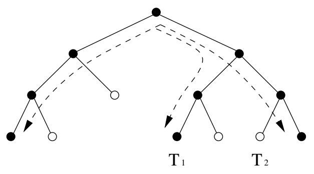
Scalable Concolic Testing of RTL Models

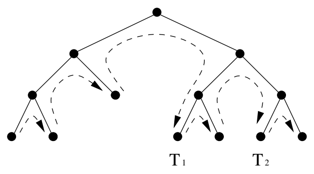
Yangdi Lyu and Prabhat Mishra, *Senior Member, IEEE*

***Abstract*—Simulation is widely used for validation of Register-Transfer-Level (RTL) models. While simulating with millions of random or constrained-random tests can cover majority of the functional scenarios, the number of remaining scenarios can still be huge (hundreds or thousands) in case of today's industrial designs. Hard-to-activate branches are one of the major contributors for such remaining/untested scenarios. While directed test generation techniques using formal methods are promising in activating branches, it is infeasible to apply them on large designs due to state space explosion. In this paper, we propose a fully automated and scalable approach to cover the hard-to-activate branches using concolic testing of RTL models. While application of concolic testing on hardware designs has shown some promising results in improving the overall coverage, they are not designed to activate specific targets such as uncovered corner cases and rare scenarios. In other words, existing concolic testing approaches address state space explosion problem but leads to path explosion problem while searching for the uncovered targets. Our proposed approach maps directed test generation problem to target search problem while avoiding overlapping searches involving multiple targets. This paper makes two important contributions. (1) We propose a directed test generation technique to activate a target by effective utilization of concolic testing on RTL models. (2) We develop efficient learning and clustering techniques to minimize the overlapping searches across targets to drastically reduce the overall test generation effort. Experimental results demonstrate that our approach significantly outperforms the state-of-the-art methods in terms of test generation time (up to 205X, 69X on average) as well as memory requirements (up to 31X, 7X on average).**

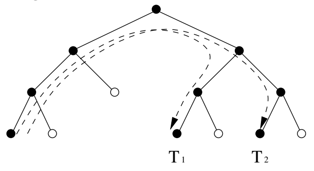
***Index Terms*—Concolic testing, RTL validation.**



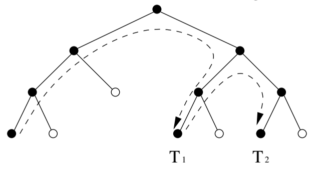
(a) Random simulation cannot guarantee the coverage of all targets in a reasonable time.



(b) Uniform strategy can take very long time (may be infeasible) to cover all the targets.



(c) Single-target method may lead to wasted effort (overlapping search) to cover targets.



(d) Multi-target approach utilizes the previous search and starts from a profitable path.

Fig. 1: Test generation to cover two targets T_1 and T_2 using four approaches: (a) random simulation, (b) uniform test generation, (c) single-target and (d) multi-target methods. The figures show the control flow graph (CFG) of the design, where each node represents one basic block[1]. The black ones and white ones represent the covered and uncovered basic blocks, respectively. The dashed lines represent simulation paths.

I. Introduction

During functional validation of hardware designs, a wide variety of coverage metrics are employed, such as functional coverage, finite state machine (FSM) coverage, statement coverage, branch coverage, and path coverage. These metrics are helpful to instruct the test suite to cover as many scenarios as possible. A common practice in industry is to simulate Register-Transfer Level (RTL) models of the design using millions of random or constrained-random tests to cover the majority of the scenarios. However, it may not be feasible to cover all scenarios using these tests for multi-million line RTL models of today's System-on-Chip (SoC) designs. We refer to these remaining uncovered scenarios as *hard-to-activate scenarios*. Verification engineers usually write specific (directed) test cases manually to cover the hard-to-activate scenarios such as corner cases and rare events. While manual test development is possible for small designs, it would be infeasible to develop directed tests manually for complex SoC designs. Moreover, manual development of test cases can be both error-prone and time-consuming due to many trial-and-error iterations in complex designs. Automated test generation is necessary to overcome these issues. An important observation is that hard-to-activate branches contribute to the vast majority of uncovered scenarios. In other words, it is crucial to cover hard-to-activate branches to improve the overall quality of functional validation as a complementary approach to traditional validations. In this paper, our goal is to generate tests to cover these hard-to-activate branches, which are referred as **targets**.

Figure 1(a) shows that it may be infeasible to cover all targets in a reasonable time using random simulation. To address the inherent limitation of random tests, there are significant prior efforts in automated generation of directed tests [1]–[7]. Test generation using formal methods [8], [9] can cover specific targets directly, but suffer from state space explosion for large designs. Semi-formal approaches, such as concolic testing [10], [11], combine the advantages of random simulation and formal methods to activate targets efficiently. Concolic testing interleaves concrete simulation and symbolic execution, and explores one path at a time to address the state explosion problem.

Concolic testing is successful in generating directed tests in software domain [10]–[12]. Chen *et al.* [13], [14] extended the

Y. Lyu and P. Mishra are with the Department of Computer and Information Science and Engineering at the University of Florida, Florida, 32611-6120, USA. (email: lvyangdi@ufl.edu).

[1]a basic block is a piece of code without any branches in (except of the entry) or out (except at the exit).

software domain tool KLEE [12] as concolic testing engine for hardware/software co-validation on virtual platforms. These approaches are not directly applicable to RTL models, since it has to deal with unrolling multiple CFGs with complicated communication between different models and different clock domains. While there are some early efforts on applying concolic testing to RTL models [15], [16], they are applicable on simple designs with restricted features. Most importantly, they do not address the fundamental challenge in concolic testing - path explosion problem. As a result, they are not suitable for large designs.

While there are recent efforts in applying concolic testing to uniformly cover as many targets as possible in RTL models [17], they did not address the fundamental "path explosion problem" in concolic testing, as shown in Figure 1(b). Uniform test generation tries to maximize the overall branch/statement coverage by utilizing various search techniques, such as depth-first search (DFS) and breadth-first search (BFS). Since the number of paths grow with unrolled cycles, uniform test generation will suffer from path explosion problem, and will not be able to finish within the time limit. As it ignores the priority of activating specific targets, approaches based on uniform test generation usually lead to longer test generation time to cover a specific rare branch. In this paper, we propose a promising approach to guide path exploration to reach a specific target. We refer it as *single-target* method. However, iterative application of this approach to activate thousands of targets is not suitable since a lot of effort will be wasted in overlapping searches, as shown in Figure 1(c). To reduce the number of overlapping searches, we propose efficient learning and clustering techniques to activate multiple targets. Our approach utilizes information from the previous searches, as shown in Figure 1(d). We refer it as *multi-target* method.

In this paper, we make two major contributions:

- We propose a scalable test generation technique using concolic testing of RTL models to activate a specific target. We develop a novel contribution-aware edge re-alignment technique to effectively evaluate the distance between a simulated path and a specific target. The realigned edges are used to guide alternative branch selection to improve both branch coverage and test generation efficiency.
- In order to exploit learning across test generation instances involving multiple targets, we explore two optimization techniques to effectively utilize previous search results. We utilize target pruning to eliminate targets that are covered by the tests generated for activating other targets. We also minimize the overlapping search efforts by employing clustering of related targets to drastically reduce the overall test generation time.

The remainder of the paper is organized as follows. Section II surveys existing test generation techniques using both formal methods and concolic testing. The overview of our framework is outlined in Section III. Section IV presents our proposed test generation approach for activating a specific target. Section V describes various optimizations for activating multiple targets. Section VI presents experimental results. Finally, Section VII concludes the paper.

II. Background and Related Work

In this section, we briefly describe the existing test generation efforts using formal methods as well as concolic testing.

A. Test Generation using Formal Methods

Model checking is widely used for formal (property) verification of RTL models [18]. Bounded model checking (BMC) is promising in automated generation of directed tests [4], [19]. To activate a specific target (functional behavior), the negated version of a property (functional behavior) is fed into a model checker, which will return a counterexample as the test that can activate the target. Binary Decision Diagrams (BDD) based BMC [20] and SAT-based BMC [21] are two widely used formal verification methods [18]. Due to the state explosion problem, model checking approaches are not suitable for large designs. Extensive research have been devoted to reduce the model checking complexity during test generation using various design/property decomposition as well as learning techniques [4], [22], [23]. In spite of these extensive efforts, it is infeasible to generate directed tests using model checking based approaches due to inherent state explosion problem while dealing with complex properties as well as large designs.

B. Test Generation using Concolic Testing

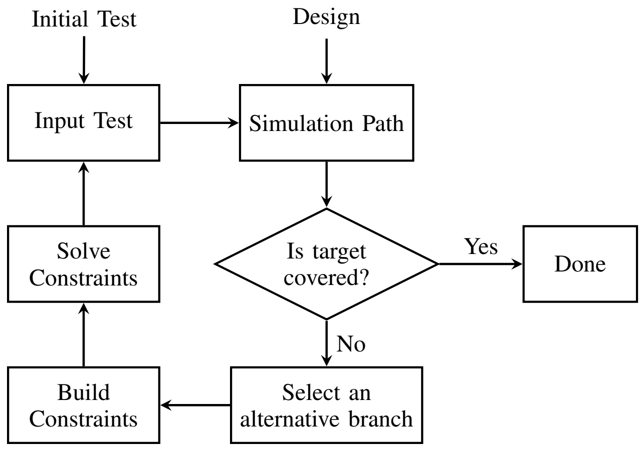


Fig. 2: Overview of concolic testing.

Concolic testing is a promising semi-formal test generation technique by interleaving concrete simulation and symbolic execution. Unlike formal method based approaches that explore all possible (exponential) execution paths at the same time (and leads to state space explosion), concolic testing explores only one execution path at a time. The major steps of path exploration are shown in Figure 2. The first step is to simulate the design using an input vector. If the simulated path covers the target, the input vector is added to the test set. Otherwise, an ***alternative branch*** is selected to create new constraints. Then, the new constraints are fed into a constraint solver. If they are satisfiable, an input vector will be returned to start a new iteration. This process continues until the target is finally covered or the test generation time exceeds a limit.

As concolic testing examines one path at a time, it avoids state explosion problem associated with test generation using formal methods [24]. However, the performance of concolic testing is decided by how the alternative branch is selected. When profitable branches are selected, the simulated path will quickly reach the target. On the other hand, selecting wrong branches may lead to longer test generation time, or even failure to activate the target. Another factor that affects concolic testing is the ***initial test*** (or ***initial path***) that we choose to start concolic testing, which is usually a random test or a manually developed test. When the initial path is already closer to the target, it is easier to reach the target and less likely to be lost in the enormous possible paths.

1) Concolic Testing of Software Designs: There are extensive research efforts in applying concolic testing on software designs [10]–[12]. To quickly cover targets in software domain, structural information from control flow graph is analyzed to guide path exploration [25], [26]. Another semi-formal method that is similar to concolic testing in generating test for a specific target is called symmetric backward execution [27]–[29]. While these approaches are successful in software domain, they are not directly applicable on hardware (SoC RTL models) designs since they have to deal with unrolling multiple CFGs with complicated communication between different models and different clock domains.

2) Concolic Testing of Hardware Designs: Concolic testing has been shown effective in hardware/software co-validation on virtual platforms [13], [14], [30], [31] and high-level modeling using SystemC [32]. While there are some early efforts on applying concolic testing to RTL models [15], [16], they are applicable on simple designs with restricted Hardware Description Language (HDL) features. There are some recent efforts in applying concolic testing of RTL models. Ahmed *et al.* [17] proposed QUEBS to balance exhaustive and restrictive search techniques by limiting the number of times a branch can be selected. While uniform test generation is promising, it suffers from the exponentially growing number of paths which makes exhaustive searching impractical (path explosion problem) for covering the selected (hard-to-activate) targets. As a result, is it not suitable for large designs.

In this paper, we propose an efficient path exploration scheme to improve the quality of explored paths to address the path explosion problem in concolic testing. Moreover, we propose clustering and learning techniques to minimize wasted efforts in overlapping searches while generating tests to activate multiple targets.

III. Overview and Problem Formulation

Given an RTL description of a hardware design, our proposed approach will generate a set of compact tests to cover all the hard-to-activate branch targets. This section is organized as follows. We first describe the modeling of targets. Next, we provide an overview of our proposed approach and outline the organization of the remainder of this paper.

A. Modeling of Targets

In this paper, a validation target (**target**, in short) in RTL model represents a branch condition that the verification engineer would like to cover during validation of RTL models. Specifically, we are interested in the hard-to-activate branches in a traditional simulation-based validation methodology. In addition to branch target in the design, our approach can also be used to validate other scenarios that can be converted to equivalent branch statements, such as assertions [33].

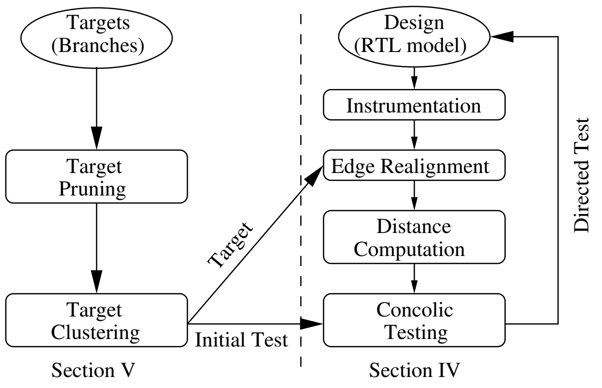


Fig. 3: Overview of our test generation framework.

B. Overview

Figure 3 shows an overview of our test generation framework. The left hand side of the diagram performs optimizations when there are multiple targets (Section V). The right hand side of the diagram utilizes concolic testing to activate a single target (Section IV). In other words, the left hand side will provide one target at a time to the right hand side to produce a directed test. To cover a specific target, the most important step in concolic testing is to decide the alternative branch in each iteration of path exploration, as introduced in Section II-B and Figure 2. To improve the alternative branch selection, we apply three steps to preprocess the RTL code, i.e., instrumentation, edge realignment and distance computation, and utilize the distance information as a heuristic to determine the most profitable alternative branches with regard to a specific target. By selecting the profitable alternative branches, the explored simulation paths are expected to get closer and closer to the target. The details of the preprocessing and path exploration techniques are discussed in Section IV. To cover multiple targets, we propose two optimizations to reuse search information to minimize wasted overlapping efforts, i.e., target pruning and target clustering. The goal of target pruning is to dynamically eliminate the targets that can be covered by the simulation paths for other targets. It decides which target to focus on in the next loop after concolic testing is done for one target. Target clustering dynamically learns and groups targets that are close to each other. For each group, we select the closest simulation path (initial test) to start concolic testing. As introduced in Section II-B, starting with a close path to the target will benefit the path exploration. These two optimizations are introduced in Section V.

Our framework consists of four main steps. First, it applies target pruning and target clustering to decide the next target and the initial test to start concolic testing. Then, it preprocesses the design with regard to the target to assist path exploration. Next, it utilizes concolic testing to generate directed test. Last, the generated test is validated in the original design (without instrumentation and edge realignment) to check if the desired target is covered. After concolic testing finishes for this target, next iteration begins with another target and its corresponding initial test.

IV. Test Generation using Concolic Testing

A major challenge in concolic testing is how to efficiently explore profitable paths to activate a specific target. As shown in Figure 3, our test generation framework to activate a single target consists of four major steps: instrument the code to add print statements, realign edges to reveal contributions of each assignment, compute distance and explore different paths to cover the target. With the help of edge realignment, we are able to heuristically evaluate the distance between a path and the target. As shown in Figure 1, both random and uniform test generation do not consider the quality of a selected path. In contrast, our directed path selection tries to explore paths that are "closer" to a specific target. This section describes these steps in detail.

A. RTL Code Instrumentation

The first step is to instrument the original design. The goal of instrumentation is to provide information of a concrete path for symbolic execution. There are two possible ways to do symbolic execution. One option is to modify the RTL simulator directly such that symbolic execution is performed along with concrete simulation. The other one is to instrument the original design such that path execution information can be dumped. After simulation is done, the symbolic execution engine will parse and analyze the dumped traces. Our framework utilizes the latter method, since it is more adaptive to different simulators and languages. The simulation traces would be the same irrespective of the simulator. Note that the instrumented RTL code is just for test generation. In our experimental evaluation, the original design is used to verify that the generated tests activate the expected branches.

Our framework first parses the design and constructs its control flow graph. Then, it marks each basic block with a unique identifier (BBi for the i^{th} basic block). The unique identifiers of basic blocks help symbolic engine build the constraints which contain all the assignments inside each basic block. Then, we instrument the design by adding a print statement at the end of each basic block. The example of an instrumented design is shown in Listing 1 with the added print statements shown in gray. Its corresponding CFG is shown in Figure 4 (without dashed lines). At the same time, our framework generates a testbench module that is able to read the tests generated by our approach and provide the stimuli to the instrumented design. After simulating one input vector, a trace showing the executed basic blocks is dumped and analyzed. For example, with a random input test, it is highly likely to get a trace [BB1, BB8, BB6, BB8, BB6, BB8, ...]. After reconstructing the simulation path from the trace, next step of concolic testing is to choose a new path to explore and generate a corresponding test to exercise the path.

Listing 1: Example 1

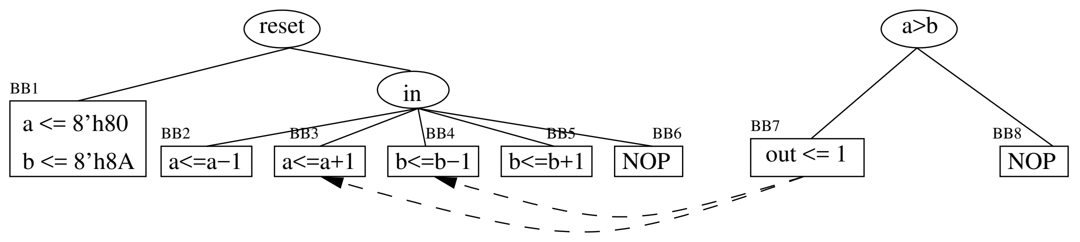
```
module top(clock, reset, in, out);
reg [7:0] a, b;
reg out = 1'b0;
always @(posedge clock) begin
    if (reset == 1'b1) begin
        a <= 8'h80; b <= 8'h8A;
        $display("BB1");
    end
    else case (in)
        8'h01: a <= a - 1; $display("BB2");
        8'h23: a <= a + 1; $display("BB3");
        8'h45: b <= b - 1; $display("BB4");
        8'h67: b <= b + 1; $display("BB5");
        default:           $display("BB6");
    endcase
end
always @(posedge clock) begin
    if (a > b) begin
        out <= 1'b1; $display("BB7");
    end
    else begin
        $display("BB8");
    end
end
endmodule
```

B. Contribution-aware Edge Realignment

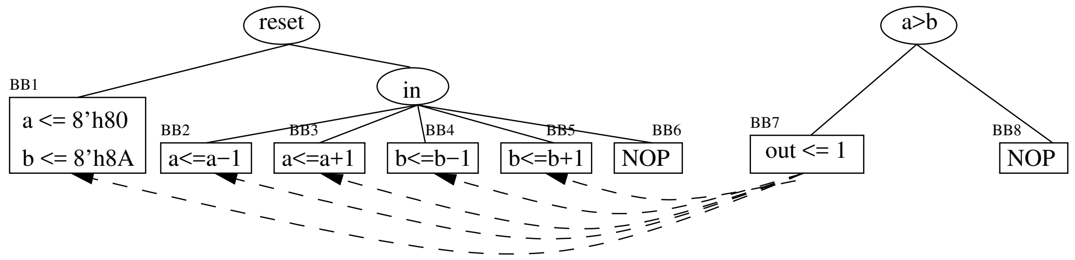
Our framework tries to explore paths that can take it "closer" to a specific target. Alternative branch selection in concolic testing is essentially "forcing" the next execution path to pass through a specific block. When a "good" alternative branch (block) is selected, the simulation path will get closer to the target. On the other hand, a "bad" alternative branch (block) will lead the simulation trace randomly or far away from the target. The goal of our edge realignment is to figure out the contribution of each block in activating a specific target. In this section, we propose a contribution-aware edge realignment scheme to enable smart selection of profitable alternative branches while exploring new paths.

Before we describe the details of the process, let us use an example to show the results of edge realignment. Let us consider the example in Listing 1 to activate the branch in BB7 with the branch condition of *(a>b)*. We note that the original CFG in Figure 4 (without dashed lines) provides no information about which block is good or bad. To manually write a test to activate BB7, a test writer tries all blocks containing the assignments for signals a and b. While this manually backward tracking is viable for small designs, it is not feasible for large designs due to scalability issues. Instead of manually figuring out which basic blocks are relevant in activating BB7, we create reference edges to directly connect the blocks with profitable assignments to our target block. For example, the dashed lines in Figure 4(a) connect BB3 and BB4 to our target block BB7. These realigned edges instruct our concolic testing framework to prefer BB3 and BB4 to other blocks in selecting alternative branches. Intuitively, the paths across these two blocks are more likely to activate BB7 than the other blocks.

For the ease of illustration, we use the following notations. We use s to represent a global state, containing a snapshot of the values of all registers and wires in a specific time. Let $g(\cdot)$ be the guard condition of a basic block. For example, the



(a) Our contribution-aware edge realignment. Realign each block to the assignments that contribute to the activation of that block.



(b) [34] realigns each block to the assignments that are satisfiable.

Fig. 4: Comparison of our edge realignment and [34]. The dashed lines represent the realigned edges. As edge realignment is critical in guiding alternative branch selection, one incorrectly realigned edge can lead to significant performance degradation.

global state $\boldsymbol{s_0} = \{a_0 = 8'h80, b_0 = 8'h8A\}$[2] after executing BB1. The subscript of a variable is used to keep track of the different values during the whole simulation. In other words, the subscript of a variable increments by 1 whenever an assignment to the variable is executed. The guard condition for BB7 is $g(\cdot) = a > b$. If we evaluate the guard condition directly on $\boldsymbol{s_0}$, i.e., $g(\boldsymbol{s_0}) = (a_0 > b_0) = False$, it represents that BB7 cannot be visited right after executing BB1. We use $f(\cdot) : \boldsymbol{s} \rightarrow \boldsymbol{s'}$ to represent any assignment that may change the global state. For the assignment $f = (a <= a+1)$, it changes the global state $\boldsymbol{s}_0$ to $\boldsymbol{s'} = f(\boldsymbol{s_0}) = \{a_1 = 8'h81, b_0 = 8'h8A\}$, where only the value of a is changed. For the ease of representation, a sequence of assignments is represented using composition, i.e., $\boldsymbol{s'} = \boldsymbol{s} \circ f_1 \circ f_2 ... \circ f_n$, where the assignments $f_1, ... f_n$ are executed in order. The goal of concolic testing is to find a viable sequence of assignments $f_1, ..., f_n$ that will hit a specific target, i.e., $g(\boldsymbol{s'}) = True$.

A naive way to realign edges was proposed in [34] by simply checking the satisfiability of one assignment and the guard condition of a block. For example, assuming the guard condition of a target is $(v \ \& \ 0 == 0)$, which requires v to be an even number, the naive edge realignment will realign the target to all blocks where v is possibly assigned an even number. While this approach is promising in connecting simple conditions to a single assignment, the naive checking introduces a large number of redundant edges which can mislead path selection. The result of applying the naive edge realignment to Listing 1 is shown in Figure 4(b). Compared to our expected results in Figure 4(a), the naive approach has two major problems. The first one is that the naive approach lacks the checking of contribution. The naive edge realignment scheme in [34] connects the target to all satisfiable assignments of its variables, e.g., BB7 is connected to BB2 in Figure 4(b). It is due to the satisfiability of the guard condition $a > b$ and the assignment $a <= a - 1$, i.e., $((a_1 == a_0 - 1)$ *and* $(a_1 > b_0))$ is satisfiable. However, it is easy to see that selecting this assignment is not profitable in achieving the target BB7. When $a_0 > b_0 + 1$ before the assignment is executed, the target BB7 is already activated. Otherwise, the assignment will lead the search path to be far away from BB7. The second problem is the level of satisfiability checking. The naive edge realignment checks the contribution of each individual assignment, rather than all assignments inside a block. For example, assignment level satisfiability checking will connect BB7 to the block containing the assignment $a <= 8'h80$ (BB1). However, when we consider all the assignments inside BB1 together, it is clear to see that executing BB1 will never help to activate the target BB7. While it is possible to manually check the contribution for small designs, it is infeasible when the design is large and the condition is complex. We propose a contribution-aware block-level edge realignment in Algorithm 1.

In this algorithm, the block queue BQ maintains all the basic blocks that need to be aligned. Initially, BQ contains all the targets. We first expand the guard condition g for the current block bb to get all the related variables. Then we check all the assignments that are related to any of these variables. For each of these assignments f, we first find out the basic block bb' that contains f. Then, we evaluate its contribution to the guard condition g based on Definition 4.1.

Definition 4.1: A basic block B has a *contribution* to a guard condition g, if there exists some initial global state $\boldsymbol{s}$, such that $\boldsymbol{s}$ does not satisfy the guard condition g, but the state after executing all assignments inside B satisfies the guard condition. Assume that $f_1, f_2, ..., f_n$ are the assignments inside B. The contribution of B to the guard condition g can be checked by the satisfiability, $g(\boldsymbol{s}) = False$ and $g(\boldsymbol{s} \circ f_1 \circ f_2 \circ ... \circ f_n) = True$.

[2]For the simplicity of explanation, we only show the relevant variables to our condition g in $\boldsymbol{s}$. For example, when g is $a > b$, we only show the variables a and b. In our framework, all variables are kept in the global state.

Algorithm 1 Edge Realignment

```
Input: CFG, Target Queue ($TQ$)
Output: Realigned CFG
1: Push all targets to block queue $BQ$
2: while $BQ$ is not empty do
3:   Current block, $bb \leftarrow BQ.pop()$
     // Update edge for block $bb$
4:   $g \leftarrow$ expanded guard condition of $bb$
5:   for all variables $v \in g$ do
6:     for all assignments $f$ to $v$ do
7:       $bb' \leftarrow$ the block of $f$
8:       $f_1, f_2, ..., f_n \leftarrow$ all the assignments of $bb'$
9:       if $g(\boldsymbol{s}) = False$ and $g(\boldsymbol{s} \circ f_1 \circ f_2 \circ ... \circ f_n) = True$
         for any $\boldsymbol{s}$ then
10:        Add $bb'$ to $bb.predecessors$
11:        $BQ.push(bb')$ if $bb'$ is not visited
12:      end if
13:    end for
14:  end for
15: end while
```

Contribution checking forces guard condition g to be false in the beginning, followed by executing all the assignments inside a basic block B, and then checks if g will be satisfied. If it is satisfiable, the block B has a contribution to g. For example, the block BB3 contains only one assignment $f = (a <= a + 1)$. It has a contribution to the guard condition $g = (a > b)$, because $((a_0 \leq b_0)$ *and* $(a_1 = a_0 + 1)$ *and* $(a_1 > b_0))$ has at least one solution. On the contrary, the block BB2 with the assignment $f = (a <= a-1)$ has no contribution to the guard condition, as the satisfiability equation $((a_0 \leq b_0)$ *and* $(a_1 = a_0 - 1)$ *and* $(a_1 > b_0))$ has no solution. Similarly, BB1 has no contribution since $((a_0 \leq b_0)$ *and* $(a_1 = 8'h80)$ *and* $(b_1 = 8'h8A)$ *and* $(a_1 > b_1))$ has no solution. The results of satisfiability checking for the block BB7 are shown in Table I.

TABLE I: The results of satisfiability checking in line 9 of Algorithm 1 for the target BB7.

Block	Equation	SAT
BB1	$(a_0 \leq b_0) \wedge (a_1 = 8'h80) \wedge (b_1 = 8'h8A)$ $\wedge (a_1 > b_1)$	UNSAT
BB2	$(a_0 \leq b_0) \wedge (a_1 = a_0 - 1) \wedge (a_1 > b_0)$	UNSAT
BB3	$(a_0 \leq b_0) \wedge (a_1 = a_0 + 1) \wedge (a_1 > b_0)$	SAT
BB4	$(a_0 \leq b_0) \wedge (b_1 = b_0 - 1) \wedge (a_0 > b_1)$	SAT
BB5	$(a_0 \leq b_0) \wedge (b_1 = b_0 + 1) \wedge (a_0 > b_1)$	UNSAT

After finding a good block bb' in line 7, we add it to the predecessors of bb, i.e., creating an edge to connect bb' and bb. For example, as the block BB3 has a contribution to $a > b$, it is added to the predecessors of BB7. Since BB3 is not visited before, it will be added to the end of BQ. In some future iteration, BB3 will be selected as the current block, and our algorithm will realign good assignments for its guard condition. It is easy to see that Algorithm 1 is fast since no basic block needs to be visited more than once. A good edge realignment scheme is important because even a single bad realigned edge will waste a lot of searching time in finding good alternative paths during path exploration, and it can lead to a wrong direction, which will be demonstrated in Section VI-E.

C. Distance Computation

Edge realignment connects a target to the blocks that have direct contributions. To quantify the contribution of all blocks, we use a distance measurement based on the realigned control flow graph. A block with lower distance means it is closer to the target, i.e., more likely to contribute to the activation of the target.

First, we define the distance between a basic block and the target. With the realigned CFG in Figure 4(a), we start from our target BB7 and perform breadth-first traversal in the direction along the predecessors. For BB7, we initialize the distance as 0, and increment the distance by 1 when we traverse an edge. The distances of the basic blocks in the first *always* block of Listing 1 are shown in Figure 5, which is unrolled for three cycles. Note that the distances of basic blocks that are never visited are not shown, which will be initialize to ∞ in our framework. For each basic block bb, the distance of bb is denoted as $bb.distance$. Next, we define the distance between a path and the target in Definition 4.2.

Definition 4.2: Assume a simulation path is constructed by a trace $\{\{bb^1_1, bb^1_2, \ldots, bb^1_{i_1}\}$, $\{bb^2_1, \ldots, bb^2_{i_2}\}$, …, $\{bb^k_1, \ldots, bb^k_{i_k}\}\}$, where bb^j_i represents the i^{th} basic block in j^{th} clock cycle. The distance between the path and a target is the minimum distance among all blocks, i.e., $\min_{i,j} bb^j_i.distance$.

Figure 5 shows the example of three paths. As all the blocks along P1 and P3 have the distance ∞, their distances to the target are ∞. On the other hand, as P2 passes through BB3 in the second clock cycle, the distance of P2 is 1. When we inspect the final state after executing these three paths, P2 is “closer” to the target, since P2 executed one more $a <= a+1$. In other words, the distance is a good quality indicator of a path. This distance definition also emphasizes the importance of good edge realignment schemes. If we realign the CFG using the naive approach (as shown in Figure 4(b)), the path P3 will have distance 1, same as P2. However, since P3 executed $a <= a - 1$, the final state is actually further away from our target. These realigned edges will mislead directed path exploration as described in the next section.

D. Path Exploration

In this section, we present a greedy path exploration scheme in activating a specific target based on our distance heuristic. We illustrate the usefulness of distance information in our automated path exploration scheme.

Algorithm 2 describes our greedy path exploration to quickly reach a specific target by selecting the most profitable alternative branch. To better illustrate how we select alternative branches and explore new paths, we use the example in Figure 5. It first computes the distance for all blocks as introduced in Section IV-C. Assume that the initial path p is P1. Next, it builds constraints vector from the simulation trace of p, $S = \{\{a_1 = 8'h80, b_1 = 8'h8A\}, \{\}, \{\}\}$, where

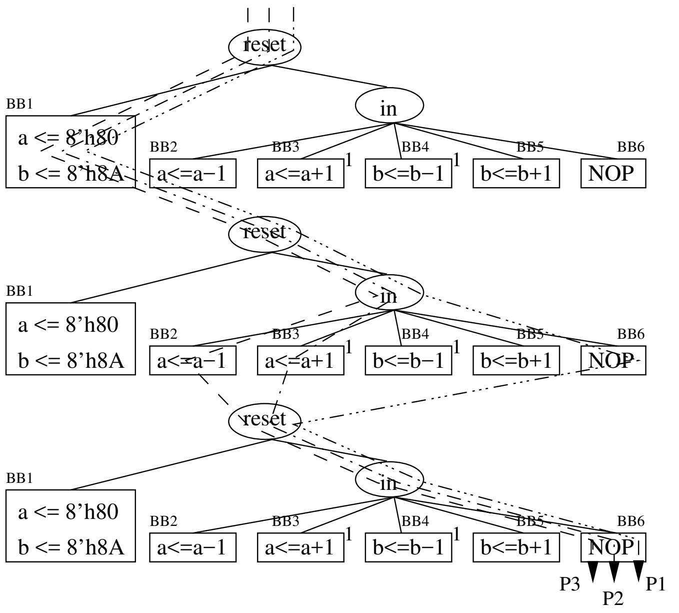


Fig. 5: The distance between a basic block and the target in realigned CFGs is marked for BB3 and BB4 (others are ∞). Only the first *always* block of Listing 1 is shown for the ease of illustration.

each inner vector represents all executed statements one clock cycle. Then, we try all alternative branches[3] from current path p and check if they are viable. For example, we want to check if BB7 is a valid block in clock 2. A new constraints vector is built by combining all constraints before BB8 in clock 2 and the new chosen branch, i.e., $\{\{a_1 = 8'h80, b_1 = 8'h8A\}, \{a_1 > b_1\}\}$. As it is not satisfiable, BB7 is not a valid block in clock 2. On the other hand, BB2 in clock 2 is a valid block, as $\{\{a_1 = 8'h80, b_1 = 8'h8A\}, \{in_2 == 8'h01\}\}$ is satisfiable. In this way, we can find all valid alternative blocks, which are BB2, BB3, BB4 and BB5 in both clock 2 and clock 3. The next step is to sort these blocks by the distance and clock cycle. Since BB3 and BB4 have smaller distance than BB2 and BB5, one of the possible order is $\{\text{BB3}^2, \text{BB4}^2, \text{BB3}^3, \text{BB4}^3, \text{BB2}^2, \text{BB5}^2, \text{BB2}^3, \text{BB5}^3\}$, where the order of BB3 versus BB4 and the order of BB2 versus BB5 in the same clock are random since each pair has the same distance. Assume that we select BB3 in clock 2 as our best alternative block in line 10. A new constraints vector will be constructed, consisting of all constraints from the beginning to BB3 in clock 2. We solve the constraints to get a test and simulate it. Let us assume that the new path is P2. Before searching in the next iteration, we set $clock = 2$ such that only the sub-path after clock 2 is checked in searching for valid alternative branches for P2, i.e., the sub-path (first two clocks) of P2 is locked. A new iteration will repeat the process until the target is activated. There are two key ideas in our algorithm.

1) The usage of our distance metric defined in Section IV-C provides our greedy algorithm a heuristic to explore relatively "close" paths to the targets. For example, in our first iteration, we would prefer P2 over P3 in Figure 5 since P2 has smaller distance than P3.
2) The usage of $clock$ maximizes the exploitation of previous good choices, and avoids toggling best alternative blocks when multiple blocks have the same distance. If the $clock$ is not enforced in our algorithm, the best alternative blocks would toggle between BB3^2 and BB4^2 in the following iterations of the previous example. If the explored path could not cover the target, which is likely since we allow the input signal to be random in the remaining cycles, the process will continue until one lucky test activates the target by chance.

[3]We assume the first clock is only used to reset all signals. Therefore, we will skip the first clock cycle in finding alternative branches.

Algorithm 2 Path Exploration

```
Input: realigned CFG, Target Queue $TQ$, search limit $limit$,
    unrolled cycles $unroll$
Output: A test set $T = \{t_1, t_2, \ldots, t_n\}$
 1: for all target $\in TQ$ do
 2:     Compute the distance from target for all blocks
 3:     Random simulation and get the path $p$
 4:     $iteration = 0$
 5:     $clock = 0$
 6:     while $iteration < limit$ do
 7:         Build constraints vector $S$ from the trace of $p$
 8:         $AB \leftarrow$ all valid alternative branches (blocks) after
            cycle $clock$
 9:         Sort $AB$ by distance and clock
10:         Randomly choose one of the best alternative branches
            (blocks) to flip
11:         $clock \leftarrow$ the clock of the chosen branch
12:         Build the new constraints vector
13:         Use a constraint solver to solve the constraints
14:         Simulate the design with returned test and get a new
            path $p$
15:         $iteration = iteration + 1$
16:         if $p$ activates the target then
17:             Add the test to $T$
18:             Break
19:         end if
20:         if $clock == unrolled$ then
21:             Increment the distance of all blocks in $p$
22:             $clock = 0$
23:         end if
24:     end while
25: end for
26: Return $T$
```

1) Dynamic Distance Update: One important thing in our algorithm is the usage of $clock$. The intuition behind $clock$ is that since we have made so much effort in finding the best alternative branch (block), we do not want it to be replaced until we have tried enough possibilities and could not find a solution. Whenever we find an alternative block in line 10, we set $clock$ to the clock cycle of the chosen block. Therefore, the sub-path from the beginning to the chosen block is "locked" in the following iterations. The next alternative branch is chosen from the remaining cycles. After $clock$ reaches the unrolled cycle, we reset the $clock$ and increment the distance of current path p in line 20-22.

The intuition behind dynamic distance update is that we cannot fully rely on the distance given by our static analysis of CFG. Since the distance is statically computed, it is likely that all paths through a block with a small distance could not activate our target, which is almost impossible to examine because the number of possible paths is exponential. Therefore, we stick with the reasonable evaluation (block-level satisfiability) and apply dynamic distance update to mitigate the shortsighted nature of our edge realignment scheme.

In Algorithm 2, distance is updated when $clock$ reaches unrolled cycles and the target is not covered (line 20). In previous iterations, we have greedily chosen a few profitable alternative branches. We increment the distance of all blocks in the current path p, which contains all the chosen blocks in previous iterations. Through dynamic distance update, our approach is able to explore other blocks instead of exploiting the good blocks from static realignment all the time. For example, assume the distance of BB2 is 10, rather than ∞ in Figure 5. After 10 times of trying BB3 and failing to activate the target, the distance of BB2 would be smaller than BB3. Therefore, in the next iteration, BB2 will have higher priority of being chosen as the alternative branch than BB3. Combining the usage of $clock$ and dynamic distance update, our approach balances the exploration and exploitation during path selection.

V. Optimizations for Covering Multiple Targets

To extend our test generation framework to handle multiple targets, we propose two optimization techniques: target pruning and target clustering. The focus of these techniques is to effectively utilize the structure of the design and previous search information to generate tests efficiently as shown in Figure 1(d). The key idea behind these two techniques are summarized below:

1) Target pruning is designed to reduce the number of targets without sacrificing the coverage. While existing target pruning techniques try to remove redundant targets after generating test for all of them, we utilize CFGs and the order of targets to efficiently prune targets prior to and during test generation.
2) Proposed target clustering connects each target with the closest simulated path, therefore, improves initial path selection. One problem of iteratively applying Algorithm 2 is that it ignores all precious search information while activating previous targets, leading to wasted effort in overlapping searches (see Figure 1(c)). Target clustering dynamically groups the remaining targets into different clusters. Each cluster has its own closest simulated path. When one target is selected as the current target, we use the closest simulated path as the initial path to replace line 3 in Algorithm 2.

A. Target Pruning

To accelerate the speed of concolic testing in multi-target scenarios, we prune redundant targets by analyzing the CFG and controlling the order of targets. We exploit both static and dynamic pruning to minimize the number of targets. To illustrate the static process using CFG, consider the simple RTL design in Listing 2 as an example. Figure 6 shows its CFG with T0, T1 and T2 to represent the three targets. If T2 is reachable, we can safely remove T0 from the target list. Formally, we can prune all the dominator nodes of the targets. Suppose the initial set of targets is TS. For each target $T \in TS$, let the dominators of T be the set $DM(T)$. Therefore, the effective target set after pruning, $TS' = TS - \cup_{T \in TS} DM(T)$.

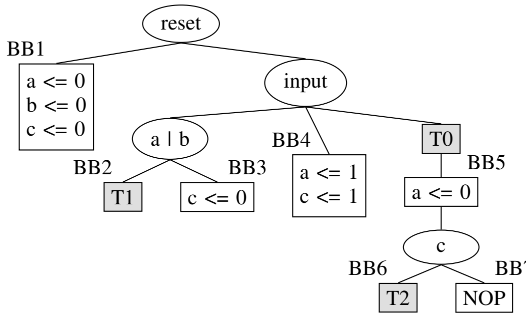


Fig. 6: CFG for the design in Listing 2. T0, T1, and T2 represent three targets.

Listing 2: Example 2

```
module top(clock, reset, in, out);
if (reset == 1'b1) begin
    a <= 0; b <= 0; c <= 0;
end
else case (input)
    2'b00:
        if (a | b)  $display("Target T1");
        else c <= 0;
    2'b10, 2'b01: begin
            a <= 1; c <= 1;
        end
    2'b11: begin
        $display("Target T0");
        a <= 0;
        if (c)      $display("Target T2");
        end
endcase
```

However, this approach may not work when T2 is not reachable, but T0 is reachable. In this case, removing T0 from the target list does not make sense. Rather, we should remove T2. One engineering choice would be to prune targets as usual, but keep track of the pruned targets. If a test cannot be generated for a target (e.g., in a reasonable time), add back the dominators that were pruned because of this target. To avoid directly pruning targets for both efficiency and coverage, we topologically sort the targets. The order ensures that the target T_d is always behind the target T_o in the target queue, where T_d is a dominator of T_o. This way, test generation for T_d will only be done if it is not covered by previously generated tests for T_o. For targets in a dominator chain, the deep targets in CFG will always be in front of the shallow ones. For examples, if the original target queue is <T0, T1, T2>, it would become <T1, T2, T0> after target pruning.

Dynamic target pruning also takes advantage of the order of targets to fully utilize previously explored paths. When the explored paths of a target can cover the other targets, the latter

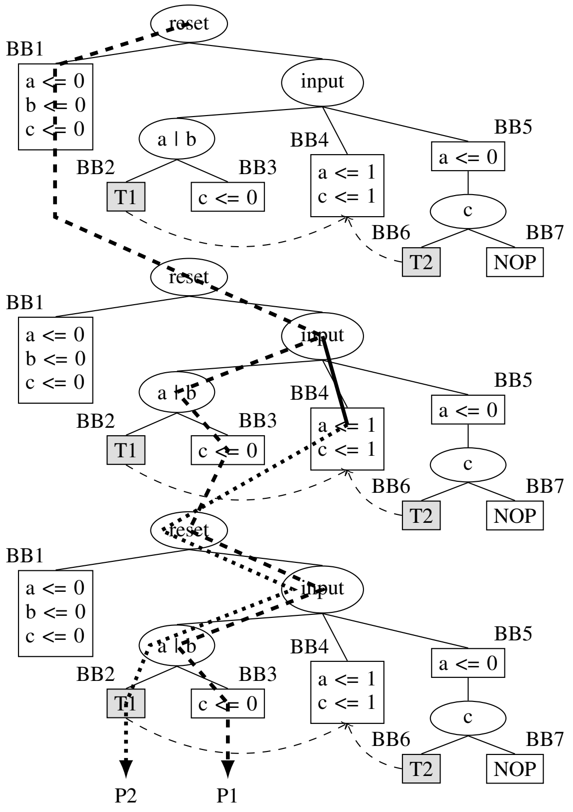


Fig. 7: The design in Listing 2 is unrolled for three cycles. The initial path is P1 with $input$ being $2'b00$ for all cycles. The selected alternative branch is shown in bold solid line, and the simulated path is P2 for the test from the constraint solver. P2 covers T1 and becomes the closest simulated path for T2 during target clustering.

can be pruned. However, it is unknown which targets can be pruned in the beginning. Therefore, we propose a round-robin scheduling in selecting targets. Instead of trying to solve one target until timeout in one round, we split the iteration limit into multiple rounds. If a target cannot be activated in one round, we put it to the end of the target queue. There are two advantages of this scheduling. First, the pruned target may be covered while generating tests for other targets. Second, target clustering may find a better initial path for this target in the following rounds, as introduced in next section.

B. Target Clustering

Our approach learns target clustering dynamically, and utilizes the clustering to achieve the most profitable initial path for concolic testing. There are mainly two advantages in selecting a profitable initial path. The first is to improve test generation efficiency. When the initial path is already close to the target, fewer concolic iterations are needed to activate the target compared to initial paths that are far away. The second advantage is to improve coverage. Although coverage is mainly controlled by how an alternative branch is selected, a better initial path means fewer concolic iterations, reducing the probability of getting lost in a large number of misleading alternative branches.

Since current designs separate different functionalities into independent modules, one random simulation path may be far away from our desired target (e.g., if it involves interaction of multiple modules). On the other hand, many targets from the same module or the same finite state machine may share a common path. For these targets, search paths for one target may be close to the other targets. To better utilize the effort of previous explorations, we propose a dynamic clustering approach to learn the most profitable initial path. For each target, we keep the simulated path with the smallest distance evaluated based on the CFGs after edge realignment, called the *closest simulated path*. We place targets in one cluster if they share a common closest simulated path. Initially, all targets are in the same cluster with the closest simulated path being a random path. The simulated path in concolic iteration is used to split clusters into smaller ones.

We use the example in Figure 7 which shows the first two steps of exploring paths for the target T1 in Listing 2. Assume that the design is unrolled for 3 cycles and $input$ is $2'b00$ for all clock cycles. Then, the initial path is P1. As BB4 has the smallest distance to T1 and it is reachable in the second cycle of P1, the alternative branch (bold solid line) is taken and an input vector is returned by the constraint solver. Assume P2 is the simulated path of the returned input vector. At the same time, targets are dynamically clustered as follows. T1 and T2 are initially in the same cluster with the closest simulated path being P1. After one concolic iteration, P2 is found and used to update the cluster. As P2 visited BB4 in the second clock cycle, it is closer to T2 than P1. Then the cluster and the closest simulated path is updated for T2. When T2 is selected as the current target, we want to start with its closest simulated path (P2) to avoid overlapping search. This technique effectively eliminates the overlapping search problem. Target clustering also emphasizes the importance of a good edge realignment and distance evaluation scheme. With an incorrect distance evaluation, a target may start from a path that is worse in activating the target, resulting in longer test generation time or failure to activate the target.

VI. Experiments

A. Experimental Setup

To evaluate the effectiveness and efficiency of our approach, we compared the performance of our proposed approach with state-of-the-art techniques including uniform test generation (QUEBS) [17] and bounded model checking (EBMC) [35], [36]. The experiments were conducted in a server machine with Intel Xeon CPU E5-2698 @2.20GHz. Our approaches utilize the Icarus Verilog Target API [37] for parsing and generation of abstract syntax tree of RTL code. Prior to applying the framework, the design is first flattened using *flattenverilog* tool from *Design Player Toolchain* [38]. Yices SMT solver is used for constraint solving [39].

B. Performance Comparison

In this experiment, we compared the performance of our approach to EBMC [35], [36] and QUEBS [17]. A variety of benchmarks are selected from ITC99 [40], TrustHub [41],

TABLE II: Comparison of target coverage using [17], [34] and our approach on 20 targets in each benchmark.

Bench	cycle	lines	EBMC [35]			QUEBS [17]			Our Approach			Impro. / EBMC		Impro. / QUEBS	
			cvr	time	mem	cvr	time	mem	cvr	time	mem	time	mem	time	mem
b10	30	182	20	4.1s	31MB	20	0.12s	9MB	20	0.02s	9.5MB	205x	3.3x	6x	-1.1x
b14	50	698	20	243s	467MB	20	21.6s	34MB	20	1.3s	15MB	187x	31x	17x	2.3x
ICache	50	258	20	6.3s	48MB	20	4371s	1.6GB	20	0.15s	18MB	42x	2.7x	29140x	89x
DCache	10	562	20	20s	138MB	20	1.27s	13MB	20	0.34s	16MB	59x	8.6x	3.7x	-1.2x
Exception	15	666	20	6.9s	40MB	20	3.3s	15MB	20	2.2s	23MB	3.1x	1.7x	1.5	-1.5x
usb_phy	20	1039	20	3.1s	26MB	12	8.2s	34MB	20	134s	138MB	-50x	-5.3x	-16x	-4.1x
T1100	10	544k	20	2386s	8.1GB	-	-	-	20	55s	1.2GB	43x	6.8x	-	-
T2000	10	456k	†	†	†	-	-	-	20	74s	1.2GB	-	-	-	-
Average*	-	-	20	381s	1264MB	19	734s	284MB	20	33s	327MB	69x	7x	4859x	13.9x

†EBMC produced errors and did not finish this benchmark.
*During average computation, we omitted the benchmarks that did not finish.

and OpenCores [42] as shown in Table II. We omit or1200 in the names of the benchmarks or1200_ICache, or1200_DCache and or1200_Exception from OpenCores [42], and omit AES in the names of AES-T1100 and AES-T2000 from TrustHub [41] for simplicity. All these benchmarks contain hard-to-activate branch targets, providing a reasonable test generation complexity. For target selection, we first ran the benchmarks with one million random tests. Then, we selected 20 rarest branches as our targets. For each Trojan-inserted benchmark form TrustHub (AES-T1100, AES-T2000), the selected targets contain 5 rare branches from the Trojan area. There is one rare branch from AES-T2000 that is not included, as it can only be covered after 2^{127} clock cycles. The number of unrolled cycles are chosen such that the hard-to-activate branches can be covered. In practice, a designer can start with a reasonable number of unroll cycles, and increment it in an iterative fashion until all the targets are covered. The number of unroll cycle problem is the same as the bound determination in bounded model checking [36]. Therefore, after we decided the number of unrolled cycles, we set the same number for the bound in EBMC. We set a new target for EBMC each time, and report the accumulated performance. Since the goal of QUEBS [17] is to cover all branches, we terminated it once it covered all of our selected targets. For the round-robin scheduling of selecting targets in our approach, we set the iteration limit to be 20 in each round.

The performance comparison is shown in Table II. The second column shows the number of unrolled cycles for each benchmark and the third column represents the number of lines of code in each flattened design. For each approach, we report the number of covered targets (*cvr*), the test generation time (*time*) and memory usage (*mem*). All 20 targets are covered in three approaches except that QUEBS only covers 12 branch targets in usb_phy. Although the main idea of QUEBS is to uniformly cover all branches using BFS or DFS, it fails to cover some branch targets due to the trade-offs made by the authors [17] to balance repeated search and overall coverage. Compared to our approach, QUEBS performed worst in two of the benchmarks - b14 and ICache. For ICache, our approach gains 29140 times improvement in test generation time and 89 times improvement in memory usage. This is because these two benchmarks are unrolled 50 cycles. If the number of unrolled cycles keeps growing, QUEBS is expected to face path explosion problem since the total number of branches grows exponentially with the number of unrolled cycles. The scalability issue of QUEBS becomes worse with the complexity of designs. For two Trojan-inserted AES designs (T1100 and T2000) with around 500k lines of code after flattening, QUEBS cannot finish within the time limit (one week). Compared to EBMC, our approach is both time efficient and memory efficient. Note that EBMC reported some errors in AES-T2000, hence the comparison does not include AES-T2000. For the largest benchmark, AES-T1100, our approach can activate 20 targets in 55 seconds with 1.2GB memory usage, while EBMC takes around 40 minutes to finish and consumes 8.1GB memory. For larger benchmarks, we will discuss in Section VI-C to explore the scalability of these two approaches. Overall, our approach provides significant improvement compared to EBMC in both test generation time (69x on average, up to 205x) and memory usage (7x on average, up to 31x improvement).

C. Scalability Comparison

In this experiment, we examined the scalability of our approach and EBMC. In particular, we compared the memory requirement of our approach to EBMC, since the main challenge of applying model checking to large benchmarks is the state explosion problem. As QUEBS faces path explosion problem for benchmarks larger than AES, we omitted QUEBS in this experiment and only compared our approach to EBMC. In other words, QUEBS is less scalable than EBMC due to path explosion problem in covering branch targets. Note that the memory requirement is also dependent on the complexity of the design and the branch target. Therefore, to minimize the impact of other factors, we utilized the same design with various unrolled cycles and sizes to examine the scalability of our approach.

First, we examined the effects of unrolled cycles on memory requirements. We used the benchmark or1200_ICache with the number of unrolled cycles increasing from 50 to 250, as shown in Table III. As we can see, the memory requirements of EBMC grow from 48MB to 218MB, while the memory requirements of our approach grow from 18MB to around 30MB. Note that the memory requirement of our approach is not consistently growing with the increasing number of unrolled cycles. It is due to the randomization introduced in path exploration. When the explored paths happen to be close to the target, the exploration process is faster, and the overall memory usage will be smaller. In contrast, the memory requirements in EBMC continuously grow with the number of unrolled cycles. As any formal methods, EBMC tries to explore all paths at a time. With more unrolled cycles, the state space of the design is expected to grow. If we check the trends of memory requirements with respect to the number of unrolled cycles in Figure 8, we can see that the memory requirements in our approach grow slower than EBMC.

TABLE III: Comparison of memory requirements using EBMC and our approach in or1200_ICache with different number of unrolled cycles.

Bench	cycles	lines	Memory Requirements (MB)		
			EBMC	Our	Reduction
ICache	50	258	48	18	2.7x
	100	258	90	26	3.5x
	150	258	132	23	5.7x
	200	258	175	30	5.8x
	250	258	218	28	7.8x
Average	-	258	133	25	5.3x

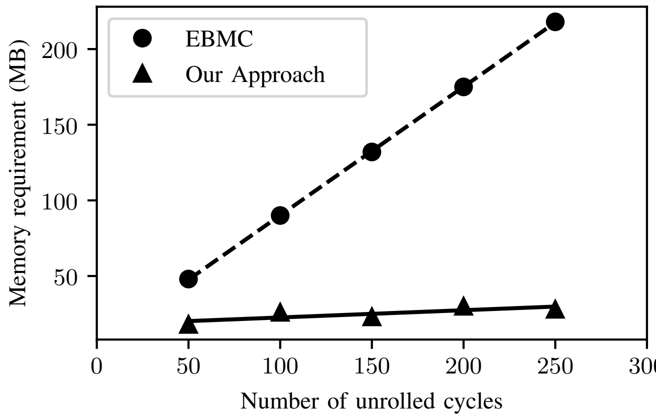


Fig. 8: Comparison of memory requirements of our approach and EBMC in or1200_ICache with various unrolled cycles.

In the previous experiment, we fixed the size of the design and varied the number of unrolled cycles. Next, we examined the impacts of the size of the benchmarks on memory requirements, using six custom AES benchmarks as shown in Table IV. These benchmarks differ only on the number of rounds, which are indicated in the names. For example, aes_20 has 20 rounds compared to 10 rounds in a typical 128-bit AES. By changing the number of rounds, the size of these benchmarks are easily controlled to demonstrate the scalability. The number of lines of code and the total number of branches are shown in the third and fourth columns, respectively. In each benchmark, we inserted one Trojan in the same way as AES-T1100, and created a branch to check if the Trojan is activated. This branch is selected as the target and it is not covered by millions of random simulations. The number of unrolled cycles are five cycles more than the number of rounds to ensure that the branches in all rounds have a chance to be activated.

The experimental results are shown in Table IV. As we can see, the average memory reduction of our approach compared to EBMC is 10 times. For the largest benchmark aes_40 with 1.7 million lines of code after flattening, EBMC needs at least 34GB memory while our approach only requires 3GB. Another

TABLE IV: Comparison of memory requirements using EBMC and our approach on one target.

Bench	cycles	lines	total branches	Memory Requirements (GB)		
				EBMC	Our	Reduction
aes_15	20	544k	123k	6.4	0.9	7.1x
aes_20	25	668k	164k	10.3	1.3	7.9x
aes_25	30	886k	205k	15.0	1.6	9.4x
aes_30	35	1003k	246k	20.7	2.1	9.9x
aes_35	40	1169k	287k	27.1	2.5	10.8x
aes_40	45	1693k	328k	34.3	3.0	11.4x
Average	-	994k	225k	19	1.9	10x

observation is that the reduction of memory requirements grows with the size of the benchmarks. For the smallest benchmark, our memory reduction is 7.1 times, and it goes up to 11.4 times for the largest benchmark. The trend of memory requirements can also be viewed from Figure 9. The x-axis represents the number of lines of code after flattening, and the y-axis represents the minimum memory requirement. As we can see, EBMC has a much steeper slope than our approach due to its state explosion problem. It is expected that when the size of the benchmarks keep growing, EBMC will give up running much faster than our approach. In other words, our approach is more scalable compared to state-of-the-art model checking tools as well as concolic testing approaches in RTL models.

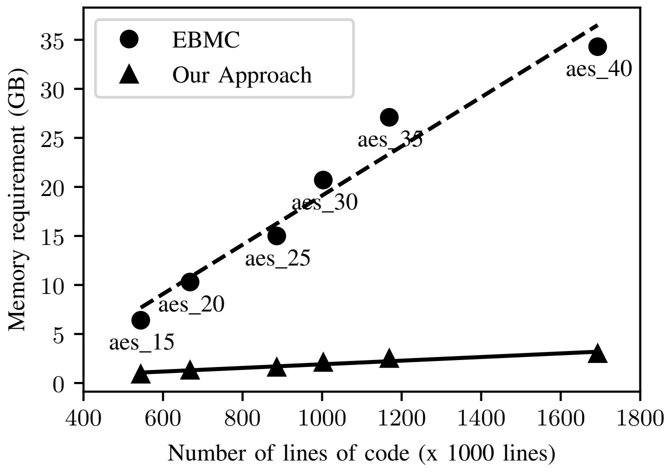


Fig. 9: The comparison of memory requirements of our approach and EBMC with various sizes of benchmarks. The circle and triangle with the same x-axis represents one benchmark in Table IV.

D. Effect of Target Pruning

Due to target pruning, some targets are covered by the explored paths for the other targets. The number of pruned targets are shown in Figure 10. We also compared the number of targets that can be pruned by EBMC. If a target is already activated by a test that is generated by EBMC to cover a previous target, this target is omitted and counted as a pruned target. Therefore, the number of pruned targets by EBMC is partially affected by the order of targets. For a fair comparison, we fed the targets to EBMC in the same order as our approach. As shown in Figure 10, both our approach and EBMC pruned over half of the targets for most of the benchmarks. However, our approach achieves consistently better results compared to EBMC. In particular, for the small benchmarks, such as b10 and ICache, the number of pruned targets are similar. On the other hand, the gap of pruned targets is becoming larger when the benchmarks become larger. There are two primary reasons for the success of our approach. First, EBMC is used as a directed test generation scheme. The generated test can cover the branch targets that reside in the same simulation path of the test. As a result, our approach is highly likely to cover these branch targets as well by our simulation. Second, our approach explores many paths from the initial path to our final path to activate one specific target. These paths may come from different parts of the design, and therefore, it is likely to cover other targets (target pruning) or be close to some future targets (target clustering).

For small benchmarks, the hard-to-activate branches are prone to reside in the same rare area. Therefore, EBMC performed well in small benchmarks. However, the hard-to-activate branches in large benchmarks are scattered in different parts of the design. The directed tests generated by EBMC pruned less targets in this scenario. On the other hand, targets are possible to be covered by some paths when our approach is searching for solutions for previous targets. The number of pruned targets reflects the effectiveness of target pruning. It also demonstrates that the extremely hard-to-activate targets dominate the overall test generation time.

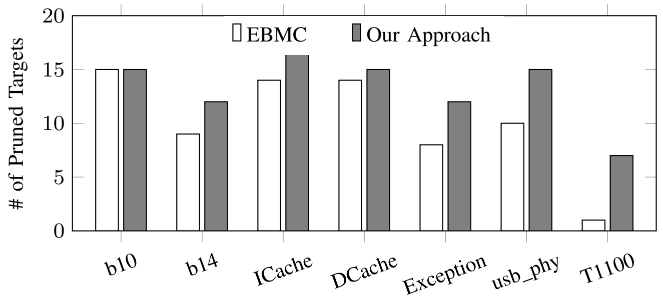


Fig. 10: The number of targets that are pruned.

E. Effect of Edge Realignment

TABLE V: The number of iterations that each block is selected as the best alternative block in exploring paths for Listing 1.

Blocks	BB2	BB3	BB4	BB5	BB6	Cover
[34]	20	19	18	18	5	No
Our approach	0	6	5	0	0	Yes

To demonstrate the contribution of an efficient edge realignment in our framework compared to a naive edge realignment [34], we applied our approach on the example shown in Listing 1 with BB7 as our target. We profiled the number of times each block is chosen as the best alternative block through all iterations in Table V. Assume that the number of selections of BB2, BB3, BB4 and BB5 are x_2, x_3, x_4 and x_5, respectively. The target is activated only when $x_3+x_4-x_2-x_5 > 10$ by statically analyzing the code in Listing 1. The first row shows the selection of [34], where the first four blocks are selected almost randomly, as expected from the realignment results shown in Figure 4(b). With this random selection, the target is not covered. On the other hand, our approach activated the target in 11 iterations with BB3 and BB4 being selected by 6 and 5 times, respectively.

VII. Conclusion

Test generation is an important step during validation and debugging of hardware designs. Conventional validation methodology using random and constrained-random tests can lead to unacceptable functional coverage under tight deadlines. While application of concolic testing on hardware designs has shown some promising results in improving the overall coverage, they are not designed for covering specific targets such as uncovered corner cases and rare functional scenarios. In this paper, we proposed a scalable test generation framework using concolic testing to automatically activate targets in RTL models. This paper made two important contributions. (1) We proposed a directed test generation framework in activating a single target utilizing contribution-aware edge realignment and effective path exploration. (2) We developed two optimization techniques to drastically reduce the overall test generation effort involving multiple targets: (i) target pruning to remove the targets that can be covered by the tests generated for other targets, and (ii) target clustering to minimize the overlapping searches by utilizing learning from previous searches. Experimental results demonstrated that our approach is significantly faster compared to state-of-the-art test generation techniques. Compared to QUEBS, our approach provides significant speedup in test generation time (up to 29140X, 4859X on average). Similarly, compared to EBMC, our approach provides drastic improvement in test generation time (up to 205X, 69X on average) and an order-of-magnitude reduction in memory requirement.

Acknowledgments

This work was partially supported by the National Science Foundation (NSF) grant CCF-1908131. We would like to acknowledge the technical discussions with Mr. Alif Ahmed.

References

[1] Y. Lyu, A. Ahmed, and P. Mishra, "Automated activation of multiple targets in RTL models using concolic testing," in *Design, Automation & Test in Europe Conference & Exhibition, Florence, Italy, March 25-29*, 2019, pp. 354–359.

[2] Y. Lyu and P. Mishra, "Efficient test generation for trojan detection using side channel analysis," in *Design, Automation & Test in Europe Conference & Exhibition, Florence, Italy, March 25-29*, 2019, pp. 408–413.

[3] Y. Lyu, X. Qin, M. Chen, and P. Mishra, "Directed test generation for validation of cache coherence protocols," *IEEE Trans. on CAD of Integrated Circuits and Systems*, vol. 38, no. 1, pp. 163–176, 2019.

[4] X. Qin and P. Mishra, "Directed test generation for validation of multicore architectures," *ACM Trans. Des. Autom. Electron. Syst.*, vol. 17, no. 3, pp. 24:1–24:21, Jul. 2012.

[5] M. Chen, P. Mishra, and D. Kalita, "Automatic rtl test generation from systemc tlm specifications," *ACM Trans. Embed. Comput. Syst.*, vol. 11, no. 2, pp. 38:1–38:25, Jul. 2012. [Online]. Available: http://doi.acm.org/10.1145/2220336.2220350

[6] F. Farahmandi and P. Mishra, "Automated test generation for debugging multiple bugs in arithmetic circuits," *IEEE Transactions on Computers*, vol. 68, no. 2, pp. 182–197, Feb 2019.

[7] M. Chen and P. Mishra, “Functional test generation using efficient property clustering and learning techniques,” *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 29, no. 3, pp. 396–404, March 2010.

[8] A. Gargantini and C. Heitmeyer, “Using model checking to generate tests from requirements specifications,” *SIGSOFT Softw. Eng. Notes*, vol. 24, no. 6, pp. 146–162, Oct. 1999.

[9] M. Chen and P. Mishra, “Property learning techniques for efficient generation of directed tests,” *IEEE Transactions on Computers*, vol. 60, no. 6, pp. 852–864, June 2011.

[10] K. Sen, D. Marinov, and G. Agha, “Cute: a concolic unit testing engine for c,” in *ACM SIGSOFT Software Engineering Notes*, vol. 30, no. 5. ACM, 2005, pp. 263–272.

[11] P. Godefroid, N. Klarlund, and K. Sen, “Dart: directed automated random testing,” in *ACM Sigplan Notices*, vol. 40, no. 6. ACM, 2005, pp. 213–223.

[12] C. Cadar, D. Dunbar, and D. Engler, “Klee: Unassisted and automatic generation of high-coverage tests for complex systems programs,” in *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation*, ser. OSDI’08. Berkeley, CA, USA: USENIX Association, 2008, pp. 209–224. [Online]. Available: http://dl.acm.org/citation.cfm?id=1855741.1855756

[13] B. Chen, K. Cong, Z. Yang, Q. Wang, J. Wang, L. Lei, and F. Xie, “End-to-end concolic testing for hardware/software co-validation,” in *2019 IEEE International Conference on Embedded Software and Systems (ICESS)*, June 2019, pp. 1–8.

[14] B. Chen, C. Havlicek, Z. Yang, K. Cong, R. Kannavara, and F. Xie, “Crete: A versatile binary-level concolic testing framework,” in *Fundamental Approaches to Software Engineering*, A. Russo and A. Schürr, Eds. Cham: Springer International Publishing, 2018, pp. 281–298.

[15] L. Liu and S. Vasudevan, “Star: Generating input vectors for design validation by static analysis of rtl,” in *High Level Design Validation and Test Workshop, 2009. HLDVT 2009. IEEE International*. IEEE, 2009, pp. 32–37.

[16] ——, “Scaling input stimulus generation through hybrid static and dynamic analysis of rtl,” *ACM Transactions on Design Automation of Electronic Systems (TODAES)*, vol. 20, no. 1, p. 4, 2014.

[17] A. Ahmed and P. Mishra, “Quebs: Qualifying event based search in concolic testing for validation of rtl models,” in *2017 IEEE 35th International Conference on Computer Design (ICCD)*. IEEE, 2017, pp. 185–192.

[18] E. Clarke, O. Grumberg and D. Peled, *Model Checking*. Cambridge, MA: MIT Press, 1999.

[19] M. Chen, X. Qin, and P. Mishra, “Efficient decision ordering techniques for sat-based test generation,” in *2010 Design, Automation Test in Europe Conference Exhibition (DATE 2010)*, March 2010, pp. 490–495.

[20] J. Burch, E. Clarke, K. McMillan, D. Dill, and L. Hwang, “Symbolic model checking: 10^{20} states and beyond,” *Information and Computation*, vol. 98, no. 2, pp. 142 – 170, 1992. [Online]. Available: http://www.sciencedirect.com/science/article/pii/089054019290017A

[21] A. Biere, A. Cimatti, E. Clarke, and Y. Zhu, “Symbolic model checking without bdds,” in *Tools and Algorithms for the Construction and Analysis of Systems*, W. R. Cleaveland, Ed. Berlin, Heidelberg: Springer Berlin Heidelberg, 1999, pp. 193–207.

[22] J. Whittemore, J. Kim, and K. Sakallah, “Satire: A new incremental satisfiability engine,” in *Proceedings of the 38th Design Automation Conference (IEEE Cat. No.01CH37232)*, June 2001, pp. 542–545.

[23] O. Strichman, “Accelerating bounded model checking of safety properties,” *Formal Methods in System Design*, vol. 24, no. 1, pp. 5–24, Jan 2004. [Online]. Available: https://doi.org/10.1023/B:FORM.0000004785.67232.f8

[24] M. Chen, X. Qin, H.-M. Koo, and P. Mishra, *System-level validation: high-level modeling and directed test generation techniques*. Springer Science & Business Media, 2012.

[25] J. Burnim and K. Sen, “Heuristics for scalable dynamic test generation,” in *2008 23rd IEEE/ACM International Conference on Automated Software Engineering*, Sep. 2008, pp. 443–446.

[26] C. Zamfir and G. Candea, “Execution synthesis: A technique for automated software debugging,” in *Proceedings of the 5th European Conference on Computer Systems*, ser. EuroSys ’10. New York, NY, USA: ACM, 2010, pp. 321–334. [Online]. Available: http://doi.acm.org/10.1145/1755913.1755946

[27] F. Charreteur and A. Gotlieb, “Constraint-based test input generation for java bytecode,” in *Software Reliability Engineering (ISSRE), 2010 IEEE 21st International Symposium on*. IEEE, 2010, pp. 131–140.

[28] S. Chandra, S. J. Fink, and M. Sridharan, “Snugglebug: a powerful approach to weakest preconditions,” *ACM Sigplan Notices*, vol. 44, no. 6, pp. 363–374, 2009.

[29] P. Dinges and G. Agha, “Targeted test input generation using symbolic-concrete backward execution,” in *Proceedings of the 29th ACM/IEEE international conference on Automated software engineering*. ACM, 2014, pp. 31–36.

[30] K. Cong, F. Xie, and L. Lei, “Automatic concolic test generation with virtual prototypes for post-silicon validation,” in *2013 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, Nov 2013, pp. 303–310.

[31] V. Herdt, D. Große, H. M. Le, and R. Drechsler, “Early concolic testing of embedded binaries with virtual prototypes: A risc-v case study*,” in *2019 56th ACM/IEEE Design Automation Conference (DAC)*, June 2019, pp. 1–6.

[32] B. Lin, K. Cong, Z. Yang, Z. Liao, T. Zhan, C. Havlicek, and F. Xie, “Concolic testing of systemc designs,” in *2018 19th International Symposium on Quality Electronic Design (ISQED)*, March 2018, pp. 1–7.

[33] Y. Lyu and P. Mishra, “Automated test generation for activation of assertions in rtl models,” in *Asia and South Pacific Design Automation Conference (ASPDAC), Beijing, China, January 13 - 16*, 2020.

[34] A. Ahmed, F. Farahmandi, and P. Mishra, “Directed test generation using concolic testing on rtl models,” in *Design, Automation & Test in Europe Conference & Exhibition (DATE), 2018*. IEEE, 2018, pp. 1538–1543.

[35] D. Kroening and M. Purandare, *EBMC: The Enhanced Bounded Model Checker*, http://www.cprover.org/ebmc.

[36] R. Mukherjee, D. Kroening, and T. Melham, “Hardware verification using software analyzers,” in *2015 IEEE Computer Society Annual Symposium on VLSI*, July 2015, pp. 7–12.

[37] S. Williams, “Icarus verilog,” *On-line: http://iverilog.icarus.com/*, 2006.

[38] “Edautils website,” *On-line: http://www.edautils.com*.

[39] B. Dutertre, “Yices 2.2,” in *International Conference on Computer Aided Verification*. Springer, 2014, pp. 737–744.

[40] F. Corno, M. S. Reorda, and G. Squillero, “Rt-level itc’99 benchmarks and first atpg results,” *IEEE Design Test of Computers*, vol. 17, no. 3, pp. 44–53, July 2000.

[41] M. Tehranipoor, D. Forte, R. Karri, F. Koushanfar, and M. Potkonjak, *Trust-HUB*, https://www.trust-hub.org/.

[42] “Opencores website,” http://www.opencores.org, 2018.

Yangdi Lyu received his B.E. degree from Department of Hydraulic Engineering, Tsinghua University, Beijing, China in 2011, and his Ph.D. degree from the Department of Computer and Information Sciences and Engineering, University of Florida in 2020. His research interests include the development of test generation techniques for hardware trust, the security validation of system-on-chip, and microarchitectural side-channel analysis.

Prabhat Mishra (SM’08) is a Professor in the Department of Computer and Information Science and Engineering at the University of Florida. His research interests include embedded and cyber-physical systems, hardware security and trust, energy-aware computing, formal verification, and system-on-chip security validation. He received his Ph.D. in Computer Science and Engineering from the University of California, Irvine. He has published 7 books, 25 book chapters, and more than 150 research articles in premier international journals and conferences. His research has been recognized by several awards including the NSF CAREER Award, IBM Faculty Award, three best paper awards, and EDAA Outstanding Dissertation Award. Prof. Mishra currently serves as an Associate Editor of ACM Transactions on Design Automation of Electronic Systems, and IEEE Transactions on VLSI Systems. He is an ACM Distinguished Scientist and a Senior Member of IEEE.