Automated Activation of Multiple Targets in RTL Models using Concolic Testing

Yangdi Lyu, Alif Ahmed and Prabhat Mishra
Department of Computer and Information Science and Engineering
University of Florida, Gainesville, Florida, USA

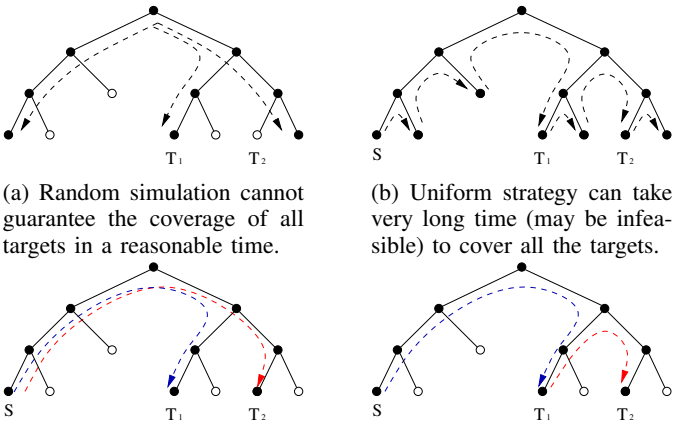
***Abstract*—Simulation is widely used for validation of Register-Transfer-Level (RTL) models. While simulating with millions of random (or constrained-random) tests can cover majority of the targets (functional scenarios), the number of remaining targets can still be huge (hundreds or thousands) in case of today's industrial designs. Prior work on directed test generation using concolic testing can cover only one target at a time. A naive extension of prior work to activate the remaining targets would be infeasible due to wasted effort in multiple overlapping searches. In this paper, we propose an automated test generation technique for activating multiple targets in RTL models using concolic testing. This paper makes three important contributions. First, it efficiently prunes the targets that can be covered by the tests generated for activating the other targets. Next, it minimizes the overlapping searches while trying to generate tests for activating multiple targets. Finally, our approach effectively utilizes clustering of related targets as well as common path sharing between the targets in the same cluster to drastically reduce the test generation time. Experimental results demonstrate that our approach significantly outperforms the existing methods in terms of overall coverage (up to 5X, 1.2X on average) as well as test generation time (up to 146X, 80X on average).**

I. Introduction

During design validation and verification, a common practice in industry is to run millions of random or constrained-random tests to quickly cover majority of functional scenarios (targets). However, it is not always possible to cover all scenarios using these tests (see Figure 1(a)). Verification engineers usually manually write test cases to cover the remaining hard-to-activate scenarios. While such manual test case development is possible for small designs, it would be infeasible to develop directed test for industrial designs. Coming up with manual test cases can be both error-prone and time-consuming due to many trial-and-error iterations. Automated test generation is necessary to overcome these issues. There are significant prior efforts in automated generation of directed tests [1]–[6]. Test generation using formal methods [7], [8] can cover specific scenarios directly, but suffer from state explosion for large designs. Semi-formal approaches, such as concolic testing [9], [10], combine the advantages of random simulation and formal methods to activate targets efficiently. Concolic testing starts from a random simulation, and continuously explores different *alternate branches* to cover a target.

Recent test generation approaches using concolic testing in RTL models can be broadly classified into two categories. The first one is *uniform* test generation [11], where the goal is to maximize overall branch/statement coverage. The second one is directed test generation [12] that can generate a test to activate a single target at a time. Unfortunately, none of these two approaches are suitable for scenarios when thousands of targets (not covered by millions of random simulation) need to be activated. Uniform test generation tends to achieve the goal of maximizing overall coverage but ignores the priority of intended targets, leading to longer test generation time, as shown in Figure 1(b). Directed test generation can be extended to cover multiple targets by activating each target iteratively. We call it *single-target method*. However, single-target method wastes a lot of effort in overlapping search, as shown in Figure 1(c). To reduce the number of overlapping search, we propose an efficient technique for multi-target test generation using concolic testing. Our approach utilizes information from the previous search, as shown in Figure 1(d). In this paper, we make four major contributions:

- We propose efficient pruning of targets that can be covered by the tests generated for activating other targets.
- We propose clustering of related targets to drastically reduce the test generation time. Targets in the same cluster usually share a common simulated path. Therefore, our approach minimizes the overlapping search while trying to generate tests for activating multiple targets.



(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 can lead to wasted effort (overlapping search) to cover targets.

(d) Our approach utilizes the previous search and starts from a profitable path.

Fig. 1: Comparison of random simulation, uniform test generation, single-target method and our approach in covering two targets T_1 and T_2. Path S is the initial path for concolic testing.

This work was partially supported by Cisco Systems.

- We propose a novel edge realignment technique to effectively evaluate the distance between a simulated path and a target. The realigned edges are used to guide alternative branch selection to improve both coverage and efficiency.
- Experimental results demonstrate the effectiveness of the proposed approach compared to the state-of-the-art uniform and single-target methods.

The paper is organized as follows. Section II introduces existing test generation techniques using concolic testing. Section III describes our test generation framework. Section IV presents experimental results. Section V concludes the paper.

II. Background and Related Work

The four major steps of concolic testing are shown in the bottom part of Figure 2 (simulate, generate constraints, select alternate branch and solve constraints). The first step is to simulate the design using a random input vector. The next step is to generate constraints from the simulated path. Then, one alternate branch is selected to create new constraints. If the new constraints are satisfiable, an input vector will be returned and used to generate a new simulated path. When the simulated path covers a target, the input vector is added to the test set. As concolic testing examines one path and one alternate branch at a time, it avoids state explosion problem in test generation using model checking. The performance of concolic testing is dependent on two main steps in Figure 2. The most important one is the ***alternate branch selection***. 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. The other one is the ***initial path***, which is usually a random simulation. When the initial path is closer to the target, it is much easier to reach the target. The existing test generation techniques based on concolic testing can be broadly classified into the following two categories.

A. Uniform Test Generation Techniques

Uniform test generation techniques utilize depth-first search (DFS), breadth-first search (BFS), etc [9]–[15] to cover as many branches as possible. Figure 1(b) shows the DFS based searching of alternate branches. Recently, Ahmed *et al.* [11] proposed QUEBS to balance exhaustive and restrictive search techniques by limiting the number of times a branch can be selected. While uniform test generation can cover all branches given enough test generation time, it suffers from the exponentially growing paths which makes exhaustive searching impractical for covering a number of selected targets.

B. Directed Test Generation Techniques

Directed test generation is efficient in generating test for a specific target using symmetric backward execution [16]–[18] in software domain. In hardware domain, Ahmed et al. [12] utilized concolic testing to cover a single target in RTL by utilizing static analysis of distance metric in control flow graphs (CFGs) to guide searching alternate branches. However, a naive extension of this approach to cover multiple targets can be infeasible due to wasted effort in overlapping searches.

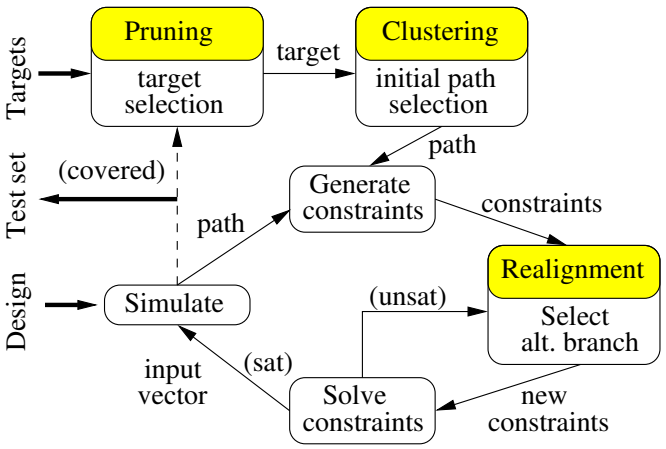


Fig. 2: The overview of our test generation framework using concolic testing. Our three major contributions (target clustering, edge realignment and target pruning) are highlighted.

III. Multi-target Test Generation

A. Overview

Given an RTL description of a hardware design and intended branch targets, our goal is to efficiently generate a set of compact tests to cover all the targets. The overview of our framework is shown in Figure 2. Our three major contributions are highlighted in the figure. While target pruning reduces the number of targets without sacrificing the coverage, edge realignment and target clustering improve the two core functionalities of concolic testing, initial path selection and selection of alternate branches. The key contributions made by these three techniques are summarized below:

1) Target pruning is designed to reduce the number of targets. 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 test generation.
2) Proposed target clustering connects each target with the closest simulated path. One problem of single-target method is that it ignores all precious search information while activating previous targets, leading to wasted effort in overlapping searches. Our approach 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.
3) Our edge realignment efficiently guides alternate branch selection. The goal of alternate branch selection in concolic testing is to find a path closer (least distance) to the target. However, the definition of "distance" is different from the distance in the original CFG. Through edge realignment, the blocks that make contribution to activate the target are realigned closer to it. By selecting the blocks containing these assignments with the most contribution, we increase the probability of activating the target. In addition, edge realignment also helps target clustering in identifying the closest simulated path.

The remainder of this section describes these three contributions in detail.

B. 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 1 as an example. Figure 3 shows its CFG with T0, T1 and T2 to represent 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, $TS' = TS - \cup_{T \in TS} DM(T)$.

Listing 1: Example RTL Design

```
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 naive 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 to achieve both efficiency and coverage, we utilize topological sorting. The targets that we want to prune are moved to the end of the target queue. This way, test generation for these targets will only be done if they are not covered by previously generated tests. 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 utilizes the order of targets. When the generated paths of a target can cover the other targets, the latter 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 at the end of the target queue, hoping that it may be covered while generating tests for other targets.

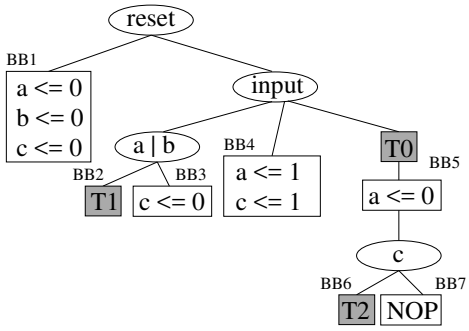
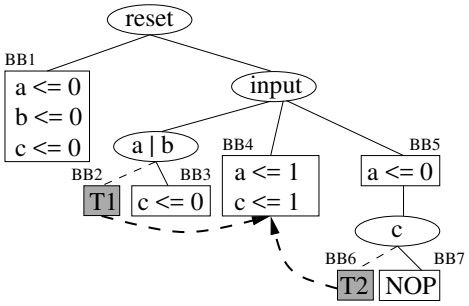


Fig. 3: CFG for Listing 1. T0, T1, and T2 represent three targets. Rectangles represent basic blocks (BB), i.e., there are no jumps in or out of the blocks. T0 will be eliminated during the static target pruning.

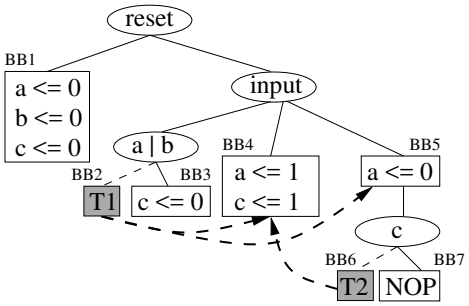
C. Edge Realignment

The main goal of edge realignment is to accurately evaluate the distance between a path and a target. For any target with condition l_g, we want to define the paths with assignments that contribute to the activation of l_g as "close" to the target. After finding such assignments, we want to align the target with blocks which contain these assignments. As shown in Figure 4(a), we want T1 and T2 to directly connect to BB4. Notice that this edge realignment is different from [12] in which the target is connected to all satisfiable assignments of its variables. The results of [12] is shown in Figure 4(b), with an extra edge connecting T1 with BB5. Although $(a|b) \wedge (a = 0)$ is satisfiable given $b = 1$, this realigned edge is inappropriate. When $b = 0$, this updated edge will lead the search path to be far away from T1. Even when $b = 1$, the assignment $a <= 0$ apparently has no contribution to activating this target. While manually checking the contribution is possible for small designs, it is infeasible when the design is large and the condition is complex. So, we propose a contribution-aware edge realignment in Algorithm 1.

In this algorithm, we put all the blocks that need to be realigned into block queue BQ. Initially, BQ contains all the targets. Whenever we realign the edge of the current block bb to point to another block bb_a, bb_a needs to be put in BQ if not realigned yet. All blocks need to be realigned at most once. For the current block bb, we first expand the condition to get all the related variables. Then we check all blocks that assign values to these variables. For each assignment l_a, we evaluate the satisfiability of $!(l_g)_0 \wedge l_a \wedge (l_g)_1$. This condition represents forcing l_g to be false in the beginning, followed by the assignment l_a, and then checking l_g, where the subscripts represent different versions of one variable. If it is satisfiable, we add the block to the predecessors of bb. For example, we will feed $(!(a_0|b_0) \wedge (a_1 = 0) \wedge (a_1|b_0))$ to check the contribution of $a <= 0$. By forcing $a|b$ to be false, both a and b should be 0 in the beginning. The assignment $a <= 0$ does not improve this situation, so it is not considered as a profitable assignment. On the other hand, the assignment $a <= 1$ will



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



(b) [12] realigns each block to the assignments that are satisfiable.

Fig. 4: Comparison of edge realignment by our approach and [12]. As edge realignment is critical in guiding alternate branch selection, one incorrectly realigned edge can lead to significant performance degradation.

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 ← BQ.pop()
        // Update edge for block bb
4:      l_g ← expanded guard condition of bb
5:      for all variables v ∈ l_g do
6:          for all assignments l_a to v do
7:              if satisfiable(!(l_g)_0 ∧ l_a ∧ (l_g)_1) then
8:                  bb_a = the block of l_a
9:                  Add bb_a to bb.predecessors
10:                 BQ.push(bb_a) if bb_a is not visited
11:             end if
12:         end for
13:     end for
14: end while
```

make $a|b$ satisfiable. So we will realign the edge of T1 to connect to BB4, but not to BB5.

To see how edge realignment guides alternate branch selection, we use the example in Figure 5 to activate T1 in Listing 1. Assume that the design is unrolled for 3 cycles and $input$ is $2'b00$ for all clock cycles. Then, the initial path is the blue one P1. As T1 is directly connected to BB4 which is reachable in the second cycle of P1, the alternate branch (red solid line) is selected and an input vector is returned by the constraint solver. Assume P2 is the simulated path of the returned input vector. It is clear to see that T1 is covered by P2.

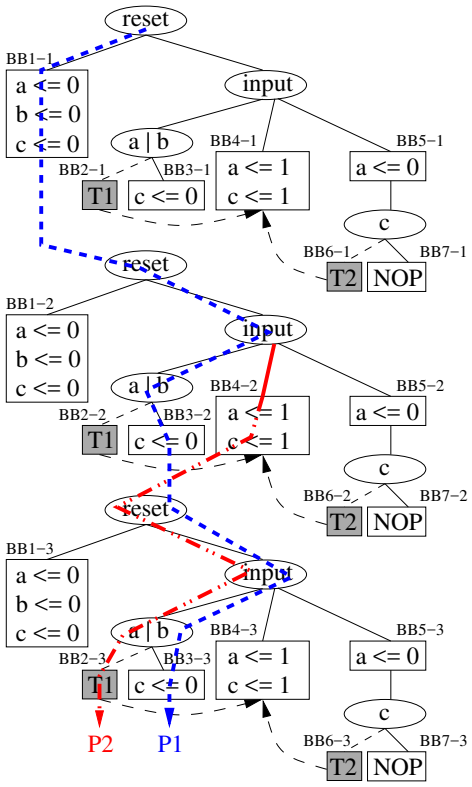


Fig. 5: The design in Listing 1 is unrolled for three cycles. The initial path is P1 with $input$ being $2'b00$ for all cycles. The selected alternate branch is shown in solid red line. The simulated path is P2 for the return input vector from constraint solver. P2 covers T1 and becomes the closest simulated path during target clustering.

From Figure 5, we notice that the distance in the original CFG is misleading. In the second clock, while P1 (passing BB3) is closer to T1 in the original CFG, P2 is actually "closer" considering the possibility of activating T1 by BB4. On the other hand, an incorrectly realigned edge is dangerous. As it receives the highest priority to be selected during concolic testing, even one extra edge may lead the direction of search to a totally different area, resulting in long test generation time or failure to activate the target.

D. 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 alternate branch is selected,

a better initial path means fewer concolic iterations which reduces the probability of getting lost in a large number of misleading alternate branches.

Since current SoCs 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 during concolic testing 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. As shown in Figure 5, 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 split the cluster. As P2 visited BB4, 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 of single-target method.

IV. Experiments

A. Experimental Setup

To evaluate the effectiveness and efficiency of our approach, we compared the performance with uniform test generation technique (QUEBS) [11] and single-target method [12]. The experiments are conducted in a server machine with Intel Xeon CPU E5-2698 @2.20GHz. Our approach utilizes the Icarus Verilog Target API [19] for parsing, instrumentation, 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* [20]. Yices SMT solver is used for constraint solving [21].

Benchmarks are selected from ITC99 [22], TrustHub [23], and OpenCores [24]. These benchmarks contain hard-to-cover branches (targets), providing a reasonable test generation complexity. For target selection, we first ran the benchmarks for one million cycles with random tests. Then, we selected 20 branches that were covered the least number of times as our targets. For each Trojan-inserted benchmark form TrustHub (AES-T1100, AES-T2000), the 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, which exceeds our unrolled cycles.

B. Results

In this section, we show the efficiency and coverage improvement over the state-of-the-art test generation techniques. Since the goal of QUEBS [11] is to cover all branches, we terminated it once it covered all of our selected targets. We set a new target for single-target method each time with the iteration limit to be 1000, and report the accumulated performance. For the round-robin scheduling of selecting targets in our approach, we set the iteration limit to be 10 in each round.

The experimental results are shown in Table I. The first column represents the benchmarks. For each approach, we compare the number of covered targets, the number of iterations in concolic testing, and the total time used for test generation. The last four columns show the improvement of coverage and time (average time per target) over QUEBS [11] and single-target method [12].

From Table I, we can see that our approach can cover all 20 targets in all benchmarks efficiently. While QUEBS and single-target method perform well in small benchmarks such as ITC99, they achieve poor coverage in benchmarks such as PCI and usb_phy. For the selected ITC99 benchmarks, majority of the selected targets can be covered by QUEBS and single-target method, except for one target from b10 that is not covered by QUEBS, and two targets from b14 that are not covered by single-target method. When the design becomes more complex, QUEBS and single-target method cannot cover all targets, and the worst performance is achieved in PCI with only 1 and 4 targets covered, respectively. For two Trojan-inserted AES designs, QUEBS cannot finish within the time limit. Even without considering the AES designs for QUEBS, our approach achieves up to 20X (1.27X on average) coverage improvement over QUEBS, and up to 5X (1.20X on average) coverage improvement over single-target method.

In terms of test generation time, our approach is comparable with QUEBS in some benchmarks and is always faster than single-target method. For or1200_ICache, our approach covered the targets extremely fast while it took QUEBS almost 43 minutes to finish. The test generation time of single-target method is very large due to the long iteration (up to 1000 each) for uncovered targets. In some benchmarks, although our approach seems slower, e.g. or1200_Exception and usb_phy, the conclusion should be the opposite considering the fact that the test generation time is usually dominated by extremely hard-to-activate targets. Even without considering the AES designs for QUEBS that it could not finish, our approach achieves up to 495X, 13X on average, time improvement over QUEBS, and up to 146X, 80X on average, time improvement over single-target method.

C. Effect of Target Pruning

Due to target pruning, some targets are activated during searching paths for other targets. The number of pruned targets are shown in Figure 6. For most of the benchmarks, over half of the targets are pruned while searching for the other targets. It demonstrates the effectiveness of target pruning. It also reflects that the extremely hard-to-activate targets dominate the total test generation time. For example, 16 targets are pruned in or1200_ICache. Therefore, the remaining 4 targets share the total 181 concolic iterations.

TABLE I: The experimental results of applying QUEBS [11], single-target method [12] and our approach on 20 targets from benchmarks. The time improvement represents the test generation improvement per target. The average for QUEBS excludes the results for AES-T1100 and AES-T2000 since it could not finish for these benchmarks.

bench	QUEBS [11]			single-target method [12]			Our approach			Impro. over [11]		Impro. over [12]	
	cover	iter	time	cover	iter	time	cover	iter	time	cover	time	cover	time
b10	19	92	0.06s	20	21	0.17s	20	10	0.03s	1.05x	2x	1x	5.7x
b14	20	3332	11.64s	18	2701	20.28s	20	31	0.29s	1x	40x	1.1x	77.7x
or1200_ICache	20	101213	2575s	19	1739	50.49s	20	181	5.20s	1x	495x	1.05x	10.2x
or1200_DCache	20	2907	41.19s	13	3895	22090s	20	87	232.3s	1x	0.18x	1.5x	146x
or1200_Exception	19	932	5.43s	18	2034	8.67s	20	141	6.22s	1.05x	0.92x	1.1x	1.5x
PCI	1	134	0.63s	4	624	7.86s	20	614	10.02s	20x	1.26x	5x	3.9x
usb_phy	12	339	5.39s	17	781	241.26s	20	81	9.33s	1.7x	0.96x	1.2x	30x
AES-T1100	-	-	-	20	29	71.84s	20	19	40.92s	-	-	1x	1.8x
AES-T2000	-	-	-	20	27	69.59s	20	20	35.29s	-	-	1x	2x
Average	15.8	15564	377s	16.6	1317	2507s	20	132	37.7s	1.27x	13x	1.20x	80x

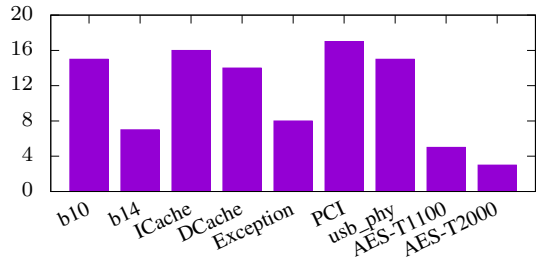


Fig. 6: The number of targets that are pruned. Benchmarks from or1200 omit the prefix "or1200" for simplicity.

D. Effect of Edge Realignment

Notice that QUEBS and single-target method perform poorly in covering the targets from PCI, but the running time is fast (within 1000 iterations). The problem is that QUEBS is not using edge realignment, and the edge realignment in single-target method has the exact problem pointed out in Section III-C. For example, one of the selected targets in PCI is controlled by $if(\sim irdy||trdy)$ where $trdy$ is set to 1 only when $enable = 2'b11$. When we inspected the CFG after edge realignment of [12], we found out that the target is connected to all assignments of $enable$ with satisfiable checking. Among the 13 realigned edges, only 4 of them are connected to $enable <= 2'b11$. In other words, less than one third of realigned edges are correct. During alternate branch selection, as these realigned edges have the highest priority, concolic testing is always trying to activate these branches. After trying to set $enable$ to wrong values several times, the path deviates from the target in state-of-the-art approaches, while our approach quickly reaches the target due to effective edge realignment.

V. Conclusion

In this paper, we proposed a test generation framework using concolic testing to automatically activate multiple targets in RTL models, which outperforms the state-of-the-art test generation techniques. Our framework made three important contributions: target pruning, contribution-aware edge realignment, and target clustering. Our contribution-aware edge realignment shows significant improvement over the existing edge realignment technique when applying to large benchmarks. Our dynamic target clustering learns the most profitable initial path from previous search. Starting from the most profitable path eliminates a lot of overlapping search compared to existing methods. Experimental results demonstrated that our approach can cover all the hard-to-detect targets, while existing approaches fail in many cases. Moreover, our approach is significantly faster (up to 146X, 80X on average) compared to state-of-the-art test generation techniques.

References

[1] Y. Lyu and P. Mishra, "Efficient test generation for Trojan detection using side channel analysis," in *DATE*, 2019.
[2] Y. Lyu et al., "Directed test generation for validation of cache coherence protocols," in *TCAD*, 2018.
[3] X. Qin and P. Mishra, "Directed test generation for validation of multicore architectures," in *TODAES*, 2012.
[4] M. Chen et al., "Automatic RTL test generation from SystemC TLM specifications," in *TECS*, 2012.
[5] F. Farahmandi and P. Mishra, "Automated test generation for debugging multiple bugs in arithmetic circuits," in *TC*, 2018.
[6] M. Chen and P. Mishra, "Functional test generation using efficient property clustering and learning techniques," in *TCAD*, 2010.
[7] A. Gargantini and C. Heitmeyer, "Using model checking to generate tests from requirements specifications," *SIGSOFT Softw. Eng. Notes*, 1999.
[8] M. Chen and P. Mishra, "Property learning techniques for efficient generation of directed tests," in *TC*, 2011.
[9] K. Sen, D. Marinov, and G. Agha, "CUTE: a concolic unit testing engine for C," in *ACM SIGSOFT Software Engineering Notes*, 2005.
[10] P. Godefroid, N. Klarlund, and K. Sen, "DART: Directed automated random testing," in *ACM Sigplan Notices*, 2005.
[11] A. Ahmed and P. Mishra, "QUEBS: Qualifying event based search in concolic testing for validation of RTL models," in *ICCD*, 2017.
[12] A. Ahmed, F. Farahmandi, and P. Mishra, "Directed test generation using concolic testing on RTL models," in *DATE*, 2018.
[13] L. Liu and S. Vasudevan, "Scaling input stimulus generation through hybrid static and dynamic analysis of RTL," in *TODAES*, 2014.
[14] Y. Li, Z. Su, L. Wang, and X. Li, "Steering symbolic execution to less traveled paths," in *ACM SigPlan Notices*, 2013.
[15] H. Seo and S. Kim, "How we get there: A context-guided search strategy in concolic testing," in *ISFSE*, 2014.
[16] F. Charreteur and A. Gotlieb, "Constraint-based test input generation for Java bytecode," in *ISSRE*, 2010.
[17] S. Chandra, S. J. Fink, and M. Sridharan, "Snugglebug: a powerful approach to weakest preconditions," in *ACM Sigplan Notices*, 2009.
[18] P. Dinges and G. Agha, "Targeted test input generation using symbolic-concrete backward execution," in *ASE*, 2014.
[19] S. Williams, "Icarus verilog," http://iverilog.icarus.com/, 2006.
[20] "EDAUtils website", http://www.edautils.com.
[21] B. Dutertre, "Yices 2.2," in *CAV*, 2014.
[22] F. Corno et al., "RT-Level ITC'99 Benchmarks and First ATPG Results," in *IEEE Design & Test of Computers*, 2000.
[23] M. Tehranipoor et al., *Trust-HUB*, https://www.trust-hub.org/.
[24] "Opencores website," http://www.opencores.org, 2018.