Coverage Models for Formal Verification

Xiushan Feng[*], Xiaolin Chen[+], Abhishek Muchandikar[+]

[*]Oracle Labs (5300 Riata Park Ct, Building 7, Austin, TX 78727 US)
Shaun.Feng@oracle.com,
[+]Synopsys (690 E Middlefield Rd, Mountain View, CA 94043 US)
{Xiaolin.Chen;Abhishek.Muchandikar}@synopsys.com

***Abstract*- As formal verification engineers, the authors always face challenges to accurately access 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 we over-constrain inputs that caused the drop of 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? We cannot get answers to these four most-asked questions without extracting information from formal engines, which is not feasible for general users. However, like coverage metrics from simulation-based verification, formal verification coverage models can be defined and used as metrics to measure formal verification progress and completeness. Some academic research on formal verification coverage and commercial formal verification tools are starting to support some coverage usages in the past two years. However, none of them clearly specified what engineers would really need and provided a good way to present formal coverage results in a standard way.**

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 metrics.

I. Introduction

As formal verification engineers, the authors always face challenges to evaluate the status and quality of test benches. Four key questions need to be answered at various stages of a project. E.g.,

1. Do we need more assertions?
2. Did we over-constrain inputs that could cause the drop of an important design scenario?
3. Are proof bounds from bounded proofs good enough to catch potential design bugs?
4. For the properties that are fully proven, do they cover the design logic that they were intended to cover?

The above four most-asked questions cannot be answered without analyzing assertions and extracting information from formal proof engines, which is not feasible for general formal tool users. However, like coverage models from simulation-based verification, formal verification coverage models can be clearly defined across the industry and be used as coverage metrics to measure formal verification progress and completeness. With such metrics from formal tools, it is possible for formal tool users to find answers to the above four questions.

In this paper, the authors will introduce four formal verification coverage models and their usages by real-life examples.

II. Background

The term of "formal verification" we use in this paper is limited to formal property verification (or model checking) for circuit designs. A circuit is given as a RTL design. Circuit behaviors under verifying are written as assertions. There are four types of assertions (assert/assume/cover/restrict).

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

Because formal verification explores circuit space exhaustively to prove assertions, it may not be able to have a conclusive proof for large designs or complicated assertions 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 of obtaining 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 determine whether an assertion passes or fails. For example, after reviewing a design and an assertion, an engineer estimates the longest pipeline stage for signals used in the assertion is around 10. Formal verification tool reports that the proof bound of the assertion is far

greater than 10 cycles. Based on that 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 by many engineers.

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!

Fig.1 shows a typical formal verification process.

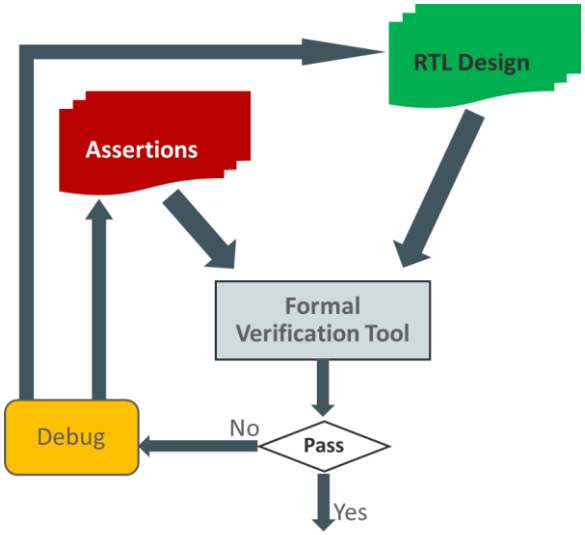

Figure 1. Formal Verification Process

A. 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 being 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 that the assertions are written properly to cover the design space intended to be covered. If there is a bug in the design space, can one of the assertions catch it? i.e., we need to ask ourselves: 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 to help us identify verification holes so that we can 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 deeper 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 is not exercised at proof bound N, it must 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.

III. 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.

A. 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 as defined by equation (1). We can target lines, conditions, or registers for coverage.

$$COI\ (tb) = \frac{\sum(\bigcup_0^{n-1} coi(ast_k))}{\sum total} \qquad (1)$$

- ast_k *is one of* n *assertions of testbench* tb. n, k *is an integer*
- $\sum$ *is a operator to get the number of coverage targets*
- $\sum$ *total is the total number of coverage targets within tb*
- *coi() is the function to compute cone of influence of an assertion*

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. Fig. 2 shows 5 assertions in blue circles with their COIs computed (green triangles). Static assertion COI coverage shows design space (in yellow) there are not covered by any COI. This is the area that potential verification holes exist.

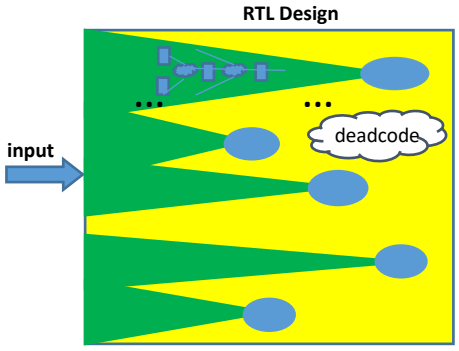

Figure 2. Static Assertion COI Coverage

This coverage model is a starting point to measure whether enough assertions have been written to cover all RTL code. Let's take a formal verification test bench as an example. This particular test bench claimed success for formal verification task. The current set of properties are considered fairly complete.

The static assertion COI coverage of register space for this test case is reported by Synopsys 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. We can clearly see which area of the design is not verified by any assertions (showed as Fig. 3). It helps us create a measurable metric for properties and see coverage holes. Similar code coverage approach can be taken for line or condition coverage.

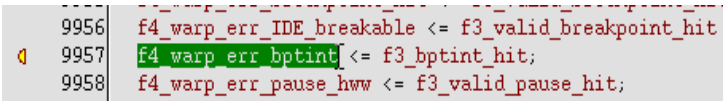

Figure 3. Link to RTL Code of Uncovered Register

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 elements and leave them to be verified by other verification plans.

B. 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 how formal tools drive inputs to reach RTL design space under the current input constraints. The same as static assertion COI coverage, we exclude dead code, so uncovered code that is only caused by formal input constraints are reported.

$$Stimuli(tb) = \frac{\sum(U_0^{\infty}(C_i))}{\sum total} \qquad (2)$$

- C_i *is covered target at cycle i.* $U_0^{\infty}(C_i)$ *is the greatest fixpoint (GFP) of all reachable targets.* $\sum(set)$ *is the number of items inside set*
- $\sum total$ *is the total number of coverage targets within tb*

For equation (2), we can see that the greatest fixpoint is computed. Inside formal verification, the greatest fixpoint is the largest state space that a circuit can read under existing input constraints and an initial state. If a fixpoint cannot be computed due to runtime or memory limitations, a bounded reachable state space can be used for stimuli coverage.

Fig. 4 shows reachable state for all possible inputs. When there is no new state can be visited (or a pre-defined bound has been reached), anything outside the reachable space will not be covered by formal verification (yellow region). Invalid constraints, e.g., conflicts within a set of constraints, over-constrained inputs, can cause unreachable space that will mask potential RTL bugs.

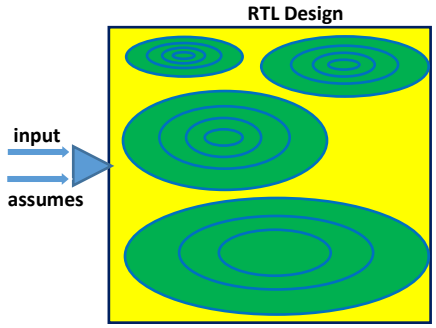

Figure 4. Input Stimuli Coverage

VC formal offers this feature called over-constraint analysis that can be used to measure input stimuli coverage. Its results can be reviewed by Verdi the same way as simulation coverage. A snapshot of Verdi coverage view is shown in Fig. 5, where uncoverable coverage goals without using any assumption are excluded (these are targets within dead code), and uncoverable under the current assumptions are highlighted. In this test, we focus on line coverage. Coverage percentage is given and source code browser can navigate through covered or not-covered items. In addition to this, we can query and find out the necessary constraints responsible for the RTL code to be uncoverable. This is very important to formal verification constraint debugging.

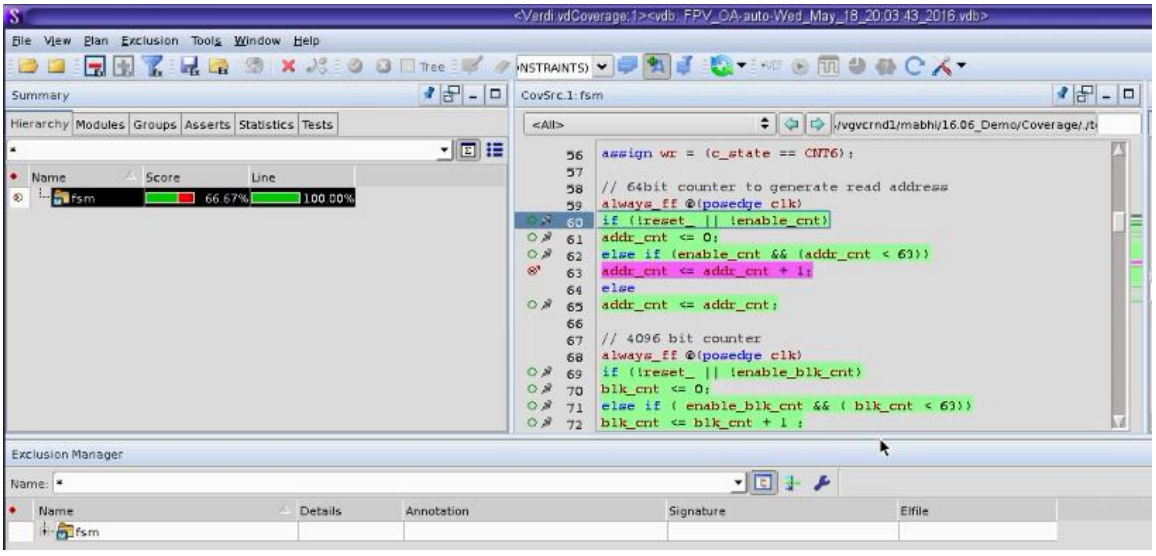

Figure 5. Coverage View of Unreachable and Over Constraint Analysis

Fig. 5 shows us that line 63 is uncoverable under current constraints. So the counter will never able to increase. With the proper query through the formal tool use interface, we narrow down to a small set of input constraints. There is an input constraint that incorrectly disables the counter:

```
enable_cnt == 0
```

Such an un-intended assumption comes from a TCL command of the formal tool setup file. It is not verified by other formal test benches or checked at simulation environment. This can be easily ignored during manual review. With input stimuli coverage, over-constraints like this can be caught easily by reviewing coverage holes. A few similar holes were identified inside our formal test benches.

With this coverage analysis, we have a better understanding of constraints. Some input constraints are intentionally over-constrained for divide-and-conquer approaches to partition the design space. With input stimuli coverage analysis, side-effects or consequences of these constraints are exposed for review. There are some constraints that haven't been formally proven within their driving units where such constraints are used as assertions for these units. These constraints may have been checked at simulation environment without failings. However, we still cannot fully trust these constraints. We need to run input stimuli coverage analysis to find out with such constraints whether formal verification drops valid scenarios.

C. Bounded Proof Coverage

The third coverage model is used to understand bounded proofs. In this model, the coverage data closely correlates with formal prove engines. The analysis will tell users code coverage under current proof bounds. It can be done to one assertion with bounded proof or a group assertion s with bounded proofs. Given a set of k assertions with maximum proof bound n, bounded proof coverage can be defined as (3):

$$bounded\_proof\ (ast\_set_k) = \frac{\sum((\bigcup_0^n(C_i))\cap coi(ast_i))}{\sum coi(ast_i)} \quad (3)$$

- $ast_i \in ast\_set_k$ *i, k is an integer*
- ast_i *is an assertion that has a proof bound* n; n, k *is an integer*
- *coi() is the function to compute cone of influence of an assertion*

This coverage model finds the common coverage between static assertion COI coverage and reachable coverage with n cycles. It gives more accurate covered items with the given proof bound. Analysis can be done for a small group of critical assertions. Fig. 6 shows bounded proof coverage for an assertion (blue circle). The overlap space between the triangle and circles is covered space of bounded proof coverage of the assertion.

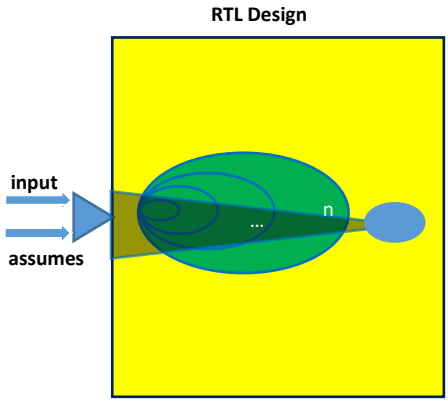

Figure 6. Bounded Proof Coverage

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 rely on the proof bound to sign off on this assertion. More efforts should be spent to converge the assertion.

Fig. 7 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.

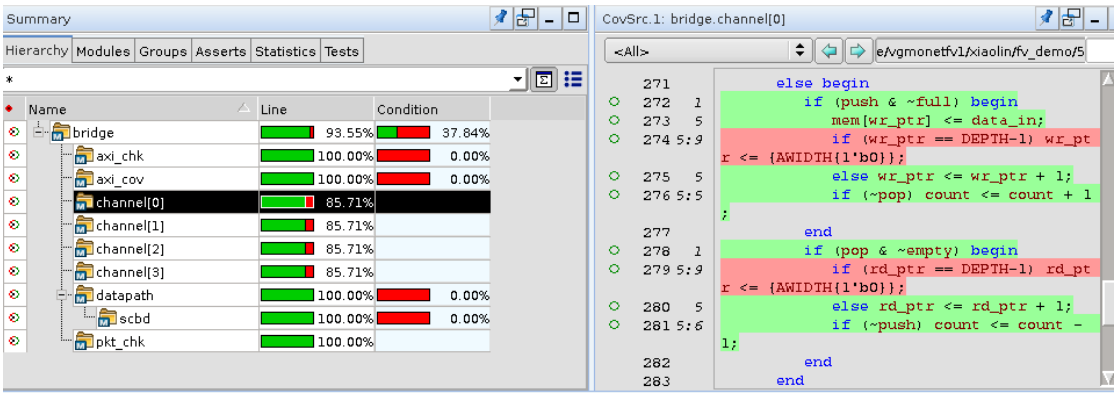

Figure 7. Bounded Proof Analysis Coverage View

In addition to COI coverage from the current proof bound, the 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.

D. 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 showed by Fig. 8) 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. Proof core coverage is defined by (4):

$$proof\_core\ (ast\_set_k) = \frac{\sum(\cup proof\_core(ast\_{}_i))}{\sum coi(ast_i)} \quad (4)$$

- $ast_i \in ast\_set_k$ *i, k is an integer*
- *proof_core(*$ast\_{}_i$*) is the set of targets actually used by formal engines to prove* $ast\_{}_i$
- *coi() is the function to compute cone of influence of an assertion*

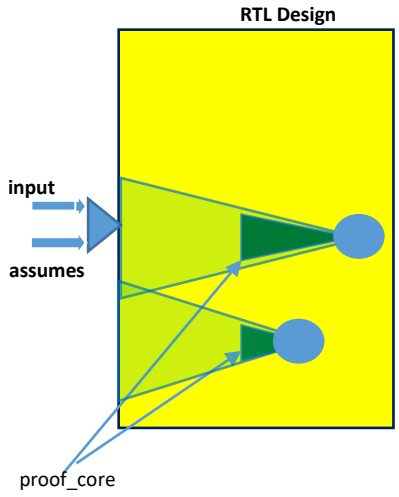

Figure 8. Proof Core Coverage

Fig. 9 shows a preliminary report of proof core coverage including primary inputs and registers that are required to prove an 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
 Message | VCF Shell
```

Figure 9. 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.

IV. Coverage Tracking

With these coverage models defined, we can track status of a formal verification test bench along the whole design cycle. As soon as RTL is ready, formal test bench is created with assertions in different types (e.g., assume/assert/cover/restrict). Here are figures for a formal verification test bench.

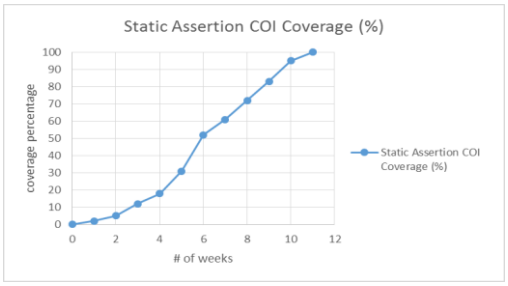

Figure 10.1 COI Coverage

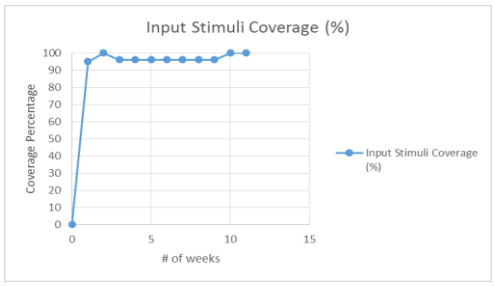

Figure 10.2 Stimuli Coverage

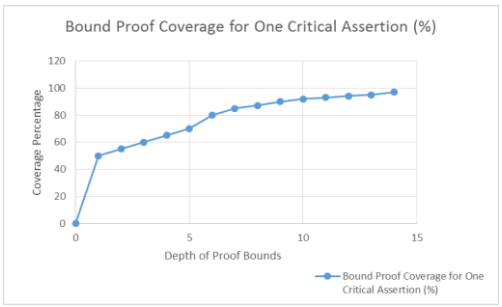

Figure 10.3 Bounded Proof Coverage

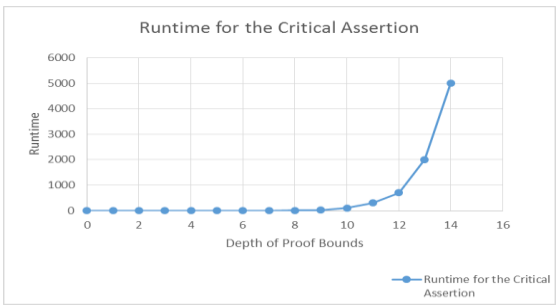

Figure 10.4 Runtime for the Critical Assertion

From the above figures, we can learn that after applying waivers and excluding targets within dead code, static assertion COI coverage hits 100% for code coverage (Fig. 10.1). We strike for COI 100% coverage because this unit relies on formal verification to fully verify -- there is no simulation test bench for this unit. Input stimuli coverage was run to collect data during the project, but it is only required at the end of project. When a formal test bench was created, there were only a few simple assumptions. So input stimuli coverage was high. With more constraints added, the number was decreasing. At the end of project, we did a final review and excluded unreachable targets and dead code. There was no intended over-constraint for this test bench, so we have 100% input stimuli coverage for the final test bench (Fig. 10.2) after resolving a few over-constraints. For a critical assertion with bounded proof, we did the bounded proof analysis. We want to understand the coverage with each proof bound and runtime. With the collected information together with some coverage properties, we feel the current proof bound 14 is good enough and it will be very hard for us to get a proof bound more than 14 with reasonable runtime without further abstractions.

We don't show proof core coverage because the proof core coverage is more expensive to collect based on the total number of assertions that we have. However proof core coverage on a small set of proven targets was very useful to detect potential holes. We wrote more assertions to cover the design space outside of proof cores.

It is worth pointing out that none of the existing formal varication tools can help to track progress of coverage (or even the percentages) and proof bounds. The data exists from log files or tool database. We have to manually harvest to prepare our tracking charts.

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. In the past two years, one author of this paper has been talking to a few major formal verification tool vendors to make sure they understand specific needs from users, enhance their exiting coverage APPs, and provide unified coverage results with useable GUI interface to represent. At this point, none of the vendors have a complete solution for all the usage models.

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