Functional Coverage – without SystemVerilog!

Alan Fitch
Doulos Ltd
Ringwood
UK
+441425471223
alan.fitch@doulos.com

Doug Smith
Doulos
Morgan Hill
California, USA
+1-888-GO DOULOS
doug.smith@doulos.com

ABSTRACT
This paper investigates the implementation of functional coverage in languages such as VHDL and SystemC®[1], when for some reason the use of SystemVerilog is not possible.

General Terms
Verification, Languages, Measurement.

Keywords
Functional Coverage, SystemVerilog, SystemC, VHDL

1. INTRODUCTION
Functional Coverage has become a key part of verification methodology in the ASIC world. Used together with testbench automation / constrained random verification, it provides a closed loop from test planning through simulation (and formal verification) via coverage measurement, back to the test plan. Over recent years, complete re-usable verification methodologies have been developed, using the object-orientated features of SystemVerilog [4] for implementation [8][12]. However programmable devices (FPGAs and CPLDs) have also grown in size and complexity; and for reasons that are not completely clear, much design and verification of these devices is carried out in VHDL [6]. Meanwhile, at the system level, there is use of SystemC [5] primarily for Virtual Prototyping but also for high level synthesis. This paper looks at the implementation of functional coverage in VHDL, and then briefly at implementing functional coverage in SystemC.

Section 2 reviews coverage and specifically functional coverage. Some of the key features of the implementation of functional coverage in SystemVerilog are shown, together with some example code. Assertions are also discussed briefly, though these are not the main focus of the paper.

Section 3 describes a simple approach to collecting coverage data using concurrent procedure calls in VHDL. This relies on post-processing of potentially large data files, but is very easy to implement. A similar approach to that used in the Open Verification Library (OVL) [7] is used to easily enable and disable coverage collection.

Section 4 shows a more complex VHDL implementation using design entities. The issue of cross coverage is also addressed. Code examples are written for maximum portability to the VHDL 1076-1993 standard, but we also show how VHDL 1076-2008 provides some very useful features which would significantly simply and improve the code.
Section 5 provides a brief summary of the performance and ease-of-use of the approaches shown. Section 6 provides an overview of how functional coverage can be implemented SystemC.

Section 7 describes recent developments in a standard for storing coverage being developed by the Accellera UCIS activity.

Finally, section 8 draws some conclusions from the work carried out, and suggests recommended approaches.

2. From Coverage to Functional Coverage

2.1 Kinds of Coverage
One simple classification for kinds of coverage is

- Structural coverage
- Property coverage
- Sample-based coverage

Structural coverage relates specifically to code, and the execution paths that may be taken through that code. Examples include code coverage (has every line of code been executed?), Finite State Machine coverage (have all allowed transitions and states been exercised?), and expression coverage (have all elements of an expression been exercised?).

Properties are Boolean statements about a design, which may include temporal aspects (that extend in time). For instance a property might state that if signal X is true at a particular clock cycle (the antecedent) then signal B must be true within 10 to 20 clock cycles (the consequent). Assertions are statements that require properties to hold in some way: for instance a property must always hold; or must never hold. Properties and assertions may be written in the Property Specification Language (PSL) [3] or SystemVerilog Assertions (SVA). Note that a property is considered to hold if the antecedent never occurs (i.e. X is never true in the example above). Hence it is useful to cover properties and assertions to check if a property was never exercised, as well as measuring how many times it held, or failed to hold.

Modern verification languages such as SystemVerilog allow measurement of the range of values that occurred during simulation for a particular object. We have referred to this above as "sample-based" coverage. The description of sample-based coverage in the SystemVerilog Language Reference Manual (LRM) [4] is quite extensive – but the essential idea is to sample values and sort them into ranges or "bins".

For instance, a design might be sending three sorts of packet, and the verification engineer might want to check that all three types of packet have been generated at the input, and received at the output.

[1] SystemC® is a registered trademark of Open SystemC Initiative

2.2 Functional Coverage

To quote from the SystemVerilog LRM

"Functional coverage is a user-defined metric that measures how much of the design specification, as enumerated by features in the test plan, has been exercised."

Functional coverage relates to design intent, that is the user must work out manually which coverage measurements demonstrate that each specification point has been met; however all the kinds of coverage mentioned above may be used. Typically different kinds of coverage measurement are combined into a common database, which may then be related back to the test plan; and the test plan is derived from the design specification.

2.3 Functional Coverage Example

A simple example [1] shows how various kinds of coverage may be implemented. This example assumes that PSL and SystemVerilog are available, and uses the SystemVerilog bind construct to connect SystemVerilog to existing VHDL code. The design example is a remote control such as might be used for a TV, consisting of a keyboard scanner and a frequency generator circuit. Figure 1. shows a block diagram.

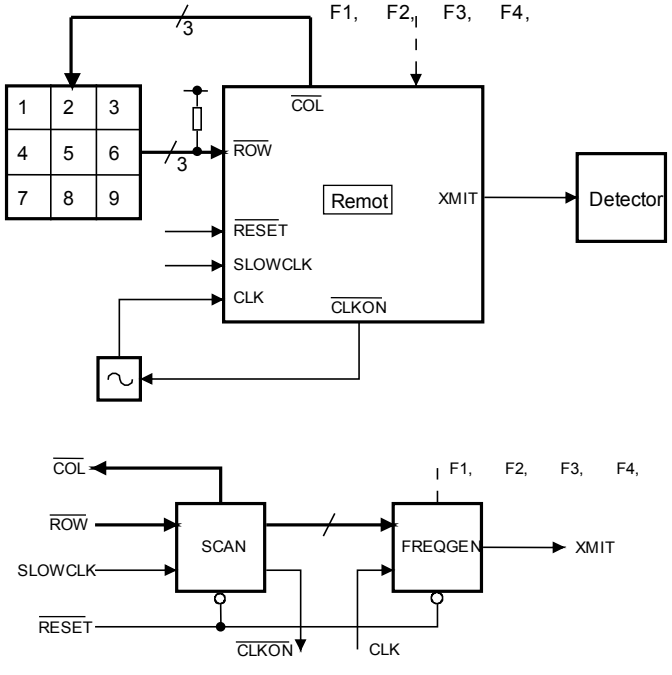

Figure 1. TV Remote Control Example Block Diagram

Certain sequences of events must happen as a consequence of button presses. These are easily captured as assertions. Figure 2 below shows an example of one of those assertions, written as embedded PSL within VHDL code.

```
-- psl B1_1: assert
     (always  rose(Button(1)) ->
         next_e[20 to 50](Found = "1000"))
                 @ rising_edge(SlowClock);
```

Figure 2. PSL Assertion Example

To measure property coverage using PSL, we may specifically use the cover directive. Figure 3 shows an example but written outside VHDL in a PSL *vunit* (verification unit).

```
vunit slot_coverage ( FreqGen(RTL) )
{
  cvg: cover { SlotNum = 0 [+];
               SlotNum = 1 [+];
               SlotNum = 2 [+];
               SlotNum = 3 [+];
               SlotNum = 0}
                   @ rising_edge(Clock);
}
```

Figure 3. PSL cover In A vunit

Next let us look at an example of sample-based coverage. The code in Figure 4 declares a SystemVerilog covergroup, containing two coverpoints and a cross coverpoint.

```
covergroup testremote_cg
                    @(posedge SlowClock);

  coverpoint RowB
  {
    ignore_bins ignore = {3'b000,
          3'b001, 3'b010, 3'b100};
    option.at_least = 100;
  }

  coverpoint ColB
  {
    ignore_bins ignore = {3'b000, 3'b001,
           3'b010, 3'b100, 3'b111};
    option.at_least = 100;
  }

  cross_row_col: cross RowB, ColB;

endgroup
```

Figure 4. SV covergroup

There are many features of SystemVerilog coverage, only some of which are shown here – for full details the reader should refer to the SystemVerilog LRM, chapter 18. Here we use the concept of *ignore bins* (values that should be ignored); and *cross coverpoints* – a coverpoint that samples when two values occur at the same sample instant (the positive edge of the signal SlowClock in the code above). The cross results in a count of the number of occurrences of *rowB* and *colB* in the following 12 combinations:

Table 1 Cross Coverage Example

RowB	ColB
011	011, 101, 110
101	011, 101, 110
110	011, 101, 110
111	011, 101, 110

The covergroup and its coverpoints are declared in a module, which is bound to the VHDL. Figure 5 shows the binding code

```
module sv_top;
  TestRemote top ();
  bind TestRemote cvg_TestRemote cvg1 (
    .SlowClock, .Button, .rowB,
    .ColB, .Found);

endmodule
```

Figure 5. The SystemVerilog *bind* statement

The module *cvg_TestRemote* above contains the covergroup and a set of port declarations, which are mapped to the signals to be monitored by the covergroup.

Let us now look at some ways of achieving coverage in VHDL.

3. VHDL Coverage with Concurrent Procedures

This section looks at a simple approach to storing coverage information. To minimize the amount of effort to collect coverage information, we will simply store data directly in a file (i.e. no attempt is made to process the data during simulation). This will be a single file declared in a package, which may be made available at any point in the design or verification environment hierarchy.

3.1 Global Declarations

To share declarations across many files, we declare a package which sets various global parameters. Part of this package is shown in Figure 6 below. The package opens a file. A separate package *configpack* is used to read global configuration data from a file. The configuration file is copied into place using an external Tcl [10] script, which means that different sets of parameters can be set without re-compilation.

```
library ieee;
use ieee.std_logic_1164.all;
use std.textio.all;
use work.configpack;

package coverpack is

  constant coverageEnable : BOOLEAN :=
       configpack.get("coverageEnable");
  constant coverageFileName   : STRING :=
       configPack.get("coverageFileName");
  constant testID : STRING :=
       configPack.get("testID");
  constant openMode : FILE_OPEN_KIND :=
       configPack.get("openMode");

  file coverageFile : text open openMode
                is coverageFileName;
```

Figure 6. Global Coverage Declarations

Note that the *open_mode* is also set from the configuration file – this means that the first test can be set to open the file in *write_mode*, while subsequent tests open the file in *append_mode*, allowing all coverage data from multiple test runs to be collected in the same file.

3.2 Concurrently-called Coverage Procedure

To collect coverage, we use concurrent procedure calls. In VHDL, a procedure may be called from a process sequentially; or it may be called concurrently within an architecture. If a procedure is called concurrently, it is equivalent to calling the procedure from a process which is sensitive to the procedure signal inputs.

```
procedure cover(
  signal s : in std_logic_vector;
  constant path : in string;
  file F : text;
  signal t : in BIT ;
  constant eventsOnly : BOOLEAN := false)
                                      is
    variable L : LINE;
  begin
    if eventsOnly and not S'EVENT then
      return;
    end if;
    write(L, path & " : ");
    write(L,s);
    write(L, " : " & testID & " : ");
    write(L, NOW);
    writeline(F, L);
  end;
```

Figure 7. Coverage Procedure Definition

Figure 7 above shows the code of such a procedure, suitable for monitoring changes in the value of a *std_logic_vector*, the standard VHDL logic vector type. The signal *t* is used to trigger the procedure: it is declared as type *bit* to allow the use of the VHDL *'transaction* attribute. The constant *testID* is declared in the package and is intended to uniquely identify the test run.

The procedure call is shown in Figure 8:

```
cpgen: if coverageEnable generate
  cover(RowB, RowB'instance_Name,
        coverageFile, RowB'TRANSACTION,
        true);
end generate;
```

Figure 8. Coverage Procedure Concurrent Call

The use of a generate statement allows coverage to easily be disabled.

The VHDL *'instance_name* attribute returns the full path to the object in question. This is an example of what is sometimes called "data introspection" – the ability of a programming language to know information about itself.

The use of the *'transaction* attribute and the final *Boolean* parameter allows the user to choose if coverage data should be written out whenever *RowB* is assigned (even if its value does not change) or only when *RowB*'s value changes. It can be useful in transaction level verification environments to store data whenever it is assigned – for instance you might want to transmit the same packet consecutively.

Note that neither of these attributes behave as you might expect if you call them from within the procedure: this is why they are passed in as arguments.

```
:testremote(bench):colb : 011 : test1 : 20000000 ns
:testremote(bench):rowb : 111 : test1 : 20000000 ns
:testremote(bench):found : 0000 : test1 : 20000000 ns
:testremote(bench):colb : 101 : test1 : 40000000 ns
:testremote(bench):rowb : 111 : test1 : 40000000 ns
:testremote(bench):found : 0000 : test1 : 40000000 ns
```

Figure 9. Extract of raw coverage data

After running the testbench, the output file contains lines of data as shown in Figure 9 above. This data must be post-processed to create coverage information. Each line contains information about the full path to the covered object, the object's value, the test identifier, and the time it was assigned or changed. This is enough to compute coverage or cross-coverage information, for instance using a Perl [9] script.

It is relatively easy to write overloaded cover procedures for all the different data types you wish to cover – even for user-defined types.

3.3 Pros and Cons

Concurrent procedures are easy to implement, and have little impact on the monitored code; also they are easily applicable to monitoring user defined data types (for instance records).

Disadvantages are that the technique creates large files – the example created an 18MByte text file (compared to less than 30kBytes in the proprietary SystemVerilog coverage database format); and of course the user must write a script to post-process the data.

4. Coverage using Design Entities

4.1 Why Use Design Entities?

Using concurrent procedures generally limits coding to one of two styles: either the concurrent procedure is effectively "combinational" (i.e. it runs from top to bottom in zero time); or it is possible to use an "implicit state machine" style using multiple wait statements. These restrict the user in writing complex behavior because any state (or memory) has to be held outside the procedure and accessed as an *inout* parameter.

Alternatively Design Entities make it easy to create local variables to hold local state – for instance to count occurrences in bins for sample-based coverage.

The next example shows how we can emulate some of SystemVerilog's sample-based coverage functionality – but with some limitations.

4.2 Limitations of Using Design Entities

The example we will show implements ignore bins and illegal bins; but does not implement automatic bins. Also for technical reasons with VHDL 2002 and earlier, it is difficult to handle arbitrary width vectors (because records may not contain unconstrained fields), so the particular implementation assumes limited range vectors of less than 32 bits width. This limitation could easily be removed by using VHDL 2008, but at the time of writing that would compromise portability.

A second limitation is that we have to write one design entity per data type we wish to cover – of course we may use good software engineering practice to factor out common code, but there will necessarily be repetition.

Again this can be removed by using VHDL 2008, due to the introduction of type and subprogram generics.

4.3 Example Design Entity

To investigate the feasibility of using Design Entities, a coverpoint Design Entity was written: this does not use the concurrent procedure calls at all, instead it encapsulates a process which collects samples when triggered and sorts the samples into a specified set of bins. Because we are now using Design Entities, it is possible to provide constant information (such as the bin specifications) using VHDL generics. First let us look at the generics of the coverage design entity, shown in Figure 10.

```
entity coverStdLogicVector is
  generic (path : string;
           h    : histIntegerArrayT;
           default : boolean
           );
```

Figure 10. Coverpoint Design Entity Generics

The first generic, *path*, is the path returned by *`instance_name*, as used before in the procedure-based approach. The last generic, *default*, tells the code to implement a default bin if set to true.

The middle generic, *h*, is the interesting one. This allows us to supply a set of bin specifications to the coverpoint. We do not have to know how many bins in advance because VHDL allows *unconstrained* array generics and ports.

The declaration of the type *histIntegerArrayT* is shown in Figure 11 below; it is an unconstrained array of records.

```
type histIntegerT is
  record
   min : natural;
   max : natural;
   count : integer; -- -1 => illegal bin,
                    -- -2 => ignore bin
   atLeast : natural;
  end record;

  type histIntegerArrayT is array
       (natural range <>) of histIntegerT;
```

Figure 11. Bin Specification Data Types

To use the array, we take advantage of the ability to declare items within the VHDL generate statement. Figure 12 shows the use of the coverpoint defined above:

```
cpgen4: if coverageEnable generate
  constant RowBHist : histIntegerArrayT :=
 --        min   max  count atLeast
   (0 => (0,    2,   -2,   0),
    1 => (4,    4,   -2,   0),
    2 => (3,    3,    0,   100),
    3 => (5,    5,    0,   100),
    4 => (6,    6,    0,   100),
    5 => (7,    7,    0,   100));

  signal rowBOut : natural;
  signal Trigger : std_logic;
begin
  Trigger <= '0', '1' after 1 ns, '0'
    after 2 ns when rising_edge(slowclock);

  cprowB: coverStdLogicVector generic map
     (path => RowB'Instance_name,
      h => rowBHist,
      default => false)
    port map ( s=> rowB,
               valout => rowBOut,
               doLog => stopTest,
               trigger => Trigger);
end generate;
```

Figure 12. Coverpoint Instantiation

The constant *RowBHist* specifies the bins we wish to use. The signal *rowBOut* is used to implement cross coverage. The *trigger* signal is used to indicate when the input *s* should be sampled. The input port *doLog* is used to trigger logging of the coverage data at the end of simulation.

Inside the coverpoint, we keep an array of bins, and update them whenever a new value is sampled. The data is again written out to a common file, declared in the package as before.

The process for the coverpoint which covers type std_logic_vector is shown in Figure 13. Note how this is much more complicated than the code for the concurrent procedure call (Figure 7) because it processes the data into bins during simulation.

(Note: the code for the helper procedures *computeMaxCoverage* and *writeCoverageResults* is not shown).

```
process
  variable vH :
    histIntegerArrayT(h'range) := h;
  variable bvint : natural;
  variable uncategorised : natural;
  variable categorised : natural;
  variable found : boolean;
  variable nIgnoreBins, nIllegalBins :
                      natural;
  variable maxCoverage : real := 0.0;
begin
  wait until rising_edge(trigger) or
          (doLog'event and dolog = '1');
  if rising_edge(trigger) then
    bvint := to_integer(unsigned(s));
    valout <= bvint;
    found := false;
    for i in vH'range loop
      if (bvint >= vH(i).min) and
         (bvint <= vH(i).max) then
        if (vH(i).count = -1) then
          report "Illegal bin " & path &
          " " & integer'IMAGE(bvint)
          severity ERROR;
        elsif vh(i).count = -2 then
        else
          vH(i).count := vH(i).count + 1;
          categorised := categorised + 1;
        end if;
        found := true;
      end if;
      exit when found;
    end loop;
  end if;
  if not found then
    uncategorised := uncategorised + 1;
  end if;
  if doLog'event and dolog = '1' then
    computeMaxCoverage(vh,s'length,
            default,  nIllegalBins,
            nIgnoreBins,maxCoverage);
    writeCoverageResults(vh, maxCoverage,
        path, categorised, uncategorised,
        coverageFile);
  end if;
end process;
```

Figure 13. Coverpoint Process

This implementation was simulated and the results compared to the original SystemVerilog – fortunately the results were the same.

Figure 14 shows a sample of the output in the coverage file (a single line representing one coverpoint for the *Button* signal): to clarify the values, each field has been put on a separate line, joined by the line continuation character "**" and annotations (which do not appear in the file) have been added in **bold**:

```
9 : \                                  number of bins
min  max  count at least  percent of target
256 256 781  10    7810.00 : \         bin 1
128 128 993 10 9930.00 : \
64  64  945 10 9450.00 : \
32  32  1920 10 19200.00 : \
16  16  1717 10 17170.00 : \
8  8  666 10 6660.00 : \
4  4  1528 10 15280.00 : \
2  2  471 10 4710.00 : \
1  1  321 10 3210.00 : \             bin 9
 :testremote(bench):button : \       object path
test1 : \                            test ID
14227 : \                            Uncategorized
9342 : \                             Categorized
100.00                               Coverage
```

Figure 14. Annotated Example of Coverpoint Output

An advantage of this file format is that it is very small.

4.4 Implementation of Cross Coverage

The coverpoint implementation outputs each value as detected. This makes it possible to design a cross coverpoint. The cross coverpoint samples the output of two coverpoints, takes in two histograms describing bins, and collates cross coverage data. The processing code is not shown, but it is similar to the basic coverpoint code in Figure 13. The entity declaration and instantiation are shown in Figure 15:

```
entity crossCover is
  generic (path1, path2 : string;
           h1, h2 : histIntegerArrayT;
           atLeast : natural := 0);
  port (i1, i2 : in natural;
        trigger : std_logic;
        doLog : std_logic);
end entity;

-- instance
cpRowBColB : crossCover
generic map (path1 => rowb'instance_name,
             path2 => colb'instance_name,
             h1 => rowBHist,
             h2 => colBHist,
             atLeast => 1)
  port map (i1 => rowBout, i2 => colBout,
            trigger => Trigger,
            doLog => stopTest);
```

Figure 15. Cross Coverpoint Entity and Instantiation

Again the results were correlated with the SystemVerilog original. The total text file size was 9kBytes.

5. Summary of Results with VHDL

The methods described both gave the same results. Using concurrent procedures created large files (18Mbytes storing changes). Implementing some coverpoint design entities in VHDL created a small 9kByte file.

Simulation speed was surprisingly similar between SystemVerilog, VHDL concurrent procedures, and VHDL Design Entities – probably limited by the performance of the NFS (Network File System) disk being used.

Table 2 Simulation Speed Measurements

Test	User Time (s)	Kernel Tme (s)
Procedures	23.0	0.66
Procedures + events	23.3	0.65
Entities	21.9	0.59
Original Code	22.83	0.52

For ease-of-use, concurrent procedures are good as they are very easy to implement and use, but of course require post-processing of large files. Design entities required more development effort, and were limited to covering essentially 32 bit integers, whereas concurrent procedures can easily handle any data type (including records).

VHDL 2008 would make the design entities considerably more flexible – type generics would allow commonality of code; and records of unconstrained arrays would remove the 32 bit limit in the implementation of the binning.

In conclusion, for a modern transaction-based verification environment, where data is processed at a high level of abstraction, concurrent procedures are the best approach as they can handle abstract data types, and would not create very large files. For sample-based coverage of data types less than 32 bits wide, the design entities can be used, at the cost of some development effort.

Of course SystemVerilog is still a much more powerful and expressive language, but the point of this paper is to see what can be done without SystemVerilog!

6. Overview of Coverage using SystemC

One of the authors of this paper created a simplified coverage implementation during a consultancy activity. This activity showed that an experienced SystemC/C++ coder can write something useful in about 3 days. While the code cannot be shown, the key features were

- Use of the SystemC Verification Library (SCV) to implement data introspection, creating generic print routines to dump information.
- Use of the SystemC hierarchy code to create as a unique identifier for each covered object.
- Storage of coverage data in a singleton class which can then be dumped at end of simulation.
- Simplified coverage sampling based on a "sample" method and the use of identical timestamps to detect cross coverage.

6.1 Other SystemC Implementations

There are two other possibilities. Firstly PSL supports SystemC, so it is possible to implement property coverage with SystemC by using PSL.

Secondly, if the reader has Cadence simulation tools, there is a thorough implementation of sample-based coverage within the Cadence extensions to the SystemC Verification Library.

6.2 Cadence Verification Extensions Example

The Cadence Verification Extensions (CVE) to the SystemC Verification Library (SCV) provide C++ classes to implement coverage. The starting point is the use of SCV smart pointers – namely the template class *scv_smart_ptr<T>*. *scv_smart_ptr<T>* takes a template data type which has been extended using SCV Extensions. The process of adding extensions allows extended data types to support data introspection amongst other things – an object (even a plain C++ integer) can be extended so that it has a full hierarchical name. The use of SCV extensions is beyond the scope of this paper: for the purpose of demonstrating SystemC coverage code, we will assume that the data types used have been extended.

First, Figure 16 shows the declaration of a smart pointer and a coverpoint. Note the inclusion of the Cadence *"cve.h"* header.

```
#include "cve.h"
void f() {
  scv_smart_ptr< sc_uint<4> > ptr("ptr");
  scv_coverpoint SCV_COVERPOINT_CTOR(cov);

  // automatic binning
  cov.cover(ptr, scv_coverbin::AUTO);

  // sample on every change
  cov.sample_at(&ptr);
}
```

Figure 16 Basic CVE Coverpoint

At the end of simulation, coverage data is saved by making a call to the function *cve_coverage_save_nc()*.

The example of Figure 16 created bins automatically. However bins may be created manually using expressions as shown in Figure 17.

```
scv_smart_ptr< short int > ptr("ptr");
scv_coverpoint SCV_COVERPOINT_CTOR(cov);
// explicit binning
cov.cover(ptr, scv_coverbin::EXPLICIT);
cov.bin("negative", ptr() < 0 );
cov.bin("small",
        ptr() >= 0 && ptr() < 256 );
cov.bin("medium",
          ptr() >= 256 && ptr() < 1024 );
cov.bin("large", ptr() >= 1024 );

cov.ignore_bins( ptr() > 32000 );
cov.illegal_bins( ptr() == -32767 );
cov.sample_at(&ptr);
```

Figure 17 CVE Coverpoint with Explicit Bins

This uses the same expression evaluation code that is used by SCV constraint expressions – note the empty parentheses when a smart pointer is referenced in an expression: this is used to build expressions which are stored within the SCV code.

Cross coverage is also possible. Figure 18 shows how a cross coverpoint may be declared, given two existing coverpoints *cov_rowB* and *cov_colB*:

```
scv_covercross SCV_COVERCROSS_CTOR(cross);

cross.cover(cov_rowB, cov_colB);
cross.sample_at(&colB);

// ... exercise coverage

cout << "Cross coverage = "
     << cross.coverage() << "%" << endl;
```

Figure 18 CVE Cross Coverage

If the reader is using Cadence tools, this provides a very extensive coverage implementation in SystemC – for more details refer to the full documentation [2].

7. Combining Coverage Data

All the VHDL implementations described so far have used text files. This section briefly describes developments in creating a standard coverage format.

Within Accellera there is an activity to create a standard approach to storing coverage data using a standard application programming interface (API). The standard under development is the Unified Coverage Interoperability Standard (UCIS) [11]. This is being developed after technology donations from the main EDA vendors. At the time of writing, a draft header file is available. Of course an implementation of the API will also be needed by any user.

When this standard is finalized, it will be straightforward to combine coverage data from SystemC (since it is a C++ class library).

Historically VHDL simulators had proprietary C APIs – with the advent of the VHPI (VHDL Programming Interface) [6] the C API of VHDL has been standardized, and tool support is maturing. This will allow portable interface code to the UCIS API to be written.

Although the UCIS header is not yet published, a draft may be obtained by registering with the UCIS working group at Accellera. An earlier superceded example of the basis of UCIS may also be downloaded from the OVM website in the Contributions section (the contribution called "UCDB API and XML Interchange Format Description"). This contains an example C application demonstrating how such an API may be used.

The UCIS API allows for specification and capture of coverage data (statement, assertion, and sample-based); creation of multiple database files; traversal of scopes (hierarchy) in a design; and merging of coverage from different simulations.

This last point is perhaps the most significant: it will be possible to merge coverage data from multiple simulation runs in different simulation languages, using different tools.

8. Conclusions

This paper has surveyed the possibilities for implementing coverage without using SystemVerilog.

For VHDL, the simplest and most flexible method is to dump data into a file for post-processing. This may be implemented in a flexible and re-usable why by writing a package of procedures and calling them concurrently. An implementation using design entities has also been shown which greatly reduces file size, slightly increases simulation speed, but has limitations in the data types it can handle using current widely implemented VHDL versions.

For SystemC, a competent C++/SystemC programmer can create a basic coverage system with some effort, but by far the easiest solution (if available) is to use the Cadence extensions to SCV.

Finally a brief overview of the forthcoming Accellera UCIS was given, and its key features outlined.

9. ACKNOWLEDGMENTS

Thanks to John Aynsley (Doulos) for his original SystemVerilog coverage code and the Remote Control example.

10. REFERENCES

[1] Aynsley, J.A., Exploiting Advances in Functional Verification Methodology from VHDL and Verilog, 2008, Doulos Ltd, Unpublished
[2] Cadence IUS 9.2 Documentation file cveref.pdf
[3] IEEE Standard for Property Specification Language (PSL), 2005, The Institute of Electrical and Electronic Engineers, ISBN 0-7381-4780-X
[4] IEEE Standard for SystemVerilog, 2005, The Institute of Electrical and Electronic Engineers, ISBN 0-7381-4811-3
[5] IEEE Standard SystemC Language Reference Manual, 2005, The Institute of Electrical and Electronic Engineers, ISBN 0-7381-4871-7
[6] IEEE Standard VHDL Language Reference Manual, 2009, The Institute of Electrical and Electronic Engineers, ISBN 978-0-7381-5800-6
[7] Open Verification Library http://www.accellera.org/activities/ovl/
[8] Open Verification Methodology http://www.ovmworld.org/
[9] PERL http://www.perl.org
[10] Tool Control Language http://www.tcl.tk
[11] UCIS http://www.accellera.org/activities/ucis/
[12] Verification Methodology Manual http://www.vmmcentral.org/