October 2011

UILU-ENG-11-2209
CRHC-11-07

EVALUATING CODE COVERAGE OF ASSERTIONS BY STATIC ANALYSIS OF RTL

Viraj Athavale, Sam Hertz, and Shobha Vasudevan

Coordinated Science Laboratory
1308 West Main Street, Urbana, IL 61801
University of Illinois at Urbana-Champaign

REPORT DOCUMENTATION PAGE

Form Approved
OMB NO. 0704-0188

1. AGENCY USE ONLY *(Leave blank)*	2. REPORT DATE	3. REPORT TYPE AND DATES COVERED
	October 2011	

4. TITLE AND SUBTITLE
Evaluating Code Coverage of Assertions by Static Analysis of RTL

5. FUNDING NUMBERS
C5505, Qualcomm 900038673

6. AUTHOR(S)
Viraj Athavale, Sam Hertz, and Shobha Vasudevan

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
Coordinated Science Laboratory, University of Illinois at Urbana-Champaign, 1308 W. Main Street, Urbana, Illinois, 61801-2307

8. PERFORMING RGANIZATION REPORT NUMBER
UILU-ENG-11-2209
CRHC-11-07

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)
Qualcomm Inc.
5775 Morehouse Drive
San Diego, CA 92120-1714

10. SPONSORING/MONITORING AGENCY REPORT NUMBER

11. SUPPLEMENTARY NOTES

12a. DISTRIBUTION/AVAILABILITY STATEMENT

Approved for public release; distribution unlimited.

12b. DISTRIBUTION CODE

13. ABSTRACT *(Maximum 200 words)*

Assertions are critical in pre-silicon hardware verification to ensure expected design behavior. While Register Transfer Level (RTL) code coverage can provide a metric for assertion quality, few methods to report it currently exist. We introduce two practical and effective code coverage metrics for assertions - one inspired by test suite code coverage as reported by RTL simulators and the other by assertion correctness in the context of formal verification. We present an algorithm to compute coverage with respect to assertion correctness, by analyzing the Control Flow Graph (CFG) constructed from the RTL source code. Our technique reports coverage in terms of lines of RTL source code which is easier to interpret and can help in efficiently enhancing an assertion suite. We apply our technique to an open source USB 2.0 design and show that our coverage evaluation is efficient and scalable.

14. SUBJECT TERMS
1. Code coverage; 2. Assertion; 3. Coverage; 4. Verification; 5. Formal verification; 6. Static analysis

15. NUMBER OF PAGES
6

16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
UNCLASSIFIED	UNCLASSIFIED	UNCLASSIFIED	UL

Evaluating Code Coverage of Assertions by Static Analysis of RTL

Viraj Athavale, Sam Hertz and Shobha Vasudevan
University of Illinois at Urbana-Champaign

***Abstract*—Assertions are critical in pre-silicon hardware verification to ensure expected design behavior. While Register Transfer Level (RTL) code coverage can provide a metric for assertion quality, few methods to report it currently exist. We introduce two practical and effective code coverage metrics for assertions - one inspired by test suite code coverage as reported by RTL simulators and the other by assertion correctness in the context of formal verification. We present an algorithm to compute coverage with respect to assertion correctness, by analyzing the Control Flow Graph (CFG) constructed from the RTL source code. Our technique reports coverage in terms of lines of RTL source code which is easier to interpret and can help in efficiently enhancing an assertion suite. We apply our technique to an open source USB 2.0 design and show that our coverage evaluation is efficient and scalable.**

I. Introduction

Assertions represent desirable properties that a hardware design should satisfy. *Assertion based verification* [1], which involves the use of assertions in pre-Silicon formal verification as well as simulation of Register Transfer Level (RTL) designs, has steadily gained popularity in recent years [2]. Assertion based verification checks if a design complies with a set of properties.

Coverage achieved during verification is the single most important parameter in determining the quality of verification results. In conventional *simulation based verification*, coverage of a set of *tests*[1] or a *test suite* is measured using various metrics such as code coverage, toggle coverage, FSM coverage [3]. *Code coverage* measures the fraction of statements in the RTL source code executed or *covered* while simulating the test suite. Since code coverage can be easily related to the RTL code and reporting it adds little overhead to simulation, it is the most popular coverage metric.

In the context of formal verification, the fraction of design states covered by an assertion is typically used as a coverage metric [4]–[7]. Coverage computation is done through injection of mutations or faults in the RTL source code in [4], [6]–[9], while [5] uses a tableau based method.

Although RTL code coverage is an important coverage metric for verification, no such metric for assertions is widely used by standard tools. In this work, we address this problem by defining and computing code coverage metrics for an assertion.

Assertions are primarily used in two fundamentally different contexts viz. formal verification and simulation. Depending on the context, the definition of coverage of an assertion needs to change. Since the simulation process is inherently not exhaustive, we base the definition of coverage of an assertion on the coverage of a test that *triggers* the assertion. In formal verification context, we expand the definition to span all possible paths in the design and consider all RTL statements whose incorrect execution can cause an assertion to fail.

In [4]–[7], coverage of assertions is reported in terms of design states covered. Coverage computation in this case typically depend on the analysis of the state transition graph of the design or its variants. However, in this work we do not construct state transition graph from the RTL design, but instead perform an analysis of the Verilog Hardware Description Language (HDL) source code for the design considered as a *Verilog program* as in [10], [11].

Coverage computation consists of an execution phase and a coverage extraction phase. In the execution phase, assuming the antecedent of the assertion holds, the CFG is used to explore all possible executions of the RTL design over number of cycles spanned by the assertion. During the CFG traversal, important dependency information between statements is stored in the form of *triggers*. These triggers are used in the coverage extraction phase to find statements covered by the assertion.

Our coverage definition and computation technique has key merits over the state space based methods. Firstly, the constructed CFG is linear in the size of RTL source code and hence cost of building a CFG is far less than a state transition graph. Therefore, any manipulations of the CFG are also more scalable. Secondly, coverage reported in terms of lines of RTL source code is closer to the designer and easier to interpret. On the other hand, state space coverage is not easily translatable to source code. Lastly, in practical verification environments, bridging coverage holes in an assertion suite can be easier when coverage information is available in the form of lines of code. Since verification is a resource and time intensive process, it is valuable to make coverage information easy to understand and use. In addition, our technique can help in quickly determining how modifications to an assertion suite can affect coverage.

We evaluate our coverage computation technique using a USB 2.0 protocol design from [12]. We inject mutations in the covered lines and see if the assertion fails formal verification in the mutated RTL design. These experiments help show that the technique is scalable and efficient and correctly computes coverage according to the proposed definition.

[1]Here we refer to pre-silicon verification stimuli and not manufacturing tests.

Recent approaches [13], [14] attempt to relate coverage metrics from formal and simulation based verification. For each of the simulation based coverage metrics, [13] presents a corresponding metric suitable for assertions in formal verification. To compute coverage of an assertion, statements in the code are removed one at a time and the assertion is checked for vacuity in the mutant design. Although a pioneering approach to code coverage of assertions, it becomes intractable as size of the RTL source code increases. In contrast, our technique does not require mutating every line and since we analyze the CFG, the technique naturally scales.

The main contributions of this paper are as follows.

- We propose two code coverage metrics, applicable in each of the two use cases of assertions- simulation and formal verification.
- We present an efficient technique to compute code coverage of assertions by statically analyzing the RTL source code.
- Our technique presents coverage in terms of lines of RTL code, which is easier to interpret for the designer as compared to state space coverage.
- Our technique can also facilitate identification of coverage holes and consequently enhance the assertion suite.

II. Verilog Code as a Control Flow Graph

In order to facilitate analysis of the Verilog RTL source code, we represent it as a *Control Flow Graph (CFG)*. Each statement in the code is mapped to a node in the CFG. A Verilog RTL design consists of a set of concurrently running `always` processes and `assign` statements. The CFG for the entire design consists of the union of CFGs for each of these processes. The CFG thus completely captures the structure of the RTL design. It is similar to a Process Dependence Graph described in [10].

Each node in the CFG stores the number of the line in the RTL code it corresponds to, as well as an expression for the RTL statement at that line. For RTL statements spanning multiple lines, the line number for each of them is recorded.

Nodes in the CFG are classified into assignment nodes and decision nodes. An *assignment node* represents a blocking or non-blocking assignment in the Verilog code. A *decision node* represents conditional statements including `if` and `case` and `always @ ()` statements.

Edges in the CFG represent the control flow between RTL statements. Each assignment node has one outgoing edge that points to the node corresponding to the next line in the RTL code. Each decision node has two outgoing edges $left$ and $right$ that point to the RTL statements executed if the corresponding condition is true or false respectively.

CFG node n_1 is a successor of another node n_2 if there exists a path from n_2 to n_1 going through the $left/right$ outgoing edges of intermediate nodes. We define two assignment nodes n_1 and n_2 as *coincident* if there exists a path between n_1 and n_2 only containing assignment nodes. Essentially during execution, n_1 must be executed given that n_2 is executed and vice versa.

The CFG is a purely syntactic representation of the RTL code. For effective coverage estimation of assertions, we need to track dependencies between variables. The CFG is therefore extended with additional data flow information. The resulting *extended CFG* is similar to the System Dependence Graph (SDG) for VHDL introduced in [10] where the additional dependencies are represented as *flow edges*. However, the data flow information we store is simpler and mainly targeted towards the coverage estimation algorithm described later, in Section IV.

We store a list of variables used in the RTL code. the variables are classified into *inputs, outputs, internals* and *parameters*. The dependence information for a variable v is recorded in the form of *initial assignment*, *decisions* and *assignments*. Initial assignment for v stores a pointer to the CFG node where the variable is assigned its initial value and is only valid when v is an *internal* or *output* variable. *Assignments* for v are assignment nodes containing statements that assign to v, while $decisions$ for v are decision nodes which use v in the corresponding condition in the RTL.

Example 1. Fig. 1 illustrates the terms defined above with the help of an example Verilog RTL code. The CFG for the module consists of the CFGs for the two processes viz. the continuous assignment on line 1 and the always process starting at line 2.

It can be seen that each statement in the RTL code is mapped to a node in the CFG. Nodes corresponding to `if (rst)`, `case (s)` and `always @ (posedge clk)` are decision nodes. Each of these have corresponding `end` nodes which are not shown for clarity. The remaining statements map to assignment nodes.

The data flow information in the form of variable dependencies which augments the CFG is also shown in Fig. 1. Again for clarity, only the information for internal variable `s` is shown. The node corresponding to statement `s <= 0` on line 4 forms *initial assignment* for variable `s` (dotted line). Variable `s` is used in the decision corresponding to `case (s)` (dashed line) and assignments on lines 10 and 14 (solid lines).

III. Defining Code Coverage of an Assertion

In this section, we define the two code coverage metrics for an assertion. We start by defining some relevant terms.

A *proposition* is a variable-value pair (v, val), where v is a variable in the RTL source code. We consider assertions of the form $ant \Rightarrow con$, where the antecedent ant is a conjunction of propositions and con is a single proposition. We also consider *temporal* assertions, which are assertions spanning multiple clock cycles. Assertions are represented in Linear Temporal Logic (LTL) [15] format.

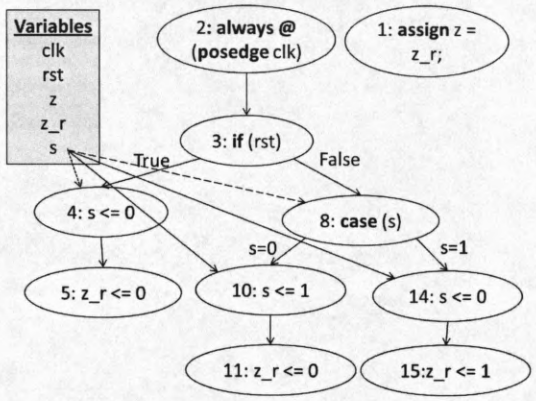
An assertion is said to *trigger* when its antecedent becomes true.

We call the set of lines reported as covered by a according to our definition the *result set* of a.

Definition 1: (Simulation based coverage) A statement s in the RTL source code is said to be covered by an assertion a if (1) s must be executed for a to trigger, (2) s must be executed

```
1. assign z = z_r;
2. always @ (posedge clk)
3.    if (rst) begin
4.       s <= 0;
5.       z_r <= 0;
6.    end
7.    else
8.       case (s)
9.          0: begin
10.              s <= 1;
11.              z_r <= 0;
12.           end
13.         1: begin
14.              s <= 0;
15.              z_r <= 1;
16.           end
17.      endcase
```

(a)



(b)

Fig. 1: **(a) Example Verilog RTL code for a with inputs `clk`, `rst`, output `z` and internal variables `s`, `z_r`. (b) Control flow graph for the example RTL design augmented with variable dependency information in the form of *assignments* (solid lines) and *decisions* (dashed line) and *initial assignment* (dotted line) for variables. For clarity, nodes containing only keywords like `begin`, `end` etc. are not shown. Also dependency information is shown only for the internal variable `s`.**

due to the triggering of a or (3) s corresponds to a CFG node coincident with a node belonging to (1) or (2).

Note that we include only those statements which are *always* executed, whenever a given assertion is triggered. As a result, if a test triggers a, the result set of a is a subset of the set of lines covered by the test. If multiple tests trigger the same assertion a, the result set is an intersection of the sets of lines covered by each of the tests.

For the RTL design from Example 1, consider the following assertion:

$a : \neg rst \wedge s \Rightarrow X(z)$.

The assertion a triggers when the antecedent $\neg rst \wedge s$ becomes true.

It can be concluded from Fig. 1 that statements on lines 3, 8, 9 and 10 must be executed prior to variable `s` getting the value 1. Apart from these, the statement `11: z_r <= 0;` is also executed whenever line 10 is executed and therefore included in the simulation based coverage.

Lines 3, 8, 13, 14, and 15 must be executed due to the triggering of a.

Lastly, continuous assignments such as `1: assign z = z_r;` are always executed and therefore covered by a. Fig. 2(a) shows the CFG nodes covered by a in this case.

In general, the RTL statements covered by a under this definition can be classified into following categories, also given by Definition 1:

1) *Backward cone*: This includes statements that must be executed to make the antecedent of a true (Lines 3, 8, 9, 10 from Example 1).
2) *Forward cone*: This includes statements that get executed due to the triggering of a (Lines 3, 8, 13, 14, 15 from Example 1).
3) *Dependent cone*: This includes statements that do not belong to the above two categories; but get executed whenever the statements belonging to the above categories are executed (Lines 1, 11 from Example 1).

Definition 2: (Correctness based coverage) A statement s in the RTL source code is said to be covered by assertion a if an error in s can cause a to fail during formal verification.

In this context an *error* in an RTL statement is a logical bug such as incorrect value assigned to a variable.

Since this definition depends on the correctness of the assertion, the number of cycles spanned by the assertion as well as the consequent are relevant to coverage.

In [13], a mutation based definition of code coverage of an assertion is presented. However, the mutation considered involves removing the RTL statement and coverage is computed by checking if the assertion becomes vacuous in the mutant RTL design. We consider logical bugs as mutations and compute coverage through analysis of the CFG for the RTL source code as described in Section IV.

Consider again the assertion a above for the RTL code in Example 1. Lines 1, 3, 8, 13, and 15 are included in the result set of a according to the correctness based definition.

Fig. 2(b) shows the CFG nodes covered by a according to this definition.

Given that the antecedent $\neg rst \wedge s$ holds, an error in one of these statements can make the consequent false and therefore make the assertion fail. For example, if the antecedent holds and either `z_r` or `z` is not assigned to the correct value, it will make the assertion fail.

A comparison between the result sets of a according to the two definitions show that lines 9, 10 and 14 are included in simulation based coverage but not in correctness based coverage. Lines 9 and 10 fall in this category because they are executed prior to the antecedent becoming true and hence irrelevant to correctness of a. Line 14 which is included in simulation based coverage also cannot affect the correctness of a and therefore not included in correctness based coverage.

IV. Correctness Based Coverage Computation

This section describes our algorithm to extract the set of RTL statements covered by a given assertion, according to the

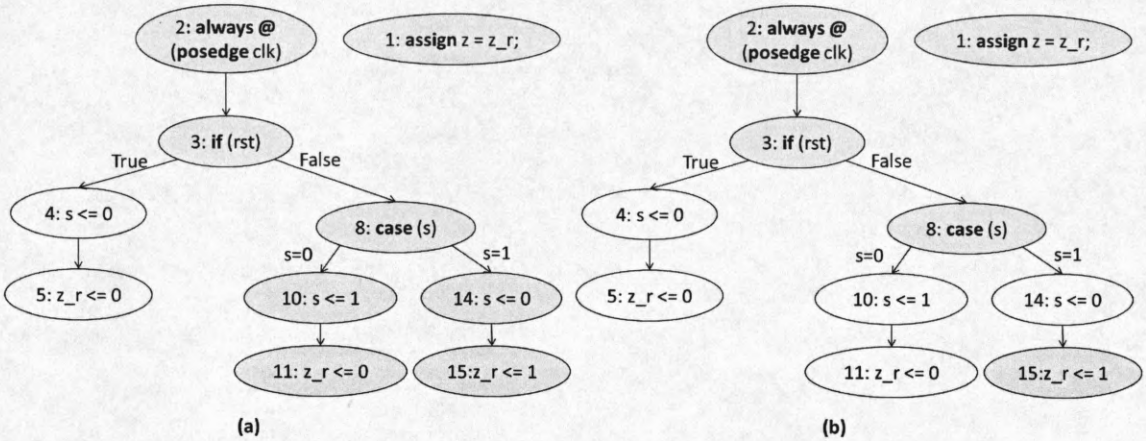


Fig. 2: **CFG nodes covered by a: (a) according to the simulation based coverage definition (b) according to the correctness based coverage definition.**

correctness based coverage definition. It consists of two phases viz. the *execution* phase and *coverage extraction* phase.

Consider an assertion $a : ant \Rightarrow con$ for RTL module M, that spans k cycles. Let V be the set of variables in M. For a k cycle assertion, con is of the form $XX \ldots X(ktimes)(v_{out})$ or $XX \ldots X(ktimes)(\neg v_{out})$, where $v_{out} \in V$. Our goal is to find C_a, the result set of a according to the correctness based coverage definition.

A. Execution phase

In this phase, we execute the RTL for k cycles starting with the information in a (Procedure 1). In this process, we record following information:

- We construct $\|V\| \times k$ tables of values of variables in k cycles. In particular the entry $B[v, i]$ in a value table B contains the value of variable v in cycle i. In other words, a value table is a table of propositions (variable-value pairs) in each of the k cycles. Multiple value tables correspond to different possible values of conditions corresponding to the decision nodes encountered during execution that cannot be evaluated to true or false.
- Apart from value tables, we also record triggers for each decision or assignment nodes that are visited. Triggers for a decision node are assignment nodes that make the decision true. Triggers for an assignment node include other assignment nodes which affect it and decisions nodes on which it depends.

When an assignment node $n : (v <= rhs)$ is encountered during execution, The value table B is updated with the value rhs if it is a constant. If it is not a constant, assignments to rhs encountered thus far are added to the set of triggers of n. the decision nodes on which this assignment depends are also added to the set of triggers of n. Essentially, these are the decision nodes that lie on the path in the CFG from n to a top level node.

When a decision node n is encountered, we attempt to evaluate the condition in n using the values available in B. If it evaluates to true, we update its triggers and take the true branch in the CFG. If it evaluates to false, we take the false branch without updating the triggers. If the decision is unknown due to a lack of sufficient data in the value table, we split B into B_{left} and B_{right} corresponding to the true and false evaluations of n respectively. As a result, at the end of execution phase, we obtain a tree of value tables rooted at B_{top}.

B. Coverage extraction phase

In the coverage extraction phase (Procedure 2), we look at each leaf B of the tree of value tables in turn. If v_{out} is assigned in cycle k in B and its value agrees with con, we have found a possible execution starting from ant that makes con true. We then include RTL lines for all nodes visited during this execution in C_a. We use the triggers recorded during the execution phase to traverse backwards recursively and obtain such nodes. This is implemented by the $BackwardTraversal()$ procedure on line 7. Pseudocode for that procedure is not shown due to space constraints.

V. Case Study: A USB 2.0 Design

We demonstrate our correctness based coverage computation technique on a USB 2.0 protocol design from [12]. We consider the packet assembler (usbf_pa), the packet disassembler (usbf_pd) and the protocol engine (usbf_pe) modules which constitute the core of USB protocol. All experiments were performed on a machine with 2.93 GHz Intel Core i3 with 4 GB RAM.

For each of the three modules considered from the USB design, we manually wrote two assertions and formally verified them against the RTL design. For each assertion, we extracted the set of lines covered according to the correctness based definition. Covered lines containing assignments and `if` statements were mutated and the assertion was run through the formal verifier again along with the mutated designs. Mutations to assignments involved changing the right hand side of the assignment to another valid value. Mutations in the `if` statements involved flipping bits in the condition.

Procedure 1 Construct a tree of value tables rooted at B_{top} using assertion a and record triggers

ConstructValueTable (G, a)

Input: Extended CFG for M (G), assertion a
Output: Tree of value tables rooted at B_{top}, triggers recorded in G

$Initialize(B_{top}, a)$ {Initialize root table of values (B_{top}) using a}
for $cycle = 1 \rightarrow k$ **do**
  for all Leaf tables B in tree rooted at B_{top} **do**
    for all Processes P in M **do**
      $n \leftarrow top(P)$
      while $n \neq NULL$ **do**
        if $IsAssignmentNode(n)$ **then**
          Let $n : (v <= rhs)$
          if $IsConstant(rhs)$ **then**
            {Update the value table}
            $B[v, cycle] \leftarrow rhs$
          end if
          $UpdateTriggers(n)$
          $n \leftarrow left(n)$
        else
          {Decision node}
          if $EvaluateDecision(n, B) = true$ **then**
            $UpdateTriggers(n)$
            $n \leftarrow left(n)$
          else if $EvaluateDecision(n, B) = false$ **then**
            $n \leftarrow right(n)$
          else
            {Unknown decision}
            $(B_{left}, B_{right}) \leftarrow split(B)$
          end if
        end if
      end while
    end for
  end for
end for

Procedure 2 Extract correctness based coverage from value table tree and triggers

ExtractCoverage (G, B_{top}, a)

Input: Extended CFG for M (G), tree of value tables rooted at B_{top}, assertion $a : ant \Rightarrow con$
Output: Set of lines C_a covered by a according to correctness based coverage definition

$C_a \leftarrow \phi$
Let $p : (v_{out}, val) \leftarrow con$
for all Leaf tables B in tree rooted at B_{top} **do**
  if $B[v_{out}, k] = val$ **then**
    {Value of v_{out} exists and matches the consequent of a}
    for all Triggers t of p **do**
      $C_a \leftarrow C_a \cup BackwardTraversal(t)$
    end for
  end if
end for

Table I summarizes the results for all assertions considered in this experiment. We show the number of lines covered and number of mutations injected and detected for each assertion. Comments, blank lines and lines with variable/port declarations etc are not included while counting covered as well as total lines of code.

Firstly, it can be seen that only about 5% of total lines were reported as covered by each assertion, which shows that our technique can effectively find the lines truly relevant to the correctness of an assertion. Secondly, mutations injected in almost all the covered lines made the corresponding assertion fail formal verification. The exceptions in case of a_2 and a_4 can be attributed to masking of the mutation as explained later in detail for a_2.

We now describe the results for assertion a_2 for the USB packet assembler module in further detail. Fig. 3 shows the relevant lines in the RTL code for the module, where lines covered by a_2 are underlined. The lines shown constitute a state machine with 5 states: `IDLE`, `WAIT`, `DATA`, `CRC1`, `CRC2`.

```
1.  always @(posedge clk)
2.    if(!rst)   state <= IDLE;
3.    else       state <= next_state;

4.  always @(state or send_data or tx_ready or
    tx_valid_r or send_zero_length_r)
5.    begin
6.       //assignments
7.       case(state)
8.       //other cases: IDLE, DATA, WAIT, CRC1
9.         CRC2:
10.        begin
11.              //assignments
12.              if(tx_ready)
13.                 begin
14.                       next_state = IDLE;
15.                 end
16.              else
17.                 begin
18.                       last = 1'b1;
19.                 end
20.        end
21.      endcase
```

Fig. 3: **Lines of RTL code from USB packet assembler module relevant to assertion a_2. Lines reported as covered by a_2 by our technique are underlined.**

Each covered line containing an assignment or `if` condition was mutated one at a time, as shown in Table II.

We found that a_2 failed formal verification in RTL designs with mutations in lines 3, 12 and 14. For instance, consider the mutation to the assignment in line 3 (`state <= next_state`). In this case the value of `next_state` variable was `IDLE` which should have been the value of `state` in the next cycle. However, since the assignment in line 3 was mutated, `state` did not get the its correct value which made a_2 fail formal verification in the mutated design.

Mutation in line 2 shown in Table II did not cause a_2 to fail. In this case, `next_state` had the value `IDLE` which is also the value of `state` on reset. Therefore `state` was assigned value `IDLE` in both branches of the `if` statement.

Module	Number of lines	Assertions	Covered lines	Detected mutations/ mutated lines
Packet assembler	182	$a_1 : rst \wedge send\_zero\_length\_r \wedge send\_data \wedge (state = IDLE) \Rightarrow XX(state = DATA)$	14	5/5
		$a_2 : rst \wedge tx\_ready \wedge (state = CRC2) \Rightarrow X(state = IDLE)$	11	3/4
Packet disassembler	183	$a_3 : rst \wedge \neg pid\_ACK \wedge \neg pid\_TOKEN \wedge pid\_DATA \wedge rx\_valid \wedge rx\_active \wedge \neg rx\_err \wedge state = ACTIVE \Rightarrow X(state = DATA)$	16	6/6
		$a_4 : rst \wedge \neg rx\_active \wedge (state = DATA) \Rightarrow X(state = IDLE)$	12	4/5
Protocol engine	415	$a_5 : rst \wedge \neg match \wedge match\_r \wedge \neg ep\_disabled \wedge \neg pid\_SOF \wedge \neg ep\_stall \wedge \neg buf0\_na \wedge \neg buf1\_na \wedge \neg no\_buf0\_dma \wedge \neg pid\_PING \wedge \neg IN\_ep \wedge \neg CTRL\_ep \wedge OUT\_ep \wedge (state = IDLE) \Rightarrow X(state = OUT)$	30	10/10
		$a_6 : rst \wedge \neg match \wedge \neg tx\_data\_to \wedge \neg crc16\_err \wedge \neg abort \wedge rx\_data\_done \wedge txfr\_iso \wedge (state = OUT) \Rightarrow X(state = UPDATEW)$	22	7/7

TABLE I: **Summary of mutation based experiments to evaluate correctness based coverage technique. For each module, number of lines of actual code and two assertions written are shown. Comments, blank lines and lines with variable/port declarations etc are not included in actual code. For each assertion, number of lines covered and number of mutations detected/total mutations are shown.**

In other words, the mutation was *masked* by the logic in the RTL design and hence had no effect on the correctness of a_2.

Line	Original RTL line	Mutated line
2	`if(!rst)`	`if(rst)`
3	`state <= next_state`	`state <= WAIT`
12	`if(tx_ready)`	`if(!tx_ready)`
14	`next_state = IDLE`	`next_state = CRC2`

TABLE II: **Mutations injected in lines covered by assertion a_2 for the USB packet assembler module. Only assignments and `if` conditions are mutated.**

We now show how runtime and memory cost of our algorithm depend on the size of the RTL source code and the number of cycles spanned by the assertion. Table III shows the time and memory costs for the three USB modules considered. The protocol engine module, being the largest one, requires the most resources.

In Table IV, we show the variation of runtime and memory requirements with length of the assertion under consideration. It can be seen that although runtime of our technique does not change considerably with number of cycles spanned by an assertion, memory increase is significant. This can be attributed to the increasing size of value tables as number of cycles increase.

Module	Size	Time (ms)	Memory (MB)
usbf_pa	182	16	1.31
usbf_pd	183	22	1.63
usbf_pe	415	46	2.79

TABLE III: **Time and memory costs to compute coverage of an assertion for the three USB modules. The values are averaged over 5 assertions spanning 1-5 cycles.**

Assertion length (cycles)	Time (ms)	Memory (MB)
1	44	1.22
2	45	1.74
3	47	2.62
4	46	3.59
5	48	4.75

TABLE IV: **Increase in time and memory costs to compute coverage of an assertion USB protocol engine module with increasing number of cycles spanned by assertion.**

VI. Conclusion

In this work, we propose code coverage metrics for assertions, both in the context of formal verification and simulation. Code coverage, being a widely used coverage metric for test suite coverage, can evaluate the quality of assertions in practical verification environments. We also present an algorithm to compute correctness based coverage which is entirely based on static analysis of RTL source code.

References

[1] H. D. Foster, A. C. Krolnik, and D. J. Lacey, *Assertion-Based Design*, 2010.

[2] A. Gupta, "Assertion-based verification turns the corner," *IEEE Des. Test*, vol. 19, July 2002.

[3] S. Tasiran and K. Keutzer, "Coverage metrics for functional validation of hardware designs," *IEEE Design and Test of Computers*, vol. 18, 2001.

[4] Y. Hoskote, T. Kam, P.-H. Ho, and X. Zhao, "Coverage estimation for symbolic model checking," in *Proc. of DAC '99*.

[5] S. Katz, O. Grumberg, and D. Geist, ""Have I written enough properties?" - a method of comparison between specification and implementation," in *Proc. of CHARME '99*.

[6] H. Chockler, O. Kupferman, and M. Y. Vardi, "Coverage metrics for temporal logic model checking," *Form. Methods Syst. Des.*, vol. 28, May 2006.

[7] H. Chockler, O. Kupferman, R. P. Kurshan, and M. Y. Vardi, "A practical approach to coverage in model checking," in *Proc. of CAV '01*.

[8] G. Fey and R. Drechsler, "Sat-based calculation of source code coverage for bmc," in *GI/ITG/GMM-Workshop*, 2006.

[9] A. Fedeli, F. Fummi, and G. Pravadelli, "Properties incompleteness evaluation by functional verification," *IEEE Trans. Comput.*, vol. 56, April 2007.

[10] E. M. Clarke, M. Fujita, S. P. Rajan, T. W. Reps, S. Shankar, and T. Teitelbaum, "Program slicing for vhdl," *Int. J. Softw. Tools Technol. Transf.*, vol. 4, no. 1, 2002.

[11] S. Vasudevan, E. A. Emerson, and J. A. Abraham, "Improved verification of hardware designs through antecedent conditioned slicing," *Int. J. Softw. Tools Technol. Transf.*, vol. 9, February 2007.

[12] "Opencores," http://www.opencores.org.

[13] H. Chockler, O. Kupferman, and M. Vardi, "Coverage metrics for formal verification," *Int. J. Softw. Tools Technol. Transf.*, vol. 8, August 2006.

[14] A. Hazra, A. Banerjee, S. Mitra, P. Dasgupta, P. P. Chakrabarti, and C. R. Mohan, "Cohesive coverage management for simulation and formal property verification," in *Proc. of ISVLSI '08*.

[15] A. Pnueli, "The temporal logic of programs," *Foundations of Computer Science, Annual IEEE Symposium on*, vol. 0, 1977.