	$S(\texttt{0}) \equiv S(E); B(\texttt{0}) \equiv B(E)$
A19	$\texttt{0} \leftrightarrow \text{ashr}(E, r),\ r \in [B(E), \text{MAX\_V}]$	$S(E) \equiv S(\texttt{0}) \equiv \texttt{unsigned}$
Category B: Control Flow Equivalence Rules		
B1	$\text{if}(E_1 == E_2)\ P_1\ \text{else}\ P_2$ $\updownarrow$ $\text{case}(E_1)\ \ E_2 : P_2;\ \text{default} : P_1;\ \text{endcase}$	–
B2	$\text{if}(E)\ P_1\ \text{else}\ P_0$ $\updownarrow$ $\text{case}(E)\ \ \texttt{1'b0} : P_0;\ \text{default} : P_1;\ \text{endcase}$	–
B3	$\text{case}(E_1)\ \ E_2 : P_2;\ E_3 : P_3;\ \ldots\ E_n : P_n;\ \text{default}{:}P_d;\ \text{endcase}$ $\updownarrow$ $\text{case}(E_1)\ \ E_2 : P_2;\ E_3 : P_3;\ \ldots\ E_n : P_n;\ E_{n+1} : P_{n+1};\ \text{default}{:}P_d;\ \text{endcase}$	$S(E_i) \equiv \texttt{unsigned},\ \forall i \in [2, n]$, $E_{n+1} \in \mathbb{Z}$, $E_{n+1} > 2^{B(E_1)} - 1$
B4	$\text{case}(E_1)\ \ E_2 : P_2;\ E_3 : P_3;\ \ldots\ E_n : P_n;\ \text{default}{:}P_d;\ \text{endcase}$ $\updownarrow$ $\text{if}(E_1 == E_2)\ P_2;\ \text{else if}(E_1 == E_3)\ P_3;\ \ldots\ \text{else if}(E_1 == E_n)\ P_n;\ \text{else}\ P_d;$	$S(E_i) = S(E_1),\ \forall i \in [2, n]$, $B(E_i) = B(E_1),\ \forall i \in [2, n]$

expression chains. Meanwhile, because our design incorporates a set of fundamental semantic-preserving transformation rules (e.g., A2), there is typically at least one applicable equivalence

rule for any given program. As a result, VeriEQ will not encounter cases where no transformation rule can be applied. To illustrate how these rules work in practice, we refer to the motivating example in Section 3. Starting from the initial code `reg_10 <= 2'd3`, VeriEQ applies Rule A15 to transform `2'd3` into `{1'b1, 1'b1}`. Then, Rule A4 transforms the second `1'b1` into `~(1'b0)`, and Rule A6 further transforms `1'b0` into `~((in1 & ~in1))`. This results in the final expression `{1'b1, ~((in1 & ~in1))}`, which is semantically equivalent to the original but produces different simulation results, thereby successfully exposing a BDB.

5.4 Bug Detection

To efficiently detect the occurrence of BDBs, it is necessary to compare the outputs of a set of semantically equivalent Verilog programs generated in the previous step under identical input conditions. If we follow the conventional approach of elaborating, simulating, and comparing outputs for each Verilog program independently, the overall cost grows linearly with the number of equivalent programs under test. This becomes unacceptable when dealing with some slow Verilog simulators or synthesizers.



(a) Simulator Testbench (b) Synthesizer Testbench

Fig. 6. The inline deviation testbench connects external inputs to equivalent modules and compares all outputs through internal equality checks.

Inline Deviation Checking Mechanism: To address this, we propose an inline deviation checking mechanism. As shown in Figure 6, VeriEQ first defines a set of external input signals in the testbench, implemented as reg variables, whose bit-width and signedness match those of the original Verilog program's input signals. These external signals are assigned values in each simulation round according to the stimulus patterns derived from the original code. Next, the equivalent Verilog modules generated through program transformation are instantiated within the same testbench. All modules share the same external input signals, while their outputs are connected to separate external output wires. To determine whether these modules produce consistent results, VeriEQ connects all output signals in the testbench using a series of equality expressions. For synthesis tools, VeriEQ feeds the same set of semantically equivalent programs into the synthesizer to obtain the optimized modules. It then uses the fastest simulator to execute the previously described testbench comparison process, in order to detect any semantic deviation introduced during synthesis and thereby expose potential BDBs. This mechanism improves the efficiency of differential testing and also avoids behavioral discrepancies across different simulators and synthesizers.

Anomaly Monitor: Beyond detecting incorrect outputs, our investigation reveals that Verilog simulators and synthesizers also suffer from hangs. To ensure comprehensive bug detection, VeriEQ monitors for two categories of anomalous behavior during inline deviation checking: incorrect outputs and hangs. Incorrect results are identified by simulating the generated testbench and checking whether the final output expression evaluating the equality of all outputs is always true. If not, VeriEQ records the violating modules along with the original code for further analysis. For

hang detection, we employ two mechanisms. First, if the simulation or synthesis process exceeds a predefined timeout threshold, the process is forcibly terminated, and the corresponding test case is preserved for further analysis. In our experiments, we set this threshold to 30 seconds, since almost all generated Verilog programs complete synthesis or simulation within 5 seconds. Second, if the size of the log file generated during simulation or synthesis exceeds a predefined limit, the process is also terminated. We set this limit to 50 MB, as the log files of normal runs are typically smaller than 5 MB. Once a proof-of-concept (PoC) for a potential hang bug is obtained, we repeat the test five times, and report it to the developers only if the issue is consistently reproducible. The second mechanism helps prevent excessive log generation caused by infinite loops, which could otherwise occupy system resources and block other testing processes.

6 Implementation

We implemented VeriEQ and evaluated it on four widely used Verilog simulators and synthesizers: Verilator v5.035 [63], Icarus Verilog v13.0 [60], CXXRTL v0.50 [59], and Yosys v0.50 [74]. Among them, Verilator, Icarus Verilog, and CXXRTL serve as simulators, each supporting different simulation models and workflows. Verilator elaborates Verilog code into cycle-accurate C++ simulation models, which are then compiled and executed to perform high-speed simulation. Icarus Verilog follows a traditional two-stage workflow: it elaborates Verilog code into an intermediate vvp bytecode format, which is subsequently interpreted by the vvp runtime engine during simulation. CXXRTL, built as a backend within Yosys, also elaborates Verilog into C++ code, but uses a bit-accurate signal representation and requires writing a C++ testbench to drive the simulation. Yosys, as a synthesis tool, performs a series of logic transformations, constant propagation, and technology mapping passes to optimize Verilog code before translating it into gate-level netlists. The implementation and evaluation across these diverse tools demonstrate VeriEQ's simulation- and synthesis-compatible, and scalable capabilities.

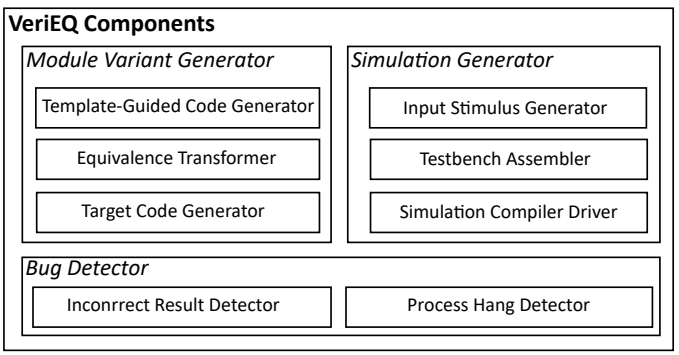


Fig. 7. The components of VeriEQ are divided into three parts: the Module Variant Generator for generating multiple equivalent Verilog modules, the Simulation Generator for producing simulation files for target-specific platforms, and the Bug Detector for identifying BDBs.

Figure 7 illustrates the components of VeriEQ, which are divided into three main parts. The first part is the Module Variant Generator, responsible for generating the initial Verilog module based on a predefined template, constructing a set of semantically equivalent modules using equivalent transformation rules, and converting them into the target-specific module formats required by simulators such as CXXRTL and Verilator (e.g., C++ style wrappers). The second part is the Simulation Generator, which generates the input stimuli and testbench for the modules, and

conducts the simulation process to produce executable outputs. The third part is the Bug Detector, which checks for possible bug manifestations at each stage of the workflow. Most components of VeriEQ are designed to be independent of the specific simulator or synthesizer being tested. When porting VeriEQ to a new Verilog simulator or synthesizer, only a small portion needs to be adapted, while the majority of the framework remains unchanged. Specifically, the Target Code Generator must be updated to map intermediate representations and testbenches to the input format expected by the new simulator or synthesizer. This step typically involves straightforward one-to-one mappings. Additionally, the Simulation Compiler Driver must be modified to reflect the new elaboration and simulation commands, which in practice usually requires only minor adjustments to a few lines of code.

7 Evaluation

We evaluate VeriEQ on four Verilog simulators and synthesizers: three open-source simulators (Verilator, CXXRTL, and Icarus Verilog) and one open-source synthesizer, Yosys. Our evaluation aims to answer the following three research questions:

- **RQ1:** Can VeriEQ detect real-world BDBs in Verilog simulators and synthesizers?
- **RQ2:** How does VeriEQ perform compared to state-of-the-art testing tools for Verilog simulators and synthesizers?
- **RQ3:** How efficient and accurate is VeriEQ?

7.1 Evaluation Setup

Metrics and Settings: In our evaluation, we employ two primary metrics: (1) the number of unique bugs detected, which reflects the bug-finding capability of the tool, and (2) the execution throughput, measured as the number of test cases executed within a fixed period of time, which evaluates the testing efficiency. Our experiments are conducted on four mainstream Verilog simulators and synthesizers: Verilator v5.035, Icarus Verilog v13.0, CXXRTL v0.50, and Yosys v0.50. All experiments are conducted on an Ubuntu 22.04.3 system with Linux kernel version 5.15.0, deployed on a 64-bit physical machine equipped with 256 CPU cores (AMD EPYC 7763 64-Core Processor) and 1 TB of memory. To facilitate comprehensive vulnerability detection, we instrumented all four Verilog simulators and synthesizers with AddressSanitizer (ASan) and UndefinedBehaviorSanitizer (UBSan).

Compared Tools: We compare VeriEQ with three state-of-the-art Verilog testing tools: TransFuzz, Verismith and VeriXmith. TransFuzz randomly generates cell-level netlists. It runs the same netlist on different simulators and compares the results to find potential BDBs. Verismith focuses on synthesis testing. It generates large behavioral-level Verilog programs using predefined syntax rules. Then, it synthesizes the code into gate-level netlists and uses formal verification to check if the synthesized behavior matches the original. VeriXmith, on the other hand, mutates Verilog code collected from open-source repositories as well as code randomly generated by Verismith. It then uses formal verification to check whether the mutated code produces identical results across multiple simulators and synthesizers.

7.2 BDB Detection in Real-World Verilog Simulators and Synthesizers

We run VeriEQ on the four Verilog simulators and synthesizers for a total of 48 hours to discover potential bugs. For the three simulators (Verilator, CXXRTL, and Icarus Verilog), we directly feed the generated Verilog code into each tool, elaborate and simulate it, and then check the output results. For the synthesis tool Yosys, we input the Verilog file into Yosys, apply its optimization passes, and extract the optimized code. We then simulate both the original and optimized code using Icarus Verilog and compare their outputs. We choose Icarus Verilog because it is the fastest

Table 2. Bugs detected in mainstream Verilog simulators and synthesizers. Among them, 29 are classified as BDB and additional 4 hang bugs. Bug identifiers are partially anonymized to support the double-blind review process.

#	Platform	Bug Type	The Root Cause Analysis	Status	Identifier
1	Yosys	Hang	synth_greenpak4 enters an infinite loop due to unguarded signal processing condition.	Fixed	Bug#xxx2
2	Yosys	Hang	synth_efinix repeatedly inserts enable logic without reaching a termination condition.	Fixed	Bug#xxx9
3	Yosys	Hang	Integer overflow in AUTONAME pass leads to unexpected looping behavior in naming logic.	Fixed	Bug#xxx3
4	Yosys	Incorrect Result	synth_coolrunner2 yields incorrect output due to mishandling of uninitialized control paths.	Fixed	Bug#xxx4
5	Yosys	Incorrect Result	MICROCHIP_DSP logic dereferences null pointer during synth_microchip execution process.	Fixed	Bug#xxx5
6	Yosys	Incorrect Result	synth_efinix asserts signal_list[id1].bit.wire is null during pass signal linking.	Fixed	Bug#xxx7
7	Yosys	Incorrect Result	Multiple $add cells drive same signal; assertion fails inside signal merging routine.	Fixed	Bug#xxx1
8	Yosys	Incorrect Result	False-positive proc_arst detection causes invalid reset logic to be incorrectly inserted.	Fixed	Bug#xxx3
9	Yosys	Hang	read_aiger -xaiger causes hang in synth_intel_alm due to recursive feedback path.	Confirmed	Bug#xxx7
10	Yosys	Incorrect Result	Shift optimization fails when shift amount exceeds safe constant boundary range.	Fixed	Bug#xxx9
11	Yosys	Incorrect Result	Integer overflow during large constant shift causes invalid wire assignment result.	Fixed	Bug#xxx1
12	Yosys	Incorrect Result	Signed right-shift with large constant not handled correctly during optimization.	Fixed	Bug#xxx5
13	Yosys	Incorrect Result	read_verilog mishandles directory path inputs, skipping or misinterpreting files.	Fixed	Bug#xxx3
14	Yosys	Incorrect Result	Signal declaration silently fails due to overflowed bit-width in internal pass.	Confirmed	Bug#xxx1
15	Yosys	Incorrect Result	Power operator folding loses x-propagation when signed exponent is statically folded.	Confirmed	Bug#xxx5
16	Yosys	Incorrect Result	nlutmap triggers heap-use-after-free when LUT reuse detection is improperly triggered.	Confirmed	Bug#xxx6
17	CXXRTL	Incorrect Result	Signed value not extended before shift operation, leading to inconsistent behavior.	Fixed	Bug#xxx4
18	CXXRTL	Incorrect Result	udivmod returns incorrect results when input width exceeds logical limit due to ctlz() misbehavior.	Fixed	Bug#xxx8
19	Verilator	Incorrect Result	Signed value and unsigned constant matched improperly in behavioral case expression.	Fixed	Bug#xxx8
20	Verilator	Incorrect Result	One-bit wire emits multi-bit value in simulation due to faulty internal evaluation.	Fixed	Bug#xxx3
21	Verilator	Incorrect Result	Constant $display propagation fails with signed value, producing wrong formatted output.	Fixed	Bug#xxx3
22	Verilator	Incorrect Result	4-state variable with -fno-expand causes unexpected internal assertion failure.	Fixed	Bug#xxx9
23	Verilator	Incorrect Result	Assignment behavior diverges under -fno-expand due to inconsistent netlist flattening.	Fixed	Bug#xxx2
24	Verilator	Incorrect Result	$display truncates signed slices with %b and misinterprets values with %d format.	Fixed	Bug#xxx3
25	Verilator	Incorrect Result	Ternary with bit-slice and shift optimized incorrectly, producing wrong output.	Fixed	Bug#xxx3
26	Verilator	Incorrect Result	Arithmetic right-shift with large constant results in incorrect sign-bit propagation.	Fixed	Bug#xxx4
27	Verilator	Incorrect Result	DFG-level transform misoptimizes signed comparison between unrelated bit-widths.	Fixed	Bug#xxx0
28	Verilator	Incorrect Result	Stack buffer overflow triggered by large-width division or modulo simulation path.	Fixed	Bug#xxx3
29	Verilator	Incorrect Result	Optimizer transforms conditional expression and silently alters output computation.	Fixed	Bug#xxx7
30	Verilator	Incorrect Result	Optimization on shifted-out variable fails to preserve original signal semantics.	Fixed	Bug#xxx6
31	Verilator	Incorrect Result	Signed int return type causes truncation when used in high-width signal assignment.	Confirmed	Bug#xxx3
32	IVerilog	Incorrect Result	StringHeap fails due to unsafe realloc on malformed token stream.	Confirmed	Bug#xxx6
33	IVerilog	Incorrect Result	case statement fails due to mismatch between signed reg and unsigned constant.	Fixed	Bug#xxx7

simulator in our experiments. When a mismatch is found between the pre- and post-synthesis designs, we check the same programs with other simulators. If all simulators show the mismatch, we treat it as a bug in the synthesizer; otherwise, it is considered a bug in Icarus Verilog. In both cases, we keep the proof-of-concept (PoC) and report it to the developers.

Bug Overview: In total, VeriEQ uncovered 33 previously unknown bugs: 16 in Yosys, 2 in CXXRTL, 13 in Verilator, and 2 in Icarus Verilog. A summary of these bugs is provided in Table 2. All of these bugs have been confirmed by the developers, and 27 of them have already been fixed. This reflects the developers' recognition of and attention to the bugs we discovered. Among the 33 bugs, 29 are BDBs that cause the simulators or synthesizers to produce outputs inconsistent with semantically equivalent code during simulation or synthesis. These subtle errors do not raise explicit failures but can silently introduce functional deviations, potentially leading to incorrect chip behavior and serious reliability or security issues. Another 4 bugs cause the tools to hang, either by freezing during elaboration or flooding logs until disk space runs out. These hangs waste resources and can stall the entire design or verification workflow. Notably, VeriEQ successfully uncovered BDBs across all four platforms, indicating that such bugs are widespread in mainstream Verilog tools. This also demonstrates VeriEQ's strong adaptability and practical effectiveness across diverse Verilog simulators and synthesizers.

Feedback from Developers: During our investigation of BDBs, we observed that some past issues related to BDB had been rejected or closed by developers without thorough examination. This was often because the code triggering the bug was randomly generated by fuzzers and tended to be overly long and complex, making it time-consuming for developers to identify the root cause. Thanks to the use of Verilog templates, the code generated by VeriEQ is typically concise and well-structured. We randomly sampled 5,000 initial Verilog programs used in our experiments. These programs have an average size of approximately 70.17 lines of code, with the largest one containing about 120 lines. Developers have responded promptly and positively to our reports, often confirming the bugs within a day or two. This makes it easier for developers to locate and minimize the problematic code fragments that trigger BDB. Furthermore, the equivalence rules defined in our system allow VeriEQ to generate many unexpected and interesting test cases that are often overlooked by developers. As a result, the bug reports submitted by our tool are generally of high quality. Notably, several bugs we reported for Verilator have been added to its unit test suite. We also received positive feedback from developers acknowledging our contributions. For example, the developers of Verilator and CXXRTL commented on our issues with remarks such as: "*Nice find, thanks!*" "*Thanks for the interesting case.*" and "*I think your bug reports are fantastic.*"

```
1 module top (in3, in4, out10);
2     input wire  [7:7] in3;
3     input wire [23:16] in4;
4     wire  [29:29] wire_2;
5     output wire  [28:14] out10;
6     // assign wire_2 = in3 ? {4{14'b010111101}} : (1'b0);
7     assign wire_2 = in3 ? {4{14'b010111101}} : (in4[18] >> 8'b1);
8     assign out10 = wire_2;
9 endmodule
```

Fig. 8. A BDB detected by VeriEQ via Arithmetic Equivalence Rule A17. This BDB is triggered because Verilator's BitOpTree construction missed the implicit masking operation & 1.

Bug Study 1: Incorrect Bit Shift. Figure 8 shows a BDB detected by VeriEQ. The equivalence between line 6 and line 7 is derived from our Arithmetic Equivalence Rule A17: `0` $\leftrightarrow$ `shr(E, r)`, $r \in [B(E), \text{MAX\_V}]$. VeriEQ identifies the constant `1'b0` and replaces it with a logical right shift of a 1-bit expression by a value greater than its bit-width. Theoretically, these two lines should be semantically equivalent, as `in4[18]` is a single-bit value, and shifting it by 1 should discard the only bit and result in `1'b0`. However, Verilator's BitOpTree construction did not handle this correctly. It missed the implicit masking operation `& 1`, and as a result, the expression `(in4[18] >> 1)` was incorrectly treated as preserving the original bit value. This led to `wire_2` being wrongly computed as 1 instead of `0` under certain input vectors, making the output diverge from that of the equivalent code. To fix this bug, developers added special handling for single-bit shifts: when the width is 1 and the shift amount removes the bit entirely, the result is forcibly set to `0`. Thanks to our shift-related equivalence transformation rules, VeriEQ was able to discover this bug within ten minutes, demonstrating its powerful detection capability.

Bug Study 2: Incorrect Relational Operator Optimization Invocation. Figure 9 presents another BDB discovered by VeriEQ. The two expressions on line 6 and line 7 are constructed using the Arithmetic Equivalence rule A7: $\text{ge}(E_1, E_2) \leftrightarrow \text{le}(E_2, E_1)$. Although this rule appears simple, it exposes a deeply hidden BDB. Specifically, when evaluating expressions like `(-a <= a)`, the simulator may apply a peephole optimization to fold it into a constant when the bit-width of `a` is 1. However, due to a developer oversight in the `foldOp<DfgLteS>` function, the simulator incorrectly

```
1 module top();
2     reg  in4;
3     wire out88;
4     wire signed wire_2;
5     assign wire_2 = in4 ? 2'b10 : 0;
6     //assign out88 = (wire_2 >= -(wire_2) ? 1 : 0);
7     assign out88 = (-(wire_2) <= wire_2 ? 1 : 0);
8 endmodule
```

Fig. 9. A BDB detected by VeriEQ via Arithmetic Equivalence Rule A7. This BDB is triggered because the peephole optimization incorrectly calls the opLtS function for the <= operator, whereas the correct behavior should be to call the opLteS function.

invokes opLtS instead of the intended opLteS, resulting in a faulty optimization for the <= expression. In contrast, the equivalent >= expression is correctly optimized via the foldOp<DfgGteS> function, which properly calls opGteS. This inconsistency leads to divergent outputs from two semantically equivalent expressions. The developer fixed the bug by simply renaming the incorrect function call from opLtS to opLteS. Notably, this bug had existed unnoticed since the initial implementation of the DFG peephole optimization module three years ago. Upon confirmation, the developer praised the discovery, commenting: "*Nice find, thanks!*" Remarkably, VeriEQ detected this issue within just one hour of execution, highlighting its effectiveness in uncovering BDB.

```
1 module top();
2   case (3'sb111)
3     4'b111: begin
4         $display("the correct branch")
5     end
6     default: begin
7         // enter this branch
8         $display("the wrong branch")
9     end
10  endcase
11 endmodule
```

(a) Initial Verilog code with a case statement generated by VeriEQ

```
1 module top();
2   case (3'sb111 * 3'sb1)
3     4'b111: begin
4         // enter this branch
5         $display("the correct branch")
6     end
7     default: begin
8         $display("the wrong branch")
9     end
10  endcase
11 endmodule
```

(b) Code after transforming the expression in the case statement

Fig. 10. A BDB detected by VeriEQ via Arithmetic Equivalence Rule A3. The root cause of this BDB is that, when the bit-widths of the expression inside the case parentheses and the branch expressions do not match, the developer mistakenly used the function for extending the left-hand value instead of the function for extending the expression when performing the extension on the case statement expression.

Bug Study 3: Incorrect Case Expression Comparison. Figure 10 presents a BDB discovered by VeriEQ, which can only be triggered in the presence of behavioral-level case statements. Due to the lack of such constructs in their input generation, both TransFuzz and Verismith failed to detect this bug. VeriEQ applied the Arithmetic Equivalence Rule A3: $\text{mul}(E, 1) \leftrightarrow E$ to construct two semantically equivalent expressions on line 2 of the two subfigures in Figure 10. In the original code on the left, when comparing 3'sb111 and 4'b111 in a case statement, their bit-width mismatch causes 3'sb111 to be zero-extended, resulting in 4'b0111, which matches the first branch. Therefore, the correct behavior should be to enter the first branch. However, due to a simulator implementation bug, the extension of 3'sb111 is performed using EXTEND_LHS instead of EXTEND_EXP, leading to incorrect extension and a failure to match the first branch. In the

transformed code on the right, the equivalent expression `3'sb111 * 3'sb1` is optimized differently and matches the correct branch, causing the two forms to produce inconsistent outputs. This inconsistency was successfully detected by VeriEQ, demonstrating its ability to expose subtle bugs related to signedness and behavioral constructs.

```
module top();
  // in4 = 1
  reg signed in4;
  // (0 ? 1'h0 : in4) equal to in4
  case (0 ? 1'h0 : in4)
    //in4 match this branch
    8'b000001: begin
      // enter this branch
      $display("the correct branch");
    end
    default: begin
      $display("the wrong branch");
    end
  endcase
endmodule
```

(a) The initial Verilog code generated by VeriEQ contains a case statement with two branches.

```
module top();
  reg signed in4; // in4 = 1
  case (0 ? 1'h0 : in4)
    8'b000001: begin // should also match this
      $display("the correct branch");
    end
    5'b01101: begin
      $display("the added branch");
    end
    default: begin
      // enter this branch
      $display("the wrong branch");
    end
  endcase
endmodule
```

(b) The code transformed by VeriEQ through equivalence transformation applies Rule C3 to add an extra branch.

Fig. 11. A BDB detected by VeriEQ via Control Flow Equivalence Rule C3. The initial code correctly enters the intended branch, but after adding the extra branch, it falls into the wrong one. This BDB arises because the case condition was resized using the sign of the base expression instead of the properly padded expression during pruning.

Bug Study 4: Incorrect Case Branch Selection. Figure 11 illustrates a BDB detected by VeriEQ, which can only be triggered by the combined effect of a `case` statement and the application of Control Flow Equivalence Rules. Specifically, in the code shown in Figure 11a, the expression inside the `case` condition, `(0 ? 1'h0 : in4)`, can be simplified to `in4`. When `in4` is set to 1, although it is a signed value, it is compared against the branch label `8'b000001`, which is unsigned. Therefore, `in4` is treated as unsigned and zero-extended during the comparison, resulting in `8'b000001`, which matches the first branch. We apply Control Flow Equivalence Rule C3 to the initial code by inserting a dead-code branch `5'b01101` into the `case` block, resulting in the transformed code shown in Figure 11b. This branch is considered dead because its value exceeds the maximum value that `(0 ? 1'h0 : in4)` can represent, meaning the control flow should never reach it. Semantically, the addition of such dead code should not affect the execution of the control flow. However, the simulator attempts to optimize simulation performance by pruning the `case` statement, aiming to minimize the width of the condition and branch labels as much as possible. During this pruning process, the resizing operation mistakenly uses the sign of the base expression rather than the correctly padded expression. This implementation flaw causes the control flow in Figure 11b to incorrectly fall into the `default` branch, resulting in output inconsistency between the semantically equivalent versions. This discrepancy is captured and reported as a BDB by VeriEQ, demonstrating the effectiveness of its Control Flow Equivalence Rules.

7.3 Comparison with Existing Tools

To evaluate the effectiveness of VeriEQ, we compare it against TransFuzz, Verismith, and VeriXmith in terms of their bug-finding capabilities. Since Verismith is designed exclusively for synthesis

testing, we evaluate it only on Yosys, whereas TransFuzz is tested across all four Verilog simulators and synthesizers. VeriXmith is expected to use Verilog code collected from open-source repositories as well as code randomly generated by Verismith as its initial seeds. However, these seed programs were not released by the authors. Therefore, we use code generated by Verismith ourselves as the initial seed set for VeriXmith in our evaluation. Each tool is executed for 48 hours and the results are summarized in Table 3. The 48-hour window refers to the total runtime of each testing tool, and the execution of every test case already includes the time spent invoking the simulator (or synthesizer). TransFuzz successfully identified 3 bugs in Yosys (#10, #11, #12) and 4 in Verilator (#23, #25, #29, #30), but failed to detect any bugs in CXXRTL and Icarus Verilog. Verismith was only able to find one bug on Yosys (#10). VeriXmith detected one bug in Yosys (#10) and one bug in Icarus Verilog (#32). This is primarily because several Yosys bugs are only triggered when the input signals contain unknown value x. Unlike VeriEQ, TransFuzz, Verismith, and VeriXmith all lack the ability to generate inputs with initial x values, and therefore fail to detect this class of bugs. Additionally, Verismith and VeriXmith rely on formal verification techniques to detect bugs, which significantly reduces their testing throughput. As a result, they all failed to discover the 3 bugs that could be detected by TransFuzz within the same 48-hour time window. Most of the bugs in Verilator and CXXRTL are triggered by signals with signedness or bit-width mismatches. However, TransFuzz generates netlists using macro cells, which always produce unsigned signals with matched widths, making it less effective in exposing such issues. Furthermore, TransFuzz lacks behavioral-level constructs and cannot generate constructs like case statements, leading it to miss related bugs that VeriEQ successfully identifies.

To evaluate the impact of initial seeds on testing effectiveness, we provide the initial Verilog programs generated by VeriEQ to the other tools and compare the number of bugs discovered within 48 hours. For the generation-based testing tools TransFuzz and Verismith, we replace their generated programs with our initial Verilog programs. For the mutation-based testing tool VeriXmith, we replace its initial seed corpus with our initial Verilog programs. The results show that, when supplied with our seeds, TransFuzz, Verismith, and VeriXmith discover only 14, 3, and 6 bugs respectively, all of which are subsets of the bugs found by VeriEQ. This demonstrates that even under identical initial seeds, the bug detection capability of these tools remains limited.

Table 3. Bug counts and bug IDs detected by each tool across different Verilog simulators and synthesizers

Tool	**Simulators and Synthesizers (Bug Count)**				**Bug IDs**
	Yosys	**Verilator**	**CXXRTL**	**Icarus Verilog**	
VeriEQ	16	13	2	2	#1-#33
TransFuzz	3	4	0	0	#10, #11, #12, #23, #25, #29, #30
Verismith	1	–	–	–	#10
VeriXmith	1	0	0	1	#10, #32

In terms of test case generation efficiency, VeriEQ outperforms TransFuzz, Verismith, and VeriXmith. Since TransFuzz employs differential testing between Verilator and another simulator, or between pre- and post-optimization versions of a Yosys design, we constrain VeriEQ to the same comparison targets for a fair evaluation. We measured the number of test cases generated by each tool within 10 minutes on three configurations: Verilator/Icarus Verilog, Verilator/CXXRTL, and Yosys/opt. VeriEQ was able to generate 332,050, 53,350, and 107,770 test cases for each respective configuration, whereas TransFuzz only generated 7,192, 6,564, and 78,009 test cases, which represents an improvement of 138.1% to 4161.9% in throughput. Verismith, on the other hand,

generated only 15 test cases on Yosys within the same time period. VeriXmith performed even worse, completing only a single test case in the same period. This is primarily because both tools rely on formal verification, which introduces substantial verification overhead. The impact is especially significant for VeriXmith, as it performs verification across all platforms.

7.4 Efficiency and Accuracy of VeriEQ

To evaluate the practical value of VeriEQ, we assess both the efficiency of its inlined deviation checking mechanism and stimulus generation process, together with the accuracy of bug detection.

Efficiency. We compare the inlined deviation checking mechanism implemented in VeriEQ against VeriEQ$^-$. VeriEQ integrates multiple semantically equivalent modules into a single testbench and verifies their output consistency during simulation. In contrast, VeriEQ$^-$ adopts the traditional approach of elaborating each module individually and comparing their outputs. As shown in Table 4, the execution throughput of the inlined strategy yields improvements of 281.1%, 411.1%, 221.6%, and 489.2% across four Verilog simulators and synthesizers, respectively. These results validate the effectiveness of inlined checking mechanism in enhancing testing efficiency. This design eliminates redundant simulation runs and reduces costly file I/O operations, enabling faster detection of behavioral discrepancies.

Table 4. Comparison of Inlined Deviation Checking and Traditional One-by-One Comparison

Method	**Simulators and Synthesizers (Testcase Count)**			
	Yosys	**Verilator**	**CXXRTL**	**Icarus Verilog**
VeriEQ$^-$	36,562	25,312	21,776	144,296
VeriEQ	102,770	104,060	48,260	705,880
Improvement	+281.1%	+411.1%	+221.6%	+489.2%

To evaluate whether the inlined deviation checking mechanism impacts the bug-finding capability of VeriEQ, we conduct another ablation study. Specifically, we compare VeriEQ$^-$ against the original VeriEQ. Both tools run for 48 hours (the same time budget used in Section 7.3) across four Verilog simulators and synthesizers. The number of bugs found by each configuration is shown in Table 5. The results show that VeriEQ$^-$, due to its significantly lower execution efficiency, explores fewer test cases and consequently detects only 23 bugs. In contrast, the original VeriEQ, with inlined deviation checking enabled, executes a much larger number of test cases within the same time window and uncovers more bugs. These findings indicate that the inlined deviation checking mechanism contributes to improved bug-finding capability by significantly accelerating test execution.

Table 5. Bug counts detected by VeriEQ$^-$ and VeriEQ across different Verilog simulators and synthesizers

Method	**Simulators and Synthesizers (Bug Count)**			
	Yosys	**Verilator**	**CXXRTL**	**Icarus Verilog**
VeriEQ$^-$	11	9	2	1
VeriEQ	16	13	2	2

Beyond the inlined checking mechanism, the efficiency of VeriEQ is also influenced by the number of stimulus cycles used in each test case. To quantify this effect, we randomly sampled

1,000 code snippets that can trigger BDBs together with their corresponding random stimuli and measured the number of stimulus cycles required to expose the bug. The distribution is summarized in Table 6. The vast majority of bugs are triggered within a small number of cycles: 94.6% of the sampled programs expose the bug within 10 cycles, and over 98% within 15 cycles. Increasing the number of stimulus cycles beyond this point provides little additional benefit while incurring higher simulation overhead and reducing overall testing throughput. In particular, no obvious improvement is observed once the number of cycles exceeds 20. Based on this observation, VeriEQ adopts 20 stimulus cycles as a reasonable and effective heuristic that balances bug detection capability and execution efficiency.

Table 6. Distribution of Stimulus Cycles Required to Trigger BDBs

Trigger Cycles (#cycles)	Number of Code Snippets	Percentage
≤ 10 cycles	946	94.6%
11–15 cycles	37	3.7%
15–20 cycles	17	1.7%

Accuracy. To assess accuracy, we manually inspected all bug reports produced by the Bug Detector throughout the entire testing process. After deduplication and discussions with tool developers, only two false positives were identified.

The first false positive occurred when VeriEQ reported a crash in an experimental feature of Yosys. The developers clarified that this feature was still under active development and not intended for formal testing. The second false positive was observed in CXXRTL, where certain scenarios require two consecutive calls to the `eval()` function for register values to stabilize; otherwise, inconsistent simulation results may arise. Although the developers acknowledged this as a valid issue, they considered it extremely rare and indicated that resolving it would require a major redesign. Based on the developers' recommendations, VeriEQ skips experimental features in Yosys and invokes `eval()` twice in CXXRTL to avoid such false positives and ensure stable results.

To evaluate the false negatives of VeriEQ's Bug Detector, we reproduced all 54 historical BDBs collected in our empirical study. We configured the simulators and synthesizers to the corresponding versions and ran VeriEQ on them for 48 hours. The results show that VeriEQ successfully detected all 54 historical BDBs: every program that triggers a BDB produces either incorrect outputs or a hang during execution, and thus is flagged by VeriEQ's Bug Detector. Therefore, within the 54 known historical BDB test cases examined in this experiment, we did not observe false negatives from the Bug Detector. Notably, the code templates employed by VeriEQ are derived from historical BDB cases and are restricted to synthesizable Verilog constructs. To evaluate the coverage of the synthesizable Verilog language, we measure the code coverage of the Icarus Verilog parser while executing our generated programs, as an approximation of the language constructs exercised by our approach. Overall, our generated programs achieve 85.7% coverage. This focus reflects the fact that synthesizable code is ultimately deployed in hardware and therefore carries greater practical relevance. As a result, VeriEQ is unable to detect bugs that are exclusively triggered by non-synthesizable constructs, since such code is not generated. Nevertheless, with appropriate modifications, VeriEQ can be extended to support non-synthesizable code and uncover the corresponding bugs. In summary, VeriEQ achieves both high efficiency and strong accuracy in detecting BDB across Verilog Simulators and Synthesizers.

8 Discussion

Lessons Learned for Verilog Simulator and Synthesizer Development. During our analysis of the bugs discovered by VeriEQ, we identified an interesting pattern across Verilog simulators

and synthesizers. Most Verilog simulators and synthesizers, including the four we evaluated, are implemented in C or C++ to ensure high performance. In C/C++ development, it is common practice to use the `int` type for integers, which is signed by default. In contrast, Verilog treats integers as unsigned unless explicitly declared otherwise. If developers overlook this semantic mismatch, conversions between signed and unsigned integers may result in overflows, ultimately causing bugs. We found this issue to be particularly prevalent in aggressively optimizing simulators and synthesizers such as Yosys and Verilator. For instance, a core Yosys developer commented under our reported issue: *"I wonder what proportion of the bugs fuzzers have found in the last few years have been exclusively due to* `as_int`*..."*. We therefore advocate for greater awareness of signedness-related overflow risks among developers of Verilog simulators and synthesizers.

Supporting More Semantic Equivalence Rules. Currently, VeriEQ supports a core set of semantic equivalence transformations. These include arithmetic identity transformations and control flow equivalence transformations. In practice, these rules have proven effective in uncovering multiple real-world bugs in simulators and synthesizers that were previously missed by existing tools. However, since the current rules are manually defined, they may be limited in completeness. In future work, we plan to leverage large language models to automatically generate additional equivalence rules and integrate them into VeriEQ after verifying their semantic correctness. A key advantage of VeriEQ is its extensibility: supporting new equivalence rules typically requires only a few lines of additional transformation code. This makes it easy to broaden the diversity of semantic transformations with minimal engineering effort.

Supporting More Bug Types. Currently, VeriEQ primarily focuses on detecting BDBs in Verilog simulators and synthesizers, which manifest as incorrect simulation or synthesis results caused by faulty optimizations or code implementation. In addition, it can identify runtime anomalies such as hangs that occur during simulation or synthesis. However, hardware design and compilation flows involve other categories of potential issues, such as performance regressions and abnormal resource utilization. At present, our work focuses exclusively on the logic synthesis stage. In principle, our approach could be extended to later stages in the hardware compilation flow, such as post-placement and routing. In future work, we plan to extend VeriEQ's detection capability beyond BDBs and logic synthesis to cover a broader range of simulation and synthesis errors, including performance- and resource-related anomalies across multiple stages of the hardware design flow. This enhancement would further improve the comprehensiveness and practical value of VeriEQ as a general testing framework for hardware design tools.

9 Related Work

Verilog Simulators and Synthesizers Testing: Some prior works have focused on testing Verilog simulators and synthesizers. For example, TransFuzz [49] and SynFuzz [43] perform differential testing on randomly generated gate-level netlists across different synthesizers to detect BDBs. Verismith [23] and VlogHammer [72] generate random behavioral-level Verilog code and employ formal equivalence checking to verify that the synthesized output matches the original design. VeriXmith [78] mutates code samples from an existing Verilog corpus and uses formal verification to detect semantic mismatches. LegoHDL [66] constructs high-level hardware designs by assembling models from existing cyber-physical system (CPS) libraries, converts them into HDL code, and performs differential testing on the netlists synthesized by different tools. However, these approaches overlook the structural characteristics of previously discovered BDBs, which limits their effectiveness in exposing deeper semantic issues. As a result, the majority of bugs detected by these tools are simple crashes in simulators and synthesizers, with only a small portion revealing more severe behavioral deviation errors. Furthermore, they do not leverage Verilog's unique capability

for inlined deviation checking through multi-module integration, leading to suboptimal testing efficiency and missed opportunities for detecting high impact bugs.

Tools Using Metamorphic Testing: The idea of metamorphic testing has been widely applied across various domains to uncover subtle bugs in different systems [2, 4, 7, 13, 21, 28, 65]. For instance, Equivalence Modulo Inputs [68] has been successfully used for C/C++ compiler validation by generating program variants that preserve behavior on a fixed input but expose behavioral deviations through divergent outputs. DeepXplore [41] tests deep neural networks by exploiting metamorphic relations such as invariance to image perturbations to detect errors. MT-DLComp [64] uses MT to identify mis-compilations in deep learning compilers. PolyJuice [77] constructs semantically equivalent computation graphs via equality saturation and leverages their structural diversity to validate complex tensor compiler pipelines However, these approaches overlook the semantic intricacies of bitwidth and signedness in Verilog. Applying such methods directly to Verilog simulators and synthesizers testing may break the validity of metamorphic relations, thereby undermining the effectiveness of the testing process.

Compilers Testing: There has been extensive research on compiler testing. Some of these works focus on checking the correctness of peephole optimizations and avoiding issues caused by undefined behavior. For example, Alive [36] introduces a domain-specific language for LLVM peephole optimizations, which formally models optimization rules and uses SMT solvers to automatically prove their correctness or find counterexamples while considering undefined behavior. Alive2 [35] further proposes an SMT-based bounded translation validation approach that precisely models LLVM undefined behavior and automatically checks refinement between the original and optimized IR. Peek [39] presents a framework for expressing, verifying, and executing meaning-preserving assembly-level transformations in CompCert [11], where proving a set of local properties is sufficient to guarantee the correctness of global transformations. In contrast, VeriEQ targets behavioral deviation bugs in Verilog synthesizers and simulators caused by various optimizations or implementation errors, rather than focusing only on peephole optimizations. In Verilog, X-values may introduce semantic inconsistencies across different tools, which can interfere with bug detection. VeriEQ addresses this problem using metamorphic testing, which eliminates the impact of different ways that simulators and synthesizers handle X-values.

Main Difference: Different from the above work, VeriEQ detects BDBs in Verilog simulators and synthesizers by applying semantic-preserving transformations to Verilog code. It first generates high-quality seed programs using templates derived from historical BDBs, then performs equivalence-preserving transformations. These transformations are guided by a set of bitwidth- and signedness-aware transformation rules, applied only when their constraints are satisfied. To improve efficiency, VeriEQ introduces an inlined deviation checking approach, which embeds multiple semantically equivalent modules into a single testbench for simultaneous simulation, reducing redundancy and increasing throughput.

10 Conclusion

In this paper, we present VeriEQ, an automated framework that uses metamorphic testing to detect BDBs by generating semantically equivalent Verilog programs. First, we design code templates based on patterns from historical BDBs to generate high-quality seed programs. Second, we apply circuit transformations with bit-width and signedness constraints to create semantically equivalent variants. Finally, we use inlined deviation checking by placing multiple equivalent modules in one testbench to improve efficiency. We implemented VeriEQ and evaluated it on four mainstream Verilog simulators and synthesizers, successfully uncovering 33 previously unknown bugs including 29 BDBs, as well as 4 hang bugs identified as additional findings. In contrast, other state-of-the-art tools were only able to detect 1-7 of the bugs that VeriEQ discovered.

Acknowledgments

We would like to express our sincere gratitude to the anonymous reviewers for their valuable feedback and constructive comments on this paper. This research is sponsored by the NSFC Program (No. 62525207).

Data-Availability Statement

The artifact that supports Section 5 has been implemented and is available on Zenodo [67]. It includes the usable tools and four test targets.

References

[1] 2006. IEEE Standard for Verilog Hardware Description Language. *IEEE Std 1364-2005 (Revision of IEEE Std 1364-2001)* (2006), 1–590. doi:10.1109/IEEESTD.2006.99495

[2] Jubril Gbolahan Adigun, Linus Eisele, and Michael Felderer. 2022. Metamorphic testing in autonomous system simulations. In *2022 48th Euromicro Conference on Software Engineering and Advanced Applications (SEAA)*. IEEE, 330–337. doi:10.1109/SEAA56994.2022.00059

[3] Elad Alon, Krste Asanović, Jonathan Bachrach, and Borivoje Nikolić. 2019. Open-source EDA tools and IP, a view from the trenches. In *Proceedings of the 56th Annual Design Automation Conference 2019*. 1–3. doi:10.1145/3316781.3323481

[4] Domenico Amalfitano, Misael Júnior, Anna Rita Fasolino, and Marcio Delamaro. 2025. A GUI-based Metamorphic Testing technique for detecting authentication vulnerabilities in Android mobile apps. *Journal of Systems and Software* (2025), 112364. doi:10.1016/j.jss.2025.112364

[5] Janick Bergeron. 2007. *Writing testbenches using SystemVerilog*. Springer Science & Business Media. doi:10.1007/0-387-31275-7

[6] Alan Mishchenko Satrajit Chatterjee Robert Brayton and Peichen Pan. [n. d.]. Integrating Logic Synthesis, Technology Mapping, and Retiming. ([n. d.]).

[7] Nazanin Bayati Chaleshtari, Fabrizio Pastore, Arda Goknil, and Lionel C Briand. 2023. Metamorphic testing for web system security. *IEEE Transactions on Software Engineering* 49, 6 (2023), 3430–3471. doi:10.1109/TSE.2023.3256322

[8] Tsong Y Chen, Shing C Cheung, and Shiu Ming Yiu. 2020. Metamorphic testing: a new approach for generating next test cases. *arXiv preprint arXiv:2002.12543* (2020). doi:10.48550/arXiv.2002.12543

[9] Tsong Yueh Chen, Fei-Ching Kuo, Huai Liu, Pak-Lok Poon, Dave Towey, TH Tse, and Zhi Quan Zhou. 2018. Metamorphic testing: A review of challenges and opportunities. *ACM Computing Surveys (CSUR)* 51, 1 (2018), 1–27. doi:10.1145/3143561

[10] Kelvin Chung. 2025. fsm_expand deleting design if state register is not initialised. https://github.com/YosysHQ/yosys/issues/5014.

[11] CompCert. 2025. The CompCert project investigates the formal verification of realistic compilers usable for critical embedded software. https://compcert.org/.

[12] Samuel Coward, George A Constantinides, and Theo Drane. 2022. Automatic datapath optimization using e-graphs. In *2022 IEEE 29th Symposium on Computer Arithmetic (ARITH)*. IEEE, 43–50. doi:10.1109/ARITH54963.2022.00016

[13] Anurag Dwarakanath, Manish Ahuja, Samarth Sikand, Raghotham M Rao, RP Jagadeesh Chandra Bose, Neville Dubash, and Sanjay Podder. 2018. Identifying implementation bugs in machine learning based image classifiers using metamorphic testing. In *Proceedings of the 27th ACM SIGSOFT international symposium on software testing and analysis*. 118–128. doi:10.1145/3213846.3213858

[14] flaviens. 2023. Wrong simulation result with adders and single-bit exclusive or gates. https://github.com/verilator/verilator/issues/4709.

[15] flaviens. 2024. Arithmetics after a flip-flop causes wrong runtime value in specific circumstances. https://github.com/steveicarus/iverilog/issues/1074.

[16] flaviens. 2024. Fix lost NOT in const-bit-op-tree. https://github.com/verilator/verilator/issues/4857.

[17] flaviens. 2024. Misoptimization of wide shifts. https://github.com/YosysHQ/yosys/issues/4164.

[18] flaviens. 2024. Runtime value issue under some specific conditions. https://github.com/verilator/verilator/issues/4864.

[19] FSY369. 2023. UB in celledges shift handling. https://github.com/YosysHQ/yosys/issues/4844.

[20] Steve Golson et al. 1994. State machine design techniques for Verilog and VHDL. *Synopsys Journal of High-Level Design* 9, 1-48 (1994), 12.

[21] Pablo Gómez-Abajo, Pablo C Cañizares, Alberto Núñez, Esther Guerra, and Juan de Lara. 2023. Automated engineering of domain-specific metamorphic testing environments. *Information and Software Technology* 157 (2023), 107164. doi:10.1016/j.infsof.2023.107164

[22] Soheil Hashemi, Hokchhay Tann, and Sherief Reda. 2018. BLASYS: Approximate logic synthesis using Boolean matrix factorization. In *Proceedings of the 55th Annual Design Automation Conference*. 1–6. doi:10.1145/3195970.3196001

[23] Yann Herklotz and John Wickerson. 2020. Finding and understanding bugs in FPGA synthesis tools. In *Proceedings of the 2020 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*. 277–287. doi:10.1145/3373087.3375310

[24] Xu Huang, He Xin, Lintao Liu, and Luncai Liu. 2015. Eliminate the X-Optimization during RTL Verification. In *3rd International Conference on Mechatronics, Robotics and Automation*. Atlantis Press, 381–385. doi:10.2991/icmra-15.2015.75

[25] Peter Jamieson and Jonathan Rose. 2005. A verilog RTL synthesis tool for heterogeneous FPGAs. In *International Conference on Field Programmable Logic and Applications, 2005.* IEEE, 305–310. doi:10.1109/FPL.2005.1515739

[26] Jiabao1125. 2024. Incorrect result from shift operation. https://github.com/steveicarus/iverilog/issues/1165.

[27] Jiabao1125. 2024. Incorrect result from shift operation. https://github.com/steveicarus/iverilog/issues/1165.

[28] Andrei Lascu. 2022. *Metamorphic Testing for Software Libraries and Graphics Compilers*. Ph. D. Dissertation. Imperial College London.

[29] Andrei Lascu, Matt Windsor, Alastair F Donaldson, Tobias Grosser, and John Wickerson. 2021. Dreaming up metamorphic relations: Experiences from three fuzzer tools. In *2021 IEEE/ACM 6th International Workshop on Metamorphic Testing (MET)*. IEEE, 61–68. doi:10.1109/MET52542.2021.00017

[30] Vu Le, Mehrdad Afshari, and Zhendong Su. 2014. Compiler validation via equivalence modulo inputs. *ACM Sigplan Notices* 49, 6 (2014), 216–226. doi:10.1145/2594291.2594334

[31] lejar. 2024. Proof engine is going into wrong case in case statement. https://github.com/YosysHQ/yosys/issues/4317.

[32] lejar. 2024. Proof engine is going into wrong case in case statement. https://github.com/YosysHQ/yosys/issues/4317.

[33] Rui Li, Huai Liu, Pak-Lok Poon, Dave Towey, Chang-Ai Sun, Zheng Zheng, Zhi Quan Zhou, and Tsong Yueh Chen. 2024. Metamorphic Relation Generation: State of the Art and Visions for Future Research. *arXiv preprint arXiv:2406.05397* (2024). doi:10.48550/arXiv.2406.05397

[34] Andreas Lööw. 2025. The simulation semantics of synthesisable Verilog. *Proceedings of the ACM on Programming Languages* 9, OOPSLA1 (2025), 1295–1320. doi:10.1145/3720484

[35] Nuno P Lopes, Juneyoung Lee, Chung-Kil Hur, Zhengyang Liu, and John Regehr. 2021. Alive2: bounded translation validation for LLVM. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation*. 65–79. doi:10.1145/3453483.3454030

[36] Nuno P Lopes, David Menendez, Santosh Nagarakatte, and John Regehr. 2015. Provably correct peephole optimizations with alive. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation*. 22–32. doi:10.1145/2737924.2737965

[37] Phu X Mai, Fabrizio Pastore, Arda Goknil, and Lionel Briand. 2020. Metamorphic security testing for web systems. In *2020 IEEE 13th International Conference on Software Testing, Validation and Verification (ICST)*. IEEE, 186–197. doi:10.1109/ICST46399.2020.00028

[38] Steven Meyer. 2016. CVC Verilog Compiler–Fast Complex Language Compilers Can be Simple. *arXiv preprint arXiv:1603.08059* (2016). doi:10.48550/arXiv.1603.08059

[39] Eric Mullen, Daryl Zuniga, Zachary Tatlock, and Dan Grossman. 2016. Verified peephole optimizations for CompCert. In *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation*. 448–461. doi:10.1145/2980983.2908109

[40] Rachit Nigam, Samuel Thomas, Zhijing Li, and Adrian Sampson. 2021. A compiler infrastructure for accelerator generators. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*. 804–817. doi:10.1145/3445814.3446712

[41] Kexin Pei, Yinzhi Cao, Junfeng Yang, and Suman Jana. 2017. Deepxplore: Automated whitebox testing of deep learning systems. In *proceedings of the 26th Symposium on Operating Systems Principles*. 1–18. doi:10.1145/3132747.3132785

[42] Richard Rudell. 1996. Tutorial: Design of a logic synthesis system. In *33rd Design Automation Conference Proceedings, 1996*. IEEE, 191–196. doi:10.1145/240518.240554

[43] Raghul Saravanan, Sudipta Paria, Aritra Dasgupta, Venkat Nitin Patnala, Swarup Bhunia, et al. 2025. SynFuzz: Leveraging Fuzzing of Netlist to Detect Synthesis Bugs. *arXiv preprint arXiv:2504.18812* (2025). doi:10.48550/arXiv.2504.18812

[44] Fabian Schuiki, Andreas Kurth, Tobias Grosser, and Luca Benini. 2020. LLHD: A multi-level intermediate representation for hardware description languages. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*. 258–271. doi:10.1145/3385412.3386024

[45] sdjasj. 2025. Incorrect assignment behavior with -fno-expand. https://github.com/verilator/verilator/issues/5972.

[46] Sergio Segura, Gordon Fraser, Ana B Sanchez, and Antonio Ruiz-Cortés. 2016. A survey on metamorphic testing. *IEEE Transactions on software engineering* 42, 9 (2016), 805–824. doi:10.1109/TSE.2016.2532875

[47] Sergio Segura, Dave Towey, Zhi Quan Zhou, and Tsong Yueh Chen. 2018. Metamorphic testing: Testing the untestable. *IEEE Software* 37, 3 (2018), 46–53. doi:10.1109/MS.2018.2875968

[48] Wilson Snyder. 2004. Verilator and systemperl. In *North American SystemC Users' Group, Design Automation Conference*, Vol. 79. 122–148.
[49] Flavien Solt and Kaveh Razavi. 2025. Lost in Translation: Enabling Confused Deputy Attacks on EDA Software with TransFuzz. *USENIX Security. Paper= https://comsec. ethz. ch/wp-content/files/mirtl_sec25. pdfURL= https://comsec. ethz. ch/mirtl* (2025).
[50] Yuhe Sun, Zuohua Ding, Hongyun Huang, Senhao Zou, and Mingyue Jiang. 2023. Metamorphic testing of relation extraction models. *Algorithms* 16, 2 (2023), 102. doi:10.3390/a16020102
[51] Tze Sin Tan and Bakhtiar Affendi Rosdi. 2014. Verilog HDL simulator technology: a survey. *Journal of Electronic Testing* 30 (2014), 255–269. doi:10.1007/s10836-014-5449-5
[52] Alessandro Tempia Calvino. 2024. *Technology Mapping and Optimization Algorithms for Logic Synthesis of Advanced Technologies*. Ph. D. Dissertation. EPFL.
[53] Mike Turpin and Principal Verification Engineer. 2003. The Dangers of Living with an X (bugs hidden in your Verilog). In *Synopsys Users Group Meeting*.
[54] Hans Van der Schoot, Anoop Saha, Ankit Garg, and Krishnamurthy Suresh. 2011. Off to the races with your accelerated SystemVerilog testbench. In *Design and Verification Conference and Exhibition (DVCon)*.
[55] João Victor Amorim Vieira, Luiza de Melo Gomes, Rafael Sumitani, Raissa Maciel, Augusto Mafra, Mirlaine Crepalde, and Fernando Magno Quintão Pereira. 2025. Bottom-Up Generation of Verilog Designs for Testing EDA Tools. *arXiv preprint arXiv:2504.06295* (2025). doi:10.48550/arXiv.2504.06295
[56] Jianxin Wang, Xiangze Chang, Chaoen Xiao, and Lei Zhang. [n. d.]. Research on the Principle and Architecture of Icarus Verilog System. ([n. d.]). doi:10.23977/acss.2024.080604
[57] Yiting Wang, Wanghao Ye, Ping Guo, Yexiao He, Ziyao Wang, Bowei Tian, Shwai He, Guoheng Sun, Zheyu Shen, Sihan Chen, et al. 2025. SymRTLO: Enhancing RTL Code Optimization with LLMs and Neuron-Inspired Symbolic Reasoning. *arXiv preprint arXiv:2504.10369* (2025). doi:10.48550/arXiv.2504.10369
[58] whitequark. 2020. CXXRTL, a Yosys Simulation Backend. https://tomverbeure.github.io/2020/08/08/CXXRTL-the-New-Yosys-Simulation-Backend.html.
[59] whitequark. 2025. yosys - Yosys Open SYnthesis Suite. https://github.com/YosysHQ/yosys/tree/main/backends/cxxrtl.
[60] Stephen Williams. 2025. The ICARUS Verilog Compilation System. https://github.com/steveicarus/iverilog.
[61] Stephen Williams and Michael Baxter. 2002. Icarus verilog: open-source verilog more than a year later. *Linux Journal* 2002, 99 (2002), 3. doi:10.5555/513581.513584
[62] Clifford Wolf, Johann Glaser, and Johannes Kepler. 2013. Yosys-a free Verilog synthesis suite. In *Proceedings of the 21st Austrian Workshop on Microelectronics (Austrochip)*, Vol. 97.
[63] wsnyder. 2025. Verilator. https://github.com/verilator/verilator.
[64] Dongwei Xiao, Zhibo Liu, Yuanyuan Yuan, Qi Pang, and Shuai Wang. 2022. Metamorphic testing of deep learning compilers. *Proceedings of the ACM on Measurement and Analysis of Computing Systems* 6, 1 (2022), 1–28. doi:10.1145/3508035
[65] Xiaoyuan Xie, Joshua WK Ho, Christian Murphy, Gail Kaiser, Baowen Xu, and Tsong Yueh Chen. 2011. Testing and validating machine learning classifiers by metamorphic testing. *Journal of Systems and Software* 84, 4 (2011), 544–558. doi:10.1016/j.jss.2010.11.920
[66] Zhihao Xu, Shikai Guo, Guilin Zhao, Peiyu Zou, Xiaochen Li, and He Jiang. 2024. A novel HDL code generator for effectively testing FPGA logic synthesis compilers. *arXiv preprint arXiv:2407.12037* (2024). doi:10.48550/arXiv.2407.12037
[67] Zhen Yan. 2026. VeriEQ. doi:10.5281/zenodo.18454209
[68] Xuejun Yang, Yang Chen, Eric Eide, and John Regehr. 2011. Finding and understanding bugs in C compilers. In *Proceedings of the 32nd ACM SIGPLAN conference on Programming language design and implementation*. 283–294. doi:10.1145/1993498.1993532
[69] Xufeng Yao, Yiwen Wang, Xing Li, Yingzhao Lian, Ran Chen, Lei Chen, Mingxuan Yuan, Hong Xu, and Bei Yu. 2024. Rtlrewriter: Methodologies for large models aided rtl code optimization. In *Proceedings of the 43rd IEEE/ACM International Conference on Computer-Aided Design*. 1–7. doi:10.1145/3676536.3676775
[70] YikeZhou. 2023. CXXRTL: incorrect result of shl operator. https://github.com/YosysHQ/yosys/issues/3820.
[71] YikeZhou. 2023. write_smt2: bugs caused by the » operator under certain circumstances. https://github.com/YosysHQ/yosys/issues/3748.
[72] Yosys. 2019. VlogHammer. https://github.com/YosysHQ/VlogHammer.
[73] Yosys-HQ. 2018. Optimization passes. https://yosyshq.readthedocs.io/projects/yosys/en/0.40/using_yosys/synthesis/opt.html.
[74] YosysHQ. 2025. yosys - Yosys Open SYnthesis Suite. https://github.com/YosysHQ/yosys.
[75] zachjs. 2024. unsigned port connection sign extends. https://github.com/steveicarus/iverilog/issues/1099.
[76] Huaien Zhang, Yu Pei, Junjie Chen, and Shin Hwei Tan. 2023. Statfier: Automated testing of static analyzers via semantic-preserving program transformations. In *Proceedings of the 31st ACM Joint European Software Engineering*

Conference and Symposium on the Foundations of Software Engineering. 237–249. doi:10.1145/3611643.3616272

[77] Chijin Zhou, Bingzhou Qian, Gwihwan Go, Quan Zhang, Shanshan Li, and Yu Jiang. 2024. PolyJuice: Detecting Mis-compilation Bugs in Tensor Compilers with Equality Saturation Based Rewriting. *Proceedings of the ACM on Programming Languages* 8, OOPSLA2 (2024), 1309–1335. doi:10.1145/3689757

[78] Yike Zhou, Yanyan Jiang, and Jian Lu. 2025. Unveiling Cross-checking Opportunities in Verilog Compilers. *ACM Transactions on Design Automation of Electronic Systems* (2025). doi:10.1145/3715325

Received 2025-10-10; accepted 2026-02-17