An Assertion Based Approach to Implement VHDL Functional Coverage

Susan Eickhoff, Michael DeBole, Michael Wazlowski
IBM Corporation
Systems and Technology Group
Poughkeepsie, NY 12601, USA
{srubow, mvdebole, mew}@us.ibm.com

Tagbo Ekwueme-Okoli
Cadence Design Systems
Systems and Verification Group
Cary, NC 27518, USA
tokoli@cadence.com

***Abstract*— This paper describes a process developed to leverage VHDL functional coverage in a mixed cycle and event driven simulation flow. The VHDL functional coverage is defined using an extension to VHDL. During functional simulation, functional coverage statistics are collected and maintained. A script is used to convert the combined simulation functional coverage statistics into an XML format using a predefined XML schema. At a high level, the combined functional coverage is made to look like covergroups. Finally, an Accellera Unified Coverage Interoperability Standard (UCIS) conversion utility is used to convert the generated XML into a coverage database. This conversion allowed for the use of the coverage database in industry available coverage and Verification Planning and Management tools. This paper will address the challenges resolved in creating this flow including mapping the functional coverage generated to an industry standard data model. This paper will also present future challenges and opportunities.**

Keywords—functional verification; functional coverage; Coverage Driven Verification; Metric Driven Verification; Verification Planning and Management; Unified Coverage Interoperability Standard; UCIS; VHDL;

I. INTRODUCTION

The micro-electronics industry’s continuous decrease in time-to-market requirements and simultaneous increase in design complexity have necessitated enhancements to traditional functional verification methodologies. One such enhancement is the use of coverage driven verification (CDV). In recent years the use of functional coverage has become the dominant method for managing the verification process for Intellectual Property (IP) level to System on Chip (SoC) level development, using SystemVerilog-based methodologies such as the Universal Verification Methodology (UVM). In Verilog or VHDL however, direct language support for functional coverage does not exist. Specifically for VHDL, the addition of functional coverage is often not enough motivation to move to SystemVerilog as VHDL is a very rich and robust design and verification language. There are many industries (military), design types (Field Programmable Gate Array (FPGA)) and geographies (Europe) where the language of choice is VHDL. For those industries where custom methodologies or legacy processes already exist (such is the case with the verification methodologies used by IBM) the overheads and costs of conversion are prohibitive.

IBM verification and design teams use custom tools and languages for functional coverage. This functional coverage is used increasingly as a metric to drive decision making and schedules, guide resource allocation, and achieve verification closure. With the increases in design complexity, Verification Planning and Management, and the tools and processes to enable it become increasingly important. However, the creation and maintenance of the tools and processes for Verification Planning and Management can consume large amounts of time. This time often comes from the project verification leads whose attention is diverted from more important verification tasks. The addition of various features, like the filtering of information to present the appropriate view of data to relevant stakeholders can be time and resource consuming. IBM developed a solution to this problem by leveraging Accellera’s Unified Coverage Interoperability Standard (UCIS) [1] to allow the use of commercial Verification Planning and Management (VPM) tools for internally generated functional coverage data. This paper describes how to implement VHDL functional coverage in a usable and efficient way without using SystemVerilog, with the simplicity and low overhead of simple assertions. This paper also describes the creation of a process to convert that functional coverage data so that it can be used in a commercially available Verification Planning and Management tool. This paper will be of interest to designers of large SoCs who believe in functional coverage and its benefits, but who do not currently have access to SystemVerilog and are not willing to change their VHDL code

This paper is organized as follows. Section I introduces the topic, Section II provides background information, Section III presents an overview and the implementation details of the process, and finally, Section IV summarizes and concludes the paper.

II. Background/Prior Work

A. Functional Coverage

"Are we done yet? No…….. Well, when will we be done then?" These two questions drive most functional verification schedules. Functional verification entails changing the state of a logic design and measuring that the response generated by the design is correct. Modern verification environments change the state of designs by driving constrained random inputs. The power of these constrained random inputs is that input vectors are automatically generated without having verification engineers spell out each set of input vectors manually. With this power also comes great responsibility. With the constrained random inputs come constrained random changes in state. In this environment it is imperative that the verification team be able to specify which states have been explored and that for those explored states, the correct response to input vectors has been observed. One of the first challenges in this approach, and in functional verification in general, is the size of the state space [2]. For a design with *n* state elements, the number of states to be explored is 2^n [3]. Even for a small design, the extent of this space can quickly become daunting. For instance, with a simple design consisting of a single 32-bit counter, the state space explodes to a whopping 2^{32} or 4294967296 states. Exploring the state space of today's designs without some method of recording the states which have been explored is akin to travelling across the continental United States without signs or a map! The size of the state space also necessitates a prioritization of features and testing in order to complete verification in a reasonable amount of time. Functional coverage is defined for and captures scenarios that are important for the verification environment to observe and capture for later analysis [4]. As such, functional coverage serves as a guide in the process of determining the state of the verification effort and when that effort has been completed to satisfaction of the relevant stakeholders.

B. Unified Coverage Interoperability Standard

The Unified Coverage Interoperability Standard (UCIS) is an Accellera standard for defining and standardizing the interface to coverage data. The purpose of UCIS is to have a common interface which users can incorporate into their host applications for collecting coverage across any UCIS compliant tool, whether that be simulation, acceleration, formal, system level tool, or other verification engines [UCIS reference?]. UCIS can also be used as a way to interface between vendors proprietary coverage data formats, however the level of interoperability and the feature set defined only partially address interoperability. Since UCIS can provide a unified view of coverage data and metrics, it allows users to generate a more complete view of their coverage data. This enhanced view of coverage provides an improved snapshot of the state of the verification effort and can help speed verification closure.

The UCIS 1.0 specification has defined two types of formats for coverage data information exchange by users – an Application Programming Interface (API) data access mechanism, and a XML Interchange Format [UCIS reference]. Either of these formats can be used but each provides its own benefits and challenges. The API data access method serves as an abstraction layer to the physical database and allows users to build applications to access a UCIS database for analysis or reporting. The XML Interchange Format data access method involves writing out coverage data information into the XML-based Interchange Format. This XML file can then be accessed by any UCIS-compliant or other XML-based tool [5].

The UCIS 1.0 specification has defined a common data model for a consistent interpretation and utilization of the coverage data created by various coverage data producers by a variety of consumers. This model has been designed to be flexible and extensible while maintaining the requirements for universally recognizable information storage. The UCIS 1.0 specification defines structures as part of the coverage data model. Some of the structures within this data model are as follows:

- Scopes to create the hierarchical structure that describes the design. These scopes may or may not contain child scopes. These scopes are analogous to the design hierarchy while extended to include coverage constructs that may or may not exist in the HDL design [6].
- Attributes which serve as data decoration elements for scopes and coveritems [8]. This includes both static information such as the filename as well as dynamic information such as thresholds.
- Coveritems to hold the counts of recorded events, which are used to compute coverage [7]. The coveritem is an integral count, decorated with enough information to describe what was counted.

To present a consistent interpretation of the information stored in each coveritem, the UCIS specification describes an abstract theory of coverage. For this abstract theory of coverage the collection of coverage data by the tool producing the data has the form:

$$\textit{@event if(condition) event\_counter++} \qquad (1)$$

Essentially, at an event of interest, *event*, if a particular condition of interest, *condition*, is satisfied, then a counter, *event_counter*, is incremented. Examples of the event of interest include a variable value change, a clock edge, or a finite state machine (FSM) transition. Examples of conditions that may be tested include sequences of variable values and start and destination FSM states.

C. Bugspray

BugSpray is an IBM internally developed VHDL extension used for functional coverage and assertions. [9] BugSpray is used by both design and verification engineers to add coverage and assertion verification objects to the RTL. BugSpray allows these verification objects to be portable across verification engines (formal, simulation, acceleration) and enables hierarchical design reuse. BugSpray allows the designers to define interesting events and scenarios that, based on their knowledge of the design, need to be covered for verification completeness. As such, it serves as a metric for measuring the quality of the input vectors driven into the design. [10]. For instance this BugSpray statement

[count; event.event_name_0 ; clk] : (M1,unit,chip) <= signal_name_d(0) AND NOT signal_name_q(0) ;

defines a BugSpray count event, *event.event_name_0*. This event is evaluated upon a change in signal *clk*. The count for this event is incremented when at a change in *clk*, *signal_name_d(0)* is asserted and *signal_name_q(0)* is deasserted. The complexity of BugSpray statements can range from the simple combinatorial statement specified above to coverage of a complex FSM as illustrated in Fig 1 below. Fig. 1 describes a state machine with 12 states. In the BugSpray FSM coverage statement the state vector variable is defined and the states and their encoding are specified. Finally, the transition arcs are defined for coverage to be collected on the state transitions.

D. Current Typical Analysis and Usage Flow

Design engineers leverage their knowledge of both the functionality and implementation of the design to create BugSpray assertions and evaluate coverage metrics. These assertions and coverage metrics are used to grade the robustness of the verification stimulus and the progress towards verification completeness. Verification engineers can also add BugSpray coverage and assertions to augment the verification environment. Various types of BugSpray events can be defined, however, in simulation the most commonly used are events that provide counts of whatever particular logic has been exercised. These events typically follow two main categories:

- Mainline logic count events: Counts which indicate various functional logic paths that have been exercised.
- Error logic count events: Counts which indicate that logic has been exercised in the event of an error scenario (e.g. ECC, parity or some other error detection/correction algorithm)

During simulation, the BugSpray assertions and coverage events are monitored to identify if the defined events or sequences occur. If so, the events are counted and logged. Event statistics from each simulation are maintained and sent

```
Bugspray FSM: fsm.fsm_state1, clk_event
            : (M1, unit, chip);
    state_vector    := ( w_fsm_stm_q(10
                downto 0));
    states          := ( s00, s01, s02,
s03, s04, s05, s06, s07, s08, s09, s10,
                   s11);
         state_encoding := ( s00 :=
               '00000000001',
               s01 := '00000000010',
               s02 := '00000000100',
               s03 := '00000001000',
               s04 := '00000010000',
               s05 := '00000100000',
               s06 := '00001000000',
               s07 := '00010000000',
               s08 := '00100000000',
               s09 := '01000000000',
               s10 := '10000000000',
              s11 := '00000000000' );

  arcs            := ( s00 => s00, s10 =>
 s00,    s11 => s00, s00 => s01, s01 =>
s01, s01 => s02, s02 => s02, s10 => s02,
   s11 => s02, s02 => s03, s03 => s03,
s01 => s04, s05 => s04, s11 => s04, s01
 => s05, s05 => s05, s11 => s05, s05 =>
  s06,   s01 => s07, s07 => s07, s08 =>
       s07, s09 => s07, s11 => s07,
      s07 => s08, s08 => s09, s09 => s09,
      s09 => s10, s00 => s11, s01 => s11,
s02 => s11, s03 => s11, s04 => s11, s05
 => s11, s06 => s11, s07 => s11, s08 =>
s11, s09 => s11, s10 => s11, s11 => s11
                     );
                 end FSM;
```

Figure 1. BugSpray coverage statement for a Finite State Machine

to a central server at the end of simulation. These statistics are maintained across simulations and reports are made available upon request. Engineers can then review the reports to measure and track coverage. The BugSpray report provides a well defined text format for conveying the collected functional coverage information stored on the central server. Coverage information is presented based on both the design hierarchy and individual design unit types.

E. Room for Growth

As the team's needs and desires change, it is important that the desired solution be flexible enough to change as well. The addition of other types of data, such as source and line information for improved assertion and coverage debug and analysis is an example of such a desire. In addition, while the current solution provides some automation, it is automation built and maintained by the verification leads. The maintenance of the existing automation and the addition of features has to be performed by the same verification lead and takes time away from more valuable verification tasks such as trend analysis or methodology definition and refinement. For this reason, among others, it is important to leverage existing industry available solutions wherever possible. Using these types of solutions not only alleviates the tools development and maintenance burden on the verification leads but enables the team to take advantage of industry advances as they arise in the future.

F. *Support the Existing Flow*

The requirements for any new approach are as follows. There is an ongoing need to support the continued use of legacy BugSpray assertions. New BugSpray assertion development also needs to be supported as BugSpray is an important part of the verification flow with multiple benefits. The use of both PSL and BugSpray in the same modules/entities also needs to be supported as design and verification teams need to have the flexibility to adopt either or both as their requirements change. Support for mixed cycle and event simulator environments is important as IBM continues to leverage its cycle simulator for its performance benefits.

III. METHOD

A. *Process Overview*

The conversion involves obtaining the combined coverage statistics from the BugSpray server in the form of a BugSpray report. The report is then parsed and an UCIS compliant XML file is generated. Another utility is then used to generate a coverage database from the coverage data contained in the XML file. This coverage database is then loaded into Incisive vManager for review and analysis. The overall process is presented in Figure 2

B. *Selection of the Appropriate UCIS Interface*

While the UCIS 1.0 specification attempts to address coverage mechanics across heterogeneous tools [11], what has been defined thus far is insufficient for complete interoperability. The two types of formats defined within the UCIS document – the API mechanism, as well as the simpler exchange mechanism using XML offer different restrictions and opportunities. The API mechanism provides for the creation of user defined types and operations. The API also supports the use of limited random data access in the in-memory or read-streaming access modes. However the functions contained in the API necessitate the provision and compilation of both source and destination vendor library files. The UCIS 1.0 specification provides a World Wide Web Consortium (W3C) XML compliant schema to represent the coverage data in XML format [12]. The XML approach makes it fairly easy to manipulate and translate formats, including importing and exporting information, which for our application is all that is needed.

In this process the XML approach was used for ease of implementation and use. With the XML interface, we were able to easily prototype and test the flow without changes to any software library files. The XML file is also human

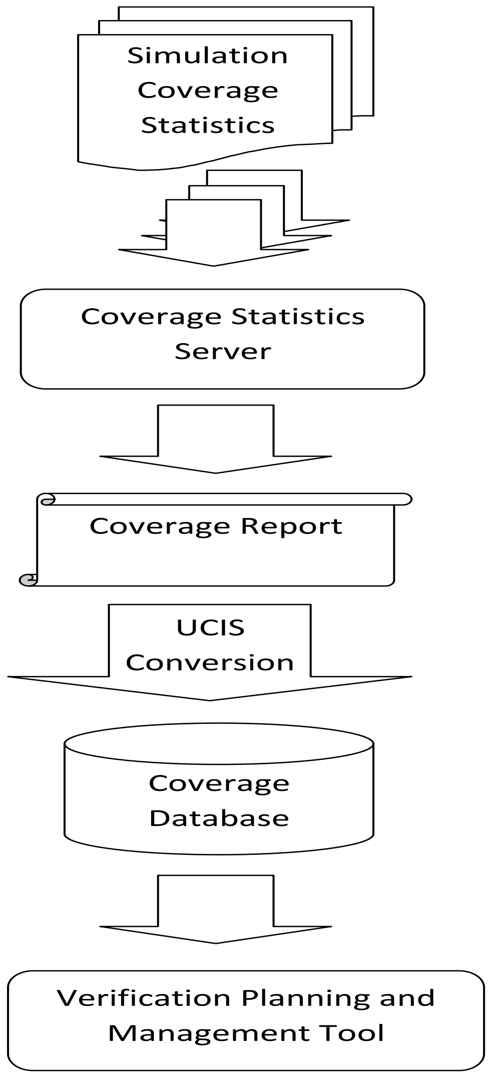

Simulation Coverage Statistics
Coverage Statistics Server
Coverage Report
UCIS Conversion
Coverage Database
Verification Planning and Management Tool

Figure 2 Simulation Functional Coverage Data to Coverage Database Conversion Flow

readable and serves as an additional source of documentation. As stated above, the UCIS was developed to enable coverage metric exchange between a heterogeneous system of coverage producers and consumers. In this case the producer is the IBM proprietary cycle based simulation regression and the consumer is Incisive vManager.

The UCIS XML exchange format provides a generic common coverage data model which supports access to the coverage data from the producer and the ability for the consumer to understand it within the context of the design. The XML exchange mechanism provides naming and mapping conventions to facilitate this consistent exchange and interpretation of coverage data between the simulation coverage statistics server and Incisive vManager.

C. *Conversion of Coverage Data to XML format*

Each BugSpray event line in the report is parsed and separated into its components. The Hierarchy section is split into its individual instance names. A hierarchy is built utilizing the instance names provided by the hierarchy section. The parsing solution script can support BugSpray properties specified in the report in a non hierarchical fashion. For each terminating level of hierarchy, the design entity and variable class information is stored. Within each variable class, the tag(s), variable name, and counter value information is stored. In the case where multiple variable names exist for a given variable class for a given level of hierarchy, the variable names, tag(s) and counter value information are added to the existing variable class. In the case where multiple variable classes exist for a given level of hierarchy, the new variable class, with associated variable name(s) and counter value(s), is added to the existing level of hierarchy. Once this tree structure is completely built, the XML file is generated. Figure [5] shows an example XML file with the associated hierarchies and coverage information.

D. *Data Mapping*

Initially, we decided to map the BugSpray assertions and coverage to PSL assertions and cover properties due to the absence of coverage crosses. In addition, the existence of PSL to BugSpray conversion tools like IBM's Formal Checkers (FoCs) tool led to this thinking. [13] However, BugSpray allows for the definition of classes of event counter variables. These classes enable multiple event counter variables of the same name at the same level of hierarchy, as long as they are defined as part of different variable classes. BugSpray also allows for the definition of tags, which serve as comments. The existence of BugSpray variable classes and tags drove the mapping in another direction, to mapping to the UCIS covergroup construct. This data model was more consistent as this allowed us to take advantage of the covergroup construct's names and comments in the mapping.

E. *The Data Model and Rationale for Data Mapping Decisions*

For the purposes of creating a common reference point for the multiple producers and consumers of coverage data, the UCIS defines a common information model which specifies that the data contained in a coverage database is a collection of counts that have meaning with respect to the design. A simple example of the coverage data model is illustrated in Figure 3. The basic concept is that these counts are integral numbers with enough associated information to make them understandable in the context of the design [1]. For practical purposes the implemented data model is not the information model but can (and does) represent a useful subset of it. There is an inherent difficulty in mapping real examples and data to the unifying principle of UCIS. While the informational model defines a set of generic counters with relevant design information, the semantics of the counters are not always the same. With this flow, we are able to maintain enough information in the coverage database that a designer with knowledge of the design RTL would be able to identify the meaning of the coverage data and perform useful post-conversion analysis on it. However, the UCIS covergroup model defines a set of constructs that matches the information that we are trying to model. At some predefined event, if some predefined condition is true, increment a counter. This definition was also consistent with the BugSpray statement data that was being imported. Consider this view of the BugSpray assertions. At the event (statement test) if the predefined sequence of events was encountered, then the counter variable is incremented. Figure 4 illustrates the various constructs and properties that are utilized in our coverage mapping. To make mapping to a covergroup data model easier and to support the UCIS conversion utility requirements, we also defined a CVGBINSCOPE and CVGBIN for each COVERPOINT (BugSpray statement).

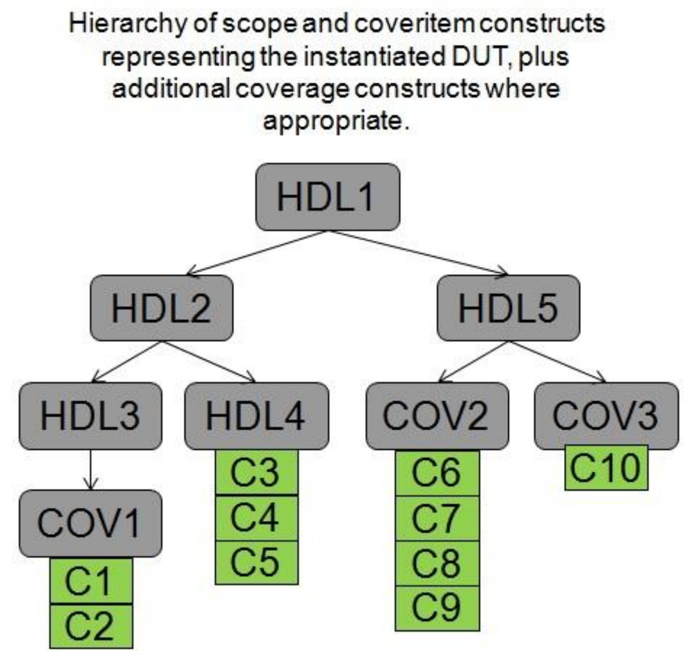

Hierarchy of scope and coveritem constructs representing the instantiated DUT, plus additional coverage constructs where appropriate.
HDL1
HDL2
HDL5
HDL3
HDL4
COV2
COV3
COV1
C1
C2
C3
C4
C5
C6
C7
C8
C9
C10

Figure 3 Representation of the UCIS Generic Data Model

Within the coverage model, multiple DU_MODULEs, which correspond to design units (Verilog modules or VHDL entities) , can be defined. In the case where a design unit is replicated within a larger design, multiple INSTANCEs of a DU_MODULE can be defined. Within either a DU_MODULE or an INSTANCE one or more COVERGROUPs can be defined. For each COVERGROUP multiple COVERPOINTs can be defined, these correspond to individual BugSpray event variables.

The currently defined data model for the UCIS conversion utility allows for the use of many properties. Future plans may call for the addition and usage of additional properties to the coverage data XML file. However, we defined these properties as the initial set for implementation in the XML file.

The XML coverage properties which can be associated with a DU_MODULE or INSTANCE are

- NAME*
- LANGUAGE*
- TYPE_NAME* (for INSTANCE for referring to the respective DU_MODULE name)

The XML coverage properties which can be associated with a COVERGROUP or COVERINSTANCE are

- NAME*

The XML coverage properties that can be associated with a COVERPOINT are

- NAME*

The XML coverage properties that can be associated with a CVGBINSCOPE are

- NAME*

The XML coverage properties that can be associated with a CVGBIN are

- NAME*
- COUNT*

* indicates mandatory properties.

Each BugSpray variable class is mapped to a COVERGROUP name and the corresponding individual BugSpray event are mapped to COVERPOINTs within that COVERGROUP. The BugSpray variable counts for each event are mapped to the corresponding COVERPOINT's COUNT property which is associated with its CVGBINSCOPE and CVGBIN properties. Another question was how to map the error scenario events. Should they be treated as a special type of coverage in the context of the covergroup? If so, what kind of data decoration should be used to denote this special case? In the context of the conversion algorithm, BugSpray fail events can be considered a special version of the count events since they are designed to track a error scenarios. Therefore they still match the @event, if (condition) , counter++ data model and this dictated that failure events are treated the same as count events within the XML and coverage database. The data decoration occurs in the comment field where the coverage is specified as being coverage of an error. Figure 5 shows an example of the coverage XML format.

F. *Conversion of XML to Coverage Database*

The Unified Coverage Interoperability Standard (UCIS) conversion utility converts a well-formed valid XML instance of a coverage database XML schema into a coverage database. The overall process flow is shown in Fig. 6. If invalid XML is specified as input to the UCIS utility, error(s) are reported and the functional coverage database is not generated. When valid XML is provided, the UCIS utility generates a corresponding coverage database based solely on the coverage data specified in the XML. This XML coverage data is not manipulated in any way. For instance, if the input XML only contains instance based coverage data, then the UCIS utility does not merge instance based coverage data to create type based coverage data. A benefit of creating a tree structure in the initial coverage report to XML conversion was that even if the related coverage information in the report is not grouped hierarchically or even located in the same portion of the report, the coverage data XML file generated is. This structure also allowed the XML to be structured in such a way that if there is a Many to One relationship between several INSTANCEs and their respective DU_MODULE, the DU_MODULE can precede the INSTANCEs in the XML file. This arrangement allows the UCIS conversion utility to better manage memory and improve coverage database conversion performance.

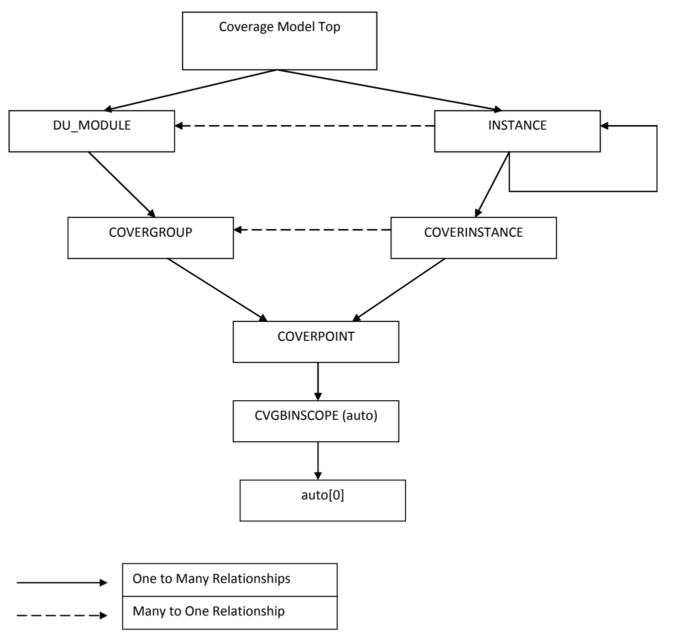


Figure 4 Representation of Implemented Coverage Data Model

IV. Results

This new approach allows for the verification and design teams to combine coverage from event and cycle simulations. This combination allows for a more complete view of coverage and verification completeness. The use of the UCIS conversion utility allows for the use of industry standard functional coverage analysis and Verification Planning and Management tools like Incisive vManager while still

```
<?xml version="1.0"?>
<unicov:unicov
xmlns:xsi="http://www.w3.org/2001/XMLSchema-
instance"
xsi:schemaLocation="http://www.cadence.com/unicov
unicov.xsd"
xmlns:unicov="http://www.cadence.com/unicov">
<!—Design Unit definition -->
<unicov:scope>
<unicov:type>DU_MODULE</unicov:type>
<unicov:name>top</unicov:name>
<unicov:lang>SV</unicov:lang>
<!—Covergroup definition -->
<unicov:type>COVERGROUP</unicov:type>
<unicov:name>cg</unicov:name>
<!—Coverpoint definition -->
<unicov:scope>
<unicov:type>COVERPOINT</unicov:type>
<unicov:name>A</unicov:name>
<!-- Auto Bins -->
<unicov:scope>
<unicov:type>CVGBINSCOPE</unicov:type>
<unicov:name>auto</unicov:name>
<unicov:src uri=" file://test.sv" line="13"/>
<unicov:bin>
<unicov:type>CVGBIN</unicov:type>
<unicov:name>auto[0]</unicov:name>
<unicov:intProperty property="COUNT" value="0"/>
</unicov:bin>
<unicov:bin>

</unicov:scope>

</unicov:scope>

</unicov:scope>
```

Figure 5. XML File Example

maintaining the current VHDL language and verification flow. With this approach the possibility of tying in other forms of coverage, like code coverage, is now available.

The UCIS conversion utility offloads the need to build and maintain various functional coverage data tracking tools from the verification leads to an industry available tool. The only tool needed is a script to convert the coverage information to the UCIS conversion utility's XML format. This can be used with a much smaller resource impact than trying to maintain a complete Verification Planning and Management toolset. The use of the UCIS conversion utility also enables the use of other tools which open the door to the possibility of more automation. As can be seen in Figure 7, the Incisive vManager GUI often makes it easier to analyze, traverse and understand the coverage data as opposed to the text reports. However, the text reports can still be generated from the tool if desired. The overhead for conversion is not excessive. A report with approximately 100000 coverage point entries takes approximately 46 seconds for the whole conversion process.

A. Proposed Next Steps

The next phase of this project will be to explore the enablement of some additional data transformation and storage in the UCIS XML and subsequently in the coverage database. For instance, at this point during coverage reviews, in some cases, the name of a property or coverage count is all that is accessible. Depending on the amount of detail included in the name, identifying the reason or meaning for the coverage statement can be difficult without help from the engineer who created the functional coverage. It is also difficult to determine what constraints might need to be applied to the verification environment to produce input vectors which could increase coverage for those properties. An easy way to correlate and view the properties which collect the coverage could be helpful in this kind of analysis. The collection of source code data from a combination of other sources may be possible. This source code data could then be stored in the coverage data XML file and coverage database. The UCIS conversion utility supports the addition of source code file and line information. Armed with this information and the vManager C/S GUI which allows for the viewing of cover properties linked to the source code of functional coverage metrics, coverage analysis can be accomplished more easily without involving design resources.

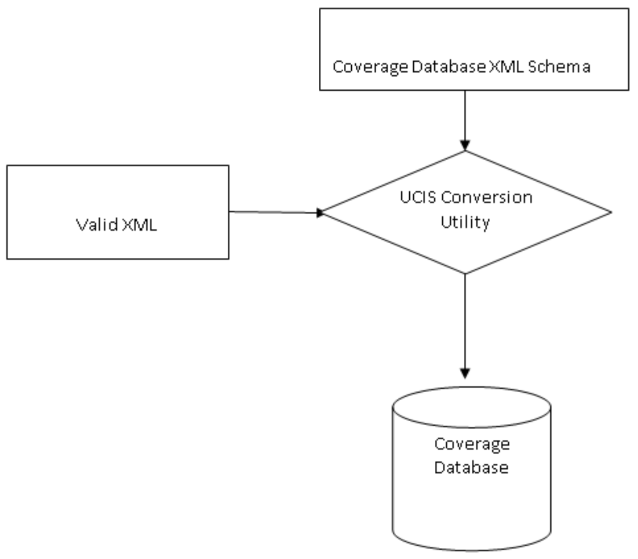
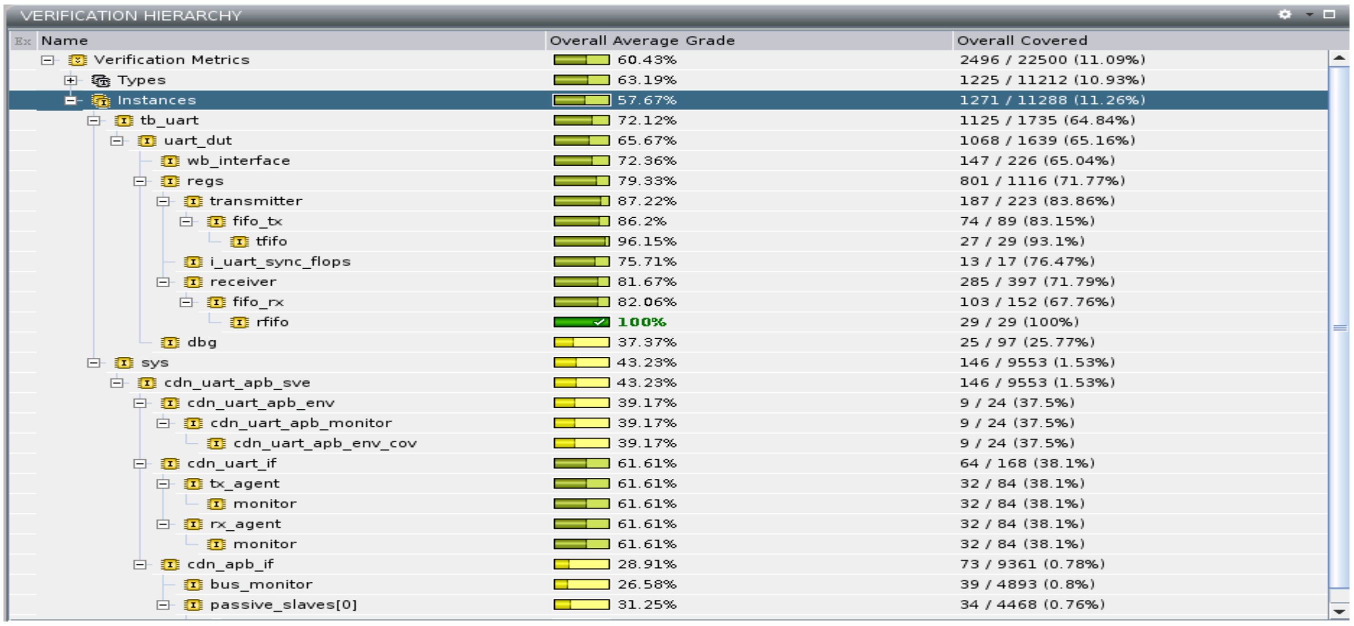


Figure 6. ***XML to Coverage Database Conversion Flow***


VERIFICATION HIERARCHY

Name	Overall Average Grade	Overall Covered
Verification Metrics	60.43%	2496 / 22500 (11.09%)
Types	63.19%	1225 / 11212 (10.93%)
Instances	57.67%	1271 / 11288 (11.26%)
tb_uart	72.12%	1125 / 1735 (64.84%)
uart_dut	65.67%	1068 / 1639 (65.16%)
wb_interface	72.36%	147 / 226 (65.04%)
regs	79.33%	801 / 1116 (71.77%)
transmitter	87.22%	187 / 223 (83.86%)
fifo_tx	86.2%	74 / 89 (83.15%)
tfifo	96.15%	27 / 29 (93.1%)
i_uart_sync_flops	75.71%	13 / 17 (76.47%)
receiver	81.67%	285 / 397 (71.79%)
fifo_rx	82.06%	103 / 152 (67.76%)
rfifo	✓ 100%	29 / 29 (100%)
dbg	37.37%	25 / 97 (25.77%)
sys	43.23%	146 / 9553 (1.53%)
cdn_uart_apb_sve	43.23%	146 / 9553 (1.53%)
cdn_uart_apb_env	39.17%	9 / 24 (37.5%)
cdn_uart_apb_monitor	39.17%	9 / 24 (37.5%)
cdn_uart_apb_env_cov	39.17%	9 / 24 (37.5%)
cdn_uart_if	61.61%	64 / 168 (38.1%)
tx_agent	61.61%	32 / 84 (38.1%)
monitor	61.61%	32 / 84 (38.1%)
rx_agent	61.61%	32 / 84 (38.1%)
monitor	61.61%	32 / 84 (38.1%)
cdn_apb_if	28.91%	73 / 9361 (0.78%)
bus_monitor	26.58%	39 / 4893 (0.8%)
passive_slaves[0]	31.25%	34 / 4468 (0.76%)

Figure 7. ***Example Coverage Hierarchy view***

Verification engineers can also review the effectiveness of their verification environments in impacting functional coverage closure more independently.

As previously mentioned in this paper, the UCIS coverage conversion utility does not manipulate the coverage data in any way. Currently the XML coverage data generated is only instance based. Instance based functional coverage is functional coverage that is collected and tracked on for each instance of any particular design unit. This kind of coverage tracking is useful for when the particular instance on which a coverage event was observed is relevant to the coverage event. For instance, in the case of some distributed arbitration algorithm, where it is important to track that each instance of the arbitration unit received some information packet from a central unit. Type based functional coverage, however, is functional coverage that is collected and tracked across all instances of any particular design unit. There are situations where type based functional coverage data can be useful to the verification effort. For instance, consider the case where, you have several instances of some instruction execution unit. For the purposes of verifying successful execution of all possible instructions, the actual instance on which the instruction was executed on may be irrelevant. In that case, the functional coverage data on instructions executed across the multiple instances of the instruction unit would be collected and aggregated for tracking and measurement purposes. Enabling the use of type based coverage data is also a possibility. Currently, two approaches are being considered. Either having the UCIS XML conversion utility automatically calculate the type based coverage based on the instances contained in the coverage data XML file or calculating the type based coverage based on the instances and type specified in the BugSpray coverage report or writing the type based coverage directly to the coverage data XML file. The latter approach has the attractive quality of being consistent with the current UCIS conversion utility generating coverage based solely on the data contained in the coverage data XML file.

Other UCIS conversion utility supported properties consistent with Coverage Driven Verification may also be explored to determine their suitability for inclusion in the coverage data XML file.

B. *Conclusion*

This paper describes a method that we developed to easily convert VHDL functional coverage data generated from cycle simulations into a form that could be consumed by an industry available Verification Planning and Management tool, Cadence's vManager C/S. This process leverages the Accellera Unified Coverage Interoperability Standard (UCIS) to create a generic common coverage data model for the exchange and transformation of the functional coverage data.

References

[1] Accellera Organization, Inc. (2012, June), Unified Coverage Interoperability Standard (UCIS). Accellera Organization, Inc. [Online] Available: http://www.accellera.org/downloads/standards/ucis

[2] Bruce Wile, John C. Goss, Wolfgang Roessner. Comprehensive Functional Verification: The Complete Industry Cycle, pp 9.

[3] Bruce Wile, John C. Goss, Wolfgang Roessner. Comprehensive Functional Verification: The Complete Industry Cycle, pp 10

[4] Bruce Wile, John C. Goss, Wolfgang Roessner. Comprehensive Functional Verification: The Complete Industry Cycle, pp 251

[5] Accellera Organization, Inc. (2012, June), Unified Coverage Interoperability Standard (UCIS). Accellera Organization, Inc. [Online] Available: http://www.accellera.org/downloads/standards/ucis pp 14

[6] Accellera Organization, Inc. (2012, June), Unified Coverage Interoperability Standard (UCIS). Accellera Organization, Inc. [Online] Available: http://www.accellera.org/downloads/standards/ucis pp 18

[7] Accellera Organization, Inc. (2012, June), Unified Coverage Interoperability Standard (UCIS). Accellera Organization, Inc. [Online] Available: http://www.accellera.org/downloads/standards/ucis pp 18

[8] Accellera Organization, Inc. (2012, June), Unified Coverage Interoperability Standard (UCIS). Accellera Organization, Inc. [Online] Available: http://www.accellera.org/downloads/standards/ucis pp 18

[9] Amir Hekmatpour, James Coulter, Azadeh Salehi. FoCus: A Dynamic Regression Suite Generation Platform for Processor Functional Verification. [Online] Available:http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.93.3577&rep=rep1&type=pdf pp 2

[10] Viresh Paruthi . Large-scale Application of Formal Verification: From Fiction to Fact [Online] Available:http://fmcad10.iaik.tugraz.at/Papers/papers/09Session8/024Paruthi.pdf

[11] Accellera Organization, Inc. (2012, June), Unified Coverage Interoperability Standard (UCIS). Accellera Organization, Inc. [Online] Available: http://www.accellera.org/downloads/standards/ucis

[12] Accellera Organization, Inc. (2012, June), Unified Coverage Interoperability Standard (UCIS). Accellera Organization, Inc. [Online] Available: http://www.accellera.org/downloads/standards/ucis pp 181

[13] IBM, Corporation. (2003, Februrary), Formal Checkers (FoCs) User Guide. IBM Corporation. [Online] . Available:https://www.research.ibm.com/haifa/projects/verification/focs/focs2.pdf