

# RLConcolic: Enhancing Concolic Testing via Multi-Step Reinforcement Learning

Yan Tan\*, Xiangchen Meng\* and Yangdi Lyu†

*Microelectronics Thrust, The Hong Kong University of Science and Technology (Guangzhou)*

Email: {ytan910, xmeng027}@connect.hkust-gz.edu.cn, yangdilyu@hkust-gz.edu.cn

**Abstract**—Chip manufacturing relies on rigorous verification to prevent costly design errors before fabrication and deployment. Branch coverage, a key metric for Register-Transfer Level (RTL) validation, ensures thorough testing of decision points in the design. However, RTL designs often contain numerous hard-to-activate branches, which can lead to hidden bugs and security vulnerabilities. While concolic testing addresses the memory explosion issues associated with formal methods, it relies on heuristics that may get stuck in local optima. In this paper, we propose a novel approach that reformulates Concolic testing as a reinforcement learning problem. Our method utilizes the agent that takes into account RTL structural characteristics and runtime simulation states to select strategies for guiding the simulation path toward target branches. Experimental results demonstrate that our approach effectively directs simulations toward branch targets, reduces search redundancy, and significantly increases branch coverage, thereby improving the efficiency and effectiveness of the test generation process.

**Index Terms**—Concolic, RTL Verification, Reinforcement Learning

## I. INTRODUCTION

In the modern large-scale integrated circuit (IC) design process, verification plays an essential role in identifying design vulnerabilities and ensuring the security and reliability of chips. As the complexity of IC designs continues to grow, ensuring correctness under all possible conditions becomes a significant challenge. To evaluate the thoroughness and effectiveness of a verification process, branch coverage [1] is utilized as one of the key metrics. This ensures that every conditional statement in the RTL code is exercised for both its true and false outcomes. This metric helps prevent bugs and vulnerabilities that might arise from inactive or untested branches.

Fuzzing [2] is a prevalent defect detection method that generates millions of test cases to simulate designs. Its scalability makes it well-suited for testing large and complex designs. For example, DifuzzRTL [3] leverages register coverage to guide the fuzzing process in a large hardware design such as a RISC-V CPU core. However, while fuzzing is a scalable method, its reliance on metric-guided mutations often makes it inefficient in achieving high coverage. It often fails to explore critical, hard-to-activate branches, which may contain significant design vulnerabilities.

To activate these hard-to-activate branches, formal methods offer a more deterministic solution. Techniques like model

checking [4]–[7] employ mathematical proofs to rigorously verify a design against given properties, which can be used to exercise hard-to-activate branches. However, formal methods encounter a significant scalability challenge known as the state space explosion problem, where the complexity of verification increases exponentially with the size of the design. While recent advancements, such as the piecewise composition in Sylvania [8], aim to mitigate this issue, applying these techniques to large, industrial-scale designs remains a formidable challenge.

To address the state explosion problem while maintaining high coverage, semi-formal methods have been introduced by integrating the scalability of fuzzing with the directness of formal methods. Concolic testing [9]–[11], as an example of a semi-formal approach, integrates simulation and symbolic execution to generate directed test cases. This strategy leverages the efficiency of simulation in large-scale designs and the guidance of formal verification on critical paths to improve the effectiveness.

Despite these advantages, the effectiveness of Concolic testing is often limited by two new challenges: the path explosion problem, caused by the immense number of possible execution paths, and the high computational cost of the SMT solver. The efficiency of Concolic testing, therefore, relies on intelligent path exploration heuristics to minimize SMT solver invocations. Current RTL frameworks often rely on static, manually-designed heuristics [11] that cannot adapt their decisions based on the dynamic state of the simulation. This reveals a critical need for a framework that can autonomously learn an optimal exploration policy.

To address this challenge, we introduce RLConcolic, a novel framework that reformulates Concolic testing as a dynamic, closed-loop learning problem. Our approach integrates a flexible Concolic test engine with an adaptive reinforcement learning (RL) agent that intelligently steers the path exploration process. Instead of relying on the static, manually-designed heuristics of prior work, our RL agent autonomously learns and continually refines its path selection policy at runtime. This learning process is driven by a novel reward function, engineered from a combination of static RTL structural features and runtime register values. As a result, RLConcolic can adapt its strategy based on the exploration history, learning from both successes and failures to effectively navigate the state space and activate critical, hard-to-activate branches. The main contributions of our paper are summarized as follows:

- 1) We introduce RLConcolic, the first scalable Concolic

This work was supported by the National Natural Science Foundation of China under Grant #62402412.

\*These authors contributed equally to this work.

†Corresponding author: yangdilyu@hkust-gz.edu.cn

testing framework<sup>1</sup> that is able to handle large and complex RTL Verilog designs like RISC-V cores.

- 2) We design a universal reward function that integrates static RTL structural details and dynamic runtime register values to guide path exploration in the reinforcement learning-based agent.
- 3) We employ a Multi-step Deep Q-Network algorithm to overcome the temporal credit assignment problem, enabling the agent to learn complex, long-term policies necessary for reaching hard-to-activate branches.
- 4) Experimental results demonstrate that our framework significantly improves both branch coverage and efficiency compared to state-of-the-art tools, with 84.7% branch coverage on the Rocket Chip.

## II. BACKGROUND AND RELATEDWORK

### A. Concolic Testing

Concolic testing is a semi-formal testing generation technique that combines concrete simulation and symbolic execution. It is widely used in both software testing [9], [10] and hardware verification [11], [12]. Unlike formal methods, in which the number of possible states grows exponentially with the complexity and size of the design, Concolic testing iteratively explores a single execution path and continuously guides the path toward the target branch.

While Concolic testing mitigates the state explosion problem of formal methods, it introduces a new challenge: the path exploration problem. For a given target, millions of paths may need to be explored before the target is activated. As each path exploration requires a call to an SMT solver, the overall efficiency of Concolic testing becomes a major concern for achieving high coverage. As a result, an effective path exploration heuristic is crucial for the effectiveness and efficiency of Concolic testing [11]. However, the heuristics in existing frameworks are typically statically designed and may not always be effective. While some recent work, such as that by Zheng *et al.* [13], has explored adaptive path exploration, the effectiveness of these approaches is limited when designs become large. This highlights a need for a more adaptable approach based on historical data and feedback to effectively explore critical paths within the state space.

### B. Reinforcement Learning

Reinforcement learning has demonstrated its effectiveness across a variety of domains by leveraging trial-and-error learning principles and optimal control frameworks from dynamic programming [14]. In RL, an agent interacts with an environment, observes the current state, and uses this information to make decisions that maximize a cumulative reward. The integration of deep learning with Q-learning [15] has led to the development of Deep Q-Network (DQN) [16], [17], which has achieved impressive results in complex tasks.

In recent years, these advanced RL techniques have been successfully applied in hardware verification and optimization to automate tasks that traditionally required significant human

intervention. For example, Shi *et al.* [18] applied a DQN to the problem of Test Point Insertion (TPI) in logic built-in self-test (LBIST) configurations to enhance test coverage. Recently, hardware fuzzing frameworks such as HFL [19] and GenHuzz [20] employ RL agents to learn policies for generating effective test programs. While these RL-based methods show promise in fuzzing, their applicability to semi-formal approaches remains an open research question.

## III. METHODOLOGY

Traditional Concolic testing is dependent on static, manually crafted heuristics for path exploration, which often fail to adapt effectively to complex designs. To mitigate this limitation, we propose RLConcolic, an innovative framework that integrates reinforcement learning into the RTL verification process. The core contribution of our work is the replacement of fixed, handcrafted heuristics with a dynamic, self-adapting RL agent, which learns an optimal policy for guided path exploration.

### A. Preliminaries: The Concolic Testing Workflow

Our framework is built upon a Concolic testing engine that follows the workflow depicted in Fig. 1. The process begins by simulating an RTL design with an initial input vector. During this simulation, the engine records the constraints of the executed path. After the simulation, the system checks if the target branch has been covered. If the branch remains uncovered, the engine selects an alternative branch to explore. The constraints for this new path are then passed to an SMT solver. If the solver finds a solution, it generates a new input vector to drive the next simulation cycle. However, the system then confronts a critical strategic decision that significantly influences its efficiency and effectiveness: which alternative branch should be prioritized for exploration?

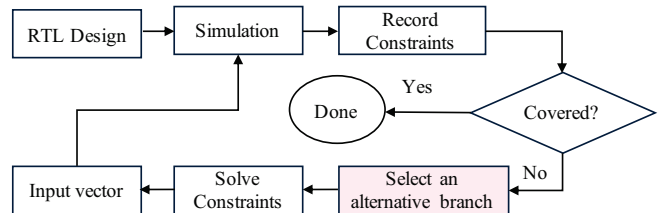


Fig. 1: The Workflow of a Concolic Testing Engine

Conventional methods rely on a fixed heuristic to resolve this. For instance, the approach in [11] ranks alternative branches based on a pre-computed structural distance within the Control Flow Graph (CFG), employing a greedy strategy to select the nearest one. The fundamental shortcoming of this approach lies in its “blindness” to the design’s runtime dynamics. A branch that appears structurally close in a static graph may be practically unreachable due to complex register state conditions that must be met. This “blind” decision-making leads to redundant exploration and frequently fails when confronted with hard-to-activate branches requiring sophisticated state manipulation.

<sup>1</sup><https://github.com/HKUSTGZ-MICS-LYU/RLConcolic>

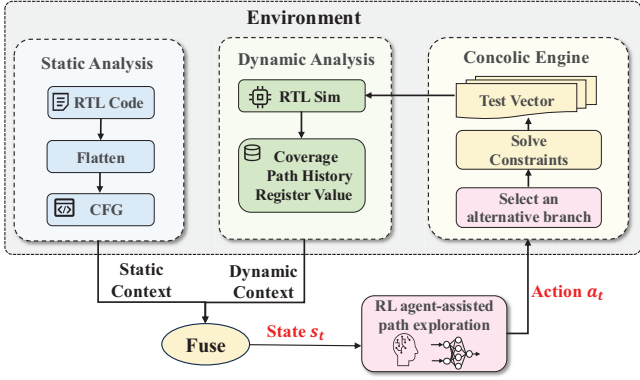


Fig. 2: RL agent-assisted path exploration

### B. Reinforcement Learning for Intelligent Path Exploration

To address the limitations of static heuristics, we reformulate the critical branch selection task as a reinforcement learning problem. Our approach replaces pre-defined heuristic rules with a dynamic RL agent that learns an optimal exploration policy through direct interaction with the simulation environment.

1) *State*: A key shortcoming of traditional heuristics is their static nature, relying on fixed strategies from pre-computed information while ignoring the real-time dynamic state. To overcome this, our RL agent utilizes a composite state space ( $S$ ) that fuses static structural information with dynamic runtime data. This approach provides the agent with a comprehensive perception, where the state ( $s \in S$ ) observed at each decision point is a rich feature vector that provides a holistic context. As shown in Fig. 2, it is composed of two information classes:

**Dynamic Context** This component provides the agent with crucial runtime information, including the path history, real-time coverage, and the values of all critical registers. The dynamic context gives the agent the runtime awareness that is absent in conventional heuristics, allowing the agent to react to the design’s dynamic conditions and direct the simulation toward hard-to-reach corner cases.

**Static Context** This component provides the agent with a structural awareness of the design. The CFG serves as a natural and standard representation for this purpose, as it models all possible execution paths. In addition to the CFG, we calculate the topological distance between the current branch and the target branch in the static context.

To form the final input vector for the DQN, these diverse pieces of information are numerically encoded and concatenated. Specifically, the path history is represented as a fixed-length vector of the last  $K$  branch IDs, and the coverage information is encoded as a bit-vector representing the coverage status of all branches. These elements, combined with the scalar CFG distance and the vector of critical register values, form a flat feature vector, which serves as the complete state representation  $s_t$ . Based on this comprehensive state, our RL agent autonomously learns and refines its path selection policy at runtime, achieving truly adaptive exploration.

2) *Action*: After gaining a comprehensive awareness of the state, the agent must translate its perception into actionable guidance for the Concolic engine. This is achieved through its

action selection mechanism, which intelligently automates the critical **Select an alternative branch** step shown in Fig. 1.

We formally define the action space ( $A$ ) as the set of all candidate branches at any given state. An agent’s action ( $a \in A$ ) corresponds to the selection of one of these branches. The agent aims to choose the action that maximizes the probability of covering a target branch.

To achieve this, the agent utilizes a policy network to assess the value of each potential action. Specifically, for a given state  $s$ , this network outputs a predicted Q-value ( $Q(s, a)$ ) for every candidate action  $a$ . This Q-value represents the expected future cumulative reward for taking action  $a$  in the current state. Subsequently, the Concolic engine then uses these Q-values to construct a dynamic priority branch queue, directing its exploration in descending order of their predicted Q-values. This mechanism replaces the traditional heuristic methods with a dynamic, learning-based policy that adaptively adjusts from experience.

3) *Reward*: To provide the agent with a precise learning signal, we design a reward function that quantifies the proximity to a target branch following each simulation trace. The function is designed so that the reward increases as the agent gets closer to the target.

The static distance  $\Delta_{D_{static}}(\mathbf{P}_t, T)$  measures the shortest topological distance from the current path  $\mathbf{P}_t$  to the target branch  $T$ . However, this metric alone is insufficient to guarantee the triggering of conditional branches, as their execution also depends on runtime factors such as register states. To address this, we introduce the dynamic register distance  $\Delta_{D_{reg}}(t)$ , which quantifies the difference between the current and target register states by calculating the Hamming distance between their corresponding bit vectors. A smaller Hamming distance indicates that the simulation is closer to satisfying the necessary register conditions for the target branch. We normalize distances into proximity scores and sum them, so the reward increases as the path approaches the target. The complete reward function  $r_t$  is then defined as a combination of this dynamic distance and the static distance:

$$r_t = \left(1 - \frac{\Delta_{D_{static}}(\mathbf{P}_t, T)}{\max(\Delta_{D_{static}})}\right) + \left(1 - \frac{\Delta_{D_{reg}}(t)}{\max(\Delta_{D_{reg}})}\right) \quad (1)$$

To balance the contributions of static and dynamic register distances, we normalize the weights using the maximum static distance  $\max(\Delta_{D_{static}})$  and maximum dynamic distance  $\max(\Delta_{D_{reg}})$  that are observed until the current exploration.

### C. RL Agent-Assisted Path Selection via Multi-step Learning

The RL agent is designed to intelligently automate the critical “Select an alternative branch” step in the Concolic testing workflow. The agent learns an optimal policy through direct interaction with the simulation environment. While obtaining feedback and updating the RL agent during each cycle is straightforward in a simulation, reaching a hard-to-activate branch in RTL verification is rarely accomplished in a single move; it usually demands a long, carefully coordinated sequence of decisions. This makes the task especially difficult for conventional one-step learning agents, which struggle to

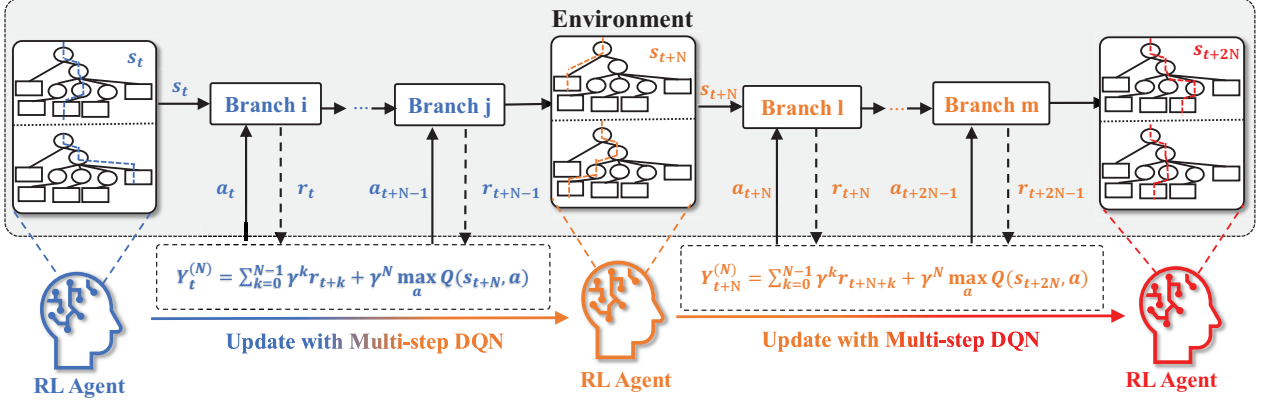


Fig. 3: RL Agent in Path Selection

attribute success to earlier actions when the reward is received only at the end of a long sequence. This leads to “short-sighted” learning, limiting the agent’s ability to develop the long-term strategies needed for thorough path exploration. To overcome this challenge, we employ a multi-step learning approach, as demonstrated in Fig. 3.

The detailed mechanism of our RL agent-assisted path selection is as follows:

- 1) **State Observation:** At the beginning of each decision cycle, the agent observes the current state,  $s_t$ . This state is a comprehensive representation that fuses static structural information from the CFG with dynamic simulation data, including path history, coverage metrics, and current register values.
- 2) **N-step Exploration:** Based on the observed state  $s_t$ , the agent uses its policy to select an action  $a_t$ , which represents a specific branch to explore. Instead of evaluating this single action immediately, our approach allows the agent and environment to interact for a sequence of  $N$  steps. During this exploration phase, the agent follows its policy, generating a trajectory of states, actions, and rewards  $(s_t, a_t, r_t, \dots, s_{t+N-1}, a_{t+N-1}, r_{t+N-1})$ , ultimately leading to the new state  $s_{t+N}$ .
- 3) **Action Evaluation:** After the  $N$ -step exploration, the agent evaluates the quality of the actions by looking at the entire sequence of outcomes, to adaptively refine its policy. This is done by calculating the  $N$ -step return target, **N-step target**  $Y_t^{(N)}$ , which aggregates the discounted rewards collected during the  $N$  steps and bootstraps the value of the resulting state  $s_{t+N}$ .

$$Y_t^{(N)} = \underbrace{\sum_{k=0}^{N-1} \gamma^k r_{t+k}}_{\text{N-step return}} + \underbrace{\gamma^N \max_a Q(s_{t+N}, a)}_{\text{bootstrapped value}} \quad (2)$$

This target value is composed of two parts: **N-step return**, which aggregates the real, observed rewards over a period of  $N$  steps, and the **bootstrapped value**, which is the Q-network’s own estimate of the long-term value from state  $s_{t+N}$ . Here,  $\gamma$  is the discount factor. This target provides a more stable and less biased estimate of the

true value of the state-action pair  $(s_t, a_t)$  compared to a single-step target.

- 4) **Update with Multi-step:** The calculated  $N$ -step target  $Y_t^{(N)}$  is then used to update the agent’s network. Specifically, the network’s parameters  $\theta$  are adjusted by minimizing a loss function between the network’s prediction  $Q(s_t, a_t; \theta)$  and the calculated accurate target  $Y_t^{(N)}$ .

$$L(\theta) = \mathbb{E} \left[ \left( Y_t^{(N)} - Q(s_t, a_t; \theta) \right)^2 \right] \quad (3)$$

This update directly propagates the reward information from the  $N$ -step trajectory back to the initial decision and improves its decision-making strategy.

- 5) **Iteration and Convergence:** The entire process from state observation to network update is carried out iteratively. With each iteration, the agent refines its network, improving its ability to estimate the long-term value of its actions. Over time, the agent’s policy converges towards an optimal strategy that discovers hard-to-activate paths and maximizes code coverage.

By iterating through this multi-step learning cycle, the agent can effectively credit previous actions based on their delayed consequences. This mechanism overcomes the myopia of single-step methods, enabling the agent to develop the sophisticated, multi-step strategies required to navigate vast state spaces and cover the hard-to-activate branches in complex RTL designs.

## IV. EXPERIMENTS

### A. Experimental Setup

We developed RLConcolic using Python. To obtain structural information, we parse RTL and extract it using Pyverilog [24]. For constraint solving within the Concolic engine, we utilize Z3 [25] for symbolic execution.

The core of our framework is a Deep Q-Network used to approximate the action-value function. The network architecture consists of five fully-connected layers. The input layer receives the state vector  $s_t$ , which is a concatenation of static and dynamic context features as described in Section III-B1. This is followed by three hidden layers, each containing 256 neurons and using the Rectified Linear Unit (ReLU) as the activation

TABLE I: Branch coverage comparison of Concolic methods across various benchmarks.

Benchmark	#Lines	#Branches <sup>‡</sup>	Concolic [21]	EBMC [22]	STSearch [23]	APE-FV [13]	RLConcolic	Impro./Concolic
b01	150	26	100%	100%	100%	100%	100%	0.00%
b06	131	23	100%	100%	100%	100%	100%	0.00%
b10	170	43	100%	100%	100%	100%	100%	0.00%
b11	118	35	94.28%	100%	100%	100%	100%	5.72%
b12	637	112	63.39%	88.39%	†	†	88.39%	25.00%
b15	553	183	53.00%	100%	†	†	100%	47.00%
b20	1510	456	†	†	†	†	90.35%	†
b21	1615	456	†	†	†	†	90.78%	†
DCache	391	48	89.58%	100%	97.87%	93.33%	93.75%	4.17%
ICache	190	24	100%	100%	100%	100%	100%	0.00%
Exception	587	64	92.18%	100%	98.71%	99.26%	100%	7.82%
Rocketchip	17715	1000	†	†	†	†	84.70%	†

<sup>‡</sup> Branch numbers are counted after flattening. Differences may arise due to variations in instrumentation or benchmark versions.

† indicates failure in test generation due to unsupported language features or excessive design complexity.

function. The output layer is a linear layer that produces a Q-value for each candidate branch (action), allowing the agent to select the most promising path for the next exploration cycle. For our learning algorithm, we employ a Multi-step DQN where the step-count N is set to 5. This value was determined through preliminary experiments on a validation set and was found to provide a good trade-off between the stability of the learning signal and the speed of credit assignment for reaching branches.

To evaluate the effectiveness and efficiency of RLConcolic, we compare its performance against state-of-the-art test generation technologies for RTL models: Concolic-based approach Concolic [21] and APE-FV [13], the fuzzing method STSearch [23], and the bounded model checker (EBMC) [6], [22]. We selected 13 benchmarks from the OpenRISC1200 [26] and ITC’99 [27] datasets. These benchmarks were chosen because they include targets with hard-to-activate branches for random testing and human experts. To further demonstrate the scalability and practical applicability of our method on a complex RTL design, we also evaluated these approaches on the RISC-V Rocket Chip core [28].

### B. Performance Comparison

To rigorously evaluate the effectiveness of our proposed method, we conducted a comprehensive performance comparison of RLConcolic against several state-of-the-art test generation tools. The results in Table I clearly demonstrate the performance advantages of RLConcolic. On small to medium-sized benchmarks, such as b01 and b10, RLConcolic achieves 100% coverage, matching the performance of other state-of-the-art tools. Furthermore, on benchmarks where the baseline Concolic method fails to achieve full coverage (e.g., b11, DCache, and Exception), RLConcolic delivers significant improvements. For instance, it successfully boosts the coverage of b11 from 94.28% to 100% and Exception from 92.18% to 100%, effectively resolving the branches that were difficult for the baseline method. For all these benchmarks except DCache, our method’s coverage is on par with the pure formal method, EBMC [22], showcasing its enhanced path exploration capability on moderately complex designs.

The advantage of RLConcolic becomes especially clear when applied to large and complex benchmarks. The design

complexity (including the richness of features and size of the design) of these benchmarks often poses significant challenges even for state-of-the-art test generation tools, as indicated by the unsupported cases (†) in the table. Traditional Concolic algorithms suffer from a combinatorial explosion of branch conditions, which drastically reduces their exploration efficiency on large-scale designs. In these challenging cases, RLConcolic provides dramatic improvements. For example, on the b12 and b15 circuits, RLConcolic boosts coverage from 63.39% and 53% to over 88%, respectively. In addition, while existing RTL test generation tools typically struggle with designs exceeding a thousand lines of code or involving complex grammars, RLConcolic demonstrates the ability to handle RTL models that are one to two orders of magnitude larger, showcasing a new level of scalability.

The most compelling evidence of its scalability is the result of the Rocketchip benchmark, a design with over 17,000 lines of code and 1,000 target branches. Existing techniques, such as Concolic [21], EBMC [22], STSearch [23], and APE-FV [13] have struggled with the scalability required for such large-scale hardware designs. In contrast, our proposed RLConcolic framework achieves an average 84.70% coverage. This demonstrates that the reinforcement learning-guided exploration strategy is effective at navigating the vast and complex state spaces inherent in large-scale hardware designs.

### C. Test Generation Time and Memory Consumption

To assess the efficiency of RLConcolic, we evaluated its runtime and memory consumption against other methods, as shown in Table II.

RLConcolic significantly outperforms the traditional Concolic method in efficiency. Even when we compare without considering its higher coverage, RLConcolic achieves approximately tenfold speedup, comparable to the other two tools. This remarkable performance gain can be attributed to its intelligent path exploration strategy. As illustrated by the green dashed lines in Fig. 4, the traditional Concolic method exhibits slow and inefficient progress due to its ineffective branch selection, which often leads the exploration to wrong directions. In contrast, RLConcolic (represented by the purple solid lines) leverages reinforcement learning to select the most profitable

TABLE II: Run time (in seconds) and memory usage (in MB) comparison between Concolic, EBMC, STSearch, and RLConcolic, along with improvement ratios (baseline / RLConcolic).

Benchmark	Concolic [21]		EBMC [22]		STSearch [23]		RLConcolic		Impro./Concolic		Impro./EBMC		Impro./STSearch	
	Time(s)	Mem	Time(s)	Mem	Time(s)	Mem	Time(s)	Mem	Time	Mem	Time	Mem	Time	Mem
b01	0.01	4.7M	0.74	11.8M	0.01	9.3M	0.01	4.3M	1.00x	1.09x	74.00x	2.74x	1.00x	2.16x
b06	0.01	4.6M	0.61	11.8M	0.01	10.4M	0.01	4.6M	1.00x	1.00x	61.00x	2.57x	1.00x	2.26x
b10	0.02	5.4M	4.92	31.0M	0.01	9.5M	0.04	5.2M	0.50x	1.04x	2.39x	5.96x	0.25x	1.83x
b11	121.67	13.8M	4.46	53.9M	4.54	11.8M	3.67	13.3M	33.15x	1.04x	1.22x	4.05x	1.24x	0.89x
b12	5657.23	162.3M	2684.45	446.9M	†	†	2245.43	57.2M	2.52x	2.84x	1.20x	7.81x	†	†
b15	5250.81	184.64M	4695.93	296.5M	†	†	2408.71	81.2M	2.18x	2.26x	1.95x	3.64x	†	†
b20	†	†	†	†	†	†	3921.82	92.3M	†	†	†	†	†	†
b21	†	†	†	†	†	†	2609.86	94.9M	†	†	†	†	†	†
DCache	101.37	31.8M	11.78	52.9M	29.99	16.1M	7.44	15.7M	13.62x	2.03x	1.58x	3.37x	4.03x	1.03x
ICache	31.58	18.0M	2.42	48.2M	6.90	12.0M	3.92	13.5M	8.06x	1.33x	0.62x	3.57x	1.76x	0.88x
Exception	73.93	69.2M	19.77	40.0M	7.17	11.0M	2.52	43.6M	29.33x	1.59x	7.84x	0.92x	2.84x	0.25x
Rocketchip	†	†	†	†	†	†	15925.10	260.0M	†	†	†	†	†	†

Note: “†” indicates that the result was not reported in the corresponding work.

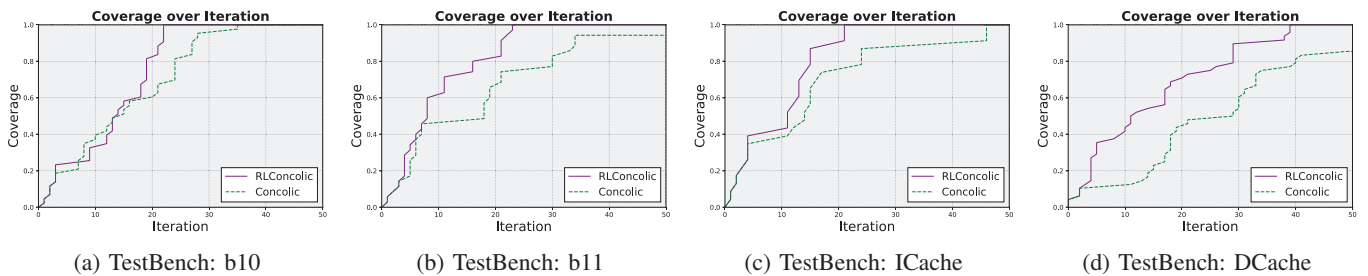


Fig. 4: Comparison of Coverage Over Iterations

branches. Consequently, RLConcolic not only achieved a higher coverage across all benchmarks but also converged in considerably fewer iterations.

In terms of memory consumption, RLConcolic demonstrates high efficiency, using slightly more memory than the fuzzing-based approach STSearch [23]. Its memory consumption is comparable to that of the traditional Concolic testing [21], with variations stemming from different implementations. In other words, our RL agent does not introduce substantial memory overhead to the overall framework. In fact, as shown in the table, RLConcolic utilizes less memory thanks to its efficient path exploration, which minimizes memory requirements during lengthy path explorations and SMT solving.

When compared to EBMC, RLConcolic demonstrates significant memory savings across most benchmarks. To understand the relationship between memory consumption and design size, we plotted memory usage against lines of code in Fig. 5. While lines of code are only an approximation of design complexity, they provide a useful metric. As the design size increases, the memory usage gap between EBMC and RLConcolic widens. For instance, on the b12 benchmark, RLConcolic consumes 7.81 times less memory than EBMC. This highlights how RLConcolic effectively optimizes computational resources during path exploration and SMT solving.

The efficiency of RLConcolic is further illustrated by the largest benchmark, Rocket Chip, which is ten times larger than b20 and b21. Despite this massive increase in size, RLConcolic’s memory consumption only triples, showcasing its scalability. The combined efficiency in both time and memory makes RLConcolic a robust and practical solution for modern

hardware verification.

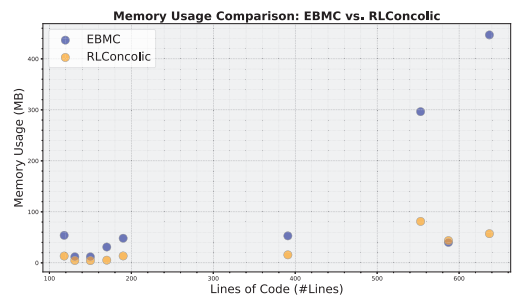


Fig. 5: Memory Usage Comparison: EBMC vs. RLConcolic

## V. CONCLUSION

In this paper, we propose RLConcolic, a novel semi-formal verification framework that combines reinforcement learning with Concolic testing to improve branch coverage in RTL verification. The core innovation of our approach is the replacement of conventional, static heuristics with an intelligent agent that autonomously learns an optimal path exploration policy.

Experimental results demonstrate that RLConcolic not only outperforms various methods in achieving superior branch coverage with reduced test generation time, but also demonstrates outstanding scalability for large designs. As the first Concolic testing framework applicable to RISC-V cores, RLConcolic achieved an outstanding 84.70% branch coverage on the Rocketchip core. These results indicate that our learning-based, dynamic exploration strategy offers a practical and highly effective solution to the persistent challenges in modern hardware verification.

## REFERENCES

- [1] V. V. Acharya, S. Bagri, and M. S. Hsiao, "Branch guided functional test generation at the rtl," in *2015 20th IEEE European Test Symposium (ETS)*. IEEE, 2015, pp. 1–6.
- [2] T. Li, H. Zou, D. Luo, and W. Qu, "Symbolic simulation enhanced coverage-directed fuzz testing of rtl design," in *2021 IEEE International Symposium on Circuits and Systems (ISCAS)*, 2021, pp. 1–5.
- [3] J. Hur, S. Song, D. Kwon, E. Baek, J. Kim, and B. Lee, "Difuzzrtl: Differential fuzz testing to find cpu bugs," in *2021 IEEE Symposium on Security and Privacy (SP)*, 2021, pp. 1286–1303.
- [4] P. E. Ammann, P. E. Black, and W. Majurski, "Using model checking to generate tests from specifications," in *Proceedings second international conference on formal engineering methods (Cat. No. 98EX241)*. IEEE, 1998, pp. 46–54.
- [5] A. Gargantini and C. Heitmeyer, "Using model checking to generate tests from requirements specifications," *ACM SIGSOFT Software Engineering Notes*, vol. 24, no. 6, pp. 146–162, 1999.
- [6] R. Mukherjee, D. Kroening, and T. Melham, "Hardware verification using software analyzers," in *2015 IEEE Computer Society Annual Symposium on VLSI*. IEEE, 2015, pp. 7–12.
- [7] E. M. Clarke, "Model checking," in *Foundations of Software Technology and Theoretical Computer Science: 17th Conference Kharagpur, India, December 18–20, 1997 Proceedings 17*. Springer, 1997, pp. 54–56.
- [8] K. Ryan and C. Sturton, "Sylvia: Countering the path explosion problem in the symbolic execution of hardware designs," in *2023 Formal Methods in Computer-Aided Design (FMCAD)*, 2023, pp. 110–121.
- [9] P. Godefroid, N. Klarlund, and K. Sen, "Dart: Directed automated random testing," in *Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation*, 2005, pp. 213–223.
- [10] K. Sen, D. Marinov, and G. Agha, "Cute: A concolic unit testing engine for c," *ACM SIGSOFT Software Engineering Notes*, vol. 30, no. 5, pp. 263–272, 2005.
- [11] Y. Lyu and P. Mishra, "Scalable concolic testing of rtl models," *IEEE Transactions on Computers*, p. 979–991, Jul 2021. [Online]. Available: <http://dx.doi.org/10.1109/tc.2020.2997644>
- [12] R. Mukherjee, D. Kroening, and T. Melham, "Hardware verification using software analyzers," in *2015 IEEE Computer Society Annual Symposium on VLSI*, 2015, pp. 7–12.
- [13] Z. Zheng, X. Meng, and Y. Lyu, "Ape-fv: Concolic testing for rtl functional verification using adaptive path exploration," in *2024 IEEE 42nd International Conference on Computer Design (ICCD)*. IEEE, 2024, pp. 373–380.
- [14] C. Szepesvári, *Algorithms for reinforcement learning*. Springer nature, 2022.
- [15] C. J. Watkins and P. Dayan, "Q-learning," *Machine learning*, vol. 8, pp. 279–292, 1992.
- [16] V. Mnih, K. Kavukcuoglu, D. Silver, A. Graves, I. Antonoglou, D. Wierstra, and M. Riedmiller, "Playing atari with deep reinforcement learning," *arXiv preprint arXiv:1312.5602*, 2013.
- [17] V. Mnih, K. Kavukcuoglu, D. Silver, A. A. Rusu, J. Veness, M. G. Bellemare, A. Graves, M. Riedmiller, A. K. Fidjeland, G. Ostrovski *et al.*, "Human-level control through deep reinforcement learning," *nature*, vol. 518, no. 7540, pp. 529–533, 2015.
- [18] Z. Shi, M. Li, S. Khan, L. Wang, N. Wang, Y. Huang, and Q. Xu, "Deeppti: Test point insertion with deep reinforcement learning," in *2022 IEEE International Test Conference (ITC)*. IEEE, 2022, pp. 194–203.
- [19] L. Wu, M. Rostami, H. Li, and A.-R. Sadeghi, "Hfl: Hardware fuzzing loop with reinforcement learning," in *2025 Design, Automation & Test in Europe Conference (DATE)*. IEEE, 2025, pp. 1–7.
- [20] L. Wu, M. Rostami, H. Li, J. Rajendran, and A.-R. Sadeghi, "Genhuzz: An efficient generative hardware fuzzer."
- [21] Y. Lyu, A. Ahmed, and P. Mishra, "Automated activation of multiple targets in rtl models using concolic testing," in *2019 Design, Automation Test in Europe Conference Exhibition (DATE)*, 2019, pp. 354–359.
- [22] D. Kroening *et al.*, "Ebmcc: The enhanced bounded model checker," <https://www.cprover.org/ebmcc/>, accessed: [Insert the date you accessed the website].
- [23] Z. Zheng and Y. Lyu, "Stsearch: State tracing-based search heuristics for rtl validation," in *2023 Design, Automation Test in Europe Conference Exhibition (DATE)*, 2023, pp. 1–6.
- [24] S. Takamaeda-Yamazaki, "Pyverilog: A python-based hardware design processing toolkit for verilog hdl," in *Applied Reconfigurable Computing*, ser. Lecture Notes in Computer Science, vol. 9040. Springer International Publishing, Apr 2015, pp. 451–460. [Online]. Available: [http://dx.doi.org/10.1007/978-3-319-16214-0\\_42](http://dx.doi.org/10.1007/978-3-319-16214-0_42)
- [25] L. De Moura and N. Bjørner, "Z3 Theorem Prover," <https://github.com/Z3Prover/z3>, 2008, [Online; accessed Year-Month-Day].
- [26] "Opencores website," <https://www.opencores.org>, 2022.
- [27] F. Corno, M. Reorda, and G. Squillero, "Rt-level itc'99 benchmarks and first atpg results," *IEEE Design & Test of Computers*, vol. 17, no. 3, p. 44–53, Jan 2000. [Online]. Available: <http://dx.doi.org/10.1109/54.867894>
- [28] K. Asanovic, R. Avizienis, J. Bachrach, S. Beamer, D. Biancolin, C. Celio, H. Cook, D. Dabbelt, J. Hauser, A. Izraelevitz *et al.*, "The rocket chip generator," *EECS Department, University of California, Berkeley, Tech. Rep. UCB/EECS-2016-17*, vol. 4, pp. 6–2, 2016.