Plugging the Holes: SystemC and VHDL Functional Coverage Methodology

Pankaj Singh
Infineon Technologies
Pankaj.Singh@infineon.com

Gaurav Kumar Verma
Mentor Graphics
Gaurav-Kumar_Verma@mentor.com

ABSTRACT

Technology advances allows for the creation of larger and more complex designs. This poses new challenges, including efforts to balance verification completeness with minimization of overall verification effort and cycle time.

It is practically impossible to enumerate all of the conditions and states to do an exhaustive test. Therefore, it is imperative to use well defined criteria to measure and check when the verification is sufficiently complete and meets a reasonable quality threshold. Code coverage is one measure of design quality. Another is functional coverage, which is used to check that all important aspects of the design are tested while perceiving the design from a user or system point of view. The verification task for complex designs is further confounded due to usage of multiple Hardware Verification Languages (HVL's) such as VHDL, SystemVerilog and SystemC in a single design. While functional coverage is provided in Vera, Specman and SystemVerilog; languages such as VHDL and SystemC have neither an inherent support for functional coverage nor a well defined methodology to facilitate it. Addressing these two issues was the main motivation behind our work.

This paper presents two unique solutions for facilitating functional coverage in VHDL and SystemC. One approach is to use post-simulation Value Change Dump (VCD) files to calculate functional coverage. The other approach, which is applicable only to SystemC, proposes extending the SystemC Verification Library (SCV) to facilitate functional coverage calculation. A utility is created based on the proposed VCD solution and validated against sample testcases. Results of these experiments are also presented.

This paper starts with an introduction section that highlights the gaps or non-availability of consistent standardized functional coverage methodology for SystemC and VHDL based designs. The second section covers the details of the proposed solutions and is divided into two main sections: VCD-based coverage methodology, which provides a generic language independent coverage solution, and SystemC functional coverage solution, which proposes a new set of extensions to the SystemC SCV library that are not currently part of the OSCI-SCV API. The conclusion summarizes results and highlights the benefits of overcoming common HVL coverage limitations.

Keywords—

VCD:	Value Change Dump
CIF:	Coverage Input File
SCV:	SystemC Verification Library
HVL:	Hardware Verification Language
OSCI:	Open SystemC Initiative

1. INTRODUCTION

Over the past decade coverage-driven verification has emerged as a means to deal with increasing design complexity and ever more constrained schedules. Functional coverage provides a reliable measure for determining what each test run has accomplished and reflects the status in terms of overall verification completeness. HVL's like SystemVerilog support key features for functional coverage such as:

- coverage of (and cross coverage between) variables and expressions
- automatic and user-defined coverage bins
- means to associate bins with sets of values, transitions, or cross products
- filtering conditions at multiple levels
- events and sequences to automatically trigger coverage sampling
- procedural activation and query of coverage
- optional directives to control and regulate coverage

However, these powerful features are unavailable to users of HVL's such as VHDL and SystemC. Coverage-driven verification flows for these two languages are still plagued by challenges on how to efficiently achieve coverage closure.

Several methodologies ([1], [2]) have been proposed to extend functional coverage to SystemC and VHDL. One [2] uses a SystemVerilog bind construct to bind a SystemVerilog module or program block containing functional coverage collection constructs inside a SystemC or VHDL region. This solution works well, though the mixed-language bind construct is not yet standardized. There is no Language reference manual (LRM) for mixed-language interactions, which forces designers to follow use models dictated by EDA tools. These use-models vary for each EDA vendor, which leads to design customization depending on which vendor's tool is in use. What works in one vendor's tool is not guaranteed to work exactly the same way in the other. Moreover, using the SystemVerilog bind construct to extend functional coverage to SystemC and VHDL introduces an extra level of hierarchy in the design, which may not always be desirable.

The next part of the introduction section describes the VCD file format, which is useful in understanding the proposed language-independent VCD-based functional coverage methodology.

Introduction to VCD

Value change dump (VCD) is an ASCII-based format for dumping files generated by EDA simulation tools that capture value changes on selected variables or signals in a simulation. The simple compact structure of the VCD format has allowed its use to become ubiquitous and to spread into other HVL domains.

Components of a VCD file

A VCD file comprises, in order: a header section with date, simulator, and timescale information; a variable definition section; and a value change section.

VCD keywords are marked by a leading **$** and in general start a section; sections are terminated by a **$end** keyword. Data in the VCD file is case sensitive and broadly organized into four sections:

1) *Header Section*: includes a timestamp, simulator version number, and a timescale.

2) Variable Definition Section: contains scope information as well as lists of signals instantiated in a given scope. Each variable is assigned an arbitrary ASCII identifier for use in the value change section.

3) *Dump Var Section ($dumpvar):* contains initial values of all variables dumped.

4) *Value Change Section:* contains a series of time-ordered value changes for the signals in a given simulation model.

2. Language-Independent Functional Coverage Measurement in HVLs

This is the implementation section which describes the details of proposed coverage methodology based on language independent VCD solution and extending functional coverage to SystemC using SCV

2.1 Using VCD To Calculate Functional Coverage

The VCD file is fundamentally a language-independent static snapshot of all value changes in the simulation. This section describes how to extend functional coverage to VHDL and SystemC based on this powerful feature. High-level flows are presented with flowcharts. Though it has been tested to work with SystemC and VHDL through a series of experiments, the proposed solution, which takes a VCD file and Coverage Input File (CIF) as its inputs, can be extended to any existing HVL.

Coverage Input File

CIF defines coverage groups — struct members that contain a list of data items for which data is collected over time. Once data coverage items have been defined in a coverage group they can be used to define special coverage group items called transition and cross items. A scope is also defined as a pragma through the $SCOPE keyword. The syntax of CIF is similar to that of SystemVerilog language, which helps to reduce the overall development effort, particularly since SystemVerilog parsers can be re-used. A sample CIF is shown in Fig. 1 below.

```
// $SCOPE=/main/
covergroup cg @ y;
   cover_point_y : coverpoint y {
      bins a = {0,1};
      bins b = {2,3};
      bins c = {4,5};
      bins d = {6,7};
   }
endgroup
```

Fig. 1 Sample Coverage Input File

A high-level overview of the proposed solution is shown is Fig. 2.

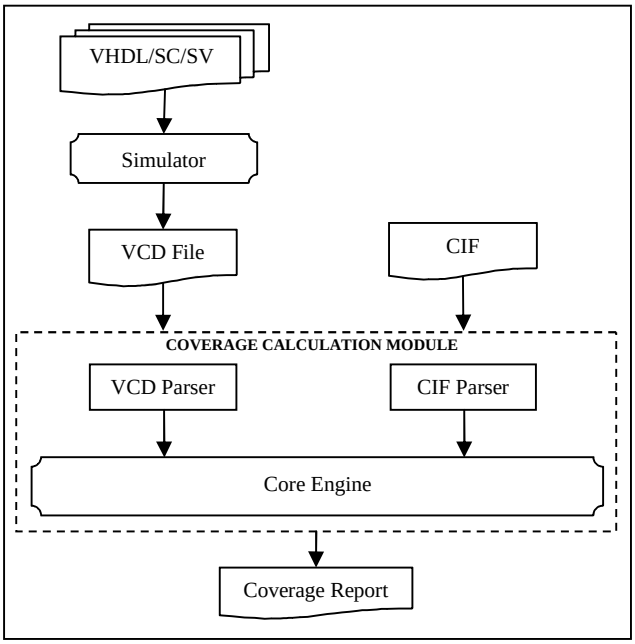

Fig. 2 High-Level Flow of VCD Based Coverage Calculation Methodology

The Design Under Test (DUT), written in any HVL, is simulated as usual in the verification environment. The generated VCD file is stored and passed to the coverage calculation module, along with the CIF. The core engine eventually generates the coverage reports. If a different set of covergroups need to be defined, the user can re-define and update the CIF file and re-use the same VCD file to generate a new coverage report.

The algorithm developed works as a post process on the VCD; hence it is not required to run the simulation again to generate a different set of coverage reports. The same VCD file can be re-used with the modified CIF to generate a different set of coverage reports.

This methodology not only provides a language-independent coverage solution but also saves a significant amount of time that otherwise would be required due to multiple simulation re-runs to

calculate the overall coverage.

The coverage calculation module described in this paper consists of three main components.

A) *CIF Parser:* In addition to its primary task of getting details of coverage groups, the CIF parser also populate a list of variables and signals that must be monitored in the VCD. This is particularly important because a VCD can contain lots of signals, and monitoring all of them adds significant performance overhead which may not be desirable.

A high-level flow of CIF parser details is shown in Fig. 3.

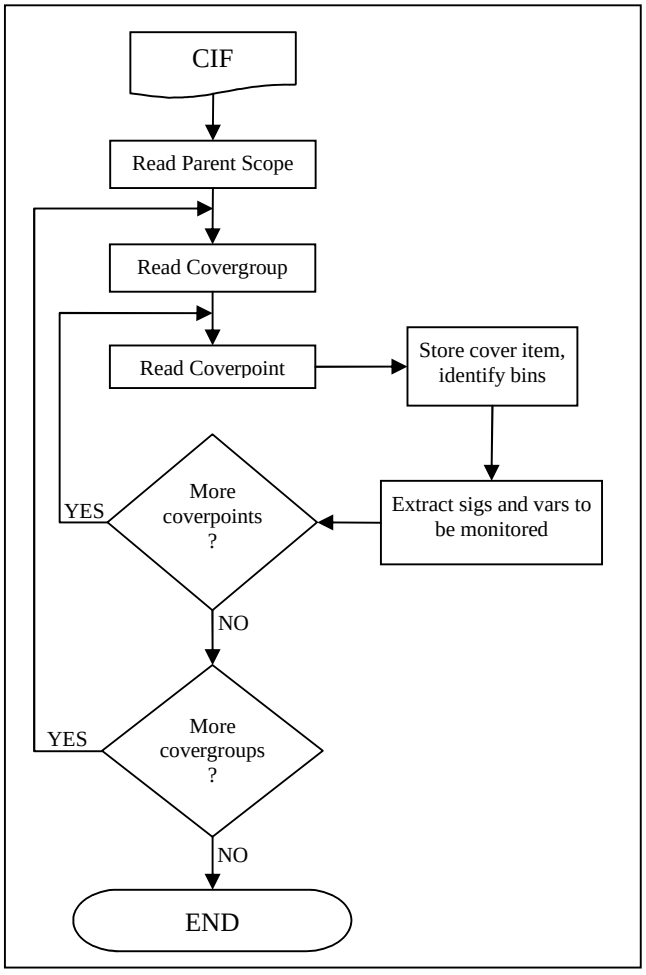

Fig. 3 CIF Parser Flow

The CIF parser retrieves the coverage item along with coverage range of the variable. Each cover item is identified as cover points and range as cover bins. The CIF parsing step involves collecting cover points from the input CIF file and creating the hash data structure. It captures bin information collected for each cover point in separate hash table which is updated later.

B) *VCD Parser:* Once the list of all signals and variables that are required to be monitored in the design is gathered by the CIF parser, the VCD file is parsed to populate a database of value changes.

A high-level flow of VCD parser is shown in Fig. 4.

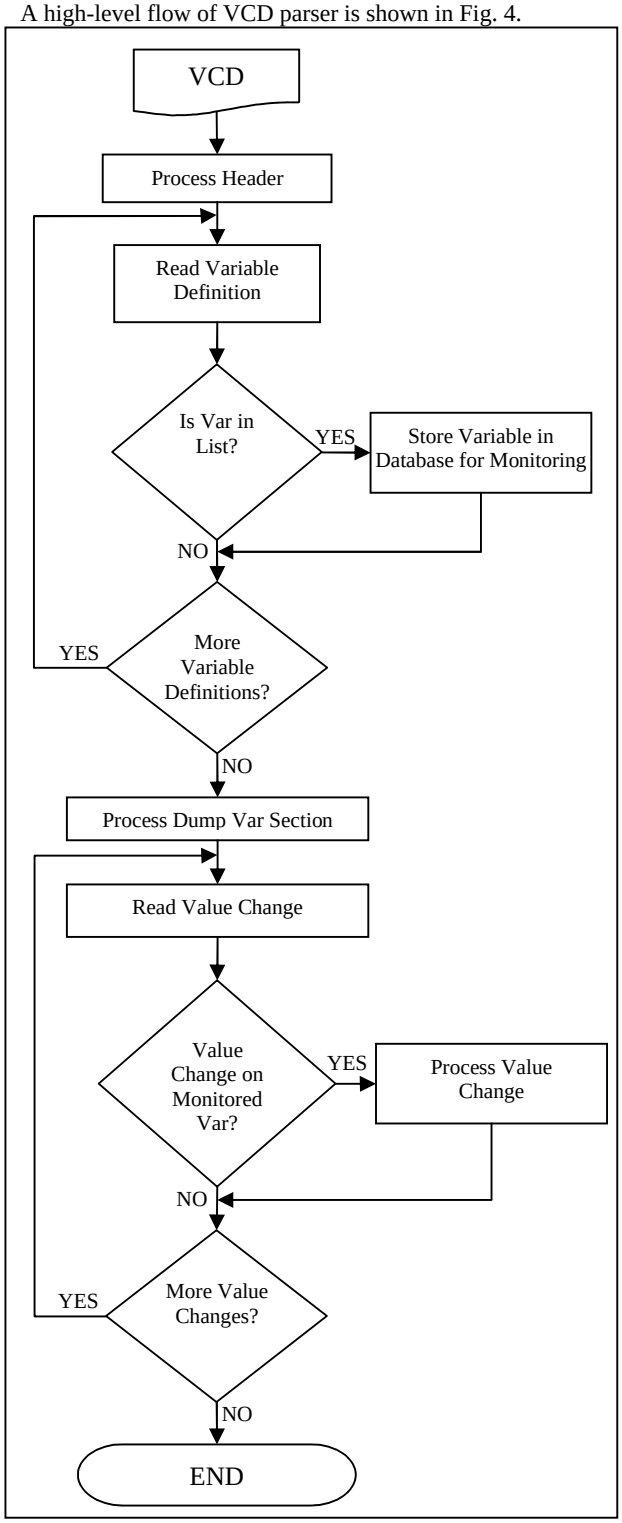

Fig. 4 VCD Parser Flow

Parsing VCD file involves two separate operations which are executed to gather coverage information from the simulation dump. The first operation parses definition part of vcd file to understand the variables. It captures the identifiers to create data structure with related details such as variable name, range, size,

type. The second parsing operation collects information related to changing characteristics of the identifier retrieved during first step. It fills the identifier data structure recorded during simulation run.

C) *Core Engine:* The core engine keeps track of value change updates coming from VCD, calculates coverage numbers based on the coverage group definitions from the CIF, and generates coverage reports. Fig. 5 illustrates the core engine flow.

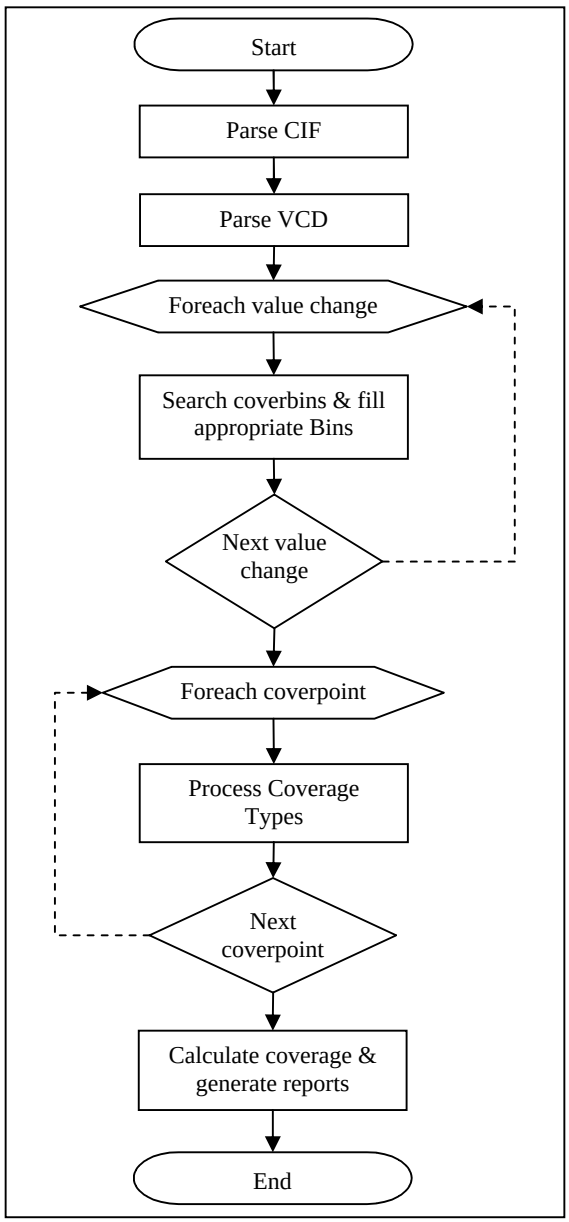

Fig. 5 Core Engine Flow

The core engine takes data structure input from the above steps of CIF and VCD parser flow. The core engine is the *main module*. It loops through the required cover items taken from CIF file. For each cover point, it identifies the type of coverage that is requested and calculates the occurrences of values within the limits of requested range of coverage point. It investigates each range value of the coverage point and if this range falls within the bin range it increments the respective bin's hash table. Once the tool captures all bin counters it calls the *ReportGen module* to generate the total coverage report.

The ReportGen module retrieves all the bin information from bin's hash table and checks counter value of the each bin that is being investigated. There will be at least one bin for each cover point. The proposed utility considers default bin if there is no explicit bin information provided in CIF. The coverage is calculated based on the counter value of each bin. Complete 100% coverage is achieved for any cover point, if the counter values of all the bins are filled. Total coverage is calculated by considering all the cover point's coverage value that is being investigated by the tool

2.2 Extending Functional Coverage To SystemC Using SCV

SystemC (IEEE Std. 1666™) is picking up as a standard to describe complex SoCs. It allows the same language to be used for early-stage and RTL-level design work, and also enables co-simulation of modules described at different levels of abstraction, including software. However, SystemC was not defined with formal analysis in mind, a limitation that many in the verification community are working to address.

This section proposes a new set of extensions to the SCV library that are not currently part of the open OSCI-SCV API. These extensions will extend functional coverage to SystemC designs, with the help of SCV. A high level overview of the proposed set of extensions is presented in Fig. 6a and Fig. 6b.

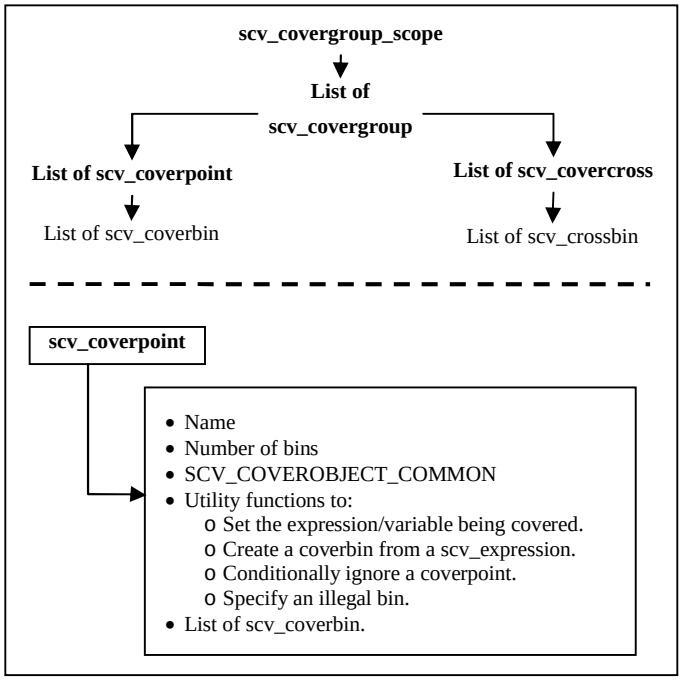

Fig. 6a Proposed set of SCV extensions to facilitate functional coverage collection in SystemC

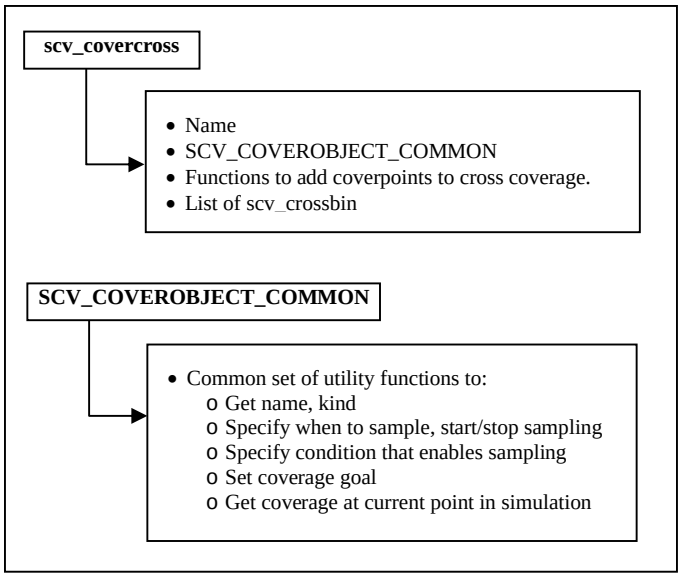

Fig. 6b Proposed set of SCV extensions to facilitate functional coverage collection in SystemC

The root of the proposed extensions data structure is the scv_covergroup_scope object, which acts as a wrapper around the list of scv_covergroups. It also contains critical scope information like name, handle to scope's parent and so on.

scv_covergroup contains lists of coverpoint and covercross, and also the core routines defined as a part of the SCV_COVEROBJECT_COMMON macro.

To help make the proposed extensions user-friendly for the verification engineers, a set of macros has been introduced to construct coverpoints, along with automatically setting the correct file name and line number.

When the name of the coverpoint is same as the name of the variable, these macros are defined as:

```
#define SCV_COVERPOINT_CTOR(x)     x(#x,__FILE__,__LINE__)
#define SCV_COVERCROSS_CTOR(x)     x(#x,__FILE__,__LINE__)
#define SCV_COVERGROUP_CTOR(x)     x(#x,__FILE__,__LINE__)
```

Additional macros are provided to add flexibility by supporting cases where the object name is not the same as its variable name:

```
#define SCV_COVERPOINT_CTOR2(x,y)  x(y,__FILE__,__LINE__)
#define SCV_COVERCROSS_CTOR2(x,y)  x(y,__FILE__,__LINE__)
#define SCV_COVERGROUP_CTOR2(x,y)  x(y,__FILE__,__LINE__)
```

With support for a well-defined means of writing functional coverage constructs in SystemC, the same SystemVerilog functional coverage calculation engine of the simulator will work for SystemC, as well.

3. RESULTS OF EXPERIMENTS

To validate the solution proposed in section II, a utility program was developed and tested. This section presents the results of our experiments.

Summarizing Experiments

40 testcases were created and tried through the utility. Following is the distribution of testcases:

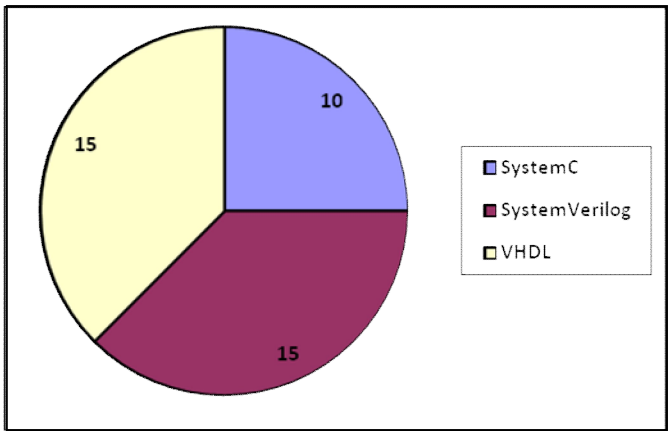

Fig. 7 Distribution of Testcases

For the purpose of validation of results, an equivalent testcase was created for each testcase by binding the covergroup directly to the design using the SystemVerilog bind construct. This modified testcase was run through a popular mixed-language simulator to generate functional coverage results, while the original testcase was run through the developed utility. Results of both these runs were compared to validate the utility.

In all 40 testcases, the results generated by the utility using the post simulation VCD file were identical to the results of the modified testcase run through the simulator.

Sample Testcase

To give a better insight into our experiments, this sub-section discusses the simplest testcase as sample example.

Consider the following SystemVerilog module, main:

```
module main;
   bit [0:2] y;
   bit [0:2] values[$]= '{3,5,6};
   initial
      foreach(values[i])
      begin
         #2 y = values[i];
      end
endmodule
```

This testcase was simulated for 8ns and the generated VCD file was stored. This VCD file was fed to the utility together with the CIF defined below:

```
// $SCOPE=/main/
covergroup cg @ y;
  cover_point_y : coverpoint y {
                    bins a = {0,1};
                    bins b = {2,3};
                    bins c = {4,5};
                    bins d = {6,7};
                    }
endgroup
```

```
C:\fcexpr\utilfinal>vcdutil
    VCD Utility v1.0
------------------------
Enter CIF Path>
./fourbins.cif

Processing CIF ... [OK]

Enter VCD Path>
./main8ns.vcd

Processing VCD ... [OK]

FUNCTIONAL COVERAGE REPORT:
--------------------------------
Covergroup
--------------------------------
 TYPE /main/cg                      75.0% coverage
    Coverpoint cg::cover_point_y    75.0% coverage
       Bin cg::cover_point_y::a     Uncovered
       Bin cg::cover_point_y::b     Covered
       Bin cg::cover_point_y::c     Covered
       Bin cg::cover_point_y::d     Covered

TOTAL COVERAGE: 75.0%  NO. OF COVERGROUPS: 1
C:\fcexpr\utilfinal>
```

Fig. 8 Screenshot of generated sample coverage report

4. FUTURE WORK

While the prototype utility discussed in section III has been tested to work on smaller testcases as a proof of concept, it's benchmarking and validation is yet to be done on larger real-world designs. It will be interesting to see how its performance compares to the simulator's built-in engine for calculating functional coverage in pure SystemVerilog designs.

Also, VCD being ASCII is very inefficient in terms of space. For large-scale real designs, VCD may not be a viable format. It will be interesting to see if WLF, or other similar formats, can be helpful in overcoming this limitation of VCD.

5. CONCLUSION

Due to the increased design complexity and distinct benefits of different languages, mixed language design is a reality. While some languages have standardized well defined coverage methodology, others suffer from limitations in measuring coverage. This paper presented solutions for functional verification coverage, specifically addressing how to implement a complete coverage-driven verification approach using VHDL, SystemC or other HVLs. The methodology proposed for VHDL utilizes a similar coverage description concept as in SystemVerilog; the addition is a new VCD-based approach. As the VCD is language-independent, the proposed methodology can be utilized for any HVL. The results for VHDL functional coverage match those of the SystemVerilog approach on sample design examples.

A set of extensions to the SCV library is also proposed to facilitate calculation of functional coverage in SystemC. The biggest factor preventing extending functional coverage to SystemC is the lack of a way of specifying these constructs in the language. With the proposed extensions, verification engineers will be able to directly define their covergroups in SystemC itself, and the native coverage engine of the simulator, which works for SystemVerilog, can be reused for calculating functional coverage of SystemC designs.

6. REFERENCES

[1] Alan Fitch and Doug Smith,"Functional Coverage without SystemVerilog", DVCon 2010.

[2] Gaurav Kumar Verma and Rudra Mukherjee, "Adding New Dimensions to Verification IP Reuse", DVCon 2009.

[3] Rich Edelmen, Mark Glassar, et al. "Inter Language Function Calls between SystemC and SystemVerilog", DVCon 2007.

[4] Pankaj Singh and Gaurav Kumar Verma, "A Holistic View of Mixed-Language IP Integration", DVCon 2010.

[5] Rudra Mukherjee and Sachin Kakkar, "System Verilog – VHDL Mixed Design Reuse Methodology", DVCon 2006.

[6] Jing-yang Jou and Chien-nan Jimmy Liu, "Coverage Analysis Techniques For HDL Design Validation", 6th Asia Pacific Conference on Chip Design Languages (APCHDL'99)

[7] Chen-Yi Chang, "Functional Coverage Analysis for Circuit Description in HDL", Master Thesis, Department of Electronics Engineering, National Chiao Tung University, Taiwan, Jun. 1999.

[8] Chien-Nan Jimmy Liu, Chen-Yi Chang, et al. "A Novel Approach for Functional Coverage Measurement in HDL", ISCAS 2000 - IEEE International Symposium on Circuits and Systems, May 28-31, 2000, Geneva, Switzerland

Note:

* In this paper the term "coverage" represents functional coverage.