The lights in the Tunnel: Coverage Analysis for Formal Verification

Xiushan Feng*, Abhishek Muchandikar+, Xiaolin Chen+

*Oracle Labs; +Synopsys

*Austin, TX, USA; +Mountain View, CA, USA

*http://labs.oracle.com; +http://www.synopsys.com/

ABSTRACT

As formal verification engineers, the authors always face challenges to know the current status of test benches. Many questions need to be answered at certain stages of a project. E.g., do we need more assertions? Did you over-constrain inputs that drop an important design scenario? Are proof bounds for bounded proofs good enough to catch potential design bugs? For the properties that are fully proven, do they cover the design logic that were intended to cover? These four most-asked questions don't have answers without extracting information from formal engines, which is not feasible for general users. However, like coverages from simulation-based verification, formal verification coverages can be defined and used as matrices to measure formal verification progress and completeness.

In this paper, the authors will introduce formal verification coverage models and their usages by real-life examples. The four most-asked questions finally have reasonable and acceptable answers supported by matrices.

Table of Contents

Table of Figures

Table of Tables

Before introducing formal verification coverage models, we will first provide background of formal verification, concept of formal verification closure, and challenges to it.

1. Background: Formal Verification

The term of "formal verification" we use in this paper is limited to formal property verification (or model checking) for circuit designs. It is a process of proving or disproving the correctness of behaviors of a circuit using mathematical methods. Behaviors of a circuit are defined as assertions and the proving process exhaustively explores the state space of the circuits under given input assumptions.

In reality, a circuit is given as a RTL design. Circuit behaviors are written as assertions. There are four types of assertions.

- assert: a property that is under verifying. A violation of an "assert" property will trigger a failing in both simulation and formal verification.
- assume: a property that formal tools use as an assumption. It is used to constrain inputs of the circuit. Inside RTL simulation, "assume" properties will be used the same as "assert" for simulation tools to check for pass or fail.
- cover: a property that formal tools monitor for coverage. A coverage hit is reported as covered. Similar definition applies to both formal and simulation tools.
- restrict: a property that formal tools use an assumption. It works the same as an "assume" property for formal tools. Simulation tools will ignore "restrict" properties.

Formal verification is assertion-based verification (ABV). A large number of assertions are written for a circuit under formal verification. Quality of formal verification highly relies on the quality of assertions.

Because formal verification process explores circuit space exhaustively, it may not be able to have a conclusive proof for large designs due to runtime or memory limits. If formal tools cannot find a trace to violate an "assert" property within certain cycles (e.g., N), it will report that the assertion is bounded proven with a proof bound N. In order to increase the proof bound or have a conclusive full proof, formal verification engineers have to spend a lot of efforts. The process to having a full proof is called convergence process. For some important assertions, a convergence may be required but cannot be guaranteed to achieve.

At the end of formal verification, a pre-defined criterion is given to decide whether an assertion passes or fails. For example, after reviewing a design and an assertion, an engineer estimates the longest pipeline stage for signals of the assertion is around 10. Formal verification tool reports that the proof bound of the assertion is far greater than 10 cycles. Based on the information, the engineer claims that the assertion passes formal verification. Such a criterion is not sound and safe. However, it is commonly used as a preliminary indicator.

If an assertion doesn't pass formal verification, we need to debug. If the assertion fails with a counter example to violate the assertion, it is possible that the assertion is written incorrectly, some input assumptions are not correct or missing, or a RTL bug is found!

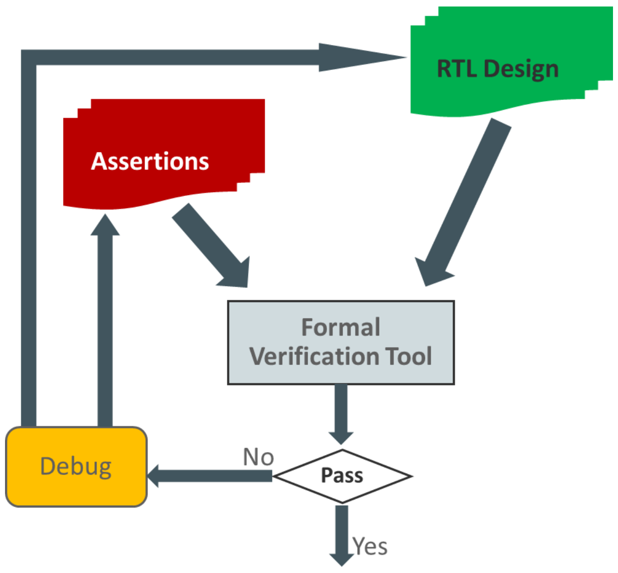


Figure 1 Formal Verification Process

Figure 1 shows formal verification process. For most formal verification test benches, more than two thirds of efforts are spent on assertion development and debugging.

2. Challenges to Formal Verification Closure

Similar to simulation-based verification, we need to know when formal verification is complete, so we can stop the process. When we stop, we should be confident that chances of bug escaping are very low. Most formal verification engineers face challenges at this point to decide when to stop and how to build the confidence with formal results.

Due to the nature of formal verification that it is built based on assertions, we need to review all assertions for verification closure.

Assert-type of assertions are used to check design behaviors. We need to understand that in the RTL code space, what have been covered and what have not been covered by assertions. We need to make sure assertions are written properly to cover the design space that they are intended to do. If there is a bug in the design space, can one of the assertions catch it? i.e., we need to ask ourselves that do we need more assertions? Without knowing how formal verification engines work and the actual design space that formal engines use, we want to get a quick answer. So we can identify verification holes and write more assertions to cover them.

For a circuit under formal verification, usually, there are a large number of "assume" type of assertions. Some assumes have been proven formally by other units that drive the assume logic. Some assumes (e.g., restrict type of assumptions) are defined only for formal verification as over-constraints to help formal tools to reduce the space as a divide-and-conquer approach. A group of assumes may have conflicts. With the current deepest proof bounds, formal tools may still be able to find valid inputs to drive the circuit. However, at some future state, formal tools may not be able to find any valid inputs to satisfy all assumptions. With the use of assumptions for formal verification, we need to make sure assumptions are correctly defined and used. Side-effects of over-constraints should be well understood.

After statically reviewing assertions, we need to understand how formal verifications tools proved the assertions.

For a bounded-proven assertion, do we understand the state space coverage between proof bound N and N+1? Do we have a deep enough proof bound that allows formal tools to hit corner cases? If so, can we stop at proof bound N for this assertion instead of spending huge efforts for N+1? At this stage, we need to understand how formal verification tool uses the design space. If a RTL line or condition was not exercised at proof bound N, which will raise an alarm for possible bug escaping.

Even an assertion has been fully proven. We still cannot draw a conclusion that all cone of influences (COIs) of the logic that the assertion was planned to check has been fully verified. Some assertions can be proven without the complete COIs. If formal verification tools are smart enough, it will use the knowledge without exploring the complete COIs.

With all these questions in mind, we are looking for hints to answer them. One possible direction is to write cover properties as functional coverage, then check whether cover properties are covered by formal engines. If they are not covered, then we need to review the proof bounds. However, cover properties are not always easy to write without knowing verification holes. Different engineers may write different cover properties based on their own opinions that make verification closure review a subjective process. In addition to that, a proof bound of a cover property may not have strong correlation with the proof bounds of assert properties – making the information hard to use.

From our experience, we learned that in addition to necessary cover properties, code coverage is another good indicator. It gives us objective metrics for line or condition coverage of a formal verification test bench. There are many research papers[1][2] on this topic, but few are successful in industry applications. In this paper, we define four code coverage models for formal verification that have been applied to the formal verification closure process.

3. Proposed Code Coverage Models for Formal Verification

In this section, four code coverage models for formal verification is proposed. We will give the definition for each model and explain them by examples.

3.1 Static Assertion COI Coverage

The first coverage model is to check whether COIs of the current assert-type assertions cover the design space that they were intended to check. In this model, constraints (assume or restrict types of assertions) are ignored. COIs of signals inside each assert-type assertion are computed. Union of COIs from all assertions are the final results. We can target lines, conditions, or registers for coverage.

If RTL has dead code, we want to exclude dead code from this analysis. Dead code is a piece of RTL code that won't be exercised by the circuit. Due to various reasons, dead code exists within many designs. In simulation, dead code will be marked as exclusion from coverage metrics. This can be done manually or automatically using formal tools[3].

For this static assertion COI coverage, we want to exclude dead code (unreachable lines or conditions) from formal coverage analysis and find out coverage targets inside and outside COIs.

This coverage model is a starting point to measure whether enough assertions have been written to cover all RTL code. Let's take one design as an example. This particular test case was done using formal verification, the current set of properties are considered fairly complete. The static assertion COI coverage for this test case is be reported by VC Formal using two simple commands:

```
analyze_fv_coverage
report_fv_coverage -list_uncovered
```

Table 1 shows a summary of assertion COI coverage for all registers of the design.

Number of instances analyzed	1612
Total number of registers	7607
Total number of assertions	5423
Total uncovered registers	18% (1375)

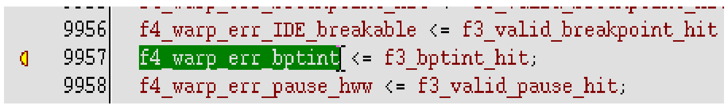
Table 1 Summary of Assertion COI Coverage

As the results showed, there are 18% of the registers not covered by any of the assertions! Some of the uncovered registers are listed in Table 2.

…
sub_partition0.f4_warp_err_bptint
…

Table 2 List of Registers Not in COI of Any Assertions

We then review these uncovered registers in the RTL code as shown in Figure 2. We can clearly see which area of the design is not verified by any assertions. It helps us to create a measurable metric for properties and see coverage holes. Similar code coverage approach can be taken for line or condition coverage.



```
f4_warp_err_IDE_breakable <= f3_valid_breakpoint_hit
f4_warp_err_bptint <= f3_bptint_hit;
f4_warp_err_pause_hww <= f3_valid_pause_hit;
```

Figure 2 Link to RTL Code of Uncovered Register

This structural analysis is light weight to formal tools. It is very fast to run and can provide us with immediate feedback of assertion coverage. We can then decide to either add more assertions to increase coverage, or review and waive those uncovered logic and leave them to be verified by other verification plans.

3.2 Input Stimuli Coverage

The second coverage model is to check constraints (assume or restrict types of assertions). In this model, assert-type assertions are ignored. The analysis focuses on under the current input constraints, how formal tools drive inputs to reach RTL design space. The same as static assertion COI coverage, we exclude dead code, so uncovered code that is only caused by formal input constraints are reported.

VC formal offers this feature called over-constraint analysis. It can be easily invoked from the line GUI, as shown in Figure 3.

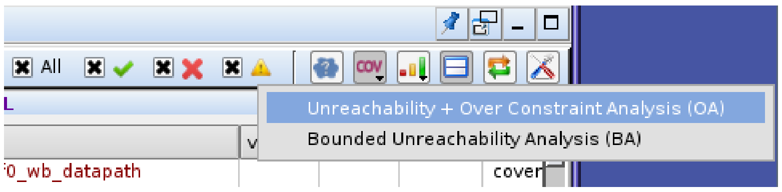
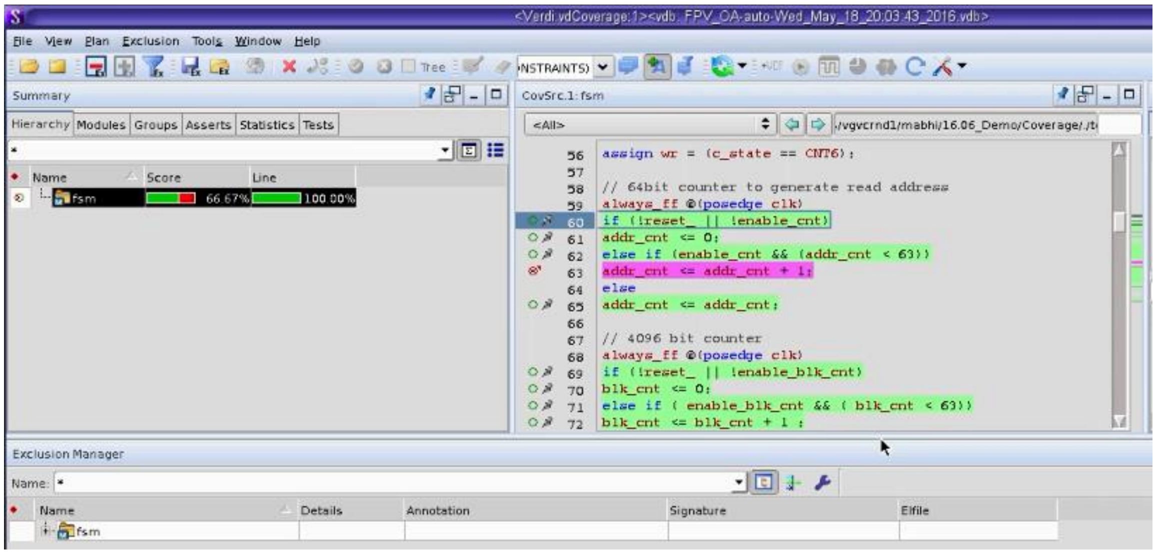

All
COV
Unreachability + Over Constraint Analysis (OA)
Bounded Unreachability Analysis (BA)
0_wb_datapath
cover

Figure 3 Unreachability and Over Constraint Analysis Feature

A snapshot of Verdi coverage view is shown in Figure 4, where uncoverable coverage goals without using any assumption are excluded, and uncoverable under the current assumptions are highlighted. We can query and find out the necessary constraints responsible for the RTL code to be uncoverable.



<Verdi:vdCoverage:1><vdb: FPV_OA-auto-Wed_May_18_20:03:43_2016.vdb>
File View Plan Exclusion Tools Window Help
Summary
Hierarchy Modules Groups Asserts Statistics Tests
Name Score Line
fsm 66.67% 100.00%
CovSrc.1: fsm
<All>
./vgvcrnd1/mabhi/16.06_Demo/Coverage/./t
Exclusion Manager
Name:
Name Details Annotation Signature Elfile
fsm

```
assign wr = (c_state == CNT6);

// 64bit counter to generate read address
always_ff @(posedge clk)
if (!reset_ || !enable_cnt)
addr_cnt <= 0;
else if (enable_cnt && (addr_cnt < 63))
addr_cnt <= addr_cnt + 1;
else
addr_cnt <= addr_cnt;

// 4096 bit counter
always_ff @(posedge clk)
if (!reset_ || !enable_blk_cnt)
blk_cnt <= 0;
else if ( enable_blk_cnt && ( blk_cnt < 63))
blk_cnt <= blk_cnt + 1 ;
```

Figure 4 Coverage View of Unreachable and Over Constraint Analysis

Figure 4 shows us that it is possible that we never turn on the counter due to this setting within formal environment.

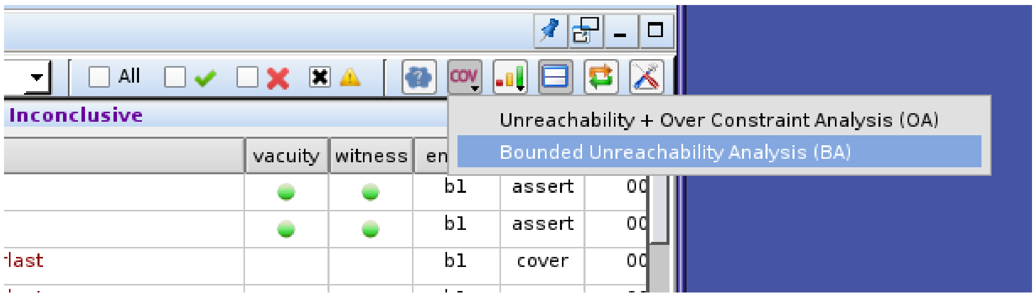
```
enable_cnt == 0
```

Such an un-intended assumption comes from a TCL command of the formal tool setup file. It is not checked by formal or simulation tools and can be easily ignored during manual review. With input stimuli coverage, over-constraints like this can be caught by reviewing coverage holes.

3.3 Bounded Proof Coverage

The third coverage model is to understand bounded proofs. In this model, the coverage data closely correlates with formal prove engines. The analysis will tell users code space coverage under current proof bounds. It can be done by per bounded proof or a group of bounded proofs.

Figure 5 shows invoking Bounded Unreachability Analysis from property table.

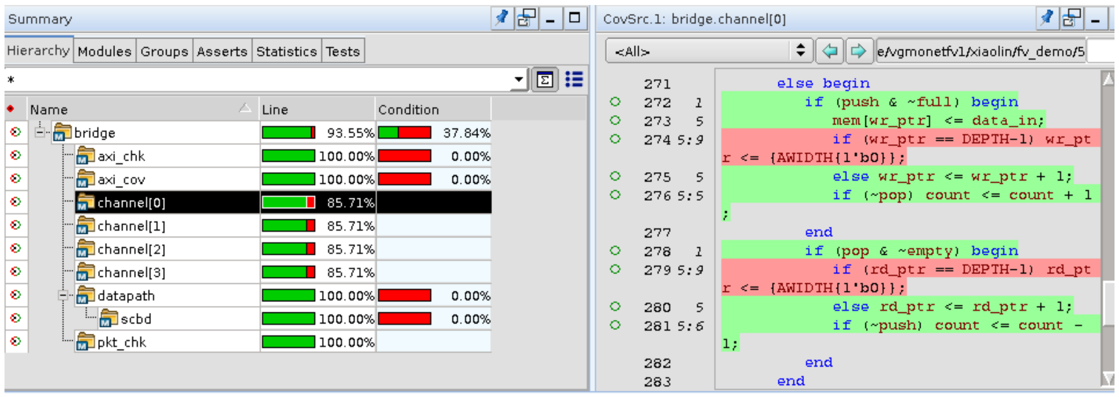


All
COV
Inconclusive
Unreachability + Over Constraint Analysis (OA)
Bounded Unreachability Analysis (BA)
vacuity
witness
en
b1
assert
00
b1
assert
00
last
b1
cover
00

Figure 5 Bounded Proof Coverage Analysis Feature

With this model, we can learn what coverage targets have been hit within the current proof bound. If a bounded proof has a very low percentage of coverage for its COI, we know that the proof bound may not be good enough and we cannot trust the proof bound. More efforts should be spent to converge the assertion.

Figure 6 shows coverage numbers for bounded proof coverage for one design. We can look into the source code to review what part of the logic is covered by a certain bound, and what part of the logic is inclusive within the specified bound. This will help us to make decisions on what the next step to take.



Name	Line	Condition
bridge	93.55%	37.84%
axi_chk	100.00%	0.00%
axi_cov	100.00%	0.00%
channel[0]	85.71%	
channel[1]	85.71%	
channel[2]	85.71%	
channel[3]	85.71%	
datapath	100.00%	0.00%
scbd	100.00%	0.00%
pkt_chk	100.00%	

```
271          else begin
272    1        if (push & ~full) begin
273    5           mem[wr_ptr] <= data_in;
274 5:9            if (wr_ptr == DEPTH-1) wr_pt
r <= {AWIDTH{1'b0}};
275    5           else wr_ptr <= wr_ptr + 1;
276 5:5            if (~pop) count <= count + 1
;
277             end
278    1        if (pop & ~empty) begin
279 5:9            if (rd_ptr == DEPTH-1) rd_pt
r <= {AWIDTH{1'b0}};
280    5           else rd_ptr <= rd_ptr + 1;
281 5:6            if (~push) count <= count -
1;
282             end
283          end
```

Figure 6 Bounded Proof Analysis Coverage View

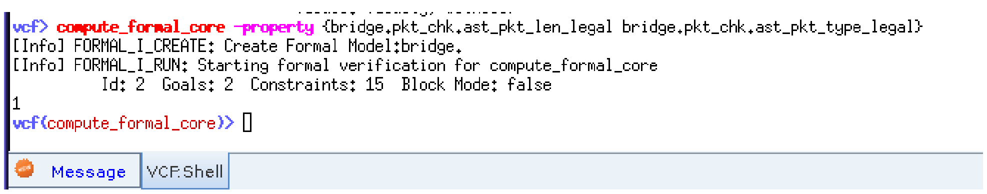
In addition to COI coverage from the current proof bound, this bounded proof coverage also gives us the progress of bounded proofs. For example, it can tell us that there are 95% condition coverage for COIs at proof bound 11. Formal engines spent 2 days from proof bound 10 to 11 for this assertion and there is no new condition covered. This gives us an indication that having a proof bound 12 for this assertion may not be feasible. We may need to stop at the current proof bound and investigate more abstractions for this design.

This type of coverage is expensive and coverage analysis needs to run together with formal engines to record coverage data. To reduce the cost, we do it only for high-value assertions.

3.4 Proof Core Coverage

The forth coverage model is proof core coverage. It analyzes coverage from proven assertions. An assertion is proven doesn't mean the whole COI of it has been covered. Formal tools have abstraction engines that can use the minimum state space (proof core) to prove an assertion. If there are design bugs outside the proof core but still within the COI of the assertion, the assertion can still be proven. This indicates that either more assertions should be added or the current assertion needs to be updated.

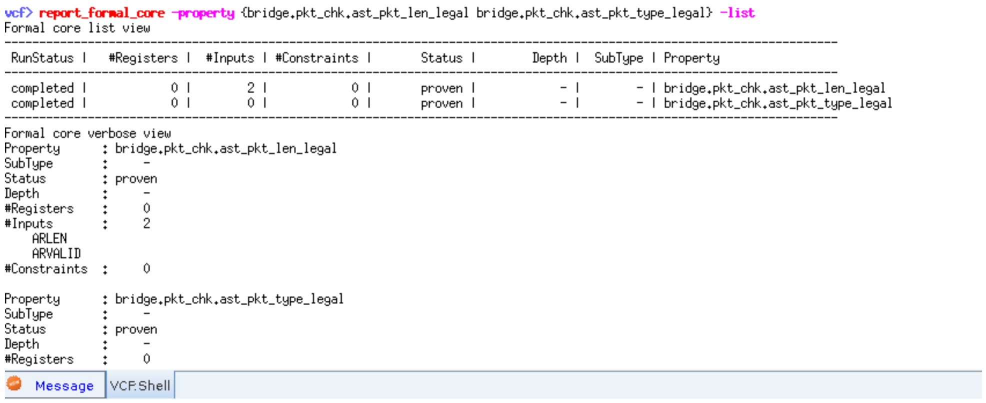
Figure 7 shows the tool invoking the feature to compute proof core coverage for two proven assertions.



```
vcf> compute_formal_core -property {bridge.pkt_chk.ast_pkt_len_legal bridge.pkt_chk.ast_pkt_type_legal}
[Info] FORMAL_I_CREATE: Create Formal Model:bridge.
[Info] FORMAL_I_RUN: Starting formal verification for compute_formal_core
         Id: 2  Goals: 2  Constraints: 15  Block Mode: false
1
vcf(compute_formal_core)>
```

Figure 7 Compute Proof Core Coverage for Proven Assertions

Figure 8 shows the report of proof core coverage including primary inputs and registers that are required to prove each assertion. Such information can be visualized by GUI windows together with RTL source code, so users can easily find out what are not used from static assertion COI coverage.



```
vcf> report_formal_core -property {bridge.pkt_chk.ast_pkt_len_legal bridge.pkt_chk.ast_pkt_type_legal} -list
Formal core list view
----------------------------------------------------------------------------------------------------------------
 RunStatus |   #Registers |  #Inputs | #Constraints |       Status |        Depth |  SubType | Property
----------------------------------------------------------------------------------------------------------------
 completed |            0 |        2 |            0 |       proven |            - |        - | bridge.pkt_chk.ast_pkt_len_legal
 completed |            0 |        0 |            0 |       proven |            - |        - | bridge.pkt_chk.ast_pkt_type_legal
----------------------------------------------------------------------------------------------------------------
Formal core verbose view
Property      : bridge.pkt_chk.ast_pkt_len_legal
SubType       :    -
Status        : proven
Depth         :    -
#Registers    :    0
#Inputs       :    2
    ARLEN
    ARVALID
#Constraints  :    0

Property      : bridge.pkt_chk.ast_pkt_type_legal
SubType       :    -
Status        : proven
Depth         :    -
#Registers    :    0
```

Figure 8 Report of Proof Core of Proven Assertions

Proof core coverage is the most accurate model for actual logic needed to prove an assertion. Different formal tools may have different results based on how abstractions are done. If an assertion is fully proven without 100% percentage of its COI covered, any RTL bug within the uncovered code space will not be able to be detected by the assertion.

Similar to bounded proof coverage, proof core coverage needs to run with formal engines dynamically. For large design, only a small set of proven assertions are selected for such an analysis.

4. Conclusions

The authors believe doing formal verification without coverage closure is the same as doing simulation without coverage closure. In this paper, we present four coverage models and provide some preliminary results. With the increasing usage of formal verification for circuit design, we expect these formal verification coverage models will become standard models for formal tools and are used by formal verification sign-off process.

5. References

[1] *Coverage estimation for symbolic model checking*, Y. Hoskote, T. Kam, Pei-Hsin Ho, Xudong Zhao, pp 300-305, DAC 1999

[2] *Coverage Metrics for Formal Verification*. Hana Chockler, Orna Kupferman, Moshe Vardi. Correct Hardware Design and Verification Methods, Volume 2860 of the series Lecture Notes in Computer Science pp 111-125, 2003

[3] *A Vendor-Independent Formal Unreachability Analysis Flow for Automated Coverage Closure*, Xiushan Feng, Abhishek Muchandikar, Sunil Keerthi, Praveen Tiwari, SNUG Austin, 2015