STSearch: State Tracing-based Search Heuristics for RTL Validation

Ziyue Zheng and Yangdi Lyu†
Microelectronics Thrust, The Hong Kong University of Science and Technology (Guangzhou)
†Corresponding author: yangdilyu@ust.hk

***Abstract*—Branch coverage is important in the functional validation of Register-Transfer-Level (RTL) models. While random tests can cover the majority of easy-to-reach branches, there are still many hard-to-activate branches in today's industrial designs. These remaining corner branches are typically the source of bugs and hardware trojans. Directed test generation approaches using formal methods effectively activate a specific branch but are limited by the state explosion problem. Semi-formal methods, such as concolic testing, improve the scalability by exploring one path at a time.**

This paper presents a novel concolic testing framework to exercise the corner branches through state tracing-based search heuristics (STSearch). The proposed approach heuristically generates and evaluates input sequences based on a novel heuristic indicator that evaluates the distance between the current state and the target branch condition. The heuristic indicator is designed to utilize both the static structural property of the design and the state from dynamic simulation. Compared to the existing concolic testing approaches, where a full new path is generated in each round by solving path constraints, the cycle-based heuristic search in the proposed approach is more effective and efficient. Experimental results show that our approach significantly outperforms the state-of-the-art approaches in both running time and memory usage.

I. Introduction

With the increasing complexity of today's chip designs, it becomes more important to ensure the correct functionality using various techniques, such as formal methods and simulation-based verification. Branch coverage in RTL models is one of the most critical metrics to evaluate the quality of tests in simulation-based verification [1]. While simulation using millions or billions of random/constrained-random tests is able to activate the majority of the easy-to-reach branches, there could still be a large number of uncovered branches which are typically the source of bugs or hardware Trojans [2]. There has been extensive research on directed test generation methods based on formal and semi-formal methods to achieve high branch coverage.

Directed test generation approaches based on formal methods [3–5], such as model checkers, transform both the RTL model and target branches into a formal description, and then generate the input sequence using a SAT solver or a Satisfiability Modulo Theory (SMT) solver. These methods can reach the corner cases that could only be activated by a minimal set of tests. However, these approaches are not scalable in generating directed tests for large designs since the number of states and transitions grows exponentially with the complexity of the design and the unrolled cycles. To address this issue, semi-formal methods, such as concolic testing, were proposed to combine concrete simulation and symbolic execution [6; 7]. In contrast to formal methods, where the whole design is transformed into a formal description and solved in one shot, concolic testing methods explore and solve one path at a time.

While concolic testing alleviates the state explosion problem by exploring one path at a time, there are two main challenges in applying concolic testing methods in large designs. The first one is the effectiveness of the test generation approaches. As it is infeasible to explore all paths within a time budget, an effective path exploration technique should be applied to cover the target branch quickly. For example, the edge realignment technique is proposed in [9; 10] to utilize the structural information in RTL models to guide path exploration. The second one is the efficiency of path exploration. The existing concolic testing methods typically apply constraint mutation to explore new paths. With the size of the design increasing, concolic testing needs to explore more paths, and each path will accumulate more constraints, which will take significantly more time to cover a target branch.

To address these two challenges, this paper proposes a state tracing-based search heuristic to improve path exploration in concolic testing. Compared to the existing concolic testing methods, where complete input sequences are solved at each iteration [10], our approach continuously generates input sequences for only a few cycles based on the current state. In searching for the best input sequences, we introduce an effective heuristic to reduce the number of paths to explore, and replace the time-consuming SMT calls with a lightweight genetic algorithm (GA). The main contributions of the paper can be summarized as follows:

- We introduce an effective and interpretable heuristic indicator, *state distance*, to estimate the distance between the current state and a target branch utilizing both the runtime states and the structural information of RTL models.
- We present a concolic testing framework based on state tracing to search for effective input sequences to cover hard-to-activate branches heuristically. To the best of our knowledge, this is the first framework that actively traces the runtime variables of the RTL design to heuristically guides the path exploration.
- We implement the tool to generate efficient tests and evaluate the branch coverage on 7 benchmarks. The tool can be directly applied to RTL models described in Verilog. The experimental results show that STSearch can achieve several times improvement in test generation time

978-3-9819263-7-8/DATE23/© 2023 EDAA

and memory usage compared to EBMC and the state-of-the-art concolic testing method.

II. Related Work

In this section, we describe the related efforts on formal methods and semi-formal methods for test generation.

A. Formal Methods

Formal methods have been extensively exploited in automated test generation [11–13]. One common approach is bounded model checking (BMC). To cover a target branch, the negation of all guard conditions of the target will be specified as the property of the model in BMC. Then, BMC will check whether the property can be satisfied or not. The counterexample that fails the property is the generated test to cover the target branch. Although this approach is able to check all possible states of an RTL model within a fixed unroll cycle, it requires a huge amount of computing resources to solve the property with a large number of states. As a result, it is not scalable to generate tests for large design models or corner cases that require lots of unrolled cycles, despite the efforts to optimize time and space complexity [14].

B. Semi-formal Methods

To address the state explosion problem, semi-formal methods, such as concolic testing, have been proposed in software testing [6–8; 15] and hardware verification [10; 16; 17]. Concolic testing combines both advantages of concrete simulation and symbolic execution. New paths are explored by path reconstruction and constraints mutation in each iteration. The effectiveness of the generated tests is dominated by the heuristics in path exploration. Many techniques are proposed in path exploration to improve the coverage of RTL models. Liu et al. [17] proposed to utilize the runtime exploration history to guide the selection of mutated constraints. Lyu et al. [10] proposed a path exploration technique by fully analyzing the static structural information of RTL models and realigning the control flow graph (CFG). Compared to these approaches, the state tracing-based search heuristics proposed in this paper is more effective in path exploration by utilizing both the structural information of RTL models and the runtime state of the design.

III. Methodology

We propose a concolic testing framework to heuristically generate tests based on state tracing to cover hard-to-activate branches in RTL models. As shown in Figure 1, our framework STSearch consists of two major stages. The first stage is the static information collector responsible for retrieving as much helpful information as possible by statically analyzing the design. The second stage is the state tracing-based heuristic path explorer, which utilizes both the information from the first stage and the runtime state to guide path exploration. This section uses the RTL model in Listing 1 as an example for demonstration. The notation b_i in the comment represents the branch name, and b_4 is the target branch.

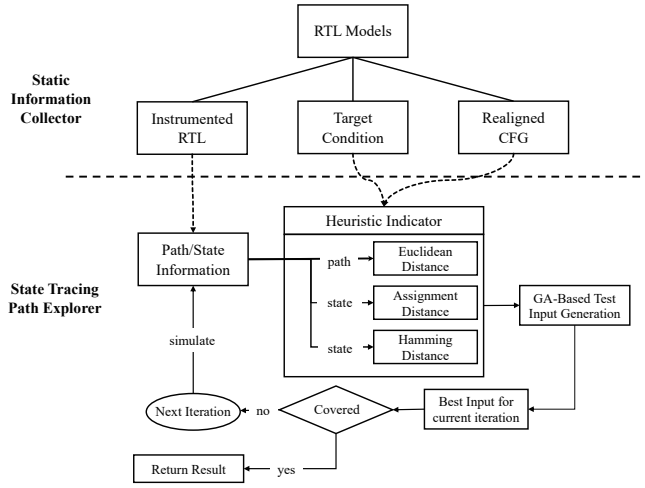


Fig. 1: The framework of STSearch. The state tracing-based path explorer utilizes the structural information from the static information collector and the runtime state to generate tests to activate a target branch.

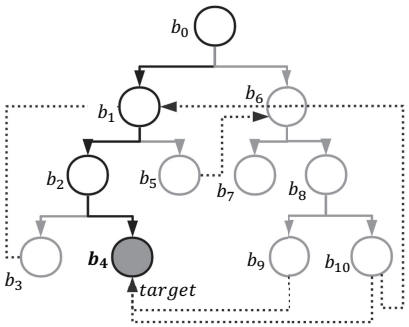
A. Static Information Collector

The static information collector is a preprocessor that statically analyzes the RTL model before simulation. It is used to help capture the state of the design during the simulation and guide the path exploration with structural information.

Path/State Instrumentation. STSearch instruments the RTL code with *$display* statements to capture the execution path and states during simulation. There are two levels of instrumentation, i.e., path-level and state-level. The path-level instrumentation labels each statement with a unique tag, and the state-level instrumentation is inserted before the end of the module to trace the state of the design during simulation. All the information will be used for calculating the heuristic indicator during path exploration.

Listing 1: An example RTL code snippet

```
1  input [1:0] i;
2  reg [1:0] a,b;
3  reg stato;
4  case (stato)
5    0:begin                    //------------------------b1
6      if (i >= 2'b10)          //------------------------b2
7        if (b <= 2'b10)        //------------------------b3
8          stato <= 0;
9        else                   //------------------------b4
10         // target
11     else                     //------------------------b5
12       stato <= 1;
13   end   // end of stato == 0
14   1:begin                    //------------------------b6
15     if (a >= 2'b11 || b >= 2'b10) begin  //-----b7
16       a <= {a[1], b[0]};
17       b <= 2'b00;
18     end
19     else                     //------------------------b8
20       if (i >= 2'b10) begin  //-----------------b9
21         a <= {b[0], 1'b1};
22         b <= b + 2'b01;
23       end
24       else begin             //-----------------b10
25         b <= b + 2'b10;
26         stato <= 0;
27       end
28   end   // end of stato == 1
29 endcase
```



(a) Illustration of CFG

	b_1	b_2	b_3	b_5	b_6	b_7	b_8	b_9	b_{10}
b_4*	2	1	∞	∞	∞	∞	∞	∞	∞
b_4**	2	1	4	4	3	∞	2	1	1

*CFG distance to b_4
**ERCFG distance to b_4

(b) Distance Matrix

Fig. 2: (a) An illustration of realigning the CFG of the RTL model in Listing 1. The dotted lines represent the newly added edges by the ERCFG; (b) Distance matrix to b_4 in the original CFG and the ERCFG.

Target Condition Collection. STSearch traverses along the CFG and extracts all the guard conditions for a nested branch to better understand the required target state. For example, all the conditions in branches $\{b_4, b_2, b_1\}$ must be satisfied to reach the target branch b_4 in Figure 2a. All these conditions are combined to form the final target condition.

Control Flow Graph Realignment. The CFG does not provide much information about the contribution of an assignment to reach a target branch. For example, consider the initial state $\mathbf{s} = \{stato = 0, a = 2'b00, b = 2'b00\}$ and the target condition $cond_{b_4} = \{b > 2'b10\}$. Based on the edges in the CFG, the basic blocks inside b_1 and b_2 are closer to b_4 than all the other blocks. However, the statements $b <= b + 2'b01$ and $b <= b + 2'b10$ inside b_9 and b_{10}, respectively, are the necessary nodes to reach the target b_4 by analyzing the code in Listing 1. This information cannot be inferred from the CFG. To address this problem, we utilize the edge realigned CFG (ERCFG) to better describe the contribution of each block in the RTL model. The process of realigning the CFG utilizes a SMT solver to symbolically check whether a statement has a potential contribution to reaching a condition, similar to the process in [10]. To indicate the contribution of the assignments in a block, ERCFG directly connects the block to the target branch, which is shown in the dotted lines in Figure 2a. For example, the two dotted lines at the bottom of Figure 2a represent the realigned edges from b_9 and b_{10} to the target branch b_4. Next, we compute the distance matrix of a specific branch target by traversing backward from the branch. A smaller distance in the distance matrix indicates that the block has a more significant contribution to reaching the target branch. As shown in Figure 2b, the distance matrix computed from the ERCFG better reflects the contribution of blocks to reaching the branch b_4 than that computed from the original CFG.

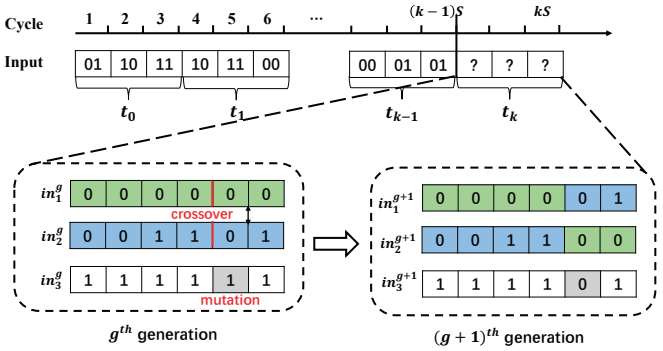


Fig. 3: An illustrative example of state tracing-based path exploration for Listing 1 with 2 bits input. The input sequences are generated stride by stride, where each stride consists of S cycles ($S = 3$ in this example). While generating the input sequence t_k for the k^{th} stride using GA-based heuristic search, the input sequences for all the preceding strides $\{t_1, t_2, ..., t_{k-1}\}$ are fixed.

B. State Tracing-based Path Explorer

The path explorer generates tests to cover the target branch based on the information provided by the static information collector. Instead of generating and simulating the whole path in each iteration [10], we group S cycles as one stride and generate input sequences stride by stride, as shown in Figure 3. To evaluate and find the best input sequence for a stride in the genetic algorithm, we propose a novel heuristic indicator called *state distance*, aiming to estimate the distance between the target state and the state after applying the input sequence. The genetic algorithm will be guided to find the input sequence with the smallest state distance. With a good design of the state distance, the explored paths are expected to get closer and closer to the target branch. In the remainder of this section, we will first introduce how to compute the state distance from the structural information and the simulation state after applying the input sequence, as well as the reason behind the design. Then, we will introduce the genetic algorithm used in path exploration.

1) State Distance Definition: State distance is designed to estimate the "distance" between the current state and the target condition. Given the condition of a target branch $cond$, state distance is an indicator to compare the distance of two states $\boldsymbol{s_1}$ and $\boldsymbol{s_2}$ from the target state. For example, the condition of the target branch in b_4 is $\{stato = 0 \ \& \ i >= 2'b10 \ \& \ b > 2'b10\}$ in Listing 1. Suppose two different input sequences in_1 and in_2 reach the states $\boldsymbol{s_1} = \{stato = 0 \ \& \ i = 2'b11 \ \& \ b = 2'b00\}$ and $\boldsymbol{s_2} = \{stato = 1 \ \& \ i = 2'b00 \ \& \ b = 2'b10\}$, respectively. The key observation is that which state is "closer" to the target condition highly depends on the RTL designs. To better evaluate the "distance", we classify the variables into three categories.

Euclidean variables. A variable v is called a Euclidean variable if most of the assignments to v are the addition or subtraction operations. In the example of Listing 1, b is a Euclidean variable. Consider the state where $b = 2'b00$ and the target condition requires $b > 2'b10$. Since the current b is less than the target, STSearch wants to give more credits to the test that increases b fast. In other words, a path that passes line 26 is more likely to be selected as the winner than a path passing line 18. Due to this heuristic, STSearch uses the Euclidean distance for this type of variables. Formally, for a Euclidean variable v, the *Euclidean distance* is defined as $\mathcal{E}_v \leftarrow |v - v_t|$, where v_t is the closest target value to v. For example, if $b = 2'b00$ and the target is $b > 2'b10$, the distance $\mathcal{E}_v = |2'b00 - 2'b11| = 3$.

Hamming variables. A variable v is called a Hamming variable if most of the assignments to v are concatenation. In the example in Listing 1, a is a Hamming variable. For Hamming variables, Euclidean distances may not help select the best test since their modifications are not continuous. Therefore, we use bit-wise *Hamming distance* $\mathcal{H}_v$ as the heuristic indicator.

Assignment variables. The remaining variables are called assignment variables. They are mostly modified through assignments of direct values or from other variables. In the example in Listing 1, $stato$ is an assignment variable. It is not as straightforward to define the distance of an assignment variable as the other two types of variables. One key observation is that the state is closer to the target condition if a path could pass through a block with a small distance in the distance matrix, as shown in Figure 2b. Therefore, we define the *assignment distance* as the smallest distance of the blocks visited in the last cycle, denoted as $\mathcal{A}_v$.

For all variables that appear in the target state guard conditions $cond$, we calculate the state distances $sd(cond)$ in the following way. First, the target condition $cond$ is transformed into the conjunctive normal form (CNF) as $\wedge_i(\vee_j cond_{ij})$. For each simple clause $cond_{ij}$, the state distance is a combination of the distances of all its variables, as shown in Eq. 1. We balance the distances by assigning different weights $\omega_{\mathcal{E}}, \omega_{\mathcal{A}}$, and $\omega_{\mathcal{H}}$ based on the type of variables. Then, the state distance is computed using Eq. 2 for the composition of clauses. For two conditions combined with an OR operation, the distance is defined as the smaller one of these two distances. For two conditions combined with an AND operation, the distance is defined as the sum of these two distances.

$$sd(cond_{ij}) = \sum_{\forall v \in cond_{ij}} \omega_{\mathcal{E}} * \mathcal{E}_v + \omega_{\mathcal{A}} * \mathcal{A}_v + \omega_{\mathcal{H}} * \mathcal{H}_v \quad (1)$$

$$sd(\wedge_i(\vee_j cond_{ij})) = \sum_i (\min_j sd(cond_{ij})) \quad (2)$$

Note that the computation of the state distance is very efficient and scalable. First, the distances for Euclidean variables and Hamming variables only depend on the final state of the design. Next, the assignment distance is computed based on the blocks visited in a single cycle, which does not accumulate over time. Finally, the computation of the state distance is efficient based on the definition.

Algorithm 1 STSearch

Input: RTL model $\mathcal{R}$, target branch B, largest number of stride M, number of cycles in a stride S, number of generations G, population n, number of selection r, mutation rate m

Output: Test sequence $T = \{t_1, t_2, ..., \}$

// static information collector

1: Instrument $\mathcal{R}$ and collect guard conditions of B
2: Realign the CFG of $\mathcal{R}$ and compute the distance matrix

// state tracing-based path explorer

3: **for** iteration $k = 1$ to M **do**
4: **chromosomes**: Randomly generate n input sequences $\{in_{k_1}, in_{k_2}, ..., in_{k_n}\}$ for cycles $(k-1)S + 1$ to kS
5: **Initialize population**: $\mathcal{P} \leftarrow \{in_{k_1}, in_{k_2}, ..., in_{k_n}\}$
// simulate and compute fitness
6: **for all** $in \in \mathcal{P}$ **do**
7: Simulate $\mathcal{R}$ with the input $\{t_1, t_2, ..., t_{k-1}, in\}$
8: Compute state distance sd_{in} using Eq. 2 and Eq. 1
9: **fitness** of in: $fit_{in} = \frac{1}{sd_{in}}$
10: **end for**
11: **for** generation $g = 1$ to G **do**
12: **selection**: Apply roulette selection on $\mathcal{P}$ to collect r chromosomes $\boldsymbol{in^g} = \{in_1^g, in_2^g, ..., in_r^g\}$ based on fitness values
13: Apply position-based **crossover** and **mutation** with mutation rate m on $\boldsymbol{in^g}$ to get $\boldsymbol{in^{(g+1)}}$
14: Simulate and compute fitness for all $in \in \boldsymbol{in^{(g+1)}}$
15: **if** branch is covered **then**
16: **return** $T = \{t_1, t_2, ..., t_{k-1}, in\}$
17: **end if**
18: Put all input sequences in $\boldsymbol{in^{(g+1)}}$ to $\mathcal{P}$
19: **end for**
20: Select in^* with the largest fitness from $\mathcal{P}$
21: The input sequence for k^{th} stride $t_k = in^*$
22: **end for**
23: **return** $T = \{t_1, t_2, ..., t_M\}$

2) State Tracing-based Heuristic Search: The algorithm of the entire framework is shown in Algorithm 1. After the static information collection stage, the framework iteratively searches for the best input sequence t_k for the k^{th} stride using a genetic algorithm. The goal of the genetic algorithm is to find the input sequence with the smallest state distance.

Genetic algorithms are lightweight optimization methods inspired by natural evolution and selection. The candidate input sequence of an iteration is called a *chromosome*. For the k^{th} iteration, each chromosome is a candidate input sequence for the cycles $(k-1)S + 1$ to kS, as shown in Figure 3. The fitness of each chromosome is the inverse of the state distance computed by Eq. 1 and Eq. 2. The optimization process of the genetic algorithm is to find a chromosome with the largest fitness value. It consists of 4 major steps: i) *Population Initialization*: At the beginning of each iteration, GA randomly generates n chromosomes and gathers

TABLE I: Comparison between EBMC, [10] and STSearch

Bench	cycle	branches	EBMC			[10]			STSearch			Impro. /EBMC		Impro. / [10]	
			Cov	time(s)	mem	Cov	time(s)	mem	Cov	time(s)	mem	time	mem	time	mem
b01	30	26	26	0.74	11.8M	26	0.01	9.1M	26	0.01	9.3M	18.50x	1.27x	1.00x	-1.02x
b06	30	24	23	0.61	11.8M	23	0.01	9.1M	23	0.01	10.4M	15.25x	1.13x	1.00x	-1.14x
b10	30	41	41	4.92	31M	41	0.02	9.4M	41	0.01	9.5M	2.58x	2.89x	2.00x	-1.13x
b11	100	32	31	4.46	53.9M	30	121.67	48.3M	31	4.54	11.8M	-1.01x	4.56x	26.80x	4.57x
ICache	50	26	25	2.42	48M	25	31.58	18M	25	6.90	12.0M	-2.85x	4.01x	4.58x	1.50x
DCache	50	46	42	11.78	52.9M	42	101.37	16M	41	29.99	16.1M	-2.55x	3.38x	3.38x	-1.01x
Exception	50	47	47	19.77	40M	47	3.8	23M	46	7.17	11.0M	2.75x	3.65x	-1.88x	2.10x
Average	-	-	33.6	19.54	35.6M	33.4	34.09	19.0M	33.3	6.95	11.4M	2.81x	3.12x	4.90x	1.66x

them into the population. The fitness of each chromosome is obtained through simulation and state distance computation. ii) *Selection*: In this stage, r well-performed chromosomes are selected for further optimization. The chosen probability of each chromosome is based on roulette probability, i.e., chromosomes with larger fitness values are more likely to be selected. Since the state distance is the inverse of the fitness value, the input sequences with smaller state distances are more likely to be chosen. iii) *Crossover and Mutation*: Crossover optimizes populations by retaining and exchanging patterns of well-performing chromosomes. The selected r chromosomes will be paired for crossover. Figure 3 illustrates a position-based crossover in in_1^g and in_2^g, where g is the generation. We specify a random crossover position and swap the left or right side of the two chromosomes to obtain two children in_1^{g+1} and in_2^{g+1}. A mutation is introduced to add randomization and expand the search domain. Every chromosome may be selected at this stage with a mutation rate m. A random signal of the chromosome will be flipped as shown in the illustration in Figure 3, iv) *Accept criterion*: In the acceptance stage, we choose the chromosome with the largest fitness as the input sequence for the k^{th} stride.

The crossover operation in GA enables our framework to co-optimize input tests that expand multiple cycles. The intuition behind this design is based on the observation that the impact of an input test may be revealed in a few cycles.

IV. Evaluation Results

A. Experimental Setup

We implemented our framework STSearch using C++ with Icarus Verilog [18] and Yices 2.6 solver [19]. The iverilog is used for simulation and collecting runtime states, while the Yices SMT solver is used for the control flow graph realignment. We performed a few experiments on a virtual machine running Ubuntu 20.04 with an i7-11800H (2.3GHz) CPU. Experiments were conducted on 7 benchmarks from OpenRISC1200 [20] and ITC'99 datasets containing hard-to-reach branches. We discard or1200 in the names of benchmarks or1200_ICache, or1200_DCache, and or1200_Exception from OpenCores without causing any confusion.

B. Algorithm Settings

We set the maximum iteration M to be 1000, and the parameters of the genetic algorithm $\{G, n, r, m\}$ to be $\{3, 15, 6, 0.1\}$ in Algorithm 1. The number of cycles in one stride S is set to be 3. Section IV-D will evaluate the effects of different cycles in a stride. Further, we set a timeout for Algorithm 1. If the largest fitness value does not change for 5 seconds, Algorithm 1 will give up and terminate.

The weights for assignment variables and Hamming variables in Eq. 1 are set to be 1. As the Euclidean distance can be very large when the variable contains many bits, we tuned the weight $\omega_{\mathcal{E}}$ for Euclidean distance to balance the three types of distances.

C. Experiment Result Evaluation

Table I compares the number of cover branches, test generation time, and memory usage of our framework STSearch with the formal method EBMC [21] and the state-of-the-art concolic testing method [10]. Columns 2-3 in Table I show the unrolled cycles and number of branches of 7 benchmarks. The unrolled cycle is determined in EBMC to cover as many branches as possible. Columns 4-12 profile the number of cover branches, runtime, and memory use of three frameworks. Without solving constraints as EBMC and [10], STSearch can achieve the same branch coverage as EBMC in 5 out of the 7 benchmarks by pure heuristic generation. In the other benchmarks, there is only 1 branch difference in the overall coverage compared to EBMC. While the branch coverage is almost the same, STSearch achieves 2.81x and 4.90x speedup over EBMC and [10], respectively, and 3.12x and 1.66x improvement in memory reduction, respectively. These improvements come from two primary factors. First, our heuristic indicator state distance effectively guides the generation of input sequences. In each cycle stride, GA optimizes the fitness of the randomly generated sequences, making RTL state closer to the target branch. Second, without heavily relying on SMT solvers, state tracing-based search poses less stress on the computing resource, making our framework much more scalable than the existing approaches.

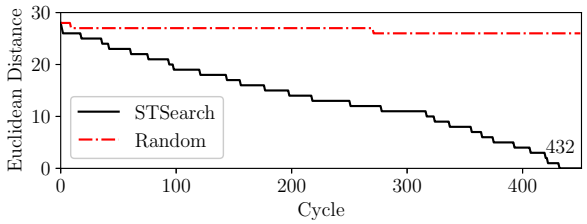
TABLE II: Comparison of the number of branches cover and time efficiency with different cycle strides

Bench	S = 1		S=2		S=3		S=4	
	cov	time(s)	cov	time(s)	Cov	time(s)	Cov	time(s)
b11	30	41.5	30	15.3	31	4.5	31	7.1
Exception	44	18.9	45	15.5	46	7.2	45	16.0

D. Cycle Stride Comparison

Cycle strides S is a critical parameter since it represents the number of cycles in each input sequence. With a larger S, tests from more cycles are grouped and co-optimized, potentially increasing the effectiveness of generated tests. But

it also significantly increases the search space, which will increase the time to find the best input sequence. As a result, there is a trade-off between the quality of the generated tests and the computing resource. Table II compares the performance in b11 and or1200_Exception with different cycle strides ($S = 1, 2, 3, 4$). The experimental result shows that STSearch achieve the best branch coverage and performance when $S = 3$.



(a) Euclidean distance for a corner branch in b11

Assignment Distance	STSearch		Random	
	Cycles	Per %	Cycles	Per %
∞	163	37.72%	136	27.20%
3	48	11.11%	32	6.40%
2	196	45.37%	327	65.40%
1	25	5.79%	5	1.00%
Total	432	100%	500	100%

(b) Assignment Distance for a corner branch in b11

Fig. 4: Comparison of the state distance computed by the tests generated by STSearch and random tests.

E. Evaluation of State Distance

To demonstrate the effectiveness of our heuristic indicator in guiding path exploration, we take one corner branch in b11 as an example to show the change of our distance over time. Figure 4 compares the components of the state distance computed by the tests generated by STSearch and random tests. Figure 4a shows the Euclidean distance for 500 cycles. The Euclidean distance of STSearch converges to 0 within 500 cycles, while the minimum Euclidean distance by random tests does not improve much. Figure 4b illustrates the distribution of assignment distances in 500 cycles, where the proportion of the blocks with the smallest assignment distance visited within 500 cycles is five times more than random tests. The result indicates that STSearch effectively guides the search to explore the blocks with high contributions frequently and reaches the states that satisfy the target condition.

V. Conclusion

This paper presents a novel semi-formal test generation framework to improve branch coverage in RTL models by combining structural information with simulating states. The framework traces the state of the design and heuristically guides the path exploration to reach a target branch using GA. The heuristic indicator, state distance, is highly explainable which evaluates the distance of the current state and the final target condition based on the properties of variables and paths. Experimental results show that STSearch can achieve several times improvement in test generation time and memory usage compared to EBMC and the state-of-the-art concolic testing method while maintaining high branch coverage.

References

[1] S. Devadas et al., "An observability-based code coverage metric for functional simulation," in *ICCAD*, 1996.

[2] Y. Lyu and P. Mishra, "Scalable activation of rare triggers in hardware trojans by repeated maximal clique sampling," *IEEE TCAD*, 2021.

[3] A. Gargantini and C. Heitmeyer, "Using model checking to generate tests from requirements specifications," *SIGSOFT Softw. Eng. Notes*, 1999.

[4] P. E. Ammann et al., "Using model checking to generate tests from specifications," in *2nd International Conference on Formal Engineering Methods*, 1998.

[5] R. Mukherjee, D. Kroening, and T. Melham, "Hardware verification using software analyzers," in *IEEE Computer Society Annual Symposium on VLSI*, 2015.

[6] P. Godefroid et al., "Dart: Directed automated random testing," in *ACM Sigplan Notices*, 2005.

[7] K. Sen et al., "Cute: A concolic unit testing engine for c," in *ACM SIGSOFT Software Engineering Notes*, 2005.

[8] C. Cadar, D. Dunbar, and D. Engler, "Klee: Unassisted and automatic generation of high-coverage tests for complex systems programs," in Proc. 8th USENIX Conf. Operating Syst. Des. Implementation, 2008,

[9] Y. Lyu and P. Mishra, "Automated Test Generation for Activation of Assertions in RTL Models," in *ASPDAC*, 2020.

[10] Y. Lyu and P. Mishra, "Scalable concolic testing of RTL models," *IEEE TC*, 2021.

[11] R. Mukherjee, D. Kroening, and T. Melham, "Hardware verification using software analyzers," in *IEEE Computer Society Annual Symposium on VLSI*, 2015.

[12] D. Kroening, "Computing over-approximations with bounded model checking," *Electron. Notes Theor. Comput. Sci.*, 2006.

[13] M. Chen and P. Mishra, "Property learning techniques for efficient generation of directed tests," *IEEE TC*, 2011.

[14] S. Ofer, "Accelerating bounded model checking of safety properties," in *Formal Methods in System Design*, 2004.

[15] C. Cadar et al., "Exe: Automatically generating inputs of death," *ACM Trans. Inf. Syst. Secur.*, 2008.

[16] L. Liu and S. Vasudevan, "Star: Generating input vectors for design validation by static analysis of rtl," in *IEEE HLDVT*, 2009.

[17] ——, "Scaling input stimulus generation through hybrid static and dynamic analysis of rtl," *ACM Trans. Des. Autom. Electron. Syst.*, 2014.

[18] S. Williams, "Icarus verilog," *http://iverilog.icarus.com/*, 2006.

[19] "Yices website," https://yices.csl.sri.com/, 2022.

[20] "Opencores website," https://www.opencores.org, 2022.

[21] D. K. and M. Purandare, "Ebmc: The enhanced bounded model checker," http://www.cprover.org/ebmc, 2022.