RL4HDL: Code Diversity Guided FPGA Logic Synthesis Compiler Testing Via Reinforcement Learning

Zhihao Xu*†
Southeast University
Nanjing, Jiangsu, China
Hosea.xu@seu.edu.cn

Hui Zeng*
Dalian Maritime University
Dalian, Liaoning, China
zh974757632@dlmu.edu.cn

Hui Li
Dalian Maritime University
Dalian, Liaoning, China
li_hui@dlut.edu.cn

Qian Ma
Dalian Maritime University
Dalian, Liaoning, China
maqian@dlmu.edu.cn

Furui Zhan
Dalian Maritime University
Dalian, Liaoning, China
izfree@dlmu.edu.cn

Shikai Guo‡
Dalian Maritime University
Dalian, Liaoning, China
shikai.guo@dlmu.edu.cn

Abstract

Logic Synthesis compilers play an indispensable role in Field Programmable Gate Arrays (FPGAs) design, translating Register-Transfer Level (RTL) designs into gate-level netlists. Defects in logic synthesis compilers may affect the security of final hardware implementations. Existing methods explored several test case generation and mutation approaches for logic synthesis testing but are limited by simple corpora and random strategies. It leads to the redundancy of test cases and further prevents the exploration of defects in logic synthesis compilers. To address the problem, we propose RL4HDL, a new method which uses Reinforcement Learning to guide logic synthesis compiler testing. Specifically, RL4HDL includes three main components, the Diversity Metamorphic Construction Component (DMC) and the Reinforcement Learning Control Component (RLC) and the Differential Testing Component(DTC). To generate diverse HDL test cases, the DMC applies a series of domain-specific metamorphic transformations to a set of seed designs, producing new test cases that are semantically equivalent to the originals while exhibiting richer structural patterns. Then, to reduce redundancy in the generated test cases, the RLC adaptively selects metamorphic transformations based on a reward of embedding-based structural diversity. It is realized by a lightweight bandit-style agent that automatically adjusts the selection probabilities of different transformations so as to avoid repeatedly generating similar test cases. Finally, to reveal defects in logic synthesis compilers, the DTC compiles each candidate HDL test case with multiple logic Synthesis compilers and compares the resulting netlists and simulation traces. If there are any discrepancies between different compilers or between semantically equivalent test cases compiled by the same compiler, the DTC reports these cases as candidate compiler defects. Comprehensive experiments conducted over a three-month period demonstrate the practical effectiveness of our method. We discovered 20 unique defects, including 12 previously unreported defects, which have been confirmed by official developers. Moreover, our approach has been shown to generate more distinct test cases than the state-of-the-art method.

Keywords

FPGA Logic Synthesis, Reinforcement Learning, Differential Testing, Compiler Testing

ACM Reference Format:
Zhihao Xu, Hui Zeng, Hui Li, Qian Ma, Furui Zhan, and Shikai Guo. 2026. RL4HDL: Code Diversity Guided FPGA Logic Synthesis Compiler Testing Via Reinforcement Learning. In *Proceedings of Design Automation Conference (DAC '26).* ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 Introduction

As highly flexible and programmable integrated circuits, Field-Programmable Gate Arrays (FPGAs) play a critical role in modern electronic design automation (EDA) [7, 9, 11, 13, 18, 20]. They have been widely used in aerospace, electronics, and medical devices [8, 9, 15, 19, 21, 24, 25]. In FPGA design, the logic synthesis compiler is a critical component, and it translates register-transfer level (RTL) descriptions into optimized hardware implementations [12, 23, 29]. Defects in logic synthesis compilers can propagate into the final hardware, leading not only to functional failures but also to exploitable security vulnerabilities [22]. Therefore, it is important to ensure the correctness of logic synthesis compilers.

Several methods have been proposed to test logic synthesis compilers, including Verismith [12], EvoHDL (also referred to as Lego-HDL) [29], and VERMEI [31]. Specifically, Verismith and EvoHDL are fuzz-based testing frameworks. Verismith constructs valid HDL programs by generating abstract syntax trees and reported 11 defects over a two-year period. In contrast, EvoHDL first creates Simulink models and then translates them to HDL via MATLAB's HDL Coder. By leveraging the breadth of the Simulink block library, it can produce more structurally complex test cases than Verismith [29]. However, such fuzzing-based methods inherit the randomness of fuzz testing, which leads to redundant and highly similar test cases. To diversify the logical complexity of generated test cases, VERMEI introduces a mutation-based approach. By identifying and pruning non-executing code in HDL test cases, VERMEI generates equivalent variants and compares the synthesis results to detect defects [31]. However, VERMEI focuses mainly on inserting

---

*Both authors contributed equally to this research.
†Zhihao Xu is also with the Faculty of Information Technology, Monash University, Melbourne, Australia, and School of Information Science and Technology, Dalian Maritime University, Dalian, China.
‡Corresponding author.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
DAC '26, July 26–29, 2026, Long Beach, CA, USA
© 2026 ACM.
ACM ISBN 978-x-xxxx-xxxx-x/YY/MM
https://doi.org/10.1145/nnnnnnn.nnnnnnn

and manipulating such non-executing code but neglects the executing code in HDL test cases. Furthermore, because its mutations are still random, VERMEI can generate many similar variants, leading to redundancy in the test cases. Therefore, to improve the capability and efficiency of logic synthesis testing, it is crucial to reduce such redundancy and increase the structural complexity of test cases.

We propose RL4HDL, a novel method utilizing Reinforcement Learning (RL) to reduce the redundancy and improve the complexity of test cases for logic synthesis compiler testing. Specifically, RL4HDL consists of three main components, the Diversity Metamorphic Construction (DMC) component, the Reinforcement Learning Control (RLC) component, and the Differential Testing Component (DTC). Due to the rich Simulink block library which can capture a wide range of realistic design patterns [29], RL4HDL chooses to build test cases from converting Simulink models to HDL code rather than directly from HDL fuzzing corpus. The Diversity Metamorphic Construction (DMC) component first uses a Simulink model generator to construct a set of valid seed models. For each seed model, DMC applies a series of domain-specific metamorphic transformations that preserve the original program semantics. Then DMC invokes MATLAB HDL Coder to translate both the original and transformed model into HDL implementations. We refer to the HDL programs obtained from these transformations as *mutants*. By working in this way, DMC can produce numerous HDL mutants that remain semantically equivalent to their seed while exhibiting more complex structural patterns than those in existing fuzzing corpus. On top of DMC, the RLC component guides the choice of metamorphic transformations and reduces redundancy in the generated test cases. It uses a lightweight bandit-style scheduler that adapts the selection of seed models and transformations based on feedback about bug novelty and structural diversity. Concretely, an embedding model is used to analyze the generated test cases and quantify how different each mutant is from previously generated tests form the reward signal for the scheduler. Through this way, RLC can effectively guides the test case generation process toward structurally diverse mutants while avoiding repeatedly producing highly similar test cases. Finally, the DTC checks whether the generated mutants actually reveal compiler defects. It compiles each original test case and its mutants with multiple logic synthesis compilers, compares the resulting netlists and simulation traces. If it observes any discrepancies either between different compilers or between semantically equivalent test cases compiled by the same compiler, DTC reports these cases as candidate compiler defects.

Over a three-month testing period, our method identified 20 unique defects, among which 12 were previously undiscovered. All reported defects were confirmed by the official developers of the respective FPGA logic synthesis compilers. Our embedding evaluation also shows that RL4HDL produces more structurally distinct and less redundant test cases than a state-of-the-art fuzzing-based baseline. The main contributions of our work are as follows:

(1) We propose RL4HDL, a novel testing method for FPGA logic synthesis compilers. By introducing a Reinforcement-Learning Control (RLC) component, RL4HDL can generate structurally diverse test cases while reducing redundancy in the generated tests.
(2) We conduct extensive experiments to evaluate the effectiveness of RL4HDL in both HDL test generation and defect detection. And over a three-month period, RL4HDL discovered 20 unique defects, 12of which were previously unknown.
(3) To support reproducibility and further research, we release our implementation as an open-source tool on GitHub [5].

2 Related Work

2.1 Defect Detection in Logic Synthesis Tools

Recent testing methods for logic synthesis compilers can be roughly divided into two categories: fuzz-based and mutation-based. Fuzz-based methods include VlogHammer [1], Verismith [12], and EvoHDL [29]. These approaches aim to generate valid and diverse test cases to exercise logic synthesis compilers. VlogHammer and Verismith directly produce Verilog programs. Compared with the purely random generation in VlogHammer, Verismith first constructs Abstract Syntax Trees (ASTs) and then generates syntactically valid Verilog test cases from the ASTs. However, the relatively simple code corpora used by VlogHammer and Verismith limit the structural diversity of their tests and make it difficult to thoroughly stress logic synthesis compilers. To address this, EvoHDL generates Simulink models instead of Verilog code directly, leveraging the rich Simulink block library as a more diverse corpus, and then uses MATLAB HDL Coder to translate the models into HDL test cases. Compared with VlogHammer and Verismith, EvoHDL can generate more diverse test cases and support multiple HDLs (VHDL, Verilog, and SystemVerilog). Experiments show that EvoHDL is more effective than Verismith, it detected 16 defects in three months, whereas Verismith reported 11 defects over two years.

For mutation-based methods, VERMEI has been proposed to generate more complex test cases for logic synthesis compiler testing [31]. VERMEI introduces several mutation strategies, such as identifying and deleting non-executing code. However, these operations mainly increase the syntactic diversity of test cases and can not improve the logical diversity of the executing code. Moreover, cause the mutations are still applied in a random manner, VERMEI may generate many similar variants of the same seed, leading to redundant test cases. Therefore, designing mutation strategies that can systematically enhance the functional and timing diversity of test cases while avoiding such redundancy remains a challenge in logic synthesis testing.

3 Framework Of RL4HDL

In this section, we first present an overview of the framework of RL4HDL. We then explain the Diversity Metamorphic Construction (DMC) Component, the Reinforcement-Learning Control (RLC) Component, and the Differential Testing Component (DTC).

3.1 Overview

The framework of RL4HDL is illustrated in Figure 1. RL4HDL starts from a set of Simulink models, referred to as seed Simulink models. The DMC first analyzes each seed Simulink model and identifies potential transformation regions, following similar procedures to prior work [10, 17]. Then the DMC introduces specific metamorphic transformations, including MUX Intensive Transformation, Arithmetic Path Extension, Control-Fanout Restructuring, and Hierarchical-Sharing Reorganization, into the identified transformation regions. The choice of which transformation to apply at each region is governed by a policy distribution over these strategies, which is provided and continuously updated by the RLC based on feedback from previous tests. After applying the metamorphic transformations, DMC generates new Simulink models, which we call *mutant Simulink models*. DMC then uses MATLAB HDL Coder to translate both the seed Simulink models and the mutant Simulink models into HDL programs. We refer to the HDL generated from seed models as *seed HDL test case*, and the HDL generated from

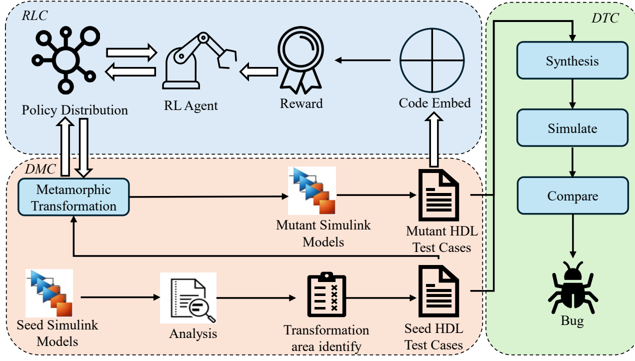

Figure 1: The framework and Basic Component of RL4HDL.

mutant models as *mutant HDL test case*. The RLC sits on top of DMC and adaptively adjusts the policy distribution based on a diversity reward. Specifically, we use a code embedding model to map each mutant HDL test case to a vector in an embedding space and compute distance to previously generated mutant HDL test cases. The distance is used as a diversity score. RLC takes this diversity score as the reward and updates the policy distribution, which can prevent RL4HDL from producing large numbers of redundant and similar test cases. Finally, the DTC synthesizes both the seed HDL test cases and the mutant HDL test cases with multiple logic synthesis compilers and simulates the resulting netlists. By comparing the netlists and simulation traces, the DTC detects discrepancies and reports the corresponding cases as candidate defects.

3.2 Diversity Metamorphic Construction

To generate diverse Simulink models and finally obtain diverse HDL test cases, DMC introduces four specific metamorphic transformations, MUX Intensive Transformation, Arithmetic Path Extension, Control Fanout Restructuring, and Hierarchical Sharing Reorganization. Before applying these metamorphic transformations, DMC first identifies potential transformation areas in each seed Simulink model and collects model information in these areas, including the involved blocks, signals, and state variables as well as their data types and connectivity, following similar procedures to prior work [10, 17]. This information is later used to check the applicability of each transformation and to ensure that the transformed model remains semantically equivalent to the original one. Based on the collected local information, DMC checks which transformations are applicable at each potential transformation area.

We design four classes of metamorphic transformations that preserve the functional semantics of the seed simulink models while changing its structure, MUX Intensive Transformation (MIT), Arithmetic Path Extension (APE), Control Fanout Restructuring (CFR), and Hierarchical Sharing Reorganization (HSR). In the following, we briefly describe each transformation and explain how it can increase diversity in the generated tests and thereby help expose defects in logic synthesis compilers.

MUX Intensive Transformation. Given a Simulink model $S = (B, E)$, where B is the set of blocks and E is the set of signal connections, RL4HDL considers a combinational region $R \subseteq B$ whose outputs are a set of signals $Y = \{y_1, \ldots, y_m\}$. For each $y \in Y$, the semantics of R induces a function

$$y = f_y(x_1, \ldots, x_n),$$

where $x_1, \ldots, x_n$ are the input signals of this region (coming from blocks outside R).

The MUX Intensive Transformation (MIT) constructs an equivalent implementation of R that contains additional multiplexer logic but preserves the original input–output behavior. Formally, for each output $y \in Y$ RL4HDL generates two expressions $e_y^{(1)}$ and $e_y^{(2)}$ such that

$$\forall (x_1, \ldots, x_n). \quad e_y^{(1)}(x_1, \ldots, x_n) = e_y^{(2)}(x_1, \ldots, x_n) = f_y(x_1, \ldots, x_n).$$

In practice, $e_y^{(1)}$ and $e_y^{(2)}$ are obtained by applying semantics-preserving rewrites to the original computation of y, such as inserting and then canceling arithmetic operations (e.g., "$+k$" followed by "$-k$"), duplicating parts of the computation, or introducing identity Boolean operations. These rewrites change the block structure but not the functional result.

MIT then replaces the original computation of y by a 2-to-1 multiplexer:

$$y = \mathrm{MUX}(c_y,\ e_y^{(1)}(x_1, \ldots, x_n),\ e_y^{(2)}(x_1, \ldots, x_n)),$$

where c_y is a fresh Boolean control signal that does not affect the functional result, because both MUX inputs are guaranteed to be equal for all reachable inputs. For every input valuation of $(x_1, \ldots, x_n)$, the transformed region still computes the same outputs Y. After HDL code generation, it can generate mutant HDL test cases containing more multiplexer logic and deeper MUX trees, which can help expose defects in logic synthesis compilers that are sensitive to complex MUX networks.

Arithmetic Path Extension. Given a Simulink model S and an arithmetic computation path that computes an output signal

$$z = f(x_1, \ldots, x_n)$$

using a sequence of add, subtract, multiply, shift, or bitwise-logic blocks, APE rewrites this path by inserting semantically neutral arithmetic operations. Concretely, APE constructs an expression g such that

$$\forall (x_1, \ldots, x_n). \quad g(x_1, \ldots, x_n) = f(x_1, \ldots, x_n),$$

by adding and then canceling the same constant, applying a shift and its inverse, or applying the same XOR with a constant twice, while respecting the data type range to avoid overflow. For example, APE may replace

$$z = f(x_1, \ldots, x_n)$$

by

$$z = \big(f(x_1, \ldots, x_n) + k\big) - k$$

for an appropriate constant k, or insert a longer chain of intermediate arithmetic blocks that ultimately computes the same z. For every input valuation, the transformed path still produces the same output value, so the model remains semantically equivalent. After HDL code generation, however, the corresponding logic cones become deeper and the arithmetic networks more complex, which increases structural diversity and can help expose defects in logic synthesis compilers related to arithmetic simplification, constant propagation, and timing-driven optimization.

Control Fanout Restructuring. Given a Simulink model $S = (B, E)$ and a Boolean control signal c that guards a set of conditional blocks $B_c = \{b_1, \ldots, b_k\} \subseteq B$, CFR rewrites the control network around c while preserving its semantics. CFR constructs a set of derived control signals

$$C' = \{c'_1, \ldots, c'_k\}$$

such that for all input valuations σ,

$$\forall i \in \{1, \ldots, k\}. \quad c'_i(\sigma) = c(\sigma).$$

Each c'_i is obtained from c by applying simple Boolean identities, possibly together with additional local signals d that cancel out, for example

$$c'_1 = c \wedge 1, \quad c'_2 = c \vee 0, \quad c'_3 = (c \wedge d) \vee (c \wedge \neg d),$$

all of which evaluate to c for any σ. CFR then rewires the guarded blocks so that b_i is controlled by c'_i instead of c, and may insert extra AND/OR blocks so that c first fans out through several intermediate signals and then reconverges.

Because each c'_i is logically equivalent to c, the activation condition of every $b_i \in B_c$ is unchanged for all inputs, and the overall model remains semantically equivalent. After HDL code generation, however, the synthesized design contains more complex control networks with higher fanout and additional reconvergence points, which increases the structural diversity of the test cases.

Hierarchical Sharing Reorganization. Hierarchical-Sharing Reorganization (HSR) operates at the subsystem (hierarchy) level. Consider a Simulink model S with two subsystems F_1 and F_2 that implement identical or structurally similar computations over inputs $\mathbf{u} = (u_1, \ldots, u_m)$ and produce outputs $\mathbf{v} = (v_1, \ldots, v_\ell)$. RL4HDL write their semantics as

$$F_1 : \mathbf{u} \mapsto \mathbf{v}_1, \qquad F_2 : \mathbf{u} \mapsto \mathbf{v}_2,$$

and assume that for all input valuations $\mathbf{u}$,

$$\mathbf{v}_1(\mathbf{u}) = \mathbf{v}_2(\mathbf{u}),$$

i.e., F_1 and F_2 are functionally equivalent with the same interface.

HSR performs factoring of shared computation. Suppose F_1 and F_2 each contain an isomorphic subgraph of blocks G_1 and G_2 that implement a common subfunction

$$g : \mathbf{u}' \mapsto \mathbf{w}$$

over some subset of inputs $\mathbf{u}'$ and intermediate outputs $\mathbf{w}$. HSR creates a new subsystem G that implements g once, removes G_1 and G_2 from F_1 and F_2, and inserts calls to G with the same inputs $\mathbf{u}'$ and outputs $\mathbf{w}$ as in the original connections. The external inputs and outputs of F_1 and F_2 are kept unchanged.

Because G computes the same subfunction g and the wiring of the top-level inputs and outputs of F_1 and F_2 is preserved, the overall input–output behavior of S remains unchanged for every input valuation. After HDL code generation, however, the resulting design has a different module hierarchy and shared-subsystem structure, which increases the variety of hierarchical and sharing patterns seen by the synthesis tools and can help expose defects in logic synthesis compilers related to hierarchy flattening, module inlining, and cross-module resource sharing.

3.3 Reinforcement-Learning Control

The process of RLC is shown in Algorithm 1. RLC takes as input the total number of rounds T, the learning rate η, the baseline update factor β, and the number of neighbors K_{nn} used in the KNN-based diversity reward (header of Algorithm 1). It maintains a preference value $h_a(t)$ and the corresponding probability $p_a(t)$ for each metamorphic strategy $a \in \{1, \ldots, 4\}$, together with a reward baseline $\bar{r}$ and an embedding corpus $\mathcal{Z}$.

RLC first initializes the preferences and initial strategy probabilities (line 1) and sets the baseline and embedding corpus to zero and empty, respectively (line 2). It then iterates over testing rounds

Algorithm 1: RLC with KNN-based Diversity Reward

Input: rounds T, learning rate η, baseline step β, neighbors K_{nn}
Output: strategy distributions $\mathbf{p}(t)$, $t = 1, \ldots, T$

1. initialize $h_a(1) \leftarrow 0$, $p_a(1) \leftarrow \frac{1}{4}$ for all $a \in \{1, \ldots, 4\}$
2. $\bar{r} \leftarrow 0$, $\mathcal{Z} \leftarrow \emptyset$
3. **for** $t \leftarrow 1$ **to** T **do**
4. $p_a(t) \leftarrow \dfrac{\exp(h_a(t))}{\sum_{b=1}^{4} \exp(h_b(t))}$ for all a
5. $(\mathbf{z}_t,\ n_1(t), \ldots, n_4(t)) \leftarrow \text{DMC}(\mathbf{p}(t))$
6.   **if** $|\mathcal{Z}| = 0$ **then**
7. $r_t \leftarrow 0$
8.   **else**
9. $\mathcal{N}_t \leftarrow \text{KNN}(\mathbf{z}_t, \mathcal{Z}, K_{\text{nn}})$
10. $r_t \leftarrow \dfrac{1}{|\mathcal{N}_t|} \sum_{\mathbf{z}_i \in \mathcal{N}_t} \text{dist}(\mathbf{z}_t, \mathbf{z}_i)$
11. $\mathcal{Z} \leftarrow \mathcal{Z} \cup \{\mathbf{z}_t\}$
12. $\bar{r} \leftarrow (1 - \beta)\bar{r} + \beta r_t$
13. $M_t \leftarrow \sum_{a=1}^{4} n_a(t)$
14.   **if** $M_t > 0$ **then**
15.   **for** $a \leftarrow 1$ **to** 4 **do**
16. $w_a(t) \leftarrow \dfrac{n_a(t)}{M_t}$
17. $h_a(t+1) \leftarrow h_a(t) + \eta\, w_a(t)\, (r_t - \bar{r})$
18.   **else**
19. $h_a(t+1) \leftarrow h_a(t)$ for all a
20. $p_a(t+1) \leftarrow \dfrac{\exp(h_a(t+1))}{\sum_{b=1}^{4} \exp(h_b(t+1))}$ for all a
21. **return** $\{\mathbf{p}(t)\}_{t=1}^{T}$

$t = 1, \ldots, T$ (line 3). At the beginning of each round, the current strategy distribution $\mathbf{p}(t)$ is computed from the preferences $h_a(t)$ using a softmax (line 4). This distribution is passed to the DMC component, abstracted as $\text{DMC}(\mathbf{p}(t))$, which generates a new mutant HDL test case and returns its embedding vector $\mathbf{z}_t$ together with the usage counts $n_a(t)$ of each strategy in this round (line 5).

Next, RLC computes a diversity-based reward r_t from $\mathbf{z}_t$. If no history is available ($|\mathcal{Z}| = 0$), it sets $r_t = 0$ (lines 6–7). Otherwise, it finds the K_{nn} nearest neighbors $\mathcal{N}_t$ of $\mathbf{z}_t$ in $\mathcal{Z}$ and defines r_t as the average distance from $\mathbf{z}_t$ to these neighbors (lines 8–10). The new embedding $\mathbf{z}_t$ is inserted into $\mathcal{Z}$ and the reward baseline $\bar{r}$ is updated using an exponential moving average with factor β (lines 11–12). RLC then updates the strategy preferences according to how much each strategy participated in this round. It first computes the total number of applied transformations $M_t = \sum_{a=1}^{4} n_a(t)$ (line 13). If $M_t > 0$, it derives the participation weight $w_a(t) = n_a(t)/M_t$ for each strategy and updates

$$h_a(t+1) = h_a(t) + \eta\, w_a(t)\, (r_t - \bar{r}),$$

(lines 14–17); otherwise, it keeps h_a unchanged (lines 18–19). Finally, the strategy distribution for the next round, $\mathbf{p}(t+1)$, is recomputed from $h_a(t+1)$ via softmax (line 20). In this way, strategies that more frequently contribute to mutants with high diversity rewards gradually obtain higher preferences and probabilities in $\mathbf{p}(t)$, guiding DMC toward generating more diverse test cases.

3.4 Differential Testing

The DTC checks whether logic synthesis compilers preserve the semantics of the seed and mutant HDL test cases. For each seed HDL test case and its mutants, DTC runs several mainstream synthesis tools (Yosys [28], Vivado [4], and Quartus [3]) to obtain gate-level

netlists, and then simulates these netlists under the same input stimuli (generated by HDL Coder [29]) using a common HDL simulator (Icarus Verilog [2]). DTC first compares the simulation traces produced by different compilers for the same HDL test case; any cross-compiler inconsistency indicates that at least one compiler is incorrect. It then compares, for each compiler, the traces of a seed HDL test case and its semantically equivalent mutant HDL test cases; any mismatch suggests that the compiler does not preserve equivalence under the applied transformations. Whenever such discrepancies are detected, DTC records the corresponding test case and compiler as a candidate logic synthesis defect.

4 Evaluation

In this section, three experiments are conducted to evaluate the effectiveness of RL4HDL. Specifically, our evaluation aims at answering the following Research Questions (RQs).

(1) *How effective is RL4HDL compared with SOTA methods?*
(2) *Can RL4HDL detect new logic synthesis compiler defects?*
(3) *How effective is RLC in defect detection, and what is its cost?*

4.1 Evaluation Setup and Settings

RL4HDL is implemented in MATLAB, and the source code and experimental data are available on GitHub [5]. All experiments are conducted on a machine running Ubuntu 22.04 (64-bit) with an Intel Core i9 CPU at 2.10 GHz and 120 GB RAM. For the logic synthesis compilers under test (LSCUT), we use Yosys 0.58+132 (git sha1 12cb8e951, g++ 11.4.0-1ubuntu1~22.04.2, `-fPIC -O3`), Vivado 2024.2, and Quartus 24.2, which are the latest versions available during our testing period. We use Icarus Verilog 11.0 (stable) as the HDL simulator to run all generated netlists under a common set of input stimuli. For the parameters in our method in Algorithm 1, following the prior work and our preliminary tuning, we fix the number of neighbours to $K_{\text{nn}} = 5$ for stable behavior [14, 16, 30]. Then we fix $\beta = 0.05$ to reduce variance without introducing bias, because it is standard practice in policy-gradient methods [6, 27, 32]. And for code embedding, we use the pre-trained code encoder which is CodeT5 to obtain vector representations of programs [26]. Following prior work on compiler testing [10, 12, 17, 29], we treat two candidate logic synthesis defects as the same defect if they trigger identical error messages and stack traces, or if they are confirmed by the tool developers to be the same reason. All defects reported in this paper are unique.

4.2 Answer to RQ1

To compare the effectiveness of RL4HDL against state-of-the-art methods, we faithfully reproduce the latest logic synthesis testing frameworks and follow their recommended settings. In particular, we configure the generators so that the produced HDL test cases contain about 700–1000 lines of code, as suggested in the original papers [12, 29, 31]. We run each method, including RL4HDL and the baselines, under the same testing budget of 200 CPU hours. We focus on two aspects.

(1) the number of unique defects uncovered in the logic synthesis compilers.
(2) the diversity of the generated HDL test cases.

To quantify structural diversity, we embed all generated HDL test cases into a vector space using our code embedding model, and for each test we compute its distance to the closest previously generated test. We then use the average nearest-neighbor distance D_{NN} as our diversity metric: a larger D_{NN} indicates that new tests are less clustered around existing ones and therefore less redundant.

Table 1: Defects detected by SOTA methods and RL4HDL within 200 CPU hours.

Method	Vivado	Yosys	Quartus	Icarus Verilog
VERMEI [31]	1	1	0	0
Verismith [12]	2	0	0	0
EvoHDL [29]	2	1	1	0
RL4HDL	2	3	1	1

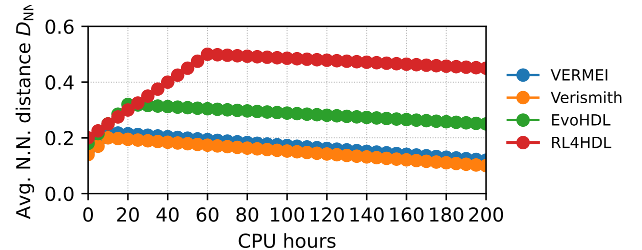

Figure 2: Evolution of test case diversity (D_{NN}) over 200 CPU hours.

For defect detection, as shown in Table 1, RL4HDL detects 2 unique defects in Vivado, matching EvoHDL and Verismith, while VERMEI finds only 1. On Yosys, RL4HDL finds 3 defects, whereas the baselines detect at most 1. For Quartus, RL4HDL and EvoHDL each uncover 1 defect and the other methods find none. On Icarus Verilog, only RL4HDL discovers a defect, caused by an implicit sensitivity list for an `always_*` construct that leads to a compiler crash; none of the baseline methods expose this issue, and we will describe this bug in more detail in the next subsection.

For the diversity of the generated HDL test cases, as shown in Figure 2, the average nearest-neighbor distance D_{NN} of the three baselines increases only slightly at the beginning and then quickly saturates at relatively low values. VERMEI and Verismith converge to the lowest D_{NN}, indicating that their later tests are strongly clustered around previously generated ones and therefore highly redundant, while EvoHDL maintains only a moderate diversity level. In contrast, the D_{NN} of our method keeps rising during the first 60 CPU hours and then stays at the highest level throughout the remaining time, showing that our RLC component effectively drives the metamorphic transformations to explore new regions of the test space instead of repeatedly producing similar HDL tests.

4.3 Answer to RQ2

To evaluate the effectiveness of RL4HDL in discovering new logic synthesis compiler defects, we ran our tool continuously for a three-month testing period from October 2024 to January 2025. As summarized in Table 2, RL4HDL exposed 20 unique defects in total, 12 of which were previously unknown. We classify each defect into two categories: Miscompilation (M) and Crash (C). A Miscompilation (M) defect means that the logic synthesis compiler finishes normally but produces an incorrect netlist, which manifests as a reproducible discrepancy in the simulation results. A Crash (C) defect means that the compiler terminates abnormally when processing the test case, for example with an internal error, assertion failure, or segmentation fault. We present two illustrative examples to explain

the defects found by RL4HDL and to illustrate why RL4HDL is able to uncover new defects.

```
1  -   always_latch begin
2  -     if (clkin_data[64])
3  -       celloutsig_1_10z = 22'h000000;
4  -     else if (clkin_data[32])
5  -       celloutsig_1_10z = in_data[190:169];
6  -   end
7  -   always_latch begin
8  -     if (!celloutsig_1_18z)
9  -       celloutsig_0_5z = 6'h00;
10 -     else if (!clkin_data[0])
11 -       celloutsig_0_5z = in_data[18:13];
12 -   end
```

Listing 1: Example of metamorphic strategies in Logic Operation Optimization

The first defect is a crash defect detected in **Iverilog#1286**. As shown in listing 1, the bug occurs because when a bit/part select is found in the implicit sensitivity list for an always_* construct, it is replaced by the entire variable. If there is more than one bit/part select from the same variable, it gets added to the list multiple times, eventually cause crash. The defect is hard for previous methods to expose because their generators rarely produce `always_*` blocks with several correlated bit/part selects from the same vector; even if such constructs appear, they are not systematically reinforced, so the probability of hitting the exact pattern is extremely low. In contrast, RL4HDL's metamorphic transformations (Control-Fanout Restructuring) naturally introduce multiple slices of the same bus signal into the control logic while preserving semantics, and the RLC keeps such structurally distinct patterns in the test distribution, making this crash much more likely to be triggered.

The second defect is a miscompilation detected in Yosys. As shown in Listing 2, the original RTL computes `out_data` by first assembling a temporary vector `shift_vec` from `data_low`, `flag_bit`, and `data_mid`, and then shifting it by `shift_amt`. In the optimized netlist, Yosys rewrites this pattern into a single concatenation-and-shift expression, but incorrectly replaces `data_low` with the constant `7'h00`. This faulty constant propagation changes the result whenever `data_low` is non-zero, so the synthesized netlist is no longer equivalent to the original RTL.

```
1 -   assign out_data = shift_vec << shift_amt;
2 -   assign shift_vec[6:0]  = data_low;
3 -   assign shift_vec[7]    = flag_bit;
4 -   assign shift_vec[9:8]  = data_mid[1:0];
5 -   assign shift_amt = ctrl_bus[17:8];
6 +   assign out_data = { data_mid[1:0], flag_bit, 7'h00 } <<
          ctrl_bus[17:8];
```

Listing 2: Example of metamorphic strategies in Logic Operation Optimization

This pattern is also hard for previous methods to expose. Their random generators seldom produce long arithmetic data paths that combine concatenation, bit-slicing, and shifts on shared signals. Even when such structures appear by chance, they are not systematically preserved or emphasized in later tests, so the probability of triggering this bug remains low. In contrast, RL4HDL's Arithmetic Path Extension transformation explicitly constructs this kind of shift-and-concatenation network while preserving semantics, and the RLC keeps these structurally distinct mutants in the test distribution, making the miscompilation more likely to be revealed.

4.4 Answer to RQ3

To isolate the impact of the Reinforcement-Learning Control (RLC) component, we compare RL4HDL with its variant RL4HDL-uniform, in which the four metamorphic strategies are always selected with a fixed uniform distribution (i.e., the bandit update in Algorithm 1 is disabled). Both variants share the same DMC and DTC components and are run under the same testing budget of 200 CPU hours.

Table 2: Defects found by RL4HDL in an evaluation period of three months.

#	Summary	Type	Fb	LSCUT
1	HARTNlUtil::isCarryInst error	C	K	Vivado
2	NNetC::singleDriver error	C	K	Vivado
3	HARTGLAddGen::regenerate error	C	K	Vivado
4	NPinC::parentModule error	C	K	Vivado
5	DFPin::disconnect error	C	K	Vivado
6	HARTTUpdateTNInstC::Cell error	C	K	Vivado
7	HARTXmsgWriter::Print error	C	K	Vivado
8	dot::openFile error	C	K	Vivado
9	GXorGen::bestSoln error	C	N	Vivado
10	ConstProp::reconnect error	C	N	Vivado
11	PrioMuxInfo::setPinArray error	C	N	Vivado
12	ConstProp::propagate error	C	N	Vivado
13	DFNode::calcConstantBinaryInt error	M	N	Vivado
14	HARTOptMux::createPartition error	M	N	Vivado
15	NDup::dupGlobalNames error	M	N	Vivado
16	NTargetLibC::findCell error	M	N	Vivado
17	NBaseModC::realModule error	M	N	Vivado
18	Sign extension of zero-width	M	N	Yosys
19	Translation Defect on latch block with incorrect assign value	M	N	Yosys
20	Assertion failure when using always_latch with constant bit selects	C	N	Iverilog

Fb = Feedback from official developers of LSCUT (N = new bug, K = known bug);

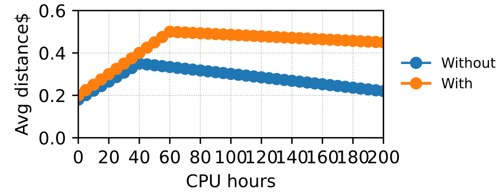

Figure 3: Diversity Distance

Table 3: Effectiveness and cost of RLC within 200 CPU hours.

Method	HDL tests	Defects	Time(s)
RL4HDL-uniform	61890↑	2 ↓	11.6335↑
RL4HDL (RLC)	57330↓	3↑	12.55↓

As shown in Figure 3, enabling RLC increases the structural diversity of the generated HDL test cases. Moreover, RLC does not degrade defect detection efficiency, so the additional runtime overhead it introduces is negligible in practice.

5 Conclusions and Future Work

In this paper, we presented RL4HDL, a reinforcement-learning guided testing framework for FPGA logic synthesis compilers that combines metamorphic HDL generation, diversity-aware bandit control, and differential testing. Our experiments show that RL4HDL generates more diverse HDL test cases and uncovers more unique defects than existing fuzzing and mutation based methods. In future work, we plan to extend RL4HDL to more logic synthesis compilers, explore stronger code representations and RL policies.

Acknowledgments

This work was supported by the National Natural Science Foundation of China (No.62472062, No.62572090), the Dalian Science and Technology Innovation Fund project (No.2024JJ12GX022), and the Applied Basic Research Project of Liaoning Province (No.2025JH2/101330109).

References

[1] 2019. VlogHammer. https://github.com/YosysHQ/VlogHammer.

[2] 2023. ICARUS Verilog. https://github.com/steveicarus/iverilog.

[3] 2023. Quartus. https://www.intel.com/content/www/us/en/products/details/fpga/development-tools/quartus-prime.html.

[4] 2023. Xilinx Vivado. https://support.xilinx.com/s/?language=en_US.

[5] 2024. *RL4HDL*. https://anonymous.4open.science/r/Lin-Hunter-F284/

[6] Arash Ahmadian, Chris Cremer, Matthias Gallé, Marzieh Fadaee, Julia Kreutzer, Olivier Pietquin, Ahmet Üstün, and Sara Hooker. 2024. Back to basics: Revisiting reinforce style optimization for learning from human feedback in llms. *arXiv preprint arXiv:2402.14740* (2024).

[7] Juan Jose Rodriguez Andina, Eduardo De la Torre Arnanz, and María Dolores Valdés Peña. 2017. *FPGAs: fundamentals, advanced features, and applications in industrial electronics*. CRC Press.

[8] NK Anish, B Kowshick, and S Moorthi. 2013. Ethernet based industry automation using FPGA. In *2013 Africon*. IEEE, 1–4.

[9] Iuliana Chiuchisan. 2013. A new FPGA-based real-time configurable system for medical image processing. In *2013 E-Health and Bioengineering Conference (EHB)*. IEEE, 1–4.

[10] Shafiul Azam Chowdhury, Sohil Lal Shrestha, Taylor T Johnson, and Christoph Csallner. 2020. SLEMI: Equivalence modulo input (EMI) basedmutation of cps models for finding compiler bugs in simulink. In *IEEE International Conference on Software Engineering(ICSE)*. 335–346.

[11] P Dillinger, JF Vogelbruch, J Leinen, S Suslov, R Patzak, H Winkler, and K Schwan. 2005. FPGA based real-time image segmentation for medical systems and data processing. In *14th IEEE-NPSS Real Time Conference, 2005*. IEEE, 5–pp.

[12] Yann Herklotz and John Wickerson. 2020. Finding and Understanding Bugs in FPGA Synthesis Tools. In *FPGA '20: The 2020 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays, Seaside, CA, USA, February 23-25, 2020*, Stephen Neuendorffer and Lesley Shannon (Eds.). ACM, 277–287. https://doi.org/10.1145/3373087.3375310

[13] Noé Monterrosa, Jason Montoya, Fredy Jarquín, and Carlos Bran. 2016. Design, development and implementation of a UAV flight controller based on a state machine approach using a FPGA embedded system. In *2016 IEEE/AIAA 35th Digital Avionics Systems Conference (DASC)*. IEEE, 1–8.

[14] Mirco Mutti, Lorenzo Pratissoli, and Marcello Restelli. 2021. Task-agnostic exploration via policy gradient of a non-parametric state entropy estimate. In *Proceedings of the AAAI conference on artificial intelligence*, Vol. 35. 9028–9036.

[15] Ahmed Sanaullah, Chen Yang, Yuri Alexeev, Kazutomo Yoshii, and Martin C Herbordt. 2018. Real-time data analysis for medical diagnosis using FPGA-accelerated neural networks. *BMC bioinformatics* 19 (2018), 19–31.

[16] Younggyo Seo, Lili Chen, Jinwoo Shin, Honglak Lee, Pieter Abbeel, and Kimin Lee. 2021. State entropy maximization with random encoders for efficient exploration. In *International conference on machine learning*. PMLR, 9443–9454.

[17] S.Guo, H.Jiang, Z.Xu, X.Li, Z.Ren, Z.Zhou, and R.Chen. 2022. Detecting simulink compiler bugs via controllable zombie blocks mutation. In *Joint European Software Eng. Conf. and Symposium on the Foundations of Software Eng.(ESEC/FSE)*. 1061–1072.

[18] Satish Sharma, Sunil Kulkarn, Vijaykumar Pujari, M Vanitha, and P Lakshminarsimhan. 2010. FPGA implementation of M-PSK modulators for satellite communication. In *2010 International Conference on Advances in Recent Technologies in Communication and Computing*. IEEE, 136–139.

[19] Shanker Shreejith and Suhaib A Fahmy. 2014. Extensible FlexRay communication controller for FPGA-based automotive systems. *IEEE transactions on vehicular technology* 64, 2 (2014), 453–465.

[20] Kanwar Jit Singh and PA Subrahmanyam. 1995. Extracting RTL models from transistor netlists. In *Proceedings of IEEE International Conference on Computer Aided Design (ICCAD)*. IEEE, 11–17.

[21] Valery Sklyarov, Iouliia Skliarova, and Alexander Sudnitson. 2011. FPGA-based systems in information and communication. In *2011 5th international conference on application of information and communication technologies (AICT)*. IEEE, 1–5.

[22] 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).

[23] Shailja Thakur, Baleegh Ahmad, Hammond Pearce, Benjamin Tan, Brendan Dolan-Gavitt, Ramesh Karri, and Siddharth Garg. 2024. Verigen: A large language model for verilog code generation. *ACM Transactions on Design Automation of Electronic Systems* 29, 3 (2024), 1–31.

[24] Brunel Happi Tietche, Olivier Romain, Bruce Denby, and Francois De Dieuleveult. 2012. FPGA-based simultaneous multichannel FM broadcast receiver for audio indexing applications in consumer electronics scenarios. *IEEE Transactions on Consumer Electronics* 58, 4 (2012), 1153–1161.

[25] Pál Varga, László Kovács, Tamás Tóthfalusi, and Péter Orosz. 2015. C-GEP: 100 Gbit/s capable, FPGA-based, reconfigurable networking equipment. In *2015 IEEE 16th International Conference on High Performance Switching and Routing (HPSR)*. IEEE, 1–6.

[26] Yue Wang, Weishi Wang, Shafiq Joty, and Steven CH Hoi. 2021. CodeT5: Identifier-aware Unified Pre-trained Encoder-Decoder Models for Code Understanding and Generation. In *EMNLP 2021-2021 Conference on Empirical Methods in Natural Language Processing, Proceedings*. Association for Computational Linguistics (ACL), 8696–8708.

[27] Lex Weaver and Nigel Tao. 2013. The optimal reward baseline for gradient-based reinforcement learning. *arXiv preprint arXiv:1301.2315* (2013).

[28] Clifford Wolf, Johann Glaser, and Johannes Kepler. 2013. Yosys-a free Verilog synthesis suite. In *Proceedings of the 21st Austrian Workshop on Microelectronics (Austrochip)*. 97.

[29] Zhihao Xu, Shikai Guo, Guilin Zhao, Peiyu Zou, Xiaochen Li, and He Jiang. 2025. A Novel HDL Code Generator for Effectively Testing FPGA Logic Synthesis Compilers. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 44, 11 (2025), 4405–4418. https://doi.org/10.1109/TCAD.2025.3565488

[30] Mingqi Yuan, Bo Li, Xin Jin, and Wenjun Zeng. 2022. Rewarding episodic visitation discrepancy for exploration in reinforcement learning. *arXiv preprint arXiv:2209.08842* (2022).

[31] Yi Zhang, He Jiang, Xiaochen Li, Shikai Guo, Peiyu Zou, and Zun Wang. 2025. A Novel Mutation Based Method for Detecting FPGA Logic Synthesis Tool Bugs. arXiv:2508.15536 [cs.SE] https://arxiv.org/abs/2508.15536

[32] Tingting Zhao, Hirotaka Hachiya, Gang Niu, and Masashi Sugiyama. 2011. Analysis and improvement of policy gradient estimation. *Advances in Neural Information Processing Systems* 24 (2011).