Automated Test Generation for Activation of Assertions in RTL Models

Yangdi Lyu Prabhat Mishra

Department of Computer and Information Science and Engineering
University of Florida
Gainesville, FL 32611
e-mail: {lvyangdi, prabhat}@ufl.edu

Abstract— A major challenge in assertion-based validation is how to activate the assertions to ensure that they are valid. While existing test generation using model checking is promising, it cannot generate directed tests for large designs due to state space explosion. We propose an automated and scalable mechanism to generate directed tests using a combination of symbolic execution and concrete simulation of RTL models. Experimental results show that the directed tests are able to activate assertions non-vacuously.

I. Introduction

Functional validation is one of the most important steps in System-on-Chip (SoC) design methodology [1]. Existing efforts use a combination of simulation-based validation and formal methods. Assertions are widely used in simulation-based validation to capture the intent of the specification [2]. One major challenge in assertion-based validation is to efficiently activate all assertions. Coverage of all assertions is fundamentally different from code coverage due to the vacuity problem outlined in Section III.

Directed tests are promising in activating assertions since a significantly smaller number of directed tests can achieve the same coverage goal compared to random or pseudo-random tests [3]–[5]. Simulation-based verification can handle large designs but cannot guarantee activation of assertions directly due to exponential input space complexity. In practice, designers need to manually write directed test patterns to cover many hard-to-activate assertions. As expected, manual test writing can be time consuming and error prone (requiring numerous trials and errors) - may not be feasible for large designs. While formal methods [6]–[23], such as SAT-based bounded model checking, are effective in automated generation of directed tests, these approaches expect formal specification and do not directly support Hardware Description Language (HDL) models. The extra procedure of conversion from HDL to formal specification may introduce errors. Most importantly, the complexity of real world designs usually exceeds the capacity of the model checking tools, leading to state space explosion. Concolic testing is a promising direction that combines the advantages of simulation-based validation and formal methods by effective utilization of symbolic execution and concrete simulation [24].

Concolic testing has been used extensively in software domain to cover functional events [24]–[27] as well as assertions [28]. While early work on concolic testing of Register-Transfer Level (RTL) models is promising, there are no prior efforts in activating RTL assertions using concolic testing. In this paper, we propose an automated mechanism to generate directed tests using concolic testing to activate assertions. *To the best of our knowledge, our approach is the first attempt in utilizing concolic testing for activation of RTL assertions.*

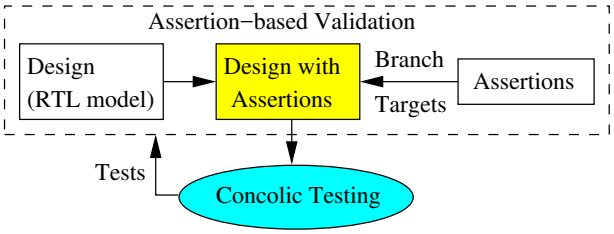

Fig. 1: Overview of our proposed methodology. Assertions can be embedded in the design or defined as separate validation goals. Our approach converts assertions to branch targets and activates them non-vacuously using concolic testing.

Our proposed methodology consists of two major steps as shown in Figure 1. The first step converts these assertions to branch statements and embed them into the design. Then, it utilizes concolic testing to generate a compact test set to efficiently cover (activate) the target branches (assertions). While formal methods try to explore all possible paths at the same time (can lead to state space explosion), concolic testing has the inherent advantage of scalability since it explores one execution path at a time. Note that the embedded branch targets are used for test generation purpose only. Once test generation is completed, these branch targets should be removed from the design (RTL model) and replaced with the original assertions. This paper makes two important contributions:

1) We map the problem of activating assertions non-vacuously to the problem of concolic testing by converting assertions to branch targets (Section IV).
2) We propose an efficient test generation method using concolic testing to cover the generated branch targets. The generated test vectors are guaranteed to activate the corresponding assertions (Section V-A).
3) To address the path explosion problem in concolic testing, we efficiently select the most profitable branches to quickly reach the target (Section V-B).

The remainder of the paper is organized as follows. We present the related work in Section II followed by problem formulation in Section III. Section IV describes the conversion from assertions to branch targets. Section V presents our test generation framework using concolic testing to activate the branch targets (assertions). Section VI presents experimental results. Finally, Section VII concludes the paper.

978-1-7281-4123-7/20/$31.00 © 2020 IEEE

II. Background and Related Work

This section describes background and existing efforts in two closely related fields.

A. Concolic Testing

Concolic testing is a directed test generation technique combining symbolic execution [29] and concrete simulation. It addresses the state explosion problem in formal methods, such as bounded model checking [30]. Concolic testing explores one path at a time by alternating one of the branches from previous simulation path until reaching the target statement. Concolic testing has been extensively explored in software domain to cover functional events [24]–[27]. These approaches utilize different path selection heuristics and optimizations to achieve specific coverage goals. Korel and Al-Yamo [28] explored concolic testing to find input that violates assertions by analyzing data dependency to guide test generation. However, these approaches are all designed for software (sequential) models, and are not suitable for hardware designs where multiple modules are running concurrently with different clock domains and interacting with each other. There are initial efforts in applying concolic testing on RTL models for test generation [31], [32]. However, there are no prior efforts in using concolic testing for activating assertions.

B. Test Generation for Hardware Assertion Coverage

Existing test generation approaches for activating hardware assertions can be broadly classified in two categories: simulation-based and formal methods. The first category uses simulation based methods [33], [34]. In transaction-level, Ferro et al. [33] proposed a framework for supervising SystemC TLM simulation of PSL temporal properties with combinatorial testing tools. In register-transfer level (RTL), Pal et al. [34] restricted that assertions are defined over the interface of a module (input and output) and proposed an approach to bias random test generation for assertion coverage. The second category uses model checking [35]–[37]. From non-deterministic finite automata (NFA), Tong et al. [35] utilized model checking to generate test for assertions with the assumption that the signals in assertions refer to the primary inputs. The above approaches have one major drawback. In order to enable test generation, they restrict the assertions to have variables of only specific types (e.g., primary inputs/outputs of modules). As a result, these approaches cannot be applied on assertions that may require complex interactions between any internal variables. Our test generation framework does not impose any restriction on assertion variables, and thereby enables test generation for activating a wide variety of assertions.

III. Problem Formulation

In this paper, **activation of assertions refers to finding counter-examples that fails the assertions non-vacuously**. Vacuity is defined in [38] as follows: if there exist a sub-formula ψ of a formula ϕ such that ψ can be replaced with arbitrary formula and does not affect the outcome of model checking, the formula ϕ is vacuous in model M. For example, in the formula $p \longrightarrow q$, it is vacuously valid if p is always false, since we can replace q with any sub-formula. We address the vacuity problem by converting the formulas into specific branch targets and applying concolic testing to activate them.

Listing 1 shows the branches that are converted from two types of assertions (immediate assertions and concurrent assertions) in Arbiter. Note that the conversions from assertions to branches are the same for these two types, except that an individual concurrently running block is needed to wrap the branches from concurrent assertions. In Listing 1, the first assertion is an immediate assertion and its corresponding branch is directly embedded in the same place as the assertion. On the other hand, the second assertion is converted into an always block that is running concurrently with all the other blocks. To find counter-examples that make the assertions fail non-vacuously, we need to generate tests to activate branch targets that are converted from the assertions.

Listing 1: An example of branch conversion in Arbiter

```
module arb(clk, rst, req1, req2, gnt1, gnt2);
input clk, rst, req1, req2;
output gnt1, gnt2;
reg state, gnt1, gnt2;
always @ (posedge clk or posedge rst)
    if (rst)
        state <= 0;
    else
        state <= gnt1;
always @ (*)
    if (state) begin
        gnt1 = req1 & ~req2;
        gnt2 = req2;
        // Assert 1: assert(req2 == gnt2)
        if (req2 != gnt2)
            // target 1
    end
    else begin
        gnt1 = req1;
        gnt2 = req2 & ~req1;
    end
// Assert 2: assert property(gnt1|->~gnt2)
always @ (*)
    if (gnt1)
        if (gnt2)
            // target 2
endmodule
```

IV. Conversion of Assertions to Branches

To generate a test to activate assertion P, we first map the assertion activation problem to branch coverage problem in concolic testing. Algorithm 1 shows our procedure to convert assertion P to blocks containing a corresponding branch target. Section V will demonstrate how to use concolic testing to generate tests to cover branch targets. In this section, we introduce the details of converting assertions to branches. In this paper, we consider assertions with logic operator, implication ($|->$) and delay ($\#\#$). Other operations are not described due to space limitation, but can be converted to branches in similar ways.

A. Simplified Abstract Syntax Tree

To understand the meaning of one assertion, we parse the assertion and build an abstract syntax tree (AST) for it. Three types of operators are selected as non-terminal for our simplified AST, i.e., logic operator, implication and delay. Others are treated as terminals. For example, if the original assertions is *assert (*$a \ \#\#7 \ b \ |-> \ \#\#[4:9] \ c$*)*, which means

Algorithm 1: Assert2Branch

```
/* Input: assertion P.                                  */
/* Output: B containing generated blocks.               */
Construct simplified AST for assertion P
Readjust AST with delay information
Empty stack S
for Post-order traversal readjusted AST do
    if current node n is an implication then
        Convert implication to logic operator
    end
    if current node n is a variable then
        Push n to S
    end
    if current node n is delay then
        Pop variable a from S
        Add delay to a
        Push the modified variable to S
    end
    if current node n is a logic operator then
        Pop all variables of its children from S
        Combine the children with its operator
        Push the result to S
    end
end
Create branch to test the variable in S
```

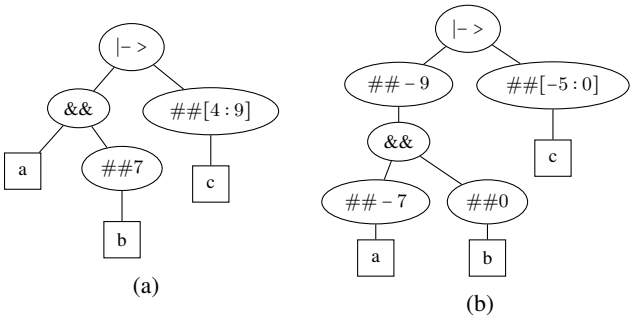

Fig. 2: (a) Simplified AST for *assert (a ##7 b |- > ##[4 : 9] c)*. Logic operator, implication and delay are non-terminal nodes (oval), and others are terminals (rectangle). (b) Readjusted AST with timing. All delays are converted to local history values.

if a is 1 in clock 0 and b is 1 in clock 7, then c must be 1 in any clock between clock 11 and clock 16. The simplified AST for this assertion is shown in Figure 2a.

B. Adjust AST with Timing

As delays represent the future events, which cannot be evaluated in the current clock cycle, we transform delays into retrieving history values. We assume that there exists a global clock counter (as shown in Listing 2), and the design remembers all the "necessary" history values. We use a[clk_cnt] to represent the history value of a in clock clk_cnt. Figure 2b shows the readjusted AST for Figure 2a. There are two things to consider:

1) Adjustment is local to its own children for each non-terminal nodes. For example, the left sub-tree in Figure 2a (a ##7 b) adjusts the delay of 7 to its left child. If we look at the whole expression, the history values of a should be at least 11 cycles ahead of c. This localization property make adjustment efficient.
2) For delay range, we adjust the longest delay to the left side and modify the range appropriately, e.g., $\#\#[4:9]$ in Figure 2a rotates the $\#\#-9$ to the left side and adjusts itself to $\#\#[-5:0]$.

Listing 2: Global clock counter

```
always @(posedge clock) begin
    clk_cnt <= clk_cnt + 1;
end
```

C. Conversion of AST to Branch Target

After we adjust AST with timing, each node is attached with non-positive delay (implicitly 0 delay). From adjusted AST, we construct branches by post-order traversal of the adjusted AST with the help of a stack S. Each part of the clause is represented by a unique variable except for the clauses which can be directly accessed. Stack S contains the visited variables that have not been combined by other clauses. Algorithm 1 shows how the target branch is generated (in italic bold text) with the help of stack S. The generated code of Figure 2b is shown in Listing 3. As shown in Algorithm 1, RTL code is generated based on the root type of each sub-tree. We consider the following three root types:

Listing 3: The branch converted from Figure 2b

```
always @(posedge clock)
begin
    // p1 = ##-7 a && b
    p1[clk_cnt] = a[clk_cnt - 7] && b[clk_cnt];
    // p2 = ##[-5:0]c
    p2[clk_cnt] = 0;
    for (i : [clk_cnt - 5, clk_cnt])
        p2[clk_cnt] = p2[clk_cnt] | c[i];
    // p3 = ##-9 p1 |-> p2
    p3[clk_cnt] = 1;
    if (p1[clk_cnt - 9])
        if (!p2[clk_cnt])
            p3[clk_cnt] = 0;
    // branch target
    if (!p3[clk_cnt])
        $display("Assertion fail");
end
```

Delay: For a single delay, we retrieve the history value of the variable, e.g., when we visit the node $\#\# - 7$ in Figure 2b, the node a is in the top of stack S. We pop a from S, and push back a[clk_cnt - 7]. A delay range represents an OR operation on all the values, e.g., $\#\#[-5:0]c$ means $c[-5]|c[-4]|...|c[0]$. Listing 3 shows the expansion of $\#\#[-5:0]c$ using for-loop and uses variable $p2$ to represent this part. When a single delay is applied, we skip generating a new variable for the clause, e.g., $\#\# - 7a$ directly utilizes the history value of a instead of generating a new variable.

Logic Operator: When the root is a logic operator, Algorithm 1 combines all its children (contains delay information) using the operator. As each child is already represented by a single variable in the stack S, we just pop all of them from S, and use a new variable to represent the combined result.

Implication: Implication, A |- > B, contains two parts: A is called the antecedent, and B is called the consequent. There are two implication operators in SVA, i.e., overlapped implication (|- >) tests consequent sequence at the clock when its antecedent sequence is activated, while nonoverlapped

implication (|=>) tests the consequent in the next clock cycle. The latter one can be converted to the previous one by adding one cycle delay to the consequent sequence. As shown in Listing 3, we convert the implication node into variable $p3$.

When we finish traversing the readjusted AST, the assertion expression is represented as a single variable in top of stack S, e.g., $p3$ in Listing 3. A branch target is created by checking the value of the final variable.

D. Complexity Analysis of Algorithm 1

For the ease of representation, we assumed that the design remembers all "necessary" values in the previous iterations. To achieve memory efficiency, the clk_cnt can be as small as the largest delay in the whole assertion, e.g., 9 for *assert(a* $\#\#7\ b\ |->\ \#\#[4:9]\ c$*)*, as a result of introducing new variables. If we look at the code in Listing 3, the impact of a[clk_cnt - 16] is already stored in p1[clk_cnt - 9]. Thus, remembering older values than the longest delay is a waste of memory. After determining the largest delay, we add a module operation to Listing 2, i.e., clk_cnt<= clk_cnt $\bmod$ (9 + 1), with an extra one to remember the current clock. Assume that b is the longest delay and n is the length of the assertion. The *memory requirement* complexity is $O(bn)$ since the memory usage of the tree structure, the stack S, and required new variables are linear to the length of assertion. The running time of Algorithm 1 is dominated by post-order traversal of the AST, compared to the AST construction and adjustment. For each node, the running time is linear to the number of children. Then, each node contributes twice to the total running time. Since the number of nodes in AST is linear to the length of the assertion, the *running time* complexity is $O(n)$.

V. Test Generation using Concolic Testing

Once the assertions are converted to branches, we apply concolic testing to generate tests to cover the generated branch targets. This section is organized as follows. First, we provide an overview of our test generation framework. Next, we briefly discuss efficient selection of alternate branches.

A. Overview

Figure 3 shows the overview of our test generation framework. As discussed in Section II-A, concolic testing combines concrete simulation and symbolic execution. In Figure 3, the left side shows the concrete simulation part, and the right side shows the symbolic execution part. To instruct symbolic execution, the concrete path needs to provide every branch it takes. Instead of changing simulator to execute symbolically in each branch and assignment, we use existing tools for simulation, and instrument the RTL design with *display* statement to show which branch the simulation has taken. For example, the instrumented first block of Listing 1 is shown in Listing 4.

Based on Algorithm 1, the assertions in RTL design are converted into branch targets in control flow graph (CFG). For every test that is generated by symbolic execution, simulation will give the concrete execution and report every branch it takes. Based on the branch information, constraints are constructed together with all the assignments inside the corresponding blocks. The most important step in concolic testing is to find the best alternative branch to flip, which will be discussed in the next section. With the selected alternative branch, new constraints are constructed, and solved by an SMT solver to generate a new test for simulation. The general idea is to efficiently explore different paths to get closer to the branch target converted from a specific assertion.

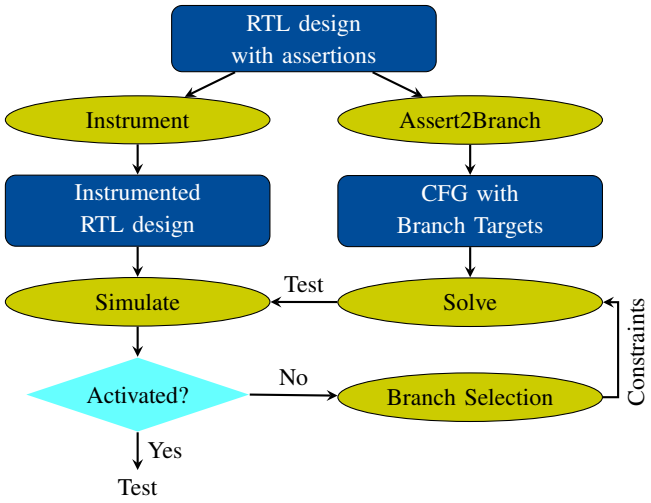

Fig. 3: Overview of our test generation framework. After converting assertions to branch targets, concolic testing is applied to generate tests to activate these branch targets (assertions).

Listing 4: Instrumented first block in Arbiter

```
always @ (posedge clk or posedge rst)
    if (rst) begin
        $display("arb2 branch 1 taken");
        state <= 0;
    end
    else begin
        $display("arb2 branch 2 taken");
        state <= gnt1;
    end
```

B. Selection of Alternate Branches in CFG

To help alternative branch selection, we first chain the relative blocks together in control flow graph. We use the second assertion in Listing 1 as an example. The branch target is controlled by the condition *gnt1* & *gnt2*. Therefore, the target block is chained to the blocks where either $gnt1$ or $gnt2$ might be assigned '1'. Similarly, since the blocks in the second CFG are controlled by the value of $state$, the blocks are chained to the blocks in the first CFG, as shown in Figure 4.

This chaining process helps alternative branch selection concentrating only on related branches. When we consider the relevance of one branch with the target, we calculate the distance from the immediate block following the alternate branch to the target. In each iteration, the most relevant and reachable branch is selected as the alternative branch to construct new constraints and generate a new test.

VI. Experiments

This section is organized as follows. First, we describe our experimental setup and an example of inserted assertions.

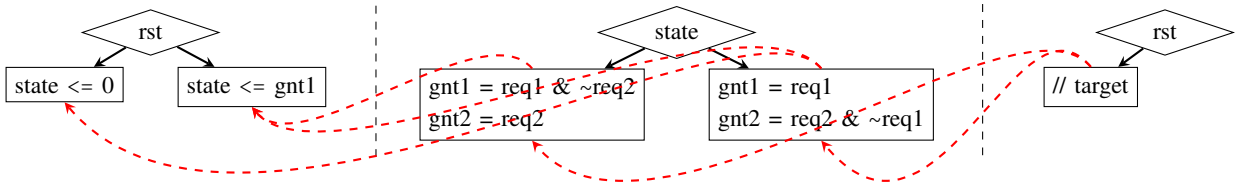

Fig. 4: Chaining of related blocks in CFGs. A block is chained to the blocks where its condition is likely to be satisfied.

Next, we present our test generation results.

A. Experimental Setup

To evaluate our test generation technique in activating assertions non-vacuously, we implemented our framework in C++ using Icarus Verilog Target API [39] with Yices [40] as the constraint solver. As shown in Section V, our framework first converted all assertions to branches and inserted them into modified designs. Next, it applied concolic testing to generate test to activate the branches. Finally, we simulated the assertion-inserted instances (before converting to branches) to validate the correctness of generated test sets. Our framework is compared with EBMC [37] to show the performance improvement. All the experiments are performed on a machine with Intel E5-2698 v4 @ 2.20GHz CPU.

B. Benchmarks and Assertions

We selected 12 benchmarks to evaluate our framework. The first three benchmarks, wb_conmax-T200, AES-T1000 and AES-T1100, are from TrustHub [41]. The remaining benchmarks are custom benchmarks of AES, named as cb_aes_n, where n is the number of rounds in AES. We varied the number of rounds to easily control the size of our benchmarks. To show the inserted assertions, we use AES-T1000 as an example. Listing 5 shows the Trojan_Trigger module from AES-T1000 benchmark [41]. The inserted assertion is $assert\ property(rst == 0)|->(trig == 0)$. In most scenarios, the extremely rare branch in Listing 5 is never executed. As a result, traditional testing approaches using millions of random tests will not activate this assertion non-vacuously. For other benchmarks, we inserted assertions in the same way.

Listing 5: Trojan_Trigger module in AES-T1000 [41]

```
module Trojan_Trigger (rst, state, trig);
  input rst;
  input [127:0] state;
  output trig;
  always @ (rst, state)
    if (rst == 1'b1) trig <= 1'b0;
    else begin
      if (state==128'h0011223344556677\
                  8899aabbccddeeff)
        trig <= 1'b1;
  end
endmodule
```

C. Test Generation Results

In this experiment, we applied our framework to generate tests for assertions. The performance comparison is shown in Table I. The number of unrolled cycles is just enough to activate the assertions. For example, the number of unrolled cycles is $n+5$ for each custom AES benchmark cb_aes_n as it requires n cycles to get results from output. As shown in Table I, our approach is significantly faster (up to 5.4x, 3.5x on average) than EBMC.

TABLE I: Performance comparison of our approach with EBMC [37] in generating tests to activate assertions.

Benchmark	EBMC [37]		Our Approach			
	Time (s)	MEM (GB)	Time (s)	Time Imp.	MEM (GB)	MEM Imp.
wb_conmax	5.42	0.74	6.86	-1.3x	0.13	5.7x
AES-T1000	116	7.99	8.91	13x	0.60	13x
AES-T1100	116	7.99	21.43	5.4x	0.69	12x
cb_aes_01	2.89	0.17	0.90	3.2x	0.06	2.9x
cb_aes_10	58.4	3.42	12.81	4.6x	0.59	5.8x
cb_aes_15	113	6.42	27.9	4.1x	0.88	7.3x
cb_aes_20	178	10.3	63.7	2.8x	1.23	8.4x
cb_aes_25	260	15.0	127	2.1x	1.58	9.5x
cb_aes_30	411	20.7	230	1.8x	1.97	11x
cb_aes_35	478	27.1	372	1.3x	2.36	12x
cb_aes_40	617	34.3	578	1.1x	2.81	12x
Average	214	12.2	132	3.5x	1.17	9.1x

Our approach is also more efficient in memory usage. As shown in Table I, our approach is up to 13x (9.1x on average) more efficient in memory usage compared to EBMC. To better visualize the relationship between the memory requirement with respect to the size of the design, we plot the memory requirement of two approaches for our custom benchmarks in Figure 5. Note that the number of lines for each custom AES benchmark is the total lines after hierarchy flattening. As we can see, the memory requirement of EBMC grows exponentially with the lines of code. It is due to the state space explosion problem of model checking. On the other hand, the memory requirement of our approach grows linearly with the lines of code since it explores one path at a time, which is linear to the code size. For the benchmark cb_aes_40 (around 1.3 million lines of code), EBMC requires over 34GB memory, while our approach only needs 2.8GB. Due to exponential memory requirement, EBMC is expected to fail for larger and more complex designs, while our approach is expected to be scalable since memory requirement increases linearly.

VII. Conclusion

Assertions are widely used in validation of hardware designs (RTL models). A major challenge in assertion-based validation is how to activate all the assertions to ensure that they are valid. While existing model checking based directed test generation is promising, it cannot generate tests for large designs due to state space explosion. We presented an automated and scalable mechanism to generate directed tests using concolic testing to activate assertions non-vacuously. Using a diverse

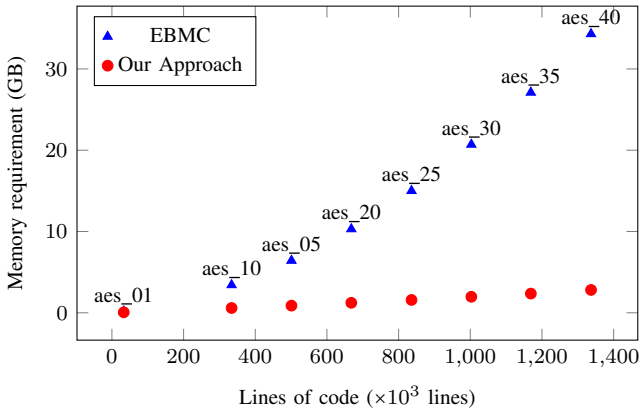

Fig. 5: Memory requirement with respect to the total lines of code in custom benchmarks of AES.

set of benchmarks, our experimental results demonstrated that our test generation approach is significantly faster (up to 5.4x, 3.5x on average) compared to state-of-the-art test generation methods. Most importantly, our approach is scalable since it has linear memory requirement, while state-of-the-art directed test generation method has exponential memory requirement.

Acknowledgments

This work was partially supported by grants from NSF (CCF-1908131) and SRC (2020-CT-2934).

References

[1] M. Chen and P. Mishra, "Functional test generation using efficient property clustering and learning techniques," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 2010.

[2] H. D. Foster, A. C. Krolnik and D. J. Lacey, *Assertion-based design*, Springer, 2004.

[3] Y. Lyu, X. Qin, M. Chen, and P. Mishra, "Directed Test Generation for Validation of Cache Coherence Protocols," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 2018.

[4] F. Farahmandi and P. Mishra, "Automated Test Generation for Debugging Multiple Bugs in Arithmetic Circuits," *IEEE Transactions on Computers (TC)*, 2019.

[5] M. Chen, X. Qin, and P. Mishra, "Efficient decision ordering techniques for SAT-based test generation," in *Design, Automation and Test in Europe (DATE)*, 2010.

[6] E. Clarke, O. Grumberg, and D. Peled, *Model Checking*, MIT Press, 1999.

[7] M. Chen, X. Qin, H. Koo, and P. Mishra, *System-Level Validation: High-Level Modeling and Directed Test Generation Techniques*, Springer, 2012.

[8] M. Chen, P. Mishra, and D. Kalita, "Automatic RTL test generation from SystemC TLM specifications," *ACM Transactions on Embedded Computing Systems (TECS)*, 2012.

[9] X. Qin and P. Mishra, "Directed Test Generation for Validation of Multicore Architectures," *ACM Transactions on Design Automation of Electronic Systems*, volume 17, no 3, article 24, 21 pages, 2012.

[10] M. Chen and P. Mishra, "Property Learning Techniques for Efficient Generation of Directed Tests," *IEEE Transactions on Computers (TC)*, 60(6), pages 852-864, June 2011.

[11] M. Chen, X. Qin, and P. Mishra, "Learning-Oriented Property Decomposition for Automated Generation of Directed Tests," *Springer Journal of Electronic Testing*, 30(3), pages 287-306, 2014.

[12] H. Koo and P. Mishra, "Functional Test Generation using Design and Property Decomposition Techniques," *ACM Transactions on Embedded Computing Systems*, volume 8, no 4, article 32, July 2009.

[13] P. Mishra and N. Dutt, "Specification-driven Directed Test Generation for Validation of Pipelined Processors," *ACM Transactions on Design Automation of Electronic Systems (TODAES)*, 2008.

[14] F. Farahmandi and P. Mishra, "Automated Test Generation for Debugging Arithmetic Circuits," in *Design Automation and Test in Europe (DATE)*, 2016.

[15] X. Qin and P. Mishra, "Automated Generation of Directed Tests for Transition Coverage in Cache Coherence Protocols," in *Design Automation and Test in Europe (DATE)*, 2012.

[16] M. Chen and P. Mishra, "Decision Ordering Based Property Decomposition for Functional Test Generation," in *Design Automation and Test in Europe (DATE)*, 2011.

[17] S. Proch and P. Mishra, "Test Generation for Hybrid Systems using Clustering and Learning Techniques," in *International Conference on VLSI Design*, 2016.

[18] X. Qin, M. Chen, and P. Mishra, "Synchronized Generation of Directed Tests using Satisfiability Solving," in *International Conference on VLSI Design*, 2010.

[19] N. Dang, A. Roychoudhury, T. Mitra, and P. Mishra, "Generating Test Programs to Cover Pipeline Interactions," in *ACM/IEEE Design Automation Conference (DAC)*, 2009.

[20] P. Mishra and M. Chen, "Efficient Techniques for Directed Test Generation using Incremental Satisfiability," in *International Conference on VLSI Design*, 2009.

[21] H. Koo and P. Mishra, "Functional Test Generation using Property Decompositions for Validation of Pipelined Processors," in *Design Automation and Test in Europe (DATE)*, 2006.

[22] P. Mishra and N. Dutt, "Functional Coverage Driven Test Generation for Validation of Pipelined Processors," in *Design Automation and Test in Europe (DATE)*, 2005.

[23] P. Mishra and N. Dutt, "Graph-based Functional Test Program Generation for Pipelined Processors," in *Design Automation and Test in Europe (DATE)*, 2004.

[24] K. Sen and G. Agha, "Cute and jcute: Concolic unit testing and explicit path model-checking tools," in *Computer Aided Verification*, 2006.

[25] P. Godefroid, N. Klarlund, and K. Sen, "Dart: Directed automated random testing," in *ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2005.

[26] C. Cadar, D. Dunbar, and D. Engler, "Klee: Unassisted and automatic generation of high-coverage tests for complex systems programs," in *USENIX Symposium on Operating Systems Design and Impl.*, 2008.

[27] V. Chipounov, V. Kuznetsov, and G. Candea, "The S2E platform: Design, implementation, and applications," *ACM Transactions on Computer Systems (TOCS)*, 2012.

[28] B. Korel and A.M. Al-Yami, "Assertion-oriented automated test data generation," in *IEEE International Conf. on Software Engineering*, 1996.

[29] R. S. Boyer, B. Elspas, and K. N. Levitt, "SELECT-a formal system for testing and debugging programs by symbolic execution," in *International Conference on Reliable Software*, 1975.

[30] A. Cimatti et al., "NuSMV 2: An opensource tool for symbolic model checking," in *Computer Aided Verification*, 2002.

[31] A. Ahmed, F. Farahmandi, and P. Mishra, "Directed test generation using concolic testing on RTL models," in *Design, Automation and Test in Europe (DATE)*, 2018.

[32] Y. Lyu, A. Ahmed, and P. Mishra, "Automated Activation of Multiple Targets in RTL Models using Concolic Testing," in *Design, Automation and Test in Europe (DATE)*, 2019.

[33] L. Ferro, L. Pierre, Y. Ledru, and L. du Bousquet, "Generation of test programs for the assertion-based verification of tlm models," in *International Design and Test Workshop*, 2008.

[34] B. Pal, A. Banerjee, A. Sinha, and P. Dasgupta, "Accelerating assertion coverage with adaptive testbenches," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems (TCAD)*, 2008.

[35] J.G. Tong, M. Boulé, and Z. Zilic, "Test compaction techniques for assertion-based test generation," *ACM Transactions on Design Automation of Electronic Systems (TODAES)*, 2013.

[36] Y. Oddos, K. Morin-Allory, D. Borrione, M. Boulé, and Z. Zilic, "Mygen: Automata-based on-line test generator for assertion-based verification," in *ACM Great Lakes Symposium on VLSI*, 2009.

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

[38] I. Beer, S. Ben-David, C. Eisner, and Y. Rodeh, "Efficient detection of vacuity in temporal model checking," in *FSMD*, 2001.

[39] S. Williams, Icarus verilog, http://iverilog.icarus.com.

[40] B. Dutertre, "Yices 2.2," In *CAV*, 2014.

[41] Trusthub, https://www.trust-hub.org/.