Directed Test Generation using Concolic Testing on RTL models

Alif Ahmed, Farimah Farahmandi and Prabhat Mishra
Department of Computer & Information Science & Engineering
University of Florida, USA

***Abstract*—Functional validation is one of the most time consuming steps in System-on-Chip (SoC) design methodology. In today's industrial practice, simulating designs using billions of random or constrained-random tests can lead to high functional coverage. However, it is hard to cover the remaining small fraction of corner cases and rare functional scenarios. While formal methods are promising in such cases, it is infeasible to apply them on large designs. In this paper, we propose a fully automated and scalable approach for generating directed tests using concolic testing of RTL models. While application of concolic testing on hardware designs has shown some promising results, existing approaches are tuned for improving overall coverage, rather than covering a specific target. We developed a Control Flow Graph (CFG) assisted directed test generation method that can efficiently generate a test to activate a given target. Our experimental results demonstrate that our approach is both efficient and scalable compared to the state-of-the-art test generation methods.**

I. Introduction

In this paper, we propose an approach to address *line reachability* problem for hardware designs. That is, given a target line or branch statement in a RTL model, can we find a test vector to cover that particular target? This type of directed test generation is important in many scenarios, like for debugging purposes, or for covering hard to reach corner cases. Goal of *directed* test generation is different from *uniform* test generation in a sense that directed test only cares about covering the specific target, rather than maximizing overall coverage. While uniform test will eventually reach the required target statement, it may take very long time.

Many formal and semi-formal directed test generation methods have been proposed over the years. However, formal methods often do not scale well with the design complexity or sequential depth [1]. For example, bounded model checking statically unrolls the design and considers all possible execution paths. Such unrolling mechanism leads to state space explosion, limiting its applicability to only small designs. In recent years, a semi-formal method named concolic testing is gaining momentum because it does not suffer from state explosion problem. This is due to the fact that the concolic testing considers only one program execution path at a time.

Concolic testing has been successfully applied on both hardware and software designs [2]–[4]. In hardware domain however, existing concolic testing methods offer uniform tests, and are not tuned for directed test generation. In software domain, structural information provided by CFG have been used to efficiently guide concolic testing towards coverage target [5], [6]. However, effectiveness of such approach on hardware designs (which is highly concurrent) is an unexplored territory. While test generation in software domain deals with only one CFG, developing a test generation method for hardware designs needs to handle multiple concurrent CFGs. Moreover, it has to deal with the complexity of unrolling CFGs across multiple clock cycles. In this paper, we propose an efficient and scalable test generation approach which addresses these challenges.

This work was partially supported by grants from NSF (CNS-1441667), SRC (2014-TS-2554) and Cisco.

Our proposed approach statically calculates distance from target statement by doing a precondition analysis on the CFG. This distance metric is used by concolic testing to efficiently guide the search, eventually converging to the target. To the best of our knowledge, this is the first attempt to apply concolic testing on hardware designs for directed test generation. Experimental results demonstrate that our approach scales well with design complexity compared to formal methods. Additionally, compared to other state-of-art test generation methods, it converges to target with significantly less number of iterations.

Our primary contributions in this paper are as follows:

- We propose a technique to statically measure target distance in a multi-process multi-cycle environment such as RTL models. This distance calculation works without unrolling the state space.
- We propose a concolic testing based scalable directed test generation approach for RTL models. This method utilizes distance feedback to efficiently guide the search towards the target.
- Provided experimental evidence supporting effectiveness and scalability of our test generation scheme.

The remainder of the paper is organized as follows. Section II describes our directed test generation methodology in details. Section III discusses some critical challenges and optimizations. Experimental results are analyzed in Section IV. Prior works on test generation techniques are discussed in Section V. Finally, we conclude our paper in Section VI.

II. Automated Generation of Directed Tests

Figure 1 presents the overview of our proposed method. Given a RTL design, our goal is to generate a test vector to cover a $target$ statement. Following sections describe four major steps of out proposed approach - CFG generation, edge realignment, priority evaluation and concolic testing.

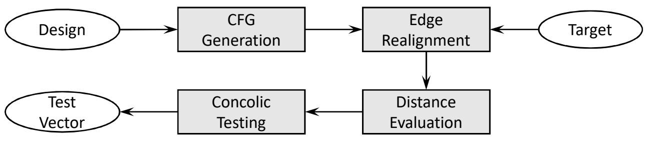


Fig. 1. Overview of the proposed test generation framework.

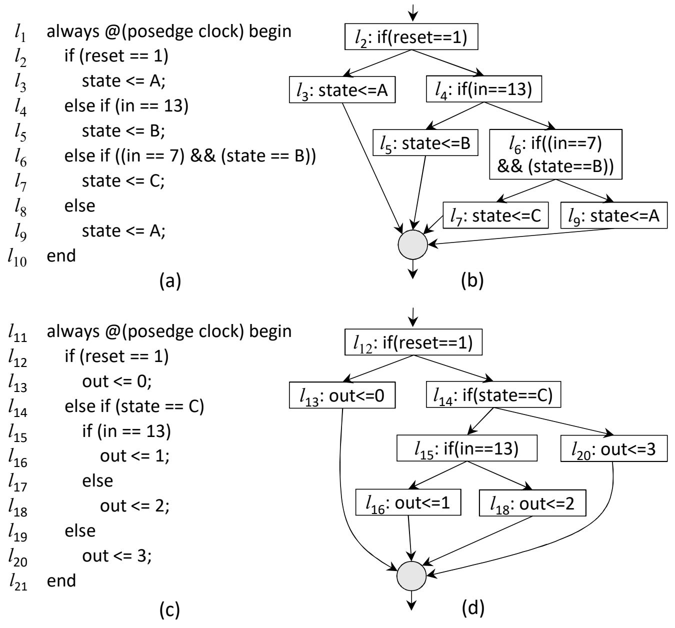
A. CFG Generation

Hardware description languages (HDL) use concurrent model of computation. The examples of concurrent statements include *process* statements in VHDL and *always* statement in Verilog. For ease of presentation, we use the term **'process'** to refer to these concurrent statements in RTL models. Rest of the section describes CFG generation and unrolling procedure for processes.

CFG generation for each process: Within a single process, execution is sequential. So the CFG of a single process can be derived easily by following the procedure established for sequential programs. Running such an algorithm will divide the program into many basic blocks. A basic block is a sequence of statements without any branching. Furthermore, each basic block have a set of successor and predecessor blocks. The blocks to which control may transfer after reaching the end are referred as successor blocks, and the blocks from which control may come are referred as predecessors.

An example Verilog code with two processes and their corresponding CFGs is shown in Figure 2. The code is from a simple sequence detector, which changes state depending on the given input sequence. In the figure, each of the rectangle represents a basic block. From now on, bb_x will be used to denote the basic block containing line l_x. For example, predecessor of bb_6 is bb_4 and successors are bb_7 and bb_9. The circle represents exit basic block.

Mulit-cycle process unrolling: Unrolling for multiple cycles



```
l1   always @(posedge clock) begin
l2     if (reset == 1)
l3       state <= A;
l4     else if (in == 13)
l5       state <= B;
l6     else if ((in == 7) && (state == B))
l7       state <= C;
l8     else
l9       state <= A;
l10  end
```
(a)

```
l11  always @(posedge clock) begin
l12    if (reset == 1)
l13      out <= 0;
l14    else if (state == C)
l15      if (in == 13)
l16        out <= 1;
l17      else
l18        out <= 2;
l19    else
l20      out <= 3;
l21  end
```
(c)

Fig. 2. An example Verilog code with two processes. Here in is input. (a) Process 1, (b) CFG of Process 1, (c) Process 2, (d) CFG of process 2. In this figure, left path is executed if the condition is true.

can be seen as padding multiple single cycle CFGs in sequence - one for each clock. However, the unrolled CFG will consist of only one structure repeated several times. Same effect can be achieved by simply connecting entry block and exit block together. This involves making the entry block successor of exit block, and by making exit block predecessor of entry block.

B. Edge Realignment

Upto this point, each process has its own CFG, and they are not connected to each other. This poses an issue as inter-process dependency information is not available. For example, assume l_{15} is the target statement in Figure 2. To reach l_{15}, $state$ must be equal to C. However, $state$ variable is manipulated in Process 1, while target l_{15} resides in Process 2. If distance values are assigned using current independent CFG structures, dependency of target's guard condition on other processes will not be correctly reflected. To overcome this issue, we have used an edge realignment procedure, which modifies the edges of the CFG to more accurately represent program flow across multiple processes and multiple cycles. First, we define the terms used to describe the algorithm.

Dominator: In a graph, if all paths leading to node n goes through node d, then node d is called a dominator of node n. For example, in the CFG of Figure 2, l_2 and l_4 are dominators of l_5.

Immediate dominator: Among all the dominators D of a node n, immediate dominator is the node which dominates only n and not any other elements of D. In other words, it is the dominator closest to n. For example, immediate dominator of l_5 is l_4.

Strict variables: In this paper, strict variables are defined as the variables to which only concrete values are assigned. In the example, out is a strict variable, because all assignments to out, which are l_{13}, l_{15}, l_{17} and l_{19} - are concrete values. Similarly, $state$ is also a strict variable. On the other hand, in and $reset$ are not strict variables.

The idea of edge realignment is to connect basic blocks to the assignments that satisfies its guard condition, instead of its current predecessors. This is necessary due to the fact that for satisfying guard condition, execution must go through at least one of these assignments.

Algorithm 1 shows this procedure. Most of its work is done within recursive $update\_edge(bb_t)$ function, which realigns the edges of bb_t block. Initially, it is called with user defined target (line 5 of main procedure). In $update\_edge()$ function, guard condition (l_g) of the bb_t is expanded first (line 3). Necessity of expansion will be described in Section III. At this point, two scenarios can occur - the expanded condition contains strict variables, or it does not. Line 5-15 shows the first scenario. In this case, current edge of bb_t is removed first (line 6). Next, all the assignments (l_a) to these strict variables are checked if they are a valid precondition of l_g. This can be done by simply giving ($l_g \wedge l_a$) to a constraint solver (line 9). If satisfiable, then the block containing l_a becomes the new target, and its edges are realigned recursively by calling

Algorithm 1 Edge Realignment

```
Input: CFG, Target Statement $l_t$
Output: Realigned CFG
 1: for all basic block, $bb \in CFG$ do
 2:     $bb.visited \leftarrow false$ // initialization
 3: end for
 4: Target basic block, $bb_t \leftarrow basic\_block(l_t)$
 5: $update\_edge(bb_t)$
 6: return

$update\_edge(bb_t)$
 1: if $bb_t$ is valid and $bb_t.visited$ is $false$ then
 2:     $bb_t.visited = true$
 3:     $l_g \leftarrow$ expanded guard condition of $bb_t$
 4:     $V_s \leftarrow$ set of strict variables in $l_g$
 5:     if $V_s$ is not empty then
 6:         $bb_t.predecessors \leftarrow \emptyset$
 7:         for all variables $v \in V_s$ do
 8:             for all assignments $l_a$ to $v$ do
 9:                 if $satisfiable(l_g \wedge l_a)$ then
10:                     $bb_a \leftarrow basic\_block(l_a)$
11:                     add $bb_a$ to $bb_t.predecessors$
12:                     $update\_edge(bb_a)$
13:                 end if
14:             end for
15:         end for
16:     else if $bb_i \leftarrow idom(bb_t)$ then
17:         $update\_edge(bb_i)$
18:     end if
19: end if
```

$update\_edge(bb_a)$. On the other hand, if the expanded guard condition does not contain any strict variable, then we go to its immediate dominator and realigns its edges. A *visited* flag is maintained to prevent multiple assessment of the same block. Note that successor information is not updated in this algorithm, because they will not be needed in the future.

Figure 3 illustrates modified CFG after edge realignment is carried out on the example of Figure 2. Here, our initial target is l_{16}. According to Algorithm 1, initially $update\_edge(bb_{16})$ will be called. The guard condition of l_{16} is $l_{15} : in = 13$. As this does not contain any strict variable, immediate dominator of l_{16} (which is l_{15}) becomes the new target. Guard condition of l_{15} is $l_{14} : state = C$. This guard condition contains one strict variable - *state*, which is assigned in l_3, l_5, l_7 and l_9. However, if we give $(l_{14} \wedge l_3)$, $(l_{14} \wedge l_5)$, $(l_{14} \wedge l_7)$ and $(l_{14} \wedge l_9)$ to the constraint solver, only $(l_{14} \wedge l_7)$ will be satisfiable. Therefore, the basic block containing l_7 becomes the predecessor of bb_{15}. Also, l_7 becomes the new target. This procedure continues until there are no new basic blocks to align.

C. Distance Evaluation

Once edge realignment is completed, every basic block is assigned a distance value. Lower distance means it is closer to target, or its precondition. Initially, distance of basic block containing target statement is assigned 0, and all the other blocks are set to infinity. Distance of these blocks are updated by carrying out breath first search starting from the target node. The search propagated in the opposite direction of program flow (target to root node) using the predecessor edges of basic blocks. Algorithm 2 realizes this procedure using a queue.

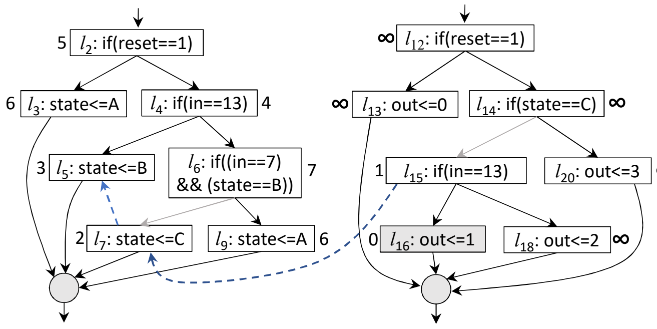


Fig. 3. CFG of example in Figure 2 after edge realignment and weight assignment. Here target is l_{16}. Omitted edges are shown using light solid arrow. New edges are depicted as blue dashed arrow.

Algorithm 2 Distance Evaluation

```
Input: CFG, Target Statement $l_t$
Output: CFG with distance values
 1: for all basic block, $bb \in CFG$ do
 2:     $bb.distance \leftarrow \infty$
 3: end for
 4: $bb_t.distance \leftarrow 0$
 5: Queue for distance evaluation, $Q \leftarrow \{bb_t\}$
 6: while $Q$ not empty do
 7:     $bb_t \leftarrow Q.pop()$
 8:     for all $bb_p \in bb_t.predecessors$ do
 9:         if $bb_p.distance > (bb_t.distance + 1)$ then
10:             $bb_p.distance \leftarrow (bb_t.distance + 1)$
11:             $Q.push(bb_p)$
12:         end if
13:     end for
14: end while
15: return
```

Figure 3 shows CFG after evaluating priority values for the target statement l_{16}. These values are self explanatory except for l_3, l_6 and l_9. These values are due to the fact that exit block is the predecessor of entry block to accommodate for multi-cycle unrolling.

D. Test Generation using Concolic Testing

In concolic testing, the design is initially executed with a random input vector. Then in each iteration of concolic testing, distance values are utilized to force execution closer to target, eventually converging.

Algorithm 3 shows this procedure. If the target is covered during initial random simulation, input vector is immediately returned (line 1-5). If not, then one of the adjacent branches of the current execution (τ) path is selected to be explored next. We have selected branches depending on its distance from target. Branches with lower distance will be selected first. If two such branches have same distance, then

Algorithm 3 Concolic Testing

```
Input: CFG with distance values
Output: Test vector
 1: Input vector, $I \leftarrow random()$
 2: Execution path, $\tau \leftarrow simulate(I)$
 3: if target statement covered in $\tau$ then
 4:     return $I$
 5: end if
 6: $B \leftarrow$ list of branches adjacent to $\tau$
 7: $sort\_by\_distance(B)$
 8: for all branch $b \in B$ and $is\_selectable(b)$ do
 9:     $C \leftarrow$ path constraints upto $b$
10:     $I \leftarrow constraint\_solver(C)$
11:     if $I$ is valid then
12:         decrease priority of $b$
13:         goto 2
14:     end if
15: end for
16: return $invalid$
```

the branch that comes earlier in the execution path will be selected. The $sort\_by\_distance()$ function in line 7 sorts branches following this scheme. After sorting, branch with lowest distance, b, is checked if it is valid for selection by calling $is\_selectable(b)$ function (line 8). This function does two important jobs. First, it checks if at least one of the preconditions of b is met. Second, it checks if selecting b will match any previously traversed path. Details of these will be discussed in Section III. Once branch selection is completed, constraints upto and including that branch is given to the constraint solver (line 9-10). If a satisfiable input vector is found, iteration starts with this new input vector (line 11-14). Otherwise, branch with next lowest distance is evaluated. This procedure is repeated until target is covered or no satisfiable branches are found.

As seen so far, this algorithm is greedy - always trying to force execution through branches with lowest distance. However, such approach will result in exhausting lower distance branches for all clock cycles before moving onto branches with higher distance. This issue is resolved by increasing distance of a branch each time it is selected (line 12).

Following table shows the concolic testing iterations for the example of Figure 3. Reset sequence at clock 0 is omitted for

iter	execution path at			sorted	sel.
	clock 1	clock 2	clock 3	branch	
0	l_2, l_4, l_6, l_9 l_{12}, l_{14}, l_{20}	l_2, l_4, l_6, l_9 l_{12}, l_{14}, l_{20}	l_2, l_4, l_6, l_9 l_{12}, l_{14}, l_{20}	l_{15}, l_7, l_5 ...	l_5
1	$l_2, l_4, \mathbf{l_5}$ l_{12}, l_{14}, l_{20}	l_2, l_4, l_6, l_9 l_{12}, l_{14}, l_{20}	l_2, l_4, l_6, l_9 l_{12}, l_{14}, l_{20}	l_{15}, l_7, l_5 ...	l_7
2	l_2, l_4, l_5 l_{12}, l_{14}, l_{20}	$l_2, l_4, l_6, \mathbf{l_7}$ l_{12}, l_{14}, l_{20}	l_2, l_4, l_6, l_9 $l_{12}, l_{14}, \mathbf{l_{15}}, l_{18}$	l_{16}, l_{15}, l_7 ...	l_{16}
3	l_2, l_4, l_5 l_{12}, l_{14}, l_{20}	l_2, l_4, l_6, l_7 l_{12}, l_{14}, l_{20}	l_2, l_4, l_6, l_7 $l_{12}, l_{14}, l_{15}, \mathbf{l_{16}}$	–	–

clarity. In this table, fifth column shows the sorted branches according to distance and sixth column shows the selected branch. We can see that sometimes branches with lower priority is selected. This is because either their preconditions are not met, or it will lead to a previously traversed path. For this specific example, our approach covers target at third iteration.

III. Discussion

A. Preventing selection of previously traversed path

Among all the alternate branches, sometimes a branch might get selected which will lead to a previously exercised path. This type of redundancy may lead to continuous looping between same branches. One way to avert this issue is to check whether the branch being selected is covered in the same clock before. However, if there are multiple paths leading to that branch, this scheme might prohibit going through some prerequisite path leading to the target. A proper way would require maintaining a set of execution paths (i.e. execution tree) T. A branch should only be selected if the speculated path does not exist in T.

To maintain a set of execution paths, each path must be identified uniquely. Sequence of branches in the path can be used for this purpose. However, maintaining a set of branch sequences and frequently searching within this set is not efficient. We used a combination of tabulation and multiplicative hashing to map branch sequences for efficient searching. This is performed within $is\_selectable()$ function in Algorithm 3.

B. Omitting conflicting clauses from other processes

Contrary to what happens in actual hardware, all the concurrent processes are executed in sequence by a simulator. This false sense of sequential execution can produce misguiding conflicting clauses.

To illustrate this issue, consider the example of Figure 2. Here Process 1 and Process 2 are concurrent. However, suppose the simulator used for generating trace executes Process 1 before Process 2. Also, assume that $state = C$ and $in = 13$ at current time frame t. This will make following execution path: $\tau = (\neg l_2, l_4, l_5, \neg l_{12}, l_{14}, l_{15}, l_{16})_t$. Now suppose we want to go to l_{18} by negating l_{15}. For this, we will provide $(\neg l_2, l_4, l_5, \neg l_{12}, l_{14}, \neg l_{15})_t$ to the constraint solver. In this case solver cannot come up with a solution because $\neg l_{15}$ will conflict with l_4. This problem will persist even if unrolled for more than two cycles. However, it is actually possible to go to l_{18} by setting in to any value other than 13. If the simulator executed parent process of l_{15} before any other process, this problem could have been avoided. For example, if Process 2 was executed before Process 1, τ would become $(\neg l_{12}, l_{14}, l_{15}, l_{16}, \neg l_2, l_4, l_5)_t$. When negating l_{15}, following constraints would be given to solver: $(\neg l_{12}, l_{14}. \neg l_{15})_t$. This time, the conflicting clause l_4 does not appear anymore, and solver can come up with a satisfiable test.

We prevented this problem with following procedure. First, all the processes that will execute at the same time frame are grouped together. Next, we determine the parent process and time frame of the branch being selected. Finally, all the clauses that occurred in the same time frame and belong to the same group, but not in the same process are ignored. In case of the previous example, both Process 1 and Process 2 falls into the same group. If we want to negate l_{15} as before, parent process of l_{15} is Process 2, and its clock cycle is t. Following our suggested method, $(\neg l_2, l_4, l_5)_t$ clauses would be skipped here, eliminating the conflict.

C. Necessity of guard condition expansion

As mentioned in the edge realignment section, before determining the satisfying assignments for a guard condition, the condition is expanded. To illustrate expansion procedure, consider the following Verilog statement: *assign a = b&c;*, where *b* and *c* are strict variables. Now, for a guard condition of *if(a==1)*, the expanded guard condition will be *if((b&c)==1)*. If we do not expand the guard condition, then edge will be aligned for the immediate dominator instead of itself. On the other hand, for the expanded guard condition, as it contains strict variables, edge realignment will be done properly. A valid concern at this point would be - why are we not using complete data flow analysis instead of guard expansion and strict variables? We avoided complete data flow analysis because it would require state unrolling of CFG and is susceptible to state space explosion.

D. Optimization to reduce solver calls

During edge realignment procedure, precondition of many branches are evaluated. This information can be used to reduce unnecessary solver calls. While selecting a branch for negation, it is evident that control flow must go through at least one of the preconditions for the branch selection to be satisfiable. More specifically, if the precondition is a blocking assignment, then it must be covered in the same or any previous time frame. On the other hand, if the precondition is a non-blocking assignment, then it must be covered in any previous time frame. If this criteria is not met, then trying to force through that particular branch will always result in conflicting constraints. This check is done by $is\_selectable()$ function of Algorithm 3. Skipping such branches saves solver calls and in turn improves performance.

IV. Experiments

A. Experimental Setup

Experiments are conducted in a machine with Intel Core-i7 6700k processor and 16GB of RAM. Benchmarks are collected from TrustHub, ITC99 and OpenCores [7]–[10]. Most of these benchmarks contain hard to cover branches. The concolic testing framework is implemented using open-source Icarus Verilog Target API [11]. Simulation for concrete execution stage is also carried out using Icarus Verilog. Yices SMT solver is used for solving symbolic constraints [12].

B. Evaluation of Scalability

To demonstrate the scalability of our approach, we have compared it with bounded model checking based tool EBMC [13]. AES cipher from OpenCores is used as base design,

TABLE I
Scalability Comparison with Model Checking

Benchmark	Unroll cycles	Lines of code*	EBMC [13]		Our Approach	
			Time (s)	Mem (MB)	Time (s)	Mem (MB)
cb_aes_01	5	33 k	1.27	179.4	0.51	55.3
cb_aes_05	10	167 k	11.47	1450.3	4.03	244.3
cb_aes_10	15	334 k	33.17	4130.6	14.47	502.4
cb_aes_15	20	501 k	70.78	8041.2	32.14	778.2
cb_aes_20	25	668 k	110.13	13202.8	86.03	1085.5
cb_aes_25	30	886 k	–	–	150.54	1405.3
cb_aes_30	35	1003 k	–	–	243.02	1780.3
cb_aes_35	40	1169 k	–	–	371.23	2112.7
cb_aes_10	15	334 k	33.17	4130.6	14.47	502.4
cb_aes_10	20	334 k	42.72	5361.8	15.33	520.2
cb_aes_10	25	334 k	53.78	6628.8	16.25	542.6
cb_aes_10	30	334 k	64.12	7871.5	17.22	563.7
cb_aes_10	35	334 k	74.32	9119.5	18.47	582.5
cb_aes_10	40	334 k	84.57	10361.3	19.36	608.0

* after hierarchy flattening

and we varied its round number to tune design complexity. These benchmarks are named as *cb_aes_xx*, where *xx* refers to the number of rounds. These rounds are cascaded sequentially and the final activation condition is chosen in such a way that it depends on all intermediate stages. This ensured that increasing circuit complexity will increase difficulty of test vector generation. Furthermore, as the activation condition depends on the last round's output, these benchmarks must be unrolled for at least the number of rounds it contain, and more if there is any reset sequence.

Table I compares the time and memory consumption of EBMC and our approach. Here, first 8 rows mainly highlight the effect of increasing design complexity. For EMBC, we can see that memory requirement grows exponentially with design complexity. It exceeds available memory after cb_aes_20 and fails to generate a test. On the other hand, we can see a linear increase in memory requirement for our approach. Time is in similar range for both, with our approach being faster.

The last 6 rows show the effect of varying unroll cycles, keeping design complexity constant. For EBMC, memory consumption increased by 150.8% and when changing unroll cycles from 15 to 40. Similarly, time requirement increased by 155%. On the other hand, unrolling more with keeping design complexity same have much less impact on our concolic testing based approach - with memory and time increasing by 21% and 33.8% respectively.

In summery, unlike bounded model checking based approaches, proposed approach scales well with both design complexity and unroll cycles.

C. Evaluation of Coverage

We compared our approach against uniform coverage concolic testing method (QUEBS [14]) and directed concolic testing method (CFG-Directed, which is a naive extension of [5] to apply on RTL models). For ITC99 and OpenCore benchmarks, target branches are selected in two ways - *random* and *rare*. For *random* target selection, five reachable branches are chosen randomly, and then results are averaged. For *rare* type,

TABLE II
PERFORMANCE COMPARISON BETWEEN TEST GENERATION METHODS

Benchmark	Unroll Cycles	Target	QUEBS [14]		CFG-Directed*		Our Approach	
			Time(s)	Iter	Time	Iter	Time(s)	Iter
b06	10	rand	0.02	13.6	0.02	9.4	0.01	3
		rare	0.02	20	0.01	7	0.01	5
b10	30	rand	0.11	245.4	0.01	4.4	0.01	1.4
		rare	0.13	301	0.01	5	0.01	1
b14	30	rand	0.61	564	0.01	8.4	0.01	3.6
		rare	0.90	814	0.01	21	0.01	1
i2c	10	rand	–	–	1.62	356.4	0.57	21.2
		rare	–	–	4.09	1123	0.98	40
OR1200 ICache	50	rand	0.23	155	0.13	27.2	0.02	5.4
		rare	0.34	224	0.20	34	0.02	9
AES-T1000	10	trojan	–	–	4.67	1	3.88	1
AES-T1100	10	trojan	–	–	19.62	7	11.80	4
wb_conmax T200	10	trojan	–	–	–	–	13.36	1
wb_conmax T300	10	trojan	–	–	–	–	11.06	1

* extension of [5] to apply on RTL models.

benchmarks are simulated with random inputs for one million cycles, and then the branch that is covered least number of times is selected as target. TrustHub benchmarks contains malicious Trojans with extremely difficult to activate trigger conditions. Triggers are used as target for these benchmarks.

Table II presents the comparison results in terms of test generation time and number of iterations. If a test vector is not generated within 2000 iterations, it is considered as a fail. As we can see, both QUEBS and CFG-directed failed for some targets with the imposed iteration limit. This result is expected from QUEBS, because it is designed with overall coverage in mind. Compared to CFG-directed, proposed approach improved number of iterations by 16.8 times on average (without considering the failed cases of wb_conmax). Overall, our method managed to cover targets with different characteristics within a small number of iterations.

V. RELATED WORKS

Extensive research has been conducted on directed test generation. Symbolic backward execution (SBE) based strategies starts from the target statement and symbolically executes the program backwards until it reaches an entry point [15]–[17]. However, backward exploration is susceptible to path explosion problem, and faces difficulty in presence of non-linearity or data dependent loops. Burnim et al. proposed an approach, which statically assigns distance values to CFG for guiding forward execution. A similar method is proposed in [6], which evaluates the control flow edges that must be covered to reach target statement. Unfortunately, all of these approaches considers only sequential (software) models, and are not directly applicable on hardware designs.

Bounded model checking has been used to enables directed test generation on hardware designs [13], [18]. However, they are prone to state explosion problem, and cannot be applied on large designs. Concolic testing has been used for covering branch statements in RTL models using depth-first search [4] as well as heuristically guiding execution towards uncovered branches [19]. Qin et al. added support of dynamic array references in concolic testing [20]. An exhaustive strategy is proposed recently, in which fully explored states are cached and prevented from redundant exploration [21]. Ahmed et al. proposed a concolic testing framework with tunable parameter to control search effort within a region [14]. Unfortunately, these methods tries to maximize overall coverage, and are not designed for directed test generation.

VI. CONCLUSION

In this paper, we have proposed an automated and scalable technique for directed test generation of RTL models. Our approach used precondition analysis to statically determine the target distance in the CFG of a design. This analysis avoided state explosion issue by exploiting the structural similarity of CFG across multiple clock cycles. Distance determined from this analysis is then effectively utilized by concolic testing framework to guide execution towards target. Experimental results demonstrated that our approach scales well with design size, both in terms of test generation time and memory requirement. Furthermore, it provided a drastic reduction in the required number of iterations compared to the state-of-the-art test generation methods.

REFERENCES

[1] Chen et al., *System-level validation: high-level modeling and directed test generation techniques*. Springer Science & Business Media, 2012.
[2] Sen et al., "CUTE: a concolic unit testing engine for C," in *SIGSOFT*, vol. 30, 2005, pp. 263–272.
[3] Godefroid et al., "DART: directed automated random testing," in *SIGPLAN*, vol. 40, 2005, pp. 213–223.
[4] L. Liu and S. Vasudevan, "STAR: Generating input vectors for design validation by static analysis of RTL," in *HLDVT*, 2009, pp. 32–37.
[5] J. Burnim and K. Sen, "Heuristics for scalable dynamic test generation," in *ASE*, 2008, pp. 443–446.
[6] C. Zamfir and G. Candea, "Execution synthesis: a technique for automated software debugging," in *EuroSys*, 2010, pp. 321–334.
[7] Tehranipoor et al., "TrustHub," *On-line: https://www.trust-hub.org*, 2017.
[8] Salmani et al., "On design vulnerability analysis and trust benchmarks development," in *ICCD*, 2013, pp. 471–474.
[9] F. Corno et al., "RT-level ITC'99 benchmarks and first ATPG results," *IEEE Design & Test of Computers*, vol. 17, pp. 44–53, 2000.
[10] "OpenCores website," *On-line: https://www.opencores.org*, 2017.
[11] S. Williams, "Icarus verilog," *On-line: http://iverilog.icarus.com/*, 2006.
[12] B. Dutertre and L. De Moura, "The yices smt solver," *Tool paper at http://yices. csl. sri. com/tool-paper.pdf*, vol. 2, pp. 1–2, 2006.
[13] Mukherjee et al., "Hardware verification using software analyzers," in *ISVLSI*, 2015, pp. 7–12.
[14] A. Ahmed and P. Mishra, "QUEBS: Qualifying event based search in concolic testing for validation of RTL models," in *ICCD*, 2017.
[15] F. Charreteur and A. Gotlieb, "Constraint-based test input generation for java bytecode," in *ISSRE*, 2010, pp. 131–140.
[16] Chandra et al., "Snugglebug: a powerful approach to weakest preconditions," *SIGPLAN*, vol. 44, pp. 363–374, 2009.
[17] P. Dinges and G. Agha, "Targeted test input generation using symbolic-concrete backward execution," in *ASE*, 2014, pp. 31–36.
[18] Cimatti et al., "NuSMV 2: An opensource tool for symbolic model checking," in *CAV*, 2002, pp. 359–364.
[19] L. Liu and S. Vasudevan, "Efficient validation input generation in RTL by hybridized source code analysis," in *DATE*, 2011, pp. 1–6.
[20] X. Qin and P. Mishra, "Scalable test generation by interleaving concrete and symbolic execution," in *VLSID*, 2014, pp. 104–109.
[21] L. Liu and S. Vasudevan, "Scaling input stimulus generation through hybrid static and dynamic analysis of rtl," *TODAES*, vol. 20, p. 4, 2014.