On Verification Coverage Metrics in Formal Verification and Speeding Verification Closure with UCIS Coverage Interoperability Standard

Rajeev Ranjan, PhD
Chief Technology Officer

Ross Weber
Principal Engineer

Ziyad Hanna, PhD
Vice President of Research & Chief Architect

Jasper Design Automation Inc.

rajeev,weberrm,ziyad.hanna@jasper-da.com

***Abstract*— This paper presents a comprehensive approach to the classical notion of hardware design verification coverage, augmenting the widespread coverage analysis and metrics from simulation with coverage obtained with formal analysis. In our work, we address major coverage questions related to the application of formal verification such as "is my formal environment capable of sufficiently analyzing the design?" "Are there parts of my design not being checked by the property set?" "What parts of my design have been verified/not verified as part of the proof process?" The formal coverage results and analysis can leverage the latest coverage interoperability standard (UCIS) that allows for interoperability of verification coverage data across tools and methods from multiple vendors. Our approach addresses the increasing verification productivity challenge by delivering the requisite coverage visibility and metrics from formal verification while simultaneously accelerating the overall verification coverage closure process.**

***Keywords*—** *(Formal Verification, Formal Coverage, Formal Specification Languages, Coverage Interoperability Standard)*

I. INTRODUCTION

The role of formal verification technology in the overall system-on-chip (SoC) design and verification flow has been growing over the years [2]. The application of this technology has spread, providing a wide range of solutions to emerging problems in design and verification, starting from architectural modeling and verification, through RTL development and block/system verification, all the way to post-silicon debug. Today, diverse engineering groups, such as architects, security engineers, RTL designers, validation engineers, and silicon debug experts, use verification solutions based on formal technologies [10]. The higher quality of the verification results, as well as the improved productivity that the formal verification methods deliver, have made it a common methodology to apply in all aspects of the design and validation flows. Consequently, the wider adoption of formal methods has led to a growing need for coverage information similar to that traditionally obtained from dynamic methods. This need is motivated by two factors. The first factor is that of achieving confidence about the effectiveness of the verification tasks performed with formal technology. The second is that of finding a way to combine engineering efforts and results from both formal verification and simulation to accelerate overall verification closure.

In this paper, we introduce several notions of coverage relevant to formal analysis that can help fulfill both objectives. The paper is organized as follows: Section II covers the necessary background to set the context; Section III provides some preliminary definitions and terminology on which our work is based; Section IV gives the background on coverage analysis in the context of formal verification; Section V, Section VI, and Section VII provide details on three different types of coverage measurements in formal verification. In Section VIII and Section IX, we touch upon the aspect of leveraging coverage information from formal verification and simulation to accelerate verification closure. Finally, Section X provides experimental data and a flow that illustrates our formal coverage analysis method.

II. COVERAGE BACKGROUND

The process of any functional verification method involves stimulating the design, propagating the effects to an observable point, and checking the response for the correct behavior. The completeness of this process is dependent on the completeness with which the design is stimulated (that is, all possible behaviors have been exercised), the ability to ensure that the effects (both good and bad) of exercising the behaviors are propagated to an observable point, and on the completeness of the checking mechanism (that is, all relevant design behaviors have been checked against the specification)[6].

Simulation-based verification is a sampling process of all of the possible input sequences that could be applied to the design. Various testbench-based tools and methodologies are widely used today to focus the validation of the design on a subset of

interesting scenarios of the design in order to limit the validation process to a manageable size and effort. Since simulation-based methods are not exhaustive, coverage methods have become an essential means to measure the completeness of the validation. Over the years, validation engineers have invented various coverage models and techniques to measure the effectiveness of the testbench, including code coverage, branch coverage, finite state machine (FSM) coverage, functional coverage, and so forth.

In formal verification, the correctness of a design is verified with respect to a desired behavior. The verification is carried out by checking whether the mathematical state of the design (typically a labeled state-transition graph) satisfies the specification of this behavior, which is expressed in terms of a temporal logic formula (specified by a set of properties) or a finite automaton.

Assuming that the implementation of the formal technique(s) is accurate, the verification completeness of a given design is then dependent on:

1. The completeness of the property set (specification) against which the design is formally verified,
2. The accuracy of the constraints (the set of legal stimuli under which the design is analyzed), and
3. Whether formal analysis was complete or partial. A partial analysis means that the computation was not able to complete due to resource constraints, such as time or memory.

In the Assertion Based Verification (ABV) methodology — one of the most common manifestations of formal method adoption — designers and verification engineers systematically add embedded assertions into their design, using some type of specification language such as SVA or PSL. When run in simulation flows, such assertions, when triggered, indicate a failure in the design. When used in formal methods, such assertions prove the correctness of the design. ABV users usually ask the questions, "have we put enough assertions in the design?" and "do the assertions capture the entire design intent?" Typically, these assertions are verified in the presence of constraints that capture the environment behavior. The soundness of these constraints is always a concern and can be addressed by measuring the stimuli coverage in the presence of these constraints. Additionally, it is possible that some of these assertions are not analyzed for the entire reachable state space of the design due to some resource constraint (such as compute memory and time). In these cases, obtaining verification coverage for the partial analysis is desired.

The most accurate coverage model is one that is based on the reachable state space of the design. The true indicator of verification progress would be based on the ratio of state space visited compared to the set of all possible states. Obviously, for coverage analysis, the state space should be computed for calculating the coverage percentage. However, it is practically impossible to represent the state-space graph of an entire industry design explicitly. In addition, the absolute number of states would be astronomically large and would not provide any meaningful value towards measuring verification progress.

For decades, a variety of different coverage models have been used that achieve different degrees of approximation of the true coverage. For example, code coverage-based models such as branch coverage, statement coverage, expression coverage, and so forth. In addition, FSM states and transition-based coverage models are widely used. For a complete treatment of a variety of coverage models, refer to [9].

III. PRELIMINARIES

Consider RTL design D represented in a standard hardware description language, such as SystemVerilog or VHDL, and formal specification S described by a set of SVA or PSL properties. For a given RTL design D, a Coverage Model M is a finite set of Coverage Items $CI(D) = \{C_0, C_1, \ldots, C_n\}$, where each cover item C_i represents an implicit or explicit event in the design. An example of an implicit event would be one corresponding to the triggering of a line/branch/expression or a set of events that can be enumerated based on some specification. An explicit event would be something specified by a user. In some cases, the collection of coverage items could also be dependent on a property or a set of properties under consideration.

Coverage Analysis computes whether a cover item C_i is triggered during the simulation of RTL design D, or was part of the reachable state space during the formal analysis, and if so, we say that such C_i is reachable. The coverage of design D for a given Coverage Model M is computed by the percentage of the reachable C_i in the Coverage Items CI. For example, a chosen coverage model can consist of all the branches in the source code of the design (classically known as branch coverage); thus, the CI is the set of all branches in D. Coverage analysis checks whether every C_i, or source code branch in D, is executed in simulation, or the corresponding state has been analyzed in formal analysis. The coverage of D according to M and CI is the percentage of reachable branches in CI. Depending on the context, the set of branches under consideration could be restricted to

those relevant only for the set of properties being analyzed. That is, a cover item is considered relevant for a property S if it falls in the cone of influence (COI) of the property. The percentage coverage is then appropriately computed by considering only the restricted subset. A Coverage Model M can be explicitly defined by the user. For example, each C_i of CI could be a functional event of interest, for example, "request ##3 grant" — a request followed by a grant in three cycles. Typically, the set of coverage points representing some functional events is referred to as the functional coverage model.

All coverage analysis-related information — set of coverage items, set of properties, and status of coverage items with respect to each property — is typically stored in a database, commonly referred to as a coverage database. These databases are proprietary to the verification technology vendors. Recently, a standard, known as the Unified Coverage Interoperability Standard, has emerged allowing all proprietary databases to support a standard API to perform read/write operations. The API standard is meant to enable uniform access to coverage data from heterogeneous sources, such as different technologies and tools from different vendors.

IV. Formal Coverage

In the context of formal analysis, a number of coverage measurement and analysis-related questions are addressed below:

Stimuli Coverage: Given a formal environment that captures, among other things, a set of constraints to limit the set of legal stimuli applied to the design, can we achieve some measurement of confidence regarding the sufficiency of the input vectors permissible by the constraints? In other words, can we obtain some coverage measurement for protection against the possibility of over-constraining the design? Do the constraints limit the state space for a set S of properties or checkers? This analysis can be referred to as *stimuli coverage*. It should be noted here that in simulation, this is the default interpretation of the "coverage" term.

Property Completeness Coverage: In formal verification, a set of assertions S is specified in some formal language, such as SVA or PSL, to check whether the design D adheres to the desired design intent. Measuring the exhaustiveness of a specification in formal verification ("do more properties need to be written?") has a similar flavor to measuring the exhaustiveness of the input sequences in simulation-based verification ("do more input sequences need to be created?"). We refer to the coverage metric to approximate the completeness of a property set as *property completeness coverage*. We should note that this coverage metric is also relevant in a simulation context; however, as previously noted, the *stimuli coverage* remains the default and main focus in the dynamic validation world.

Proof Coverage: Unlike simulation methods, which sensitize the design with input stimulus on an iterative basis, formal methods traverse the state space for the design. By default, the formal tool attempts to traverse the entire reachable state space of a design. For the cases where the entire state space is traversed, we would like to obtain the corresponding verification coverage. We refer to this type of coverage metric as *full-proof coverage*. However, depending upon the complexity of the design, the property under consideration, and the resulting mathematically complex nature of the underlying analysis, it is possible that the computation may hit resource limits, such as available time and memory. We refer to the coverage metric used to determine the verification coverage for this bounded proof analysis as *bounded proof coverage*.

Selection of Coverage Model: As stated earlier, the most accurate coverage model is one that is based on the reachable state space of the design, that is, CI, the set of reachable states of design D. Each reachable state from the reset state(s) is a cover item C_i. Obviously, for coverage analysis, the state space should be computed in order to calculate the coverage percentage. However, this coverage model is not practical from either computation or representation perspectives.

It is interesting to note that, for various coverage measurements in the formal analysis world, we can borrow on the well-established notions of coverage metrics adopted in the simulation world. While our coverage approach is independent of the chosen coverage model, for illustration purposes, we have chosen to use a specific flavor of code coverage as our coverage model. In particular, for our work, we create CI from RTL statements as well as branching conditions. The number of cover items in these two coverage models varies linearly with the size of the RTL source code. This is intuitive enough for the users, and practically manageable for the sizes of RTL designs where formal verification is applied.

V. Stimuli Coverage

Unlike simulation, where the design is sensitized by the explicit application of input vectors as generated by the testbench, in formal verification, the mathematical model of the design and the specification is analyzed for all possible combinations of input sequences. In this scenario, the stimuli coverage for the design is 100 percent because all possible sequences have been applied to the design. However, in practice, a formal environment deploys constraints that are used to eliminate the illegal input sequences and limit the analysis of the design to valid ones. These constraints are typically declarative

properties specified in SVA or PSL. The constraints are specified with the intention of preventing false failures, that is, an assertion fails in the design for an illegal input sequence. The presence of constraints in the formal environment necessitates computation of stimuli coverage in formal analysis. This is even more important to prevent "over-constraining." This is a situation where constraints collectively eliminate legal input sequences and prevent the verification from visiting legitimate reachable states where the design might fail the specification. In other words, over-constraining could lead to a false sense of confidence. Stimuli coverage metrics provide a way to establish some confidence about the correctness of the constraints in the formal environment.

In our proposal, stimuli coverage computation works in a very similar manner to that in a simulation flow. The method is independent of the coverage model chosen. As mentioned earlier, for the purposes of measuring coverage, we can borrow the widely adopted metrics in the simulation world, such as branch coverage, statement coverage, expression coverage, FSM coverage, toggle coverage, and so forth.

Assume that CI represents the set of coverage items for a given coverage model. For every cover item C_i in CI we perform a reachability analysis. Based on the results of the reachability analysis, the coverage items can be divided into three subsets: CI_R(E, D) = {C_i | C_i in CI where C_i is reachable or covered in design D under environment E}. The subset CI_R represents all the events that can be reached by a finite path from reset state(s). We denote the subset CI_U(E, D) = {C_i | C_i in CI, where C_i is unreachable or uncovered in D under environment E}. Depending upon the chosen coverage model, the subset CI_U(E, D) could represent the fragment of RTL code which cannot be exercised. This could be an artifact of (a) design error that causes the non-exercisability, or (b) some configuration parameters that disable some pieces of code (intentional disabling), or (c) the interactions of constraints preventing the design from entering a state where the code fragments could be exercised. Specifically, stimuli coverage can help detect "dead-code" situations in the design. The third set is CI_B(E, D) = {C_i | C_i in CI where C_i is unresolved or got to a bounded search in design D and environment E}. The sets CI_R(E, D), CI_U(E, D), and CI_B(E, D) depend on the complexity of the design, constraints, computational performance of the formal engines, as well as on resources applied for the computation, such as time and memory.

Applying the environmental constraints E for stimuli coverage can be done adaptively. Initially, we can choose E to be empty, that is, no constraints are applied to the design. Thus, the set CI_U(E,D) reflects the dead code in all possible environments of D. Users can define the environment incrementally by defining the clock, reset, and then the behavioral constraints, which yields a sequence of $CI\_U_i(E_i,D)$ sets, where $CI\_U_i(E_j,D)$ is included in $CI\_U_i(E_k,D)$ for E_j that is more restrictive than E_k. The computation complexity of these subsets increases for more restrictive environments. Users can obtain incremental reports for each $CI\_U_i(E_i,D)$ to understand the impact of the E_i on D. This process provides a comprehensive understanding of the impact of adding new constraints to the design in the form of differences observed in stimuli coverage.

VI. Property Completeness Coverage

Research in the area of property completeness coverage for formal verification has been focused solely on state-based coverage. This state-based coverage metric is based on mutations — small errors or mutants injected into the design — applied to the FSM. Essentially, a state "s" in the FSM is covered by the specification if modifying the value of a variable in the state renders the specification untrue. In [11], Chockler *et al*. adapted the work done on coverage in simulation-based verification to the formal-verification world in order to obtain new coverage metrics. For a number of metrics used in simulation-based verification, the authors presented a corresponding metric that is suitable for formal verification, and described an algorithmic way to check it. Examples given in the paper are code coverage, circuit coverage, FSM coverage, and mutation coverage. In fact, they claim that almost every heuristic regarding stimuli coverage in the simulation world has a corresponding metric to determine the response-checking coverage in the formal world. In [12] Koen provides a way of approximating an answer to the question, "Have we specified enough properties?" Given the interface of a design under verification, plus a property list, the technique identifies cases where some outputs of the design are not constrained at all by the properties. In practice, under some scenarios (the design states) the outputs are allowed to be under-constrained. The technique allows for easy specification of such "don't care" exceptions. Due to the computational complexity and inherent impracticality in the usage model, neither of the two methods [11, 12] have found acceptance in hardware design verification flows.

One straightforward way to measure the property completeness would be to identify the subset of design logic that is structurally in the fan-in of the collection of properties. This can be achieved by doing structural traversal of the netlist representing the design and the properties. The parts of logic that are not in the recursive fan-in of any of the properties represent an obvious hole in specification. In other words, more properties need to be written to check

these parts of the logic. It should be noted that this method of property completeness coverage measurement can identify most obvious property coverage holes. To compute this structural coverage of a given set of assertions S = {P_0, P_1, …, P_n} for a design D and environment E, we consider the coverage for each property P_i and aggregate the coverage results to form the coverage of S. We denote COI(P,E,D) to be the set of cover items C_i in CI(E,D) that intersect with the COI of property P in design D under the environment E. In short notation, we use COI(P) when D and E are known in the context. Clearly COI(P) is a subset of CI. The union of the subsets COI(P_i) is denoted by COI(S), the *coverage item set* for the specification S.

Of late, design mutation-based coverage measurement has been gaining ground in the area of verification coverage [14]. For a simulation infrastructure, a fault model for the design is applied, and the completeness of the stimulus set and the set of response-checking mechanisms is measured, based on the ability of the verification framework to catch errors resulting from those faults. Similar concepts can be adopted to obtain information about property set completeness in formal verification. First of all, a fault model can be identified, which in turn creates a fault set consisting of all the faults in the design fitting that model. For example, the fault model could be disconnecting a driver from a flop, thus making the flop a free net. The fault set would then consist of design mutations, whereby in each mutation exactly one flop has been disconnected. The property completeness metric can be obtained by checking how many of these faults in this fault set are detected, as indicated by the failure of one or more properties. The faults that are not detected, that is, none of the properties in the environment fail, indicate a hole in the property set. In other words, a user needs to write some additional properties that are sensitive to these faults that represent coverage holes.

VII. Proof Coverage

In this section, we discuss the measurement of the amount of verification performed as a result of either the full proof or the bounded proof for a set of properties. As mentioned in Section IV, these two possibilities may happen depending on the design/property complexity as well as the resources such as compute time and memory applied in the computation.

Full-Proof Coverage: To measure the verification coverage for the properties that are fully proven, we introduce a notion of *Proof Core* of a property. The proof core of a property P is the subset of the logic contained in the COI of the property, and this subset is capable of establishing the correctness of the property. In other words, parts of the design D that are outside the proof core do not impact the correctness of the property P. Computing the proof core is done inside SAT-based, model-checking algorithms using *unsatisfiable core generation* techniques [13]. We denote PC(Pj) is a set of C_i in CI(E,D) that intersects with the proof core of Pj. PC(Pj) is included in COI(Pj). The set PC(S) is the union of all PC(P_j). Similar to sets described in the stimuli coverage section, we define the subsets PC_U, PC_R, and PC_B for unreachable, reachable, and bounded cover items to form the coverage set PC(S). It should be noted that the proof core coverage for the properties that are fully proven takes into account the effect of both the constraints in the environment and the set of properties under consideration. Since it takes into account only those cover items that are found to be essential for ensuring the correctness of the property, this coverage metric is the strongest of all coverage metrics used in simulation and formal verification. The drawback of this metric is that it requires that the property is fully proven, which may not be the case for a complex design and property. Secondly, in some cases, even if the property is fully proven, it may be quite expensive to compute the corresponding unsatisfiable core.

Bounded-Proof Coverage: Due to the mathematically complex nature of formal analysis, it is possible that the computation may hit resource limits, such as time and available memory. Often, an incomplete formal analysis is measured in terms of the number of cycles, with reference to some specific clock, of analysis the design has gone through. For example, "k" cycles of formal analysis establishes that the design has been analyzed for all input sequences of length "k". This statement is true for most common types of formal analysis which traverse the state space of the design in a breadth-first manner.

In another scenario, the formal analysis can be performed under some restricted set of inputs, for example, a subset of inputs could be tied to a constant. This typically happens when a design's validity against the specification has been formally analyzed for a particular mode setting. Another scenario of partial formal analysis is when an arbitrary starting state is used. This precludes the guarantee that the property has been analyzed for all states that the design could attain.

The most accurate way to measure the stimuli coverage for partial formal analysis would be to determine the fraction of all possible reachable state space that has been covered. However, establishing this measure is not practical since determining the exact reachable state space itself is a hard problem.

Similar to the property completeness analysis, we use the COI(S) to represent all the cover items in the COI of the specification S. The subsets COI_R, COI_U, and COI_B for the reachable, unreachable,

and bounded cover items in COI(S), are computed separately using formal analysis. Observing the content of such subsets and the depth of the cover items greatly help to understand the value of the bounded proof. For example, the number of C_i in COI_R(P) within the given property bound can be an indicator of the verification coverage for that property. For example, if P is part of S with bounded proof value *k*, and there are *n* cover items in COI_R(P) that are reachable in bound < *k*, we can use "n" to be the relative indicator of the verification coverage for property P. We can aggregate the verification coverage from all the properties with a bounded proof result by taking the union of these sets.

For bounded proofs, it is interesting to know whether the given bounded proof *k* for a property P in specification S, design D, and environment E is sufficient. In other words, does the property P exercise all the interesting cover items in COI(P)? We use the subsets COI_R, COI_U, and COI_B to analyze this. Clearly, if *k* is higher than all the bounds of the cover items in COI_R and COI_B, then the bound *k* gives very good verification confidence since all the design events manifested by the cover item set are reached within this bound. Note that like any other coverage measurement, reaching 100 percent coverage for a given coverage model is only as good as the quality of the model chosen. On the other hand, if *k* is much smaller than the bounds in COI_R, it implies that the property analysis has been shallow, and therefore, the formal analysis needs to work harder on the property P to achieve a deeper bound. The COI_U can be used as helper properties for proving P to reduce the state space of P by eliminating the unreachable cover items from the search.

A sample bounded proof coverage result for an assertion is illustrated below:

a) 137: Assertion proof bound (# of cycles for which the assertion has been proven correct)

b) 1200: # of cover items in the COI

c) 865: # of cover items in b that are reached within the 137 clock cycles (the proof bound)

d) 230: # of cover items in b that are reached beyond the 137 clock cycles

e) 191: Maximum depth of the cover items reached in d

f) 55: Number of cover items in b that are proven to be unreachable

g) 50: Number of cover items in b whose results are not determined

h) 76%: Bounded proof verification coverage: 865 (# of cover items analyzed in the proof boundary) divided by 1200-55 (effective # of cover items)

To measure the collective bounded proof coverage for a group of assertions, one can take the intersection across all the assertions of cover items that are in bucket c to calculate the overall number for the numerator for the coverage computation. In the same way, the union of the cover items in f across all the assertions should be subtracted from the overall number of cover items to obtain the denominator.

VIII. Leveraging Coverage Information from Simulation and Formal Verification

Since the specific tasks performed with formal verification technology complement the rest of the verification work performed with traditional simulation methods, a question that is often asked is "How does one combine the results from formal verification with those obtained from simulation?" The intent behind this question is to eliminate any duplicate verification effort between these methodologies and improve overall verification productivity. This can be analyzed in a top-down view as well as a bottom-up view.

In the top-down view, simulation and formal verification can be leveraged in a systematic way as dictated by a verification plan. Effective verification planning adds predictability to the verification flow by specifying exactly what needs to be tested. Planning also provides the necessary structure to identify key areas where complementary verification technologies can be applied effectively. By ensuring that the verification tasks are tied to the specification and assigning an appropriate verification method (simulation or formal analysis) to each task, one can significantly reduce, if not eliminate, any redundant verification effort.

In the bottom-up view, formal analysis and simulation methods can cooperate in multiple ways to help improve overall coverage.

The first flow would be to facilitate engineering productivity in dealing with unreachable coverage targets. Traditionally, design and validation engineers spend significant time addressing coverage holes. The validation engineers tweak the testbench in an attempt to steer the simulation towards the necessary coverage targets. The design engineers either provide necessary hints for that steering effort or waive the coverage hole as harmless, in other words, the coverage hole is an artifact of the design functionality. Formal technology can greatly facilitate this manual effort. In particular, formal analysis can establish that a subset of coverage holes is caused by unreachable coverage targets — the targets that are not reachable for any legal input sequence. This eliminates unnecessary simulation effort targeting those areas. In addition,

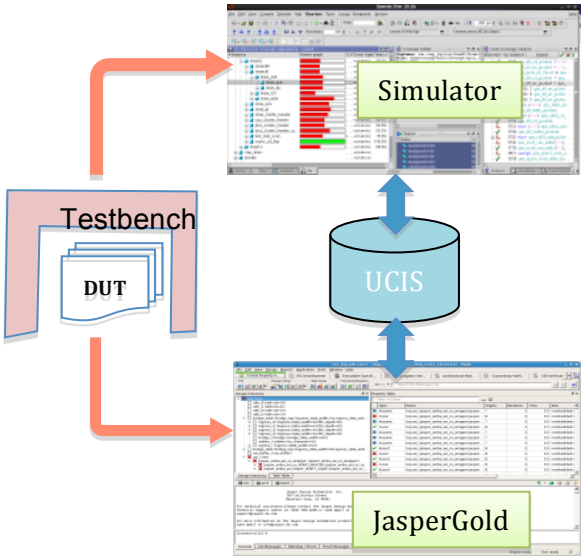
formal analysis can create specific scenarios automatically, which will fulfill specific coverage targets when simulated.

A second flow where formal analysis can reduce the simulation effort is where a design block is fully verified formally for a specific mode, perhaps the most complex one. The simulation effort can then be targeted to provide coverage in untested modes.

A third flow would be to leverage the coverage information for the fully and partially proven properties, as described in Section VII. The coverage information from formal analysis can be stored in a database that can be merged with the coverage information from simulation methods. In this use model, the information about the design sensitization from simulation and formal analysis is combined to achieve higher overall coverage. However, some technical caveats must be mentioned here for this particular flow. As described in the introduction, the verification coverage measurement is dependent on the infrastructural components of the verification environment. Typically simulation environments deploy bus functional model (BFM) based transactors for stimuli generation. This is in sharp contrast to the declarative assumptions used in formal analysis to limit the scope of legal stimuli applied to the DUT. On the response checking side, while the mechanism between simulation and formal verification may overlap — assertions being used in both formal and simulation — in practice, the checkers used in simulation are not directly usable in formal analysis. This may be due to either the non-synthesizability of the checkers or to the fact that the checkers are tied to the transactors. The bottom line is that because of these infrastructural differences, the context of the coverage measurement is different between formal verification and simulation. This should be kept in mind when combining the coverage numbers to get overall verification confidence.

IX. ACCELERATNG COVERAGE CLOSURE WITH UCIS API

In June 2012, Accellera's Unified Coverage Interoperability Standard (UCIS) sub-committee published the first draft of the coverage API standard. The standard API enables the interoperability of coverage information from heterogeneous sources. There are some limitations to the current standard specifically in the area of standardization around the definition of various coverage metrics that makes the uniform interpretation of all coverage data a bit difficult [8]. Assuming that the coverage metric is well-defined, a formal tool can query a coverage database supporting UCIS API that is populated by data from a simulation tool. The ability to read/write information into a coverage database has enabled the use models discussed in the previous section. More specifically, a formal tool can automatically identify the coverage items that are not hit in simulation, and after performing appropriate analysis, automatically update the database with information about unreachable coverage items. In addition, the coverage information for the fully proven properties as well as that for the bounded proofs can be populated in the same database that is used for simulation. Appropriate coverage merger techniques can be used to combine the metrics information from the two sources to increase the overall coverage and accelerate the coverage closure process. A conceptual representation of this flow is shown in the figure below.



X. CASE STUDY

We have applied the concepts addressed in this paper on a number of industrial designs from different industry segments. For these cases, a mixture of statement and branch coverage models was used. We provide the design statistics for five designs below in Table 1. We also provide a qualitative analysis of a coverage flow on design 5, including (a) property completeness coverage, (b) stimuli coverage, and (c) bounded proof coverage and debugging.

TABLE I. DESIGN STATISTICS

Design	*Flops*	*Latches*	*Gates*	*Nets*	*Cover Items*
1	5247	631	387409	797567	58606
2	3988	3671	156925	339872	24192
3	1084	222	113554	199478	20765
4	7732	0	341205	772132	62117
5	19892	0	414379	392794	38633

Design 5 is a cache-coherent interconnect, which is common to many SoC designs. For the rest of this

section, we have chosen this design (D) for analysis purposes. The coverage model (M) chosen for this design is statement coverage and defines the coverage items (CI) for further analysis. In addition to the design and coverage items, the formal verification environment also contains 362 assertions, which define the property set (S), and 506 assumptions, which contribute to defining the environment sets (E).

As described earlier, analyzing the COIs of a set of properties, COI(S), can be an indicator of property completeness coverage. More specifically, cover items not found in COI(S) or in only a small number of COI(P) can be an indicator of holes in the property set. For design 5, as shown in Figure 1, of the 38,633 cover items, 4,655 were outside the COI of any of the properties in S.

Figure 1 Out of COI Report

```
[<embedded>] % check_coverage -report -type outcoi -no_return -task <embedded>
|++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++|++++++++++++|
| Coverage Report: Out of COI                                                          |
| Task Name: <embedded>                                                                |
|++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++|++++++++++++|
| Cover Item                                                             | Cover Item |
| Name                                                                   | ID         |
|++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++|++++++++++++|
| i1.u_xt_iot._automatic_coveritem_stmt_condition__blocking_assignment__0 | 89         |
| i1.u_xt_iot._automatic_coveritem_stmt_condition__blocking_assignment__1 | 90         |
...
| u_war2._automatic_coveritem_stmt_condition__nonblocking_assignment__715 | 38631      |
| u_war2._automatic_coveritem_stmt_condition__nonblocking_assignment__716 | 38632      |
|++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++|++++++++++++|
| Total number of Cover Items: 4655                                                    |
|++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++|++++++++++++|
```

Further analysis also revealed that nearly 10,000 cover items were in the COI of five or fewer properties. Almost all of these cover items involved the flow of data passing through the interconnect. Clearly, the control functionality of this design is covered much better by the property set than the data functionality.

Both the out of COI(S) and low COI(P) count metrics represent holes or potential holes in the property set for design 5. The typical reaction to these holes is to write more properties — relevant for logic — or pass the verification burden on to simulation or other higher levels of verification where other properties may have the cover items in their COI.

To summarize, the property completeness metrics identified areas of the design which were either not being verified or not being well verified by the property set. This information is useful in guiding engineers to the areas of the design which need more properties written or verification focus in other verification environments.

Next, to ensure the validity of the formal verification environment constraining this design, stimuli coverage analysis was performed. Recall that to obtain information about the impact of constraints on the stimuli coverage, a baseline environment and reachability of cover items in that baseline environment need first to be established. For this design, the baseline environment consisted of a clock definition, a reset definition, constraints to tie off scan functionality, and constraints to tie off other unused ports. The reachability results, CI_R, CI_U, CI_B, for the baseline environment, E_0, are shown below in Table II. Note that in the baseline environment (without any explicit constraints), there were 1374 unreachable cover items.

TABLE II. E_0 BASELINE REACHABILITY

Design	***Total Cover Items***	***Reachable Cover Items***	***Unreachable Cover Items***	***Undetermined Cover Items***
5	38633	36726	1374	533

In the target environment with a full constraint set, where the formal verification of the assertion set was performed, there were 506 assumptions that provide many additional constraints on the environment. The reachability results for this environment, E_1, are shown below in Table III.

TABLE III. E_1 TARGET REACHABILITY

Design	***Total Cover Items***	***Reachable Cover Items***	***Unreachable Cover Items***	***Undetermined Cover Items***
5	38633	34804	1802	2027

The data shows that 428 more cover items became unreachable due to the additional constraints in E_1 vs. E_0. Also, 1,494 more cover items became undetermined. For the unreachable items, further merging of the data was done to show that there are actually only 77 more unique cover items. Figure 2 shows the relevant portion of the report.

Figure 2 Unreachability Difference Reports

```
[<embedded>] % check_coverage -report -type task -no_return -task <embedded> -reference BASELINE
|++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++|++++++++++++|++++++++++|
| Coverage Report: Unreachable                                                         |
| Task Name: <embedded>                                                                |
|++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++|++++++++++++|++++++++++|
| Cover Item                                                   | Cover Item | Caused   |
| Name                                                         | ID         | By       |
|++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++|++++++++++++|++++++++++|
| automatic_coveritem_stmt_condition__blocking_assignment__113 | 5888       | X        |
| automatic_coveritem_stmt_condition__blocking_assignment__127 | 5904       | X        |
...
| _automatic_coveritem_stmt_condition__blocking_assignment__21 | 35277      | X        |
| automatic_coveritem_stmt_condition__blocking_assignment__297 | 35371      | X        |
|++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++|++++++++++++|++++++++++|
| Total number of Cover Items: 428                                                     |
|++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++|++++++++++++|++++++++++|
[<embedded>] % check_coverage -report -type task -no_return -task <embedded> -reference BASELINE -merge
...
|++++++++++++++++++++|++++++++++++++++++++++++++++++++++++++++++|++++++++++++|++++++++++|
| Total number of Cover Items: 77                                                        |
|++++++++++++++++++++|++++++++++++++++++++++++++++++++++++++++++|++++++++++++|++++++++++|
```

Upon analyzing the 77 additional unreachable cover items, most of them were determined to be due to the known artifact of one or more constraints. However, a few of the unreachable cover items could not easily be explained and further analysis was needed. To debug a cover item that is unreachable, a good approach is to find a minimal subset of assumptions that can cause the unreachability. A manual process of determining this minimal set would be tedious. (You would need to iterate over subsets of 506 assumptions.) Instead, the formal verification tool was able to determine automatically the three specific assumptions that caused the unreachability of a particular cover item. After reviewing the assumptions, a subtle over-constraint was detected,

which essentially disallowed two parts of a transaction to enter the design in different orders, that is, the parts were forced to always enter the design in the same order. In other words, any assertion verified within this formal environment would not have been tested for the disallowed scenario above.

In summary, the stimuli coverage analysis effectively tracked the impact of constraints on the formal verification environment for design 5. It highlighted the design parts both purposely and mistakenly not being stimulated. This analysis was able to identify a corner-case constraint problem that would have otherwise gone undetected

The final area of analysis is of the bounded proof coverage data on design 5. This analysis is focused on the assertions for only one of the design's interfaces. The interface had 43 undetermined assertions. Figure 3 below is the Bounded Proof Coverage report for the targeted interface.

Figure 3 Bounded Proof Coverage Report

```
[<embedded>] % check_coverage -report -type bounded -no_return -task <embedded>
|++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++|++++++++|++++++++|++++++++|++++++++|++++++++|++++++++|
| Coverage Report: Bounded                                                                                          |
| Task Name: <embedded>                                                                                             |
|++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++|++++++++|++++++++|++++++++|++++++++|++++++++|++++++++|
| Assert                                                       | Assert | Total  | Unconf | Max    | Undet. | Max    |
| Name                                                         | Bound  | in COI | Covers | Bound  | Covers | Bound  |
|++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++|++++++++|++++++++|++++++++|++++++++|++++++++|++++++++|
| ded>::cci_tb.ace4pk_m4.AST_S_ACADDR_valid_values_DVMComplete | 16     | 23760  | 0      | 0      | 1376   | 68     |
| <embedded>::cci_tb.ace4pk_m4.AST_S_ACSNOOP_valid_values      | 15     | 23760  | 1      | 16     | 1376   | 68     |
| ::cci_tb.ace4pk_m4.AST_S_ConditionallySuppressDVMComplete_ac | 14     | 23760  | 3      | 16     | 1376   | 68     |
| <embedded>::cci_tb.ace4pk_m4.AST_S_EXOKAY_WriteResponse      | 11     | 24019  | 115    | 16     | 1385   | 68     |
| <embedded>::cci_tb.ace4pk_m4.AST_S_Barrier_WriteResponse     | 11     | 24019  | 115    | 16     | 1385   | 68     |
| <embedded>::cci_tb.ace4pk_m4.AST_S_Sequencing_AC_BRESP       | 11     | 23760  | 114    | 16     | 1376   | 68     |
| d>::cci_tb.ace4pk_m4.AST_S_SuppressBVALIDunlessValidAndReady | 10     | 23760  | 203    | 16     | 1376   | 68     |
| <embedded>::cci_tb.ace4pk_m4.AST_S_Suppress_PassDirty        | 10     | 23775  | 203    | 16     | 1378   | 68     |
| <embedded>::cci_tb.ace4pk_m4.AST_S_Suppress_SharedDirty      | 10     | 23799  | 203    | 16     | 1379   | 68     |
| >::cci_tb.ace4pk_m4.AST_S_DVMmessage_ReadResponse_secondpart | 10     | 23760  | 203    | 16     | 1376   | 68     |
| <embedded>::cci_tb.ace4pk_m4.AST_S_SuppressRVALIDunlessValid | 9      | 23760  | 355    | 16     | 1376   | 68     |
| <embedded>::cci_tb.ace4pk_m4.AST_S_Suppress_IsShared         | 9      | 23784  | 355    | 16     | 1377   | 68     |
...
| <embedded>::cci_tb.ace4pk_m4.axi4pk.AST_S_RDATA_stable       | 9      | 33743  | 1083   | 16     | 2664   | 68     |
| >::cci_tb.ace4pk_m4.axi4pk.jg_amba_axi4_abs.AST_S_RLAST_late | 8      | 23760  | 596    | 16     | 1376   | 68     |
| ::cci_tb.ace4pk_m4.axi4pk.jg_amba_axi4_abs.AST_S_RLAST_early | 8      | 23760  | 596    | 16     | 1376   | 68     |
|++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++|++++++++|++++++++|++++++++|++++++++|++++++++|++++++++|
| Total number of Bounded Assertions: 43                                                                            |
|++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++|++++++++|++++++++|++++++++|++++++++|++++++++|++++++++|
```

In this report, the column "Total in COI" shows the COI(P) for each property on the interface. The column "Unconf Covers" indicates the number of C_i in COI(P) that are reachable only beyond the bound of the property. These cover items are considered unconfirmed, implying that the assertion has not been analyzed in the context of the events represented by these cover items. The unconfirmed cover items are potential holes in the verification bound of the property. The column "Undet. Covers" indicates the number of C_i in COI(P) that are undetermined, meaning they have not been shown to be reachable or unreachable. Again, the undetermined cover items could represent holes in the verification bound of the property.

Some additional pieces of interesting information can be gleaned from the report. It appears that many of the properties have similar COIs. For example, all but one property has about 24,000 cover items in its COI. The outlier property has about 34,000 cover items in its COI. Recall from the property completeness analysis that there were nearly 10,000 cover items in the data paths that were in the COI of fewer than five assertions. This property is one of that small number of data path properties.

The data also reveal that attempting to increase each assertion bound to 14 or higher seems like a good target. The assertions with a bound of eight have nearly 600 unconfirmed cover items, whereas we see a rapid decrease in the number of unconfirmed targets (down to single digits) for the assertion at a bound of 14.

A user could react to the holes in the bounded proof coverage data in many ways:

a) Run the formal engines longer on the target assertions to increase the proof bounds.

b) Run the formal engines longer on the target cover items to move them from undetermined to covered.

c) Create appropriate abstractions for the design elements to decrease the sequential depth of the design. This will enable the cover items to be reachable at smaller depths.

d) Create a new environment with targeted over-constraints to help increase the target assertion bounds. This is akin to obtaining higher coverage for a restricted mode of operation for the design.

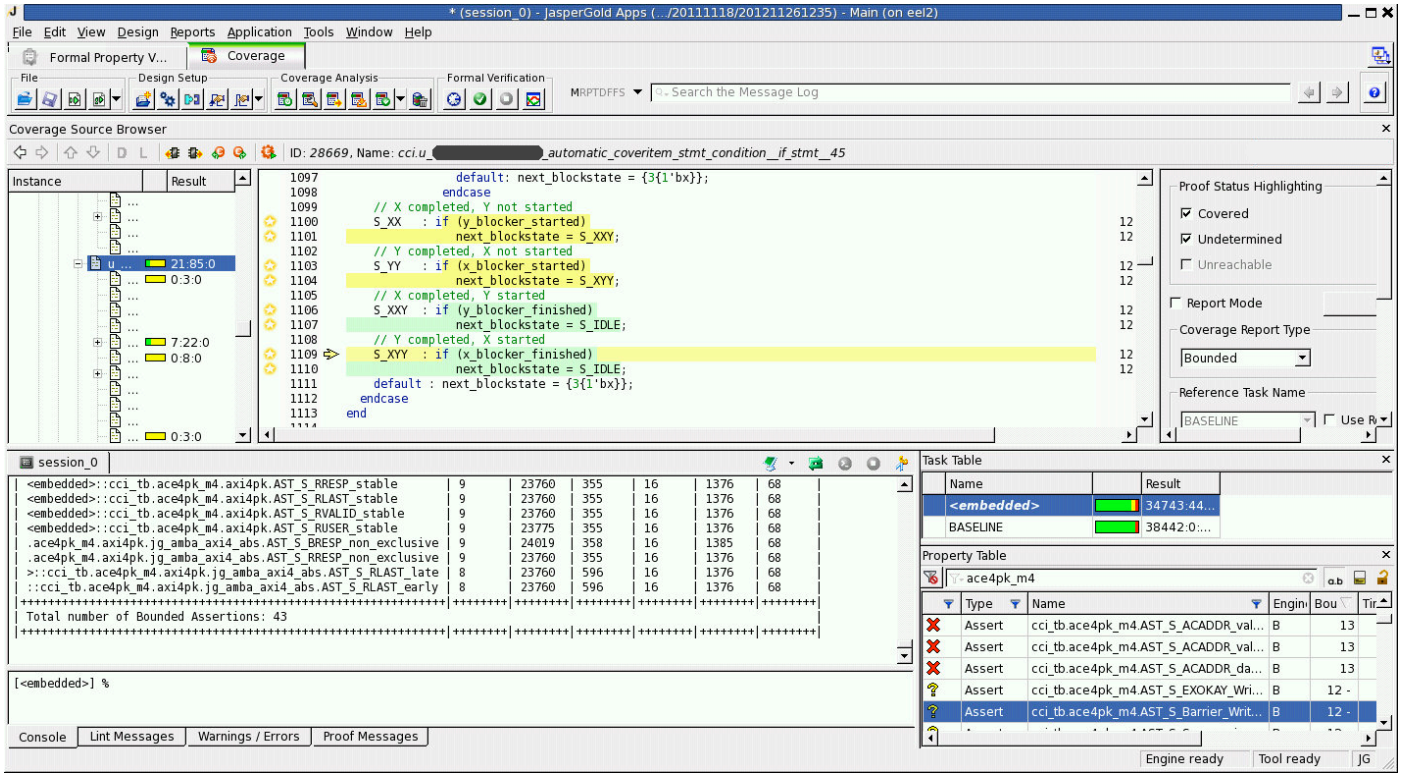
e) Pass the verification burden of the unconfirmed and undetermined cover items on to simulation.

On design 5 we have seen that when enhanced with bounded proof coverage data, there is much more meaning to the assertion bounds. The coverage data identifies the obvious holes in the proof bounds and reveals bound targets to help determine how to best plug those holes.

It should be noted that the coverage data presented in this case study is often much easier to generate, understand, and traverse when using a GUI dedicated to extracting and displaying the data compared to the textual reports shown in various figures so far. Instead of trying to use GUI snapshots throughout the case study, a single screenshot is displayed here at the end in Figure 4. Convenient access to report widgets, source code highlighting of reported cover items, hierarchical summaries of the reported data, prev/next buttons to traverse through reported items, and so forth, are all very important for efficient processing and understanding of the data.

While some additional implementations in this area are underway, the quantitative coverage results across designs from multiple industry segments and the result navigation and debugging process using the GUI have generated significant amount of interest in the end-user community.

Figure 4 Coverage GUI



REFERENCES

[1] Jasper Design Automation. Coverage: Achieving Coverage Closure with Jasper Formal. http://www.jasper-da.com/coverage_closure_seminar.

[2] John Goodenough. Jasper Formal Verification at ARM. DVCON 2011. http://www10.edacafe.com/video/ARM-Jasper-Formal-Verification-ARM-brJohn-Goodenough/34449/media.html.

[3] Yatin Hoskote, Timothy Kam, Pei-Hsin Ho, and Xudong Zhao. 1999. Coverage estimation for symbolic model checking. In Proceedings of the 36th annual ACM/IEEE Design Automation Conference (DAC '99), Mary Jane Irwin (Ed.). ACM, New York, NY, USA, 300-305.

[4] C.-N. Liu and J.-Y. Jou. Efficient coverage analysis metric for HDL design valida- tion. In Proceedings of IEEE International Conference on Computers and Digital Techniques, pages 1–6, January 2001.

[5] Hana Chockler Orna Kupferman Moshe Y. Vardi Coverage Metrics for Temporal Logic Model Checking. 528-542 2001 conf/tacas/2001 TACAS.

[6] Rajeev Ranjan, Jay Littlefield, Brian Bailey. Understanding coverage with multiple verification methods http://jasper-da.com/sites/default/files/pdfs/Understanding_coverage_multiple_verification_methods_Nov2007.pdf

[7] Rajeev Ranjan, Brian Bailey. Combining Metrics from Simulation and Formal. http://www.soccentral.com/results.asp?entryID=26340

[8] Richard Ho, Ambar Sarkar, Mike Burns, Rajeev Ranjan. Coverage Interoperability and Unification: What can be expected from the Accellera Unified Coverage Interoperability Standard? Electronic Design, September 2010.

[9] Andrew Piziali. Functional Verification Coverage Measurement and Analysis. Springer Publishing.

[10] Rajeev Ranjan. We need a simpler and faster approach to formal verification. http://www.eetimes.com/discussion/other/4391414/We-need-a-simpler-and-faster-approach-to-formal-verification

[11] H. Chockler, O. Kupferman, and M. Y. Vardi. Coverage metrics for formal verification. Correct Hardware Design and Verification Methods (CHARME), pages 111-125, 2003.

[12] Classen, Koen, A Coverage Analysis for Safety Property Lists, Workshop on Designing Correct Circuits (DCC), March 2006, http://citeseer.ist.psu.edu/587670.html, http://www.cs.chalmers.se/~koen/pubs/dcc06-coverage.pdf

[13] Nachum Dershowitz Ziyad Hanna Alexander Nadel, A Scalable Algorithm for Minimal Unsatisfiable Core Extraction. 36-41 2006 SAT Conference

[14] Certitude tool from SpringSoft. EDA company. http://www.springsoft.com/products/functional-qualification/certitude