Incremental Concolic Testing of Register-Transfer Level Designs

HASINI WITHARANA, University of Florida, USA
ARUNA JAYASENA, University of Florida, USA
PRABHAT MISHRA, University of Florida, USA

Concolic testing is a scalable solution for automated generation of directed tests for validation of hardware designs. Unfortunately, concolic testing fails to cover complex corner cases such as hard-to-activate branches. In this paper, we propose an incremental concolic testing technique to cover hard-to-activate branches in register-transfer level (RTL) models. We show that a complex branch condition can be viewed as a sequence of easy-to-activate events. We map the branch coverage problem to the coverage of a sequence of events. We propose an efficient algorithm to cover the sequence of events using concolic testing. Specifically, the test generated to activate the current event is used as the starting point to activate the next event in the sequence. Experimental results demonstrate that our approach can be used to generate directed tests to cover complex corner cases in RTL models while state-of-the-art methods fail to activate them.

1 INTRODUCTION

Functional validation is a major bottleneck for modern System-on-Chip (SoC) designs. According to the Wilson Research 2020 functional verification study [1], more than 50% of development time in hardware designs were spent in verification. Irrespective of the validation effort, only 32% of the systems can achieve the first silicon success [1]. Simulation is the most widely used form of functional validation. Even millions of random tests may not be able to activate complex corner cases such as hard-to-detect branches in Register-Transfer Level (RTL) designs. Specifically, memory and processor designs have complex hard-to-detect branches due to the nature of concurrency, shared environments and memory consistency. As a result, it is unlikely to achieve 100% functional coverage using random or constrained-random tests for industrial RTL designs. To improve the coverage, verification engineers typically write manual tests to cover the remaining functional scenarios. Manual test writing can be cumbersome and error-prone. In fact, it may be infeasible to write manual tests for complex designs. There is a critical need for automated generation of directed tests to verify such complex RTL models.

Automated test generation can be performed using formal as well as semi-formal techniques [2]. For example, SAT-based bounded model checking searches the state space to generate counterexamples (directed tests). Since the number of states increases exponentially with the increase of unroll cycles, formal methods is likely to face state space explosion for complex designs. Concolic testing is a semi-formal approach that uses an effective combination of concrete simulation and symbolic execution. Concolic testing is scalable since it explores only execution path at a time (unlike formal methods that tries to explore all possible paths).

Authors' addresses: Hasini Witharana, University of Florida, Gainesville, Florida, 32611, USA, witharana.hasini@ufl.edu; Aruna Jayasena, University of Florida, Gainesville, Florida, 32611, USA; Prabhat Mishra, University of Florida, Gainesville, Florida, 32611, USA.

XXXX-XXXX/2024/3-ART $15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

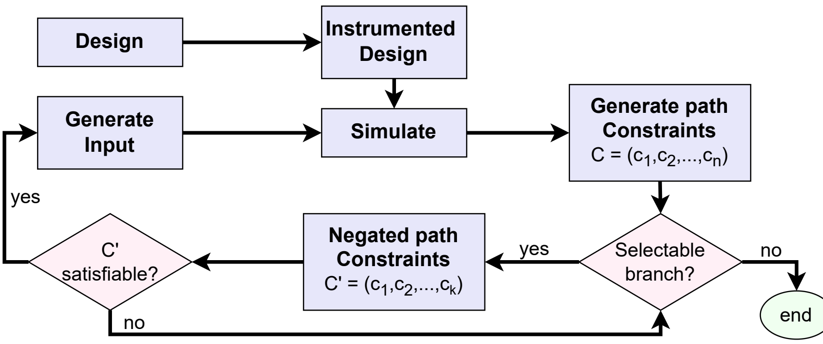


Fig. 1. An overview of concolic testing that effectively combines concrete simulation with symbolic execution.

Concolic testing has been successfully used as a directed test generation method in both software [3, 4] and hardware domains [5]. Figure 1 shows an overview of the concolic testing framework. The design is instrumented so that the tool can identify the executed path during simulation. Next, the instrumented design is simulated using an initial vector. The initial test vector can be generated using random or any other test generation methods. The execution path of the design is identified by analyzing the simulation trace. Next, an alternate path is selected by negating one of the branch constraints. The path constraints to activate the selected branch (alternate branch) will be sent to a constraint solver. Constraint solver will produce a solution if the constraints are satisfiable. This solution is used to generate a new test vector to activate the selected branch. If the constraint solver cannot solve the constraints (solution is unsatisfiable), an alternate branch is selected. This process continues until the expected coverage is achieved. Since concolic testing explores one path at a time, it overcomes the state space exploration problem. However, concolic testing faces the path exploration problem due to the exponential number of possible paths to explore. Path exploration problem can be mitigated by using a profitable alternate branch selection approach.

1.1 Motivation: An Illustrative Example

Alternate branch selection depends on the coverage goal. Existing approaches [6] try to maximize the overall coverage while try to cover specific branch target [5, 7]. In this paper, we are considering activation of hard-to-activate branches in RTL models. Some branches become hard-to-activate due to the complex temporal dependencies that should be preserved in-order to activate that branch.

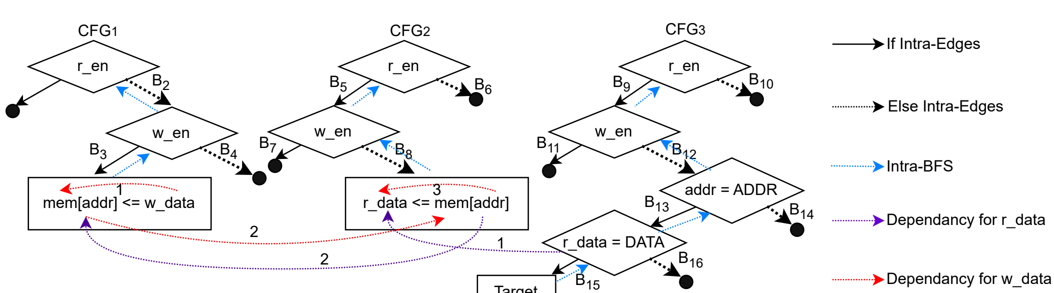


Fig. 2. Control and data flow graphs for the *ram* design in Listing 1. (BFS: Breadth First Search)

Example 1: We use a simple Verilog design (Listing 1) to describe various concepts in this paper. Listing 1 has three *always* blocks corresponding to three functionalities in a simple memory module: write functionality (line 9 - 18), read functionality (line 19 - 28), system functionality (line 29 - 42). While read and write are basic memory operations, the system functionality can be viewed as the top module (e.g., processor) trying to check a write followed by a read. For the ease of illustration, we are not showing all the else blocks for the *if* statements. Figure 2 presents the control and data flow for Listing 1. The three always blocks presented in the example corresponds to the three CFGs as CFG_1 (memory write), CFG_2 (memory read) and CFG_3 (check). The solid black lines represents control flow when the branch condition is true, while the flow for the false condition is represented using black dotted lines. □

Listing 1. Example of a memory module in Verilog

```
module ram
 input                    clk, rst,
 input  [ADDR_W-1:0]      addr, //write signals
 input                    w_en,
 input [DATA_W-1:0]       w_data, //read signals
 input                    r_en,
 output reg [DATA_W-1:0]  r_data, //memory declaration
 reg [DATA_W-1:0]  mem [2**ADDR_W-1:0];
//Memory write
 always @(posedge clk) begin
  if(r_en) begin
   //B1
  end
  else begin
   if(w_en)begin //B2
     mem[addr] <= w_data; //B3
   end
  end
 end
//Memory read
 always @(posedge clk)  begin
  if(r_en)
    if (w_en) begin //B5
     //B7
    end
    else begin
     r_data <= mem[addr]; //B8
    end
  end
 end
//Check write followed by read
 always @(*) begin
   if(r_en) begin
     if (w_en) begin  //B9
      //B11
     end
     else begin
      if(addr == ADDR) begin //B12
         if (r_data == DATA) begin //B13
           $display("Target"); //B15
         end
       end
     end
   end
```

```
42. end
43. endmodule
```

Consider line 36 in Listing 1 that reads a value (r_data) from a specific memory address ($addr$). For this condition to be true, a write should happen to that specific memory address with the exact values. The read can only happen when read flag (r_en) is true and write flag (w_en) is false. However, write can only proceed when read flag (r_en) is false and write flag (w_en) is true. These are contradictory constraints that must be satisfied in-order to activate the branch. Existing concolic testing fails unless the design is sufficiently unrolled in such cases. Unrolling for a large number of cycles is not feasible for large designs.

1.2 Contributions

In this paper, we propose a sequence-based incremental concolic testing. Our proposed technique uses edge exploration by traversing the Control Flow Graph (CFG) of the RTL design to identify the event sequence. Next, it solves each sequence while maintaining the order and preserving each solution for solving the next sequence incrementally. This paper makes the following three major contributions.

(1) Proposes an event sequence based approach for concolic testing. For a given branch, the sequence of events are identified by statically analyzing the concurrent CFGs of the RTL design.
(2) Incrementally applies concolic testing on an event sequence and preserves the test vectors to build the directed test to activate the target (hard-to-detect branches).
(3) Extensive experimental evaluation using a memory and a processor design demonstrates the effectiveness of our approach.

This paper is organized as follows. Section 2 surveys existing test generation techniques. Section 3 defines related terms. Section 4 presents our proposed test generation framework. Section 5 presents experimental results. Finally, Section 6 concludes the paper.

2 RELATED WORK

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

2.1 Verification of RTL Models

As AI and ML continue to advance, memory requirements are becoming increasingly sophisticated. Memory modules need to deliver high performance while consuming minimal power. However, the scaling of technologies has led to complex memory designs, posing challenges for verification. To bridge the verification gap, design teams must employ advanced modeling and verification techniques. These techniques ensure that the silicon behaves as expected throughout the development process. Unlike software errors, rectifying errors in memory modules at later stages of the life cycle becomes significantly more difficult. To tackle this, memory designers are utilizing various verification techniques to verify the functionality of complex interactions within the memory modules [8–11]. There are various efforts [10, 12] that rely on abstracted implementation and provides verification guarantees. In contrast, the test patterns generated by our approach can be used to simulate the actual implementation. While there is a recent effort [13] that considers simulation of processor designs, but it assumes the availability of a golden ISA specification. In this work, we explore the use of concolic testing to activate hard-to-detect branches in both processor and memory designs, enabling comprehensive verification.

2.2 Test Generation using Formal Methods

There are several test generation methods such as manual testing, random testing and formal methods. When compared to random testing formal methods are suitable for directed test generation methods [2, 5, 7, 14–20]. Formal verification techniques mathematically prove system properties based on formal models and specifications. Formal verification methods include model checking, theorem proving, property checking, etc. Formal methods can also be applied to automated testing [21–46]. For example, model checking is widely used for automated generation of directed tests [2]. Specifically, a model checker uses the model of the design and the property (the negated version of the target activity) to produce a counterexample. It performs bounded model checking using binary decision diagrams (BDD) [47] or SAT solvers [48]. Unfortunately, model checking is not scalable due to the state explosion problem. While there are promising avenues to reduce the model checking complexity, formal methods are not scalable for automated test generation when dealing with complex behaviors (e.g., hard-to-detect branches) as well as large designs.

2.3 Test Generation using Concolic Testing

Concolic testing is a promising alternative to model checking based test generation. Specifically, it provides an effective combination of concrete simulation and symbolic execution [5]. Unlike model checking that tries to explore all possible (exponential) execution paths at the same time, concolic testing explores only one execution path at a time. Concolic testing has been successfully applied on both software [3, 4, 49, 50] and hardware designs [5, 51–53].

Although concolic testing can avoid state explosion problem, it faces path explosion problem since it needs to select a profitable path is each iteration. While there are promising solutions for selecting beneficial branches [5], they are not suitable for complex corner cases such as hard-to-detect branches with complex branch conditions. We propose an efficient mechanism to activate complex branch conditions by identifying it as a sequence of simple conditions and incrementally applying concolic testing to activate these simple conditions.

3 PRELIMINARIES AND DEFINITIONS

We define few terms that are used in this paper. While our approach is applicable on both Verilog and VHDL designs, for the ease of illustration, we use Verilog examples in the remainder of this paper.

Definition 1: Branch is a conditional statement which includes statements that should be executed if the condition is satisfied. We consider 'if' and 'case' statements as branches. Note that other statements (e.g., 'for' and 'while') can also be viewed as an 'if' statement. For example, line 36-38 in Listing 1 represent a branch statement. □

Definition 2: Branch condition is a Boolean expression that can be constructed using Boolean operators (&&, ||, !) between Boolean expressions, or relational operators (<, >, >=, <= , ==, ! =) between numeric expressions. For example, ($r\_data == DATA$) is the branch condition in Listing 1 (line 36). □

Definition 3: Each branch can have up to two blocks: if-block and else-block. Each block (B) is a sequence of statements that will be executed if the condition is true (if-block) or false (else-block). For example, B13 in Listing 1 (line 36-38) represents the if-block for the branch in line 35. Similarly, B15 (line 37) is the if-block for the branch in line 36. □

Definition 4: Control Flow Graph (CFG) represents a flow of control between the blocks/branches in an 'always' or 'initial' block in Verilog designs. A CFG is a directed graph, $G = (N, E)$.

Each node $n \in N$ represents a block. Each edge $e = (n_i, n_j) \in E$ corresponds to a possible control flow from block n_i to block n_j. The edges inside a CFG are called intra-edges whereas the edges between CFGs are called as inter-edges. For example, Figure 2 shows three CFGs corresponding to the three 'always' statements in Listing 1. □

Definition 5: Simulation trace is a sequence of blocks executed by simulation for a finite number of clock cycles (c_1, c_2, ..., c_n) and corresponding test vectors (t_1, t_2, ..., t_n) . This can be represented as a tuple (c_i, <B_1^i,...., B_j^i, ...>) where $1 \leq i \leq n$ (total unroll cycles) and $1 \leq j \leq$ number of all blocks. B_j^i represent that for clock cycle c_i, the test t_i is used to simulate, and the block B_j is executed. □

Definition 6: Sequence (S) is a sequence of blocks representing an execution path that should be followed in order to get to a specific block in a CFG. Sequence S can involve blocks from different CFGs. Consider S_k = < $B_{1,k}^1$,...., $B_{j,k}^i$, ... >, where $B_{j,k}^i$ implies that the j-th block (B_j) is included in the k-th sequence (S_K) during the i-th clock cycle (c_i). For example, to activate $Target$ in Listing 1, the execution path will include the following sequence of blocks in CFG3 (Figure 2): B9, B12, B13, B15. □

Definition 7: Test sequence (T_k) is a set of test vectors to activate the sequence of blocks in S_k. Specifically, T_k consists of <t_k^1, ..., t_k^i, ... t_k^d> where $1 \leq i \leq d$ and $d \leq n$. In t_k^i, i is the clock cycle and k is the sequence id. □

Definition 8: Branch target is a block that we want to activate for a specific outcome of a branch (true or false). The block that gets activated by activating the branch condition is the target block (B). B can be activated by following a sequences stack ($B \implies < S_1, S_2, ..., S_n >$). This implies that in order to activate the branch target (B), one needs to execute a predefined sequences stack in a particular order. □

Definition 9: A hard-to-activate branch is identified as a branch that remains unactivated even after applying a substantial number of random test patterns (n) or running concolic testing up to m unroll cycles. Section 5.2 outlines the procedure for finding hard-to-activate branches as well as provides illustrative examples of hard-to-activate branches in a cache design. □

4 INCREMENTAL CONCOLIC TESTING OF RTL MODELS

Figure 3 presents an overview of our proposed incremental concolic testing framework. It consists of three major tasks: sequence identification, design instrumentation, and incremental concolic testing.

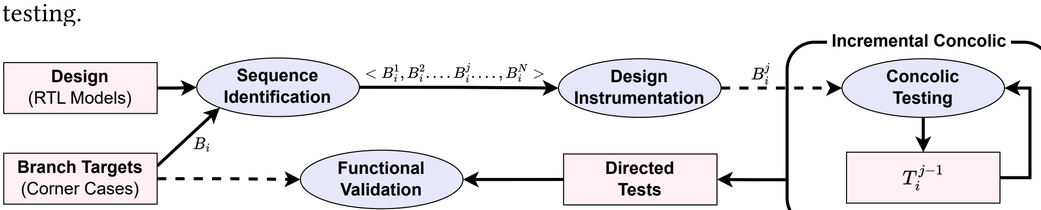


Fig. 3. Overview of our test generation framework. It consists of three important tasks: sequence identification, design instrumentation, and incremental concolic testing.

Algorithm 1 shows the relation between the three tasks. Given a design (D) and a branch target (B_i), the first step is to identify the sequences stack (SS) such that $B_i \implies < S_1, S_2, ..., S_n >$. The second step is to instrument the design by converting each sequence to a branch statement. The second step results in instrumented design (iD) and the target queue (TQ). The third step is to apply concolic testing for each of the branch statements in the order of the sequence. The generated test can be used to activate the branch target during functional validation. The remainder of this section describes these three tasks in detail.

Algorithm 1 Sequence-Based Incremental Concolic Testing

Input Design (D), Branch target (B_i)
Output Test T

SS ←SequenceIdentification(D, B_i)
<iD, TQ>←DesignInstrumentation(SS, Design)
T ←IncrementalConcolic(iD,TQ)
Return T

4.1 Sequence Identification

Algorithm 2 shows the procedure for sequence detection for a given branch target which consists of four major steps. The first step constructs the CFG for the design. This step can be performed using any existing Verilog language parser [54]. Figure 2 shows the CFG representation of the design in Listing 1. The next step extracts the branch condition for the target. This condition is an expression of the signals (SE). The third step uses *DependencySearch* function to recursively identify the assignment blocks that are relevant for each of the signal in SE. The *DependencySearch* function incorporates safeguards to prevent infinite loops caused by circular dependencies. This is achieved by introducing a mechanism to track and skip signals that have already been visited during the recursive search. The final output is a Sequence Stack (SS) containing the identified blocks representing the execution path required to reach the specific block associated with the given branch target. *FindAssignmentBlock*, gets the block which is closest to the branch target. This ensures that the shortest possible test vector is generated. When traversing the CFG to find the blocks that update a signal we traverse from the branch target. Therefore, the first block that is detected is added to the dependency search. The distance calculation used in original concolic testing [15] is used when finding the assignment block.

Example 2: In Listing 1, consider the target as line 37 where the block is (B_{15}) and this is represented in Figure 2 as the "Target". Line 1 of Algorithm 2 produces three concurrent CFGs with inter-CFG edges in Figure 2. Line 2 of Algorithm 2 produces the branch condition (line 36 in Listing 1) as $SE \leftarrow <r\_data ==$ DATA>. This signal expression consists of one signal (r_data) and one constant value ($DATA$). Since no action needed for $DATA$, the DependencySearch routine only tries to find the assignment block corresponding to signal r_data. The dependency search for r_data is shown in Figure 2 using the two purple dotted lines. The signal r_data appears in one assignment (Line 25 in Listing 1) where r_data is assigned the value of $mem[addr]$ in CFG_2 block B_8. The block B_8 is pushed into SS. Then the dependency search is executed for the signals mem and $addr$. Since the $addr$ is a primary input, the search will not continue for $addr$. An assignment exists for mem in line 15 where $mem[addr]$ is assigned the value of w_data in CFG_1 block B_3. The block B_3 is pushed into SS. Since w_data is a primary input and there are no more assignments for w_data, the recursion will end. Once the algorithm terminates, SS will have $<B_3, B_8>$. □

4.2 Design Instrumentation

Algorithm 3 shows the procedure for branch generation for a given sequence set SS. As shown in the algorithm, breadth first search is performed along the predecessors of the target block in the CFG (Intra-BFS) to extract the conditions to activate the target. Line 1 of the algorithm identifies the constraints for the target. For each sequence in the SS, it tries to identify the constraints

Algorithm 2 Sequence Identification

Input Design (D), Branch target (B_i)
Output Sequence Stack (SS)
1: $CFG \leftarrow$ ConstructCFG(D)
2: $SE \leftarrow$ GetSignalExpression(B_i.condition)
3: $SS \leftarrow$ DependencySearch(CFG, SE, $\emptyset$)
4: Return SS
5: **function** DependencySearch(CFG, SE, $visited$)
6:     **for** each signal $A \in SE$ **do**
7:         **if** A is not in $visited$ **then**
8:             $visited \leftarrow visited \cup \{A\}$
9:             $B_A \leftarrow$ FindAssignmentBlock(CFG, A)
10:             SS.push(B_A)
11:             DependencySearch(CFG, GetSignalExpression(B_A.condition), $visited$)
12:         **end if**
13:     **end for**
14:     **Return** SS
15: **end function**

using the similar intra-BFS (line 3). The constraints can have either resolved Boolean expressions or unresolved expressions. In the next step, constraints from the target are used to resolve the unresolved constraints of the sequence. First an intersection is performed between the unresolved constraints from the sequence and constraints from the target. The results of the intersection are the new resolved constraints for the sequence. If still some of the constraints are unresolved in the sequence, it searches through dependencies to identify any dependent signals for the target. If any of the dependent signals are in the target constraints, the value of the target constraint is used to resolve the sequence constraint. If there are still unresolved constraints, it implies that the scenario is untestable (target branch cannot be activated).

Example 3: To identify the constraints for "Target" block (B_{15} in Figure 2 and line 37 in Listing 1), intra-BFS is performed in CFG_3. This search is represented using blue dotted lines in Figure 2. Intra-BFS for "Target" is <B_{15}, B_{13}, B_{12}, B_9>. Based on this traversal, we get the constraints to activate "Target" as $r\_en = 1$, $w\_en = 0$, $addr = ADDR$ and $r\_data = DATA$. Next, Intra-BFS is performed for the blocks in SS (<B_3, B_8>). The constraints for B_3 are $r\_en = 0$, $w\_en = 1$, $mem = UR$, $addr = UR$ and $w\_data = UR$, and the constraints for B_8 are $r\_en = 1$, $w\_en = 0$, $mem = UR$, $addr = UR$ and $r\_data = UR$. Here, UR means unresolved. There are three unresolved constrained for B_3. We can resolve the first constraint $addr = UR$ to $addr = ADDR$. We need to search for dependencies to address the remaining two unresolved constraints (mem and w_data). The search of dependencies for w_data is shown in Figure 2 using red dotted lines. w_data is assigned to $mem[addr]$ and $mem[addr]$ is assigned to r_data. Once the search is complete, final dependency for w_data can be identified as r_data. Since r_data is included the target constraints, w_data gets the value of r_data. After discarding the unresolved constraints, the final constraints for B_3 are $r\_en = 0$, $w\_en = 1$, $addr = ADDR$ and $w\_data = DATA$ and for B_8 are $r\_en = 1$, $w\_en = 0$, $addr = ADDR$ and $r\_data = DATA$. □

In Algorithm 3, for each of the sequences in SS, conditional branches are created using the modified constraints (line 5) and these branches are embedded in the design. The newly created branches are stored in the TQ (Target Queue) preserving the order in the SS. When the first sequence

Algorithm 3 Design Instrumentation

Input Design (D), CFG, Target (B_i), Sequence Stack (SS)
Output Instrumented Design (iDesign), Target Queue (TQ)

1: Target Constraints $TC \leftarrow$ IntraBFS(CFG, B_i.block)
2: **for** each $S \in SS$ **do**
3:     Sequence Constraints $SC \leftarrow$IntraBFS(CFG, S)
4:     $SC \leftarrow$MODIFY(TC, SC, CFG)
5:     $TQ \leftarrow$CreateBranch(SC.resolved, D)
6:     iDesign $\leftarrow$ instrumentDesign(D, TQ)
7: **end for**
8: **Return** iDesign, TQ
9:
10: **function** MODIFY(TC, SC, CFG)
11:     SC.resolved $\leftarrow$ SC.unresolved $\cap$ TC
12:     **for** each $cons \in SC$.unresolved **do**
13:         Depend Signal $DS \leftarrow$Search(CFG, $cons$.signal)
14:         **if** $DS \in TC$ **then**
15:             $cons$.value $\leftarrow$ $TC[DS]$.value
16:             SC.resolved $\leftarrow$ SC.resolved $\cup$ $cons$
17:         **end if**
18:     **end for**
19:     **Return** SC
20: **end function**

is removed from the SS, corresponding branch of that sequence is the first element to insert in the TQ. This process continues until SS is empty. Finally, the modified design is instrumented (line 6). The goal of the instrumentation is to identify which path is executed by analyzing the simulation log. We achieve this goal by adding print statements for all the branch conditions and end of the blocks by using a unique identifier (block id) as illustrated in Example 4.

Listing 2. Branch creation for sequences

```
1.  if (r_en==1'b0 && w_en==1'b1 &&
     addr==ADDR  && w_data==DATA) begin
2.    $display("Target1") //B17
3.  end
4.  if (r_en==1'b1 && w_en==1'b0 &&
     addr==ADDR  && r_data==DATA) begin
5.    $display("Target2") //B19
6.  end
```

Example 4: The SS to activate the "Target" block (B_{15} in Figure 2) is <B_3, B_8>. The resolved constraints for both these sequences are presented in Example 3. Using those constraints, we can create branch statements for B_3 and B_8. The created branches using Algorithm 3 for B_3 and B_8 are shown in Listing 2 from line 1 - 3 and line 4 - 6, respectively. The corresponding block ids of these branches are stored in the TQ as $< B_{17}, B_{19} >$. After branch creation, instrumentation of the design is conducted. By analyzing the CFG, each block is given a unique block identifier. Th instrumentation of the first always block in Listing 1 (line 9 - 18) is shown in Listing 3. We add a print ($display) statement at the end of each block. This will print the blocks that got activated in each clock cycle along with clock cycle information. □

Listing 3. Example of design instrumentation

```
always@(posedge clk) begin
if(r_en) begin
 $display("B1");
end
else begin
 $display("B2");
 if(w_en) begin
  mem[addr] <= w_data;
  $display("B3");
 end
 else begin
    $display("B4");
 end
end
```

4.3 Incremental Concolic Testing

In this section, we present the incremental concolic testing scheme to activate a set of sequence events in the preserved order. Figure 4 presents a pictorial representation of incremental test generation. As shown in the figure, there are two sets: sequence set < S_1, S_2, ..., S_N > and the corresponding test set < T_1, T_2, ..., T_N >. To activate a sequence S_x, the required test is $\sum_{k=1}^{x} T_k$. For example, T_1 can activate S_1, but to activate S_2, we need both T_1 and T_2. A test set is a combination of different test vectors. A test T_x includes $\sum_{i=a}^{b} t_x^i$ where $a, b \leq n$ (unroll cycle). The test vectors in T_1 are < t_1^1, t_1^2, ..., t_1^d >, and the test vectors in T_2 are < t_2^{d+1}, t_2^{d+2}, ..., $t_2^{d'}$ >.

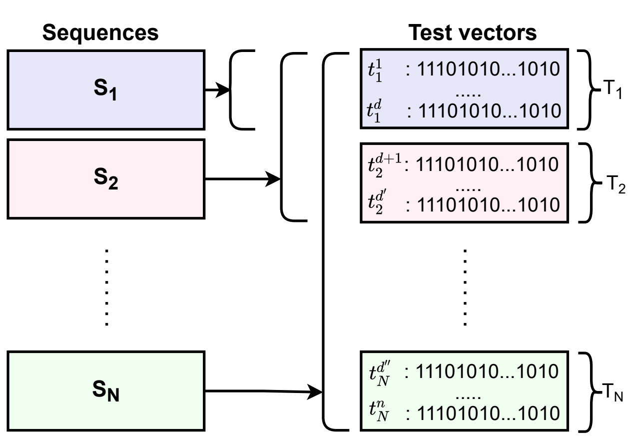


Fig. 4. Incremental test generation for a sequence set. Here S_j implies the j-th element of the sequence corresponding to branch B_i^j, i-th branch in the design.

Algorithm 4 describes the incremental test generation using concolic testing to activate a sequence of events preserving the order of events. Specifically, the test generated to activate the current event is used as the starting point to activate the next event in the sequence. For each target in TQ, we run concolic testing while changing the test set and the starting clock cycle (line 4). For the first target, the test set (T) is generated randomly and it contains test vectors up to the unroll cycle (n). The first step of concolic framework is to calculate the distance from the target to all the blocks. From the target breadth-first traversal is performed in the direction along the predecessors. The distance is initialized to 0 and incremented by 1 when an edge traversal is completed. Next, Path (P) is generated by simulating the design with test set T. All the alternate branches from the current path P is selected as the next step. When selecting the alternate branches, the clock is set to a specific starting clock cycle value so that we only select the branches after the starting clock cycle value. The path up to the starting clock cycle is set and unchanged. Then the selected alternate branches are sorted using the distance and the clock value. This will lead to the most profitable alternate branch. Using the trace of P and the chosen branch, constraint vector is generated. The constraint vector contains the value of the constraints for each of the clock cycles. Then the constraint vector is solved using a constraint solver. The constraint solver produces a new test set and this is used to simulate the design and get a new path. If the new path activates the target, the test set will be added to T. Also, the clock cycle of the selected branch will be set as the new starting clock cycle. Hence, the test set generated for the target will be preserved and used as a starting point to the next target in TQ.

Algorithm 4 Incremental Concolic Testing

```
Input Design (D), Target Queue (TQ), Unrolled Cycles (n), limit
Output Test Set T = T_1, T_2, ..., T_N
T ← Random Vectors
start ←1
for each target in TQ do
    T, start ←CONCOLIC(D, target, T, start)
end for
return T

function CONCOLIC(Design, target, T, start)
    Distance Set DS ← ComputeDistance(target, Design)
    Path P ← Simulate(T, Design)
    clock ← start
    while iteration < limit do
        AB ← AlternateBranch(P, DS, clock)
        CV ← BuildConstraints(AB, P)
        Test t ← SolveConstraints(CV)
        P ← Simulate(t, Design)
        if P activates the target then
            T.add(t)
            start ← AB.clock
            Break
        end if
    end while
    return T, start
end function
```

Example 5: Target Queue (TQ) contains 2 branch targets $<B_{17}, B_{19}>$ which are shown in Listing 2. Assume that the unroll cycle (n) is 10 and search *limit* is 10 iterations. Concolic is used to activate the first branch target (B_{17}) which is corresponding to writing a value to the memory. The *start* value is 1 and a random test set is used as initial setting. Suppose the test set to activate the target (B_{17}) is identified in unroll cycle 3. Then the starting cycle is set as 4 for the next target (B_{19}). The test set for activating B_{17} is shown in Listing 4 (line 1 - 3). This test set is used as a starting point to activate the second branch target (B_{19}) which is corresponding to reading a value form a memory (line 4 in Listing 4). □

Listing 4. Test to activate target

```
\\ Move the ADDR into R0
1. MOVQ R0, ADDR

\\ Move the DATA into R1
2. MOVQ R1, DATA

\\Store DATA in R1 in ADDR memory in R0
3. ST [R0], R1

\\ Load the value in ADDR memory in R0 to R2
4. LD R2, [R0]
```

While we utilize the core functions of concolic testing in Algorithm 4, we have incorporated our primary contributions for finding sequences, instrumenting design with new branches, and incrementally solving one sequence at a time to generate the required test to activate the target. Note that the instrumented design (including new branches) are used for test generation purpose only. We do not make any changes to the original design. During the functional validation, the generated tests are used to activate the branch targets (corner cases) on the original design.

5 EXPERIMENTS

In this section, we evaluate the effectiveness of our proposed approach using a wide variety of hard-to-activate branches in a memory and processor design. We first describe the experimental setup. Next, we outline the corner case scenarios. Finally, we present the experimental results.

5.1 Experimental Setup

To demonstrate the applicability of our framework, we have applied incremental concolic testing on two designs: (1) a re-configurable cache implementation, IOb-Cache [55], and (2) a processor design [56], which implements 32-bit RISC-V instruction set. In order to generate the abstract syntax tree of the RTL model, we use Icarus Verilog Target API [54]. We use Yices SMT solver [57] for solving constraints. Incremental concolic testing is implemented on top of the concolic testing framework proposed in [5]. In order to ensure validity of the generated test vectors, we simulate the original design with the generated test and analyze the Value Change Dump (VCD) to confirm the activation of the target (corner case scenario). We ran our experiments on Intel i7-5500U @ 3.0GHz CPU with 16GB RAM machine.

5.1.1 Memory module. Memory module interfaces with a processor and main memory as shown in Figure 5. The design of the IOb-Cache consists of four components: *Front-End*, *Cache-Memory*, *Cache-Control*, and *Back-End*. The *Front-End* implements the interface between the processor and the cache. The *Front-End* provides all data signals to the *Cache-Memory* and control signals are routed to the *Cache-Control*. The IOb-Cache is word-aligned and returns the entire word. *Cache-Memory* consists of various components including tag buffer, valid buffer, data write-through buffer, and replacement policy unit. This design can be configured as direct-mapped or set associative

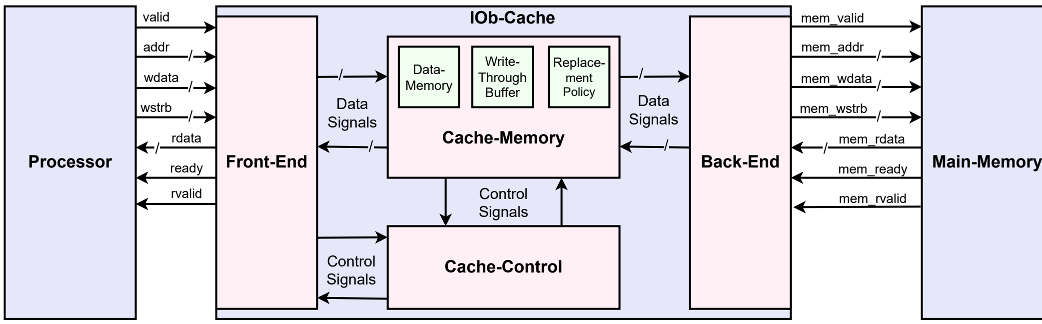


Fig. 5. IOb-Cache [55] block diagram for selected configurations outlined in Table 1.

(*N_way* as shown in Table 1). The replacement policy unit supports three different modes: Least-recently-used (LRU), Psuedo-least-recently-used (PLRU: MRU based, Binary tree-based). Finally, *Back-end* is responsible for interfacing the main memory with the cache. IOb-Cache supports both native and AXI interfaces. For the case studies in Section 5.2, we have selected configurations presented in Table 1. With the above configurations, we flattened the IOb-Cache module eliminating its hierarchy with Yosys [58] synthesis tool. The flattened RTL netlist is about 10,000 lines of code. The number of CFGs is 597. The average depth of the CFG is 2 branches and the maximum depth is 5 branches. A high-level block diagram with the inputs and outputs of the setup is presented in Figure 5. This configuration is used for validation of different functional scenarios outlined in Section 5.2.

Table 1. Configurations used for the IOb-Cache setup

Attribute	**Configuration 1**	**Configuration 2**
Addr width	16	32
Data width	32	32
Ram type	Native	AXI
Write Policy	Write Back	Write through
Replacement Policy	LRU	PLRU_mru
N_way	4	4

5.1.2 Processor. PicoRV32 [56] consists of 32 internal registers and can be configured for dual-port register implementation. During the experiments, we communicate with the processor with native memory interface. Input and output configurations of the native memory interface are presented in Figure 6. The specific configurations are listed in Table 2. Memory read operations are initiated by the picorv32 core, signaling the need for data through the assertion of *mem_valid* and specifying the target address with *mem_addr*. The read data is then communicated to the processor through *mem_rdata*. On the other hand, for memory write operations, the picorv32 core triggers writes by asserting *mem_valid*, providing the address and data through *mem_addr* and *mem_wdata*, and indicating the write strobe with *mem_wstrb*. The flattened RTL netlist of Picorv32 is about 100,000 lines of code. The number of CFGs is 8695, average depth is 2, and maximum depth is 6 branches.

5.2 Corner Case Scenarios

We refer a branch as "hard-to-activate" if it does not get activated even after simulating for a considerable amount of test patterns. In general, we use a threshold (e.g., after applying **n** test

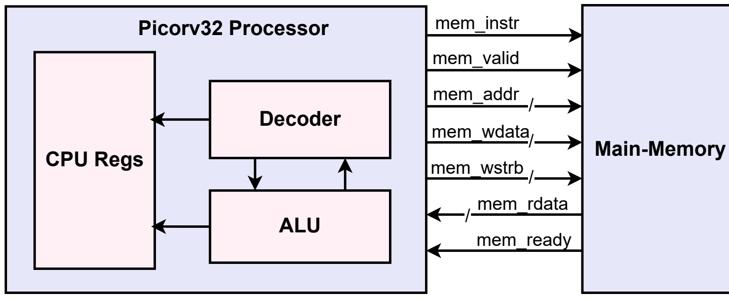


Fig. 6. Picorv32 [56] block diagram for selected configurations outlined in Table 2.

Table 2. Configurations used for the PicoRV32 setup

Attribute	**Value**
REGS_16_31	Enabled
DUALPORT_REGS	Enabled
PROGADDR_RESET	32'h 10_0000
STACKADDR	1024
TWO_STAGE_SHIFT	Enabled
Interrupt requests	Enabled

random patterns and **m** unroll cycles in concolic testing) to figure out hard-to-activate corner cases. Table 3 shows various branches and how many times they are activated with the increasing number of random test patterns for IOb cache. IOb case has 446 branches and 94 are considered hard-to-activate even after 100,000 test patterns.

Table 3. Random testing applied to activate branches in IOb cache

# Test	100	1000	10000	100000
# Branches Activated	329	331	339	352
# Uncovered Branches	117	115	107	94
% of Uncovered Branches	26.2%	25.78%	23.99%	21.07%

We can further refine hard-to-activate branches where we try to activate the branches which were not activated by previous random/constrained-random tests using concolic testing. We introduce a parameter unroll cycles (**m**) with respect to the classical concolic testing. We have considered corner cases as the branches that do not get activated after 15 unroll cycles using classical concolic testing for IOb-cache. Table 4 illustrates the percentage of hard-to-activate corner cases after unrolling the IOb design for different **m** values. In this example, concolic testing is able to activate 26 (out of 94) but it still cannot activate 68 branches, which are considered as corner cases.

Table 4. Original concolic testing applied to activate branches, that are not activated by random testing

# Unroll Cycles	5	10	15
# Branches Activated	17	24	26
# Uncovered Branches	77	70	68
% of Uncovered Branches	81.91%	74.46 %	72.34%

We identify the corner case scenarios (hard-to-detect branches) in a IOb design if it does not get activated even after running 100,000 random test cases and 15 unroll cycles with classical concolic testing. In case of Picrov32 design, we have used 100,000 random test cases and 50 unroll cycles as threshold to identify the hard-to-detect branches. In this section, we present different illustrative examples of corner case scenarios for memory and processor verification. Specifically, we consider eight corner cases related to memory modules and three corner cases for the processor design.

5.2.1 Corner case scenarios for memory. We have illustrated different hard-to-detect branches identified in memory verification.

Case 1: Write a specific value to memory as shown in Listing 5.

Listing 5. Case 1

```
if (ready == 1'b1)
 if (wstrb == 1'b1)
  if(addr == 16'h1234)
   if(w_data == 32'hCAFEFEED) begin
    $display("Target")
   end
```

Case 2: Read a specific data form a specific address as shown in Listing 6. This scenario is similar to the target in Listing 1.

Listing 6. Case 2

```
if (ready == 1'b1)
 if (wstrb == 1'b0)
  if(addr == 16'h1234)
   if(r_data == 32'hCAFEFEED) begin
    $display("Target")
   end
```

Case 3: Back to back writes to the same address as shown in Listing 7. We copied the entries in Listing 5 for 5 times and changed the data values.

Listing 7. Case 3

```
if (ready == 1'b1)
 if (wstrb == 1'b1)
  if(addr == 16'h1234)
   if(w_data == 32'hCAFEFEED) begin
    $display("Target1")
   end

if (ready == 1'b1)
 if (wstrb == 1'b1)
  if(addr == 16'h1234)
   if(w_data == 32'hABCEFEED) begin
    $display("Target2")
   end
...
```

Case 4: Back to back reads from the same address as shown in Listing 8. We copied the entries in Listing 6 for 5 times and changed the data values.

Listing 8. Case 4

```
if (ready == 1'b1)
 if (wstrb == 1'b0)
  if(addr == 16'h1234)
   if(r_data == 32'hCAFEFEED) begin
    $display("Target1")
   end

if (ready == 1'b1)
 if (wstrb == 1'b0)
  if(addr == 16'h1234)
   if(r_data == 32'hABCEFEED) begin
    $display("Target2")
   end
...
```

Case 5: Write data to a boundary location in memory as shown in Listing 9. We used the Listing 5, created two copies, and changed the address value to 16’h0000 and 16’hFFFF, respectively.

Listing 9. Case 5

```
if (ready == 1'b1)
 if (wstrb == 1'b1)
  if(addr == 16'h0000)
   if(w_data == 32'hCAFEFEED) begin
    $display("Target1")
   end

if (ready == 1'b1)
 if (wstrb == 1'b1)
  if(addr == 16'hFFFF)
   if(w_data == 32'hCAFEFEED) begin
    $display("Target2")
   end
```

Case 6: Read data from a boundary location in memory as shown in Listing 10. We used the same Listing 6, created two copies, and changed the address value to 16’h0000 and 16’hFFFF, respectively.

Listing 10. Case 6

```
if (ready == 1'b1)
 if (wstrb == 1'b0)
  if(addr == 16'h0000)
   if(r_data == 32'hCAFEFEED) begin
    $display("Target1")
   end

if (ready == 1'b1)
 if (wstrb == 1'b0)
  if(addr == 16'hFFFF)
   if(r_data == 32'hCAFEFEED) begin
    $display("Target2")
   end
```

Case 7: Verify front-end and back-end addresses for correct address translation as shown in Listing 11. The specific address translations are identified by analyzing the RTL models of front-end and back-end modules. In a write-back cache, data is only written back to the memory when a cache line is flushed. If the design doesn't perform cache line flushes in certain scenarios, the conditions inside the if statements may not always be evaluated as true, and the corresponding display statements may not be executed. In this experiment, we use explicit flush commands (for the specific address we set the *cache_memory_invalidate* bit) to flush the cache line while we try to activate Case 7.

Listing 11. Case 7

```
if (addr == 16'h1234)
 if (front_end.data_addr == addr[15:2])
  $display("Target1")
 end
if (addr == 16'h1234)
 if (back_end.write_addr == addr[15:6])
  $display("Target2")
 end
```

Case 8: Verify cache hit for a specific memory read. As shown in Listing 12, when the required write happens before the read, the cache hit should get triggered.

Listing 12. Case 8

```
if (ready == 1'b1)
 if (wstrb == 1'b0)
  if(addr==16'h1234 && r_data=32'hCAFEFEED)
   if(cache_memory.hit == 1'b1) begin
    $display("Target")
 end
```

5.2.2 *Corner case scenarios for processor.* In this section, we present three corner cases for execution of a processor. Corner cases are illustrative examples of how to check several scenarios including setting the program counter, writing some arbitrary value to internal registers and reading a value from the internal register after writing. These types of test cases are useful in situations for debugging programs on processor designs. Let's assume a scenario where the processor needs to be configured to run from the middle of a program based on the earlier execution traces. In this case, sequence-based concolic testing allows for a division of the original firmware into several segments and checking for specific coverage scenarios of the design at each segment.

Case 9: Reading from a specific register in SRAM as shown in Listing 13.

Listing 13. Case 9

```
if (mem_la_addr == 32'h00120000) begin
    if (mem_la_read) begin
        $display ("Target")$;
    end
end
```

Case 10: Setting the program counter to a specific value as shown in Listing 14.

Listing 14. Case 10

```
if (! (latched_store && latched_branch)
    && reg_next_pc == 32'h00012004) begin
    if (! irq_pending) begin
        $display ("Target")$
    end
end
```

Case 11: Writing a specific value to internal register as shown in Listing 15.

Listing 15. Case 11

```
if (cpuregs_wrdata == 32'h64) begin
    if ((latched_rd == 5'h4) && resetn &&
        cpuregs_write && latched_rd ) begin
        $display ("Target")$;
    end
end
```

5.3 Test Generation Results

In this section, we present the results of our case study. We compare our approach with EBMC [59] and the concolic framework presented in [5]. EBMC is a state-of-the art formal verification framework that uses bounded model checking. The concolic framework [5] is state-of-the-art in activating RTL branch statements using concolic testing. The number of unrolled cycles are determined based on the complexity of the scenarios. This can be achieved by starting from a reasonable number of unroll cycles and increment until the scenarios are covered. The number of unroll cycles is analogous to the bound determination for bounded model checking. We set the bound for EBMC to be equal to the number of unroll cycles for concolic testing.

The corner case activation results at system level are shown in Table 5. The first column represents different corner case scenarios outlined in Section 5.2. For IOb cache we have selected the first configuration. The second column provides the unroll cycles (bound for EBMC). For each approach, we provide information about if the target (corner case) is activated (Yes or No) within the bound, and if yes, what is the memory requirement (in MB) and run time (in seconds). As shown in Table 5, EBMC only covers one scenario, and concolic [5] covers only 4 scenarios. Our approach successfully covered all the 11 scenarios. EBMC is expected to fail for most of the scenarios due to state space exploitation problem. The concolic framework in [5] activates some of the branches, however, when dealing with contradictory and complex sequences, it fails to activate the target due to path explosion problem ([5] selects branches based on the distance heuristics).

Table 5. Comparison of system-level target activation using EBMC [59], Concolic [5], and our approach

Cases	Unroll Cycles (Bound)	EBMC [59]			Concolic [5]			Our Approach		
		Activated	Memory (MB)	Time (s)	Activated	Memory (MB)	Time (s)	Activated	Memory (MB)	Time (s)
1	20	No	-	-	Yes	82.34	20.13	Yes	20.00	14.55
2	20	No	-	-	No	-	-	Yes	34.67	25.78
3	50	No	-	-	Yes	215.84	50.67	Yes	67.89	20.78
4	50	No	-	-	No	-	-	Yes	182.56	82.91
5	20	No	-	-	No	-	-	Yes	19.78	14.43
6	20	No	-	-	No	-	-	Yes	30.24	23.91
7	20	Yes	597.81	2.01	Yes	20.56	4.93	Yes	15.23	4.81
8	20	No	-	-	No	-	-	Yes	50.67	30.88
9	100	No	-	-	No	-	-	Yes	170.91	61.71
10	100	No	-	-	No	-	-	Yes	53.59	85.60
11	100	No	-	-	Yes	210.44	369.01	Yes	49.72	47.48

The final step of our framework is the functional validation using the generated test from incremental concolic testing. To validate the generated test vectors from our approach, we simulate the original design with the generated test and analyzed the VCD to confirm the activation of the corner case scenarios. Figure 7 shows the VCD for the test generated for Case 2. For Case 2, the first sequence is writing the data value to the address. This is achieved in clock cycle 7 when the ready signal has changed to 1 with *addr* = 16'h1234, *w_data* = 32'hCAFEFEED and *wstrb* = 4'hF. The next sequence for Case 2 is reading a value from an address. This is activated in clock cycle 10. The ready signal has changed to 1 with *addr* = 16'h1234, *r_data* = 32'hCAFEFEED and *wstrb* = 4'h0.

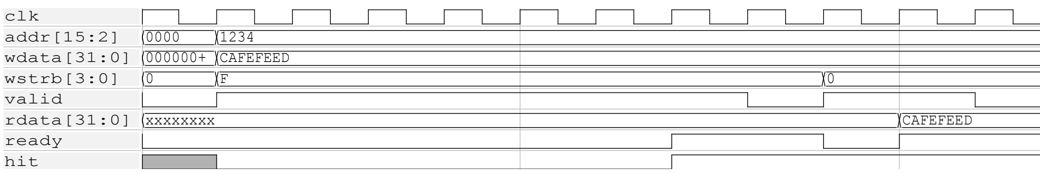


Fig. 7. Functional validation for Case 2.

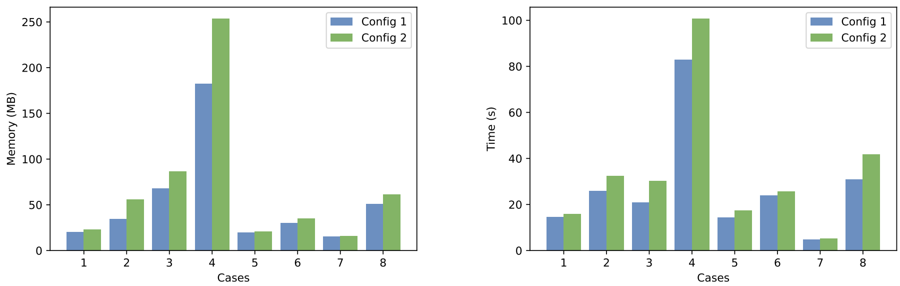
To understand the limitation of the state-of-the-art RTL concolic framework in [5], we apply [5] only on the module level. Specifically, we only consider the iob_ram module with 'Case 2' and compare the performance between our approach and [5] with respect to memory and time while increasing the unroll cycles. The experimental results are shown in Table 6. The concolic framework in [5] was able to activate the target (Case 2) only when unrolled to 50 cycles whereas our approach is able to activate the branch in 10 unroll cycles. The performance improvement of our approach compared to [5] in terms of time and memory is 24 times and 12 times, respectively. It also highlights another important aspect of the state-of-the-art concolic framework - it can activate corner cases if the design is sufficiently unrolled, which can be infeasible for industrial designs since various components in concolic testing (e.g., constraint solver) may not be able to handle such a large number of constraints. Our proposed framework solves the corner case scenarios by incrementally solving the sequence of events.

Table 6. Memory (MB) and time (s) taken to verify Case 2 at module level using [5] and our approach.

Unroll	**Concolic [5]**			**Our Approach**		
cycles	**Activated**	**Mem**	**Time**	**Activated**	**Mem**	**Time**
10	No	52.4	29.92	Yes	10.9	0.59
20	No	86.3	70.59	Yes	11.4	1.75
30	No	121.2	137.25	Yes	12.9	7.09
40	No	154.8	225.37	Yes	12.1	6.22
50	Yes	164.6	286.09	Yes	13.1	11.75

5.4 Test Generation for Different Memory Configurations

Figure 8 shows the time and memory requirements of the two configurations, presented in Table 1, in the IOb-cache design to activate 8 cases. Configuration 1, characterized by a 16-bit address width, 32-bit data width, native RAM type, write-back policy, LRU replacement policy, and a 4-way set-associative structure, consistently exhibits lower memory usage and shorter execution times compared to Configuration 2. In Configuration 2, featuring a 32-bit address width, 32-bit data width, AXI RAM type, write-through policy, PLRU_mru replacement policy, and a 4-way set-associative structure, the higher memory consumption and longer execution times can be attributed to the increased address width and the different memory access policies. The adoption of AXI RAM type and write-through policy in Configuration 2 inherently demands more memory resources and processing time. Still, the memory and the time requirements of configuration 2 remain scalable. This scalability is crucial for accommodating larger and more complex designs, making our approach suitable for complex applications.



(a) Memory comparison

(b) Time comparison

Fig. 8. Memory and time comparison for two configurations shown in Table 1 for 8 cases

6 CONCLUSION

Concolic testing provides a scalable test generation framework using an effective combination of simulation and formal methods. While it is promising for branch coverage in register-transfer level (RTL) deigns, it cannot activate complex corner cases such as hard-to-activate branches. We have developed an incremental concolic testing framework to cover such corner case scenarios in RTL models. Specifically, this paper made three important contributions. First, we show that a complex branch condition can be decomposed as a sequence of easy-to-activate events by traversing respective control and data flow graphs. Next, we map the branch coverage problem to the coverage of a sequence of events such that the test generated to activate the current event can be used as the starting point for activating the next event in the sequence. Finally, we have developed an efficient algorithm to cover the sequence of events by iterative invocation of concolic testing. Our experimental results demonstrated that our approach can be used to generate directed tests to cover complex branch targets in modern memory and processor designs, while state-of-the-art methods fail to activate them.

ACKNOWLEDGMENTS

This work was partially supported by the National Science Foundation (NSF) grant CCF-1908131.

REFERENCES

[1] Harry Foster. Wilson research group functional verification study 2020. https://blogs.sw.siemens.com/verificationhorizons/2020/11/05/part-1-the-2020-wilson-research-group-functional-verification-study/, 2020. Last accessed: 2022.

[2] Mingsong Chen, Xiaoke Qin, Heon-Mo Koo, and Prabhat Mishra. *System-level validation: high-level modeling and directed test generation techniques*. Springer, 2012.

[3] Patrice Godefroid, Nils Klarlund, and Koushik Sen. Dart: Directed automated random testing. In *Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation*, pages 213–223, 2005.

[4] Koushik Sen and Gul Agha. Cute and jcute: Concolic unit testing and explicit path model-checking tools. In *International Conference on Computer Aided Verification*, pages 419–423. Springer, 2006.

[5] Yangdi Lyu and Prabhat Mishra. Scalable concolic testing of rtl models. *IEEE Transactions on Computers*, 70(7):979–991, 2020.

[6] Alif Ahmed and Prabhat Mishra. Quebs: Qualifying event based search in concolic testing for validation of rtl models. In *ICCD*, pages 185–192. IEEE, 2017.

[7] Yangdi Lyu, Alif Ahmed, and Prabhat Mishra. Automated activation of multiple targets in rtl models using concolic testing. In *DATE*, pages 354–359. IEEE, 2019.

[8] Yatin A Manerkar, Daniel Lustig, Margaret Martonosi, and Michael Pellauer. Rtlcheck: Verifying the memory consistency of rtl designs. In *Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 463–476,

2017.
[9] Yao Hsiao, Dominic P Mulligan, Nikos Nikoleris, Gustavo Petri, and Caroline Trippel. Synthesizing formal models of hardware from rtl for efficient verification of memory model implementations. In *MICRO-54: 54th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 679–694, 2021.
[10] Yatin A Manerkar, Daniel Lustig, Margaret Martonosi, and Aarti Gupta. Pipeproof: Automated memory consistency proofs for microarchitectural specifications. In *2018 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 788–801. IEEE, 2018.
[11] Ferhat Erata, Shuwen Deng, Faisal Zaghloul, Wenjie Xiong, Onur Demir, and Jakub Szefer. Survey of approaches and techniques for security verification of computer systems. *ACM Journal on Emerging Technologies in Computing Systems*, 19(1):1–34, 2023.
[12] Hongce Zhang, Caroline Trippel, Yatin A Manerkar, Aarti Gupta, Margaret Martonosi, and Sharad Malik. Ila-mcm: integrating memory consistency models with instruction-level abstractions for heterogeneous system-on-chip verification. In *2018 Formal Methods in Computer Aided Design (FMCAD)*, pages 1–10. IEEE, 2018.
[13] Yue Xing, Aarti Gupta, and Sharad Malik. Generalizing tandem simulation: Connecting high-level and rtl simulation models. In *2022 27th Asia and South Pacific Design Automation Conference (ASP-DAC)*, pages 154–159. IEEE, 2022.
[14] Mingsong Chen, Xiaoke Qin, and Prabhat Mishra. Efficient decision ordering techniques for sat-based test generation. In *2010 Design, Automation & Test in Europe Conference & Exhibition (DATE 2010)*, pages 490–495, Dresden, Germany, 2010. IEEE.
[15] Yangdi Lyu, Xiaoke Qin, Mingsong Chen, and Prabhat Mishra. Directed test generation for validation of cache coherence protocols. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 38(1):163–176, 2018.
[16] Farimah Farahmandi and Prabhat Mishra. Automated test generation for debugging multiple bugs in arithmetic circuits. *IEEE Transactions on Computers*, 68(2):182–197, 2018.
[17] Yangdi Lyu and Prabhat Mishra. Automated test generation for activation of assertions in rtl models. In *2020 25th Asia and South Pacific Design Automation Conference (ASPDAC)*, pages 223–228. IEEE, 2020.
[18] Prabhat Mishra and Farimah Farahmandi. *Post-Silicon Validation and Debug*. Springer, 2019.
[19] Alif Ahmed, Farimah Farahmandi, and Prabhat Mishra. Directed test generation using concolic testing on rtl models. In *DATE 2018*, pages 1538–1543. IEEE, 2018.
[20] Farimah Farahmandi and Prabhat Mishra. Automated debugging of arithmetic circuits using incremental gröbner basis reduction. In *2017 IEEE International Conference on Computer Design (ICCD)*, pages 193–200. IEEE, 2017.
[21] Yangdi Lyu and Prabhat Mishra. Automated trigger activation by repeated maximal clique sampling. In *Asia and South Pacific Design Automation Conference (ASPDAC)*, pages 482–487, 2020.
[22] Mingsong Chen, Xiaoke Qin, Heon-Mo Koo, and Prabhat Mishra. *System-Level Validation: High-Level Modeling and Directed Test Generation Techniques*. Springer Publishing Company, Incorporated, Berlin, Heidelberg, 2012.
[23] Edmund M Clarke Jr. Orna grumberg, and doron a. peled. model checking. In *The MIT Press*. Springer, Berlin, Heidelberg, 1999.
[24] Mingsong Chen, Prabhat Mishra, and Dhrubajyoti Kalita. Automatic rtl test generation from systemc tlm specifications. *ACM Transactions on Embedded Computing Systems (TECS)*, 11(2):1–25, 2012.
[25] Xiaoke Qin and Prabhat Mishra. Directed test generation for validation of multicore architectures. *ACM Transactions on Design Automation of Electronic Systems (TODAES)*, 17(3):1–21, 2012.
[26] Mingsong Chen and Prabhat Mishra. Property learning techniques for efficient generation of directed tests. *IEEE Transactions on Computers*, 60(6):852–864, 2011.
[27] Mingsong Chen, Xiaoke Qin, and Prabhat Mishra. Learning-oriented property decomposition for automated generation of directed tests. *Journal of Electronic Testing*, 30(3):287–306, 2014.
[28] Heon-Mo Koo and Prabhat Mishra. Functional test generation using design and property decomposition techniques. *ACM Transactions on Embedded Computing Systems (TECS)*, 8(4):1–33, 2009.
[29] Prabhat Mishra and Nikil Dutt. Specification-driven directed test generation for validation of pipelined processors. *ACM Transactions on Design Automation of Electronic Systems (TODAES)*, 13(3):1–36, 2008.
[30] Farimah Farahmandi and Prabhat Mishra. Automated test generation for debugging arithmetic circuits. In *2016 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pages 1351–1356, Dresden, Germany, 2016. IEEE.
[31] Xiaoke Qin and Prabhat Mishra. Automated generation of directed tests for transition coverage in cache coherence protocols. In *DATE*, pages 3–8. IEEE, 2012.
[32] Mingsong Chen and Prabhat Mishra. Decision ordering based property decomposition for functional test generation. In *2011 Design, Automation & Test in Europe*, pages 1–6, Grenoble, France, 2011. IEEE.
[33] Sudhi Proch and Prabhat Mishra. Test generation for hybrid systems using clustering and learning techniques. In *2016 29th International Conference on VLSI Design and 2016 15th International Conference on Embedded Systems (VLSID)*, pages 589–590, Kolkata, India, 2016. IEEE.

[34] Xiaoke Qin, Mingsong Chen, and Prabhat Mishra. Synchronized generation of directed tests using satisfiability solving. In *2010 23rd International Conference on VLSI Design*, pages 351–356, Bangalore, India, 2010. IEEE.

[35] Thanh Nga Dang, Abhik Roychoudhury, Tulika Mitra, and Prabhat Mishra. Generating test programs to cover pipeline interactions. In *2009 46th ACM/IEEE Design Automation Conference*, pages 142–147, San Francisco, CA, USA, 2009. IEEE.

[36] Prabhat Mishra and Mingsong Chen. Efficient techniques for directed test generation using incremental satisfiability. In *2009 22nd International Conference on VLSI Design*, pages 65–70, New Delhi, India, 2009. IEEE.

[37] Heon-Mo Koo and Prabhat Mishra. Functional test generation using property decompositions for validation of pipelined processors. In *Proceedings of the Design Automation & Test in Europe Conference*, volume 1, pages 1–6, Munich, Germany, 2006. IEEE.

[38] Prabhat Mishra and Nikil Dutt. Munich, germany. In *Design, Automation and Test in Europe*, pages 678–683, Munich, Germany, 2005. IEEE.

[39] Prabhat Mishra and Nikil Dutt. Graph-based functional test program generation for pipelined processors. In *Proceedings Design, Automation and Test in Europe Conference and Exhibition*, volume 1, pages 182–187, Paris, France, 2004. IEEE.

[40] Farimah Farahmandi, Yuanwen Huang, and Prabhat Mishra. *System-on-Chip Security: Validation and Verification*. Springer Nature, 2019.

[41] Prabhat Mishra, Swarup Bhunia, and Mark Tehranipoor. *Hardware IP security and Trust*. Springer, 2017.

[42] Yangdi Lyu and Prabhat Mishra. Scalable activation of rare triggers in hardware trojans by repeated maximal clique sampling. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 2020.

[43] Zhixin Pan and Prabhat Mishra. Automated test generation for hardware trojan detection using reinforcement learning. In *Asia and South Pacific Design Automation Conference (ASPDAC)*, 2021.

[44] Alif Ahmed, Farimah Farahmandi, Yousef Iskander, and Prabhat Mishra. Scalable hardware trojan activation by interleaving concrete simulation and symbolic execution. In *2018 IEEE International Test Conference (ITC)*, pages 1–10. IEEE, 2018.

[45] Farimah Farahmandi and Prabhat Mishra. Fsm anomaly detection using formal analysis. In *2017 IEEE International Conference on Computer Design (ICCD)*, pages 313–320. IEEE, 2017.

[46] Farimah Farahmandi, Yuanwen Huang, and Prabhat Mishra. Trojan localization using symbolic algebra. In *2017 22nd Asia and South Pacific Design Automation Conference (ASPDAC)*, pages 591–597. IEEE, 2017.

[47] Jerry R Burch, Edmund M Clarke, Kenneth L McMillan, David L Dill, and Lain-Jinn Hwang. Symbolic model checking: 1020 states and beyond. *Information and computation*, 98(2):142–170, 1992.

[48] Armin Biere, Alessandro Cimatti, Edmund Clarke, and Yunshan Zhu. Symbolic model checking without bdds. In *International conference on tools and algorithms for the construction and analysis of systems*, pages 193–207. Springer, 1999.

[49] Rupak Majumdar and Koushik Sen. Hybrid concolic testing. In *29th International Conference on Software Engineering (ICSE'07)*, pages 416–426. IEEE, 2007.

[50] Shay Artzi, Julian Dolby, Frank Tip, and Marco Pistoia. Directed test generation for effective fault localization. In *Proceedings of the 19th international symposium on Software testing and analysis*, pages 49–60, 2010.

[51] Lingyi Liu and Shabha Vasudevan. Star: Generating input vectors for design validation by static analysis of rtl. In *2009 IEEE International High Level Design Validation and Test Workshop*, pages 32–37. IEEE, 2009.

[52] Lingyi Liu and Shobha Vasudevan. Scaling input stimulus generation through hybrid static and dynamic analysis of rtl. *ACM Transactions on Design Automation of Electronic Systems (TODAES)*, 20(1):1–33, 2014.

[53] Hasini Witharana, Yangdi Lyu, and Prabhat Mishra. Directed test generation for activation of security assertions in rtl models. *ACM Transactions on Design Automation of Electronic Systems (TODAES)*, 26(4):1–28, 2021.

[54] Icarus Verilog. Icarus Verilog. http://iverilog.icarus.com/, 2022. Last accessed: 2022.

[55] João V Roque, João D Lopes, Mário P Véstias, and José T de Sousa. Iob-cache: A high-performance configurable open-source cache. *Algorithms*, 14(8):218, 2021.

[56] PicoRV32. A size-optimized risc-v cpu. https://github.com/YosysHQ/picorv32. Last accessed: 2022.

[57] Bruno Dutertre. Yices 2.2. In *International Conference on Computer Aided Verification*, pages 737–744. Springer, 2014.

[58] Claire Wolf. Yosys open synthesis suite. https://yosyshq.net/yosys/. Last accessed: 2022.

[59] R. Mukherjee, D. Kroening, and T. Melham. Hardware verification using software analyzers. In *2015 IEEE Computer Society Annual Symposium on VLSI*, pages 7–12, Montpellier, France, July 2015. IEEE.