Functional Coverage in SystemC

Mark Glasser

NVIDIA Corporation
Santa Clara, CA, USA

Nvidia.com

ABSTRACT

Collecting and analyzing functional coverage information is an important part of RTL verification. It's necessary for determining the completeness of a verification project. A significant portion of the SystemVerilog language is devoted to the specification and collection of functional coverage information. However, this sort of facility is not available natively in SystemC.

To determine verification completeness using randomized tests it is necessary to understand which behaviors have been exercised and which have not. This is also true in validating transaction level and cycle-accurate SystemC models. For even moderately complex models, it is difficult, and perhaps impossible, to exercise all of the behaviors of interest in a single test. To thoroughly validate a model, it is necessary to have multiple tests (or even many tests) which collectively cover the model space. To track which behaviors have been exercised and which have not a functional coverage facility is required.

In this paper we demonstrate a functional coverage facility for SystemC. The facility is based on the SystemVerilog functional coverage facilities, and attempts to replicate those features in SystemC. Like

its SystemVerilog progenitor, it's based on a data model containing covergroups, coverpoints, crosses, and bins. A C++/SystemC API has been developed which provides both a means for users to specify a coverage model based on the data model, and for collecting data at run time to populate the model. The API and internal database is entirely in C++ and does not rely on any SystemVerilog or VCS facilities at run time. At the end of simulation, a UCIS compliant XML database is dumped which, using Synopsys' VCS tool set, can be converted to a VDB coverage database and HTML reports can be generated.

The paper will describe the elements of a complete functional coverage flow in SystemC. These include the data model, a C++/SystemC API for creating coverage models and populating them during simulation, and a tool flow used to generate coverage reports from data generated in a SystemC simulation. Additionally, the paper will provide some coverage model examples and API examples to demonstrate various use models of both in a SystemC environment.

Table of Contents

Table of Figures

Table of Tables

1. Coverage Problem

The term *functional coverage* refers to a set of concepts, tools, and features of modeling languages used to collect and report information about the exercise of behaviors during simulation. A *behavior* is any bounded activity in the modeled system. The bounds are defined as a set of elements that can change state, and time boundaries -- a start and end time for the behavior. A single signal changing state is a simple example of a behavior. There is only on element in the behavior, the signal. The time bound is the time at which the signal changes state. Behaviors can be arbitrarily complex.

To claim a behavior has occurred we need to *observe* it. Behaviors can be quite complex, but we do not need to observe all of the complexity in order to claim the behavior has occurred. Consider a read operation on a bus as another example of a behavior. The operation occurs over many clock cycles and may involve a memory or a set of registers. In the case of a cache read there may be some coherency mechanics involved as well, all of which could be quite involved. The whole operation may be initiated by the assertion of a read enable signal. For the purposes of claiming whether a read operation has occurred we only need to monitor the read enable signal. When it is asserted and later de-asserted we know the entire chain of events that constitute a read operation will be set in motion. We can then claim a read operation has occurred. At this point we are not making any claims about whether the read operation was correct or not, only that it occurred (or not).

When verifying or validating models it is important to know which behaviors have occurred during testing, so you can make correctness claims. If a behavior never occurs, then you cannot make a claim about whether it is functionally correct. The purpose of a functional coverage facility is to define a set of specific behaviors, observe their occurrence, and capture the information about those occurrences so they can be analyzed and reported.

The problem is to provide a facility in SystemC/C++ that enables observations to be made and recorded. The facility must be relatively lightweight both in terms of syntax and performance. Adding coverage observation and collection to a model should not interfere with the model itself. While it's unavoidable for the functional coverage facility to consume CPU cycles, the drag on simulation performance should be minimal. The facility should also provide analysis and reporting capabilities.

The functional coverage facility described in this paper contains several pieces. An *in-memory database* maintains the coverage information as it is generated during simulation. An *API* is used to define observation points for behaviors, collect coverage information during simulation, and store the coverage information into an *external database*. Additional *post-processing tools* are used to merge databases, analyze the coverage data, and create reports.

2. Functional Coverage in SystemC

The SystemC language [1] does not provide any sort of functional coverage facility. SystemVerilog [2], on the other hand, provides a comprehensive functional coverage facility. In the world of ASIC and SoC verification the term functional coverage usually refers to the SystemVerilog facility. To maintain consistency with terminology that is entrenched among SystemVerilog practitioners, and to enable integration with available tools, we have modeled the SystemC functional coverage facility as closely as possible to the one available in SystemVerilog.

Following is a set of definitions of terms used throughout this paper. The terminology is intended to

align with SystemVerilog [2].

- Bin

  A *bin* is a single counter that is incremented when a specified behavior occurs. This is the lowest-level element in the hierarchy of items in the coverage database.

- Coverpoints

  A *coverpoint* defines a behavior to be observed. A coverpoint consists of a collection of related bins.

- Crosses

  A cross is the Cartesian cross-product of two or more coverpoints. The number of bins contained in a cross is the product of the number of bins in each of the associated coverpoints.

- Covergroups

  A *covergroup* is a collection of coverpoints and crosses. A covergroup forms a kind of scope and coverage measurements are useful at the covergroup level.

- Sample Event

  The bins in a coverpoint are incremented when a specific event occurs. This event is called the *sample event*. When the sample event occurs the expression(s) in the coverpoint are evaluated and the bins updated appropriately.

- Coverage Model

  The *coverage model* is the complete collection of covergroups, coverpoints, and crosses that is used to define all the behaviors whose occurrence must be observed to claim a model has been thoroughly verified or validated.

3. Flow

As simulation proceeds, the coverpoints observe behaviors and gather information about their occurrence into the in-memory database. At the end of simulation, the coverage data is dumped into a UCIS-compliant XML file.

We used tools that are available as part of the Synopsys' VCS package to post process the coverage data. After the simulator concludes, the XML file is converted to a Synopsys-proprietary format called VDB. The VDB file created by the SystemC functional coverage facility is indistinguishable from any other VDB file created by a SystemVerilog simulation. Thus, standard analysis and reporting tools set can be used to process these files.

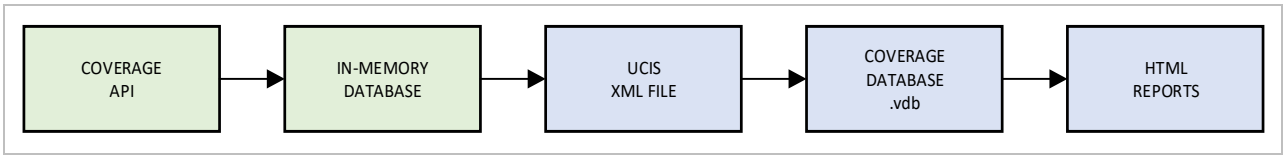

Figure 1. SystemC Coverage Flow

The model developer or verification engineer annotates models with covergroups and coverpoints that form the coverage model. As simulation proceeds, coverage information is collected and stored into an in-memory database. At the end of simulation, the database is dumped into an XML file. The

format of the XML file is compliant with the UCIS standard [4]. The XML file is converted to a VDB coverage database. Once converted into this form, the data can be processed just as any other coverage data. This includes reported generated from the coverage database using vendor-supplied tools.

The coverage metric computation is handled by the vendor-supplied tools. The SystemC coverage facility is responsible for creating a coverage model, populating it with data, and dumping it to a standard formatted XML file.

4. Database

The core of the SystemC coverage facility is an in-memory database. The database is organized as a hierarchy, with the coverage model at the top. The coverage model contains a set of covergroups, each of which contains a set of coverpoints and crosses. Coverpoints and crosses contain a collection of bins. The key elements and their relationships are depicted in the UML diagram below.

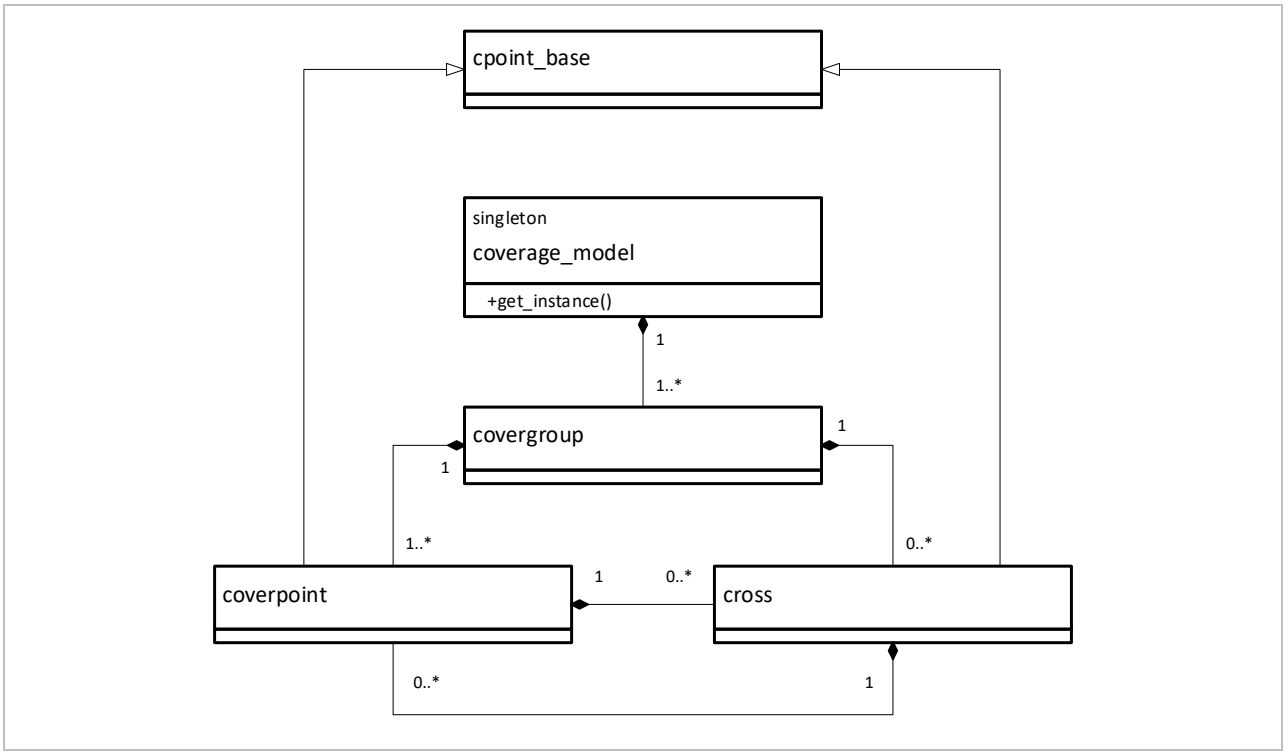

Figure 2. UML Diagram of the in-memory coverage database

Following is a description of all the elements in the data model for the coverage database.

- Coverage Model

  The coverage model is the topmost element of the hierarchy of coverage database elements. It is a singleton, meaning there is at most one instance of the coverage model. The coverage model contains one or more covergroups.

- Covergroups

  A covergroup is a collection of coverpoints and crosses. A covergroup forms a namespace for coverpoints and crosses. The name of each cross and coverpoint in a covergroup must be unique.

- Coverpoints

  A coverpoint is a set of bins that are incremented as specific behaviors occur. Coverpoints may be cross-referenced to crosses.

- Crosses

  A cross is a list of two or more coverpoints in the same covergroup, and a set of counters (bins) that represent the cross product of the associated coverpoints.

Crosses are optional. For a cross to make any sense, a covergroup must contain at least two coverpoints. The coverpoints referenced by a cross must reside in the same covergroup as the cross. The database does not support crosses that span covergroup boundaries. Of course, that also means that the crosses referenced by a coverpoint must also be in the same covergroup. That makes a covergroup a self-contained coverage entity.

Coverpoints and crosses are both derived from the same base class, `cpoint_base`. Both entities manage a collection of bins that store coverage information. The bin management code has been refactored into the `cpoint_base` class so that it does not have to be duplicated.

5. API

The SystemC functional coverage API has two parts, one for defining the coverage database, and the other for populating the database by observing and counting occurrences of behaviors. The API is implemented as a collection of text macros that hide the details of the management of and accesses to the in-memory database. The reason that text macros are used instead of a strictly classes and functions is that some of the macros expand to syntactic elements that are not function calls.

The parameters used in the macros are consistent across the entire set of macros. To aid in understanding the API here is a table that describes the arguments that are used in the macros.

Table 1. Table of API parameters

Parameter	Description
`cg`	A variable representing a covergroup.
`inst_name`	A string name of a covergroup. The name must be unique within the coverage model.
`cpname`	A string name of a coverpoint.
`name`	A string name of a cross or coverpoint
`names`	An array of strings (`char **`)
`bin`	An integer value or expression that represents the index of a bin in a coverpoint or cross.
`count`	An integer that represents the number of name strings in a `names` array
`expr`	An expression that evaluates to a Boolean value – zero or non-zero
`lo`	An integer value or expression that represents the low end of a range of integers
`hi`	An integer value or expression that represents the high end of a range of integers
`var`	An integer variable or expression that represents a bin index in a var coverpoint.

5.1 Data model creation

Data model creation is typically completed before `end_of_elaboration()`, however there is nothing inherent in the API that disallows creating the model or adding to the model after time zero. Below is a table of all the data model creation macros, followed by a detailed description of each of them.

Table 2. Data Model Creation Macros

Macro	Description
`SC_COVERGROUP(cg)`	Declare a new covergroup.
`SC_ADD_COVERGROUP(cg, inst_name)`	Add a covergroup to the coverage model.
`SC_ADD_CROSS(cg, name, cpname)`	Add a cross to a covergroup.
`SC_COVERPOINT_BIN_NAMES(cg, name, names, count)`	Supply a set of bin names to a cross or coverpoint.
`SC_COVERPOINT_IGNORE_BIN(cg, name, bin)`	Identify a bin as an IGNORE bin.
`SC_COVERPOINT_ERROR_BIN(cg, name, bin)`	Identify a bin as an ERROR bin.
`SC_COVERPOINT_DEFAULT_BIN(cg, name, bin)`	Identify a bi as a default bin.

`SC_COVERGROUP(cg)`

Declare a new covergroup. This is effectively a variable declaration and must be placed in a class declaration in place appropriate for variable declarations. `Cg` is the variable name of the covergroup.

`SC_ADD_COVERGROUP(cg, inst_name)`

Add a covergroup to the coverage model. This is an executable statement and must go in the class constructor. `Inst_name` is a string name for the covergroup. It must be a unique name amongst all the covergroups in the coverage model.

`SC_ADD_COVERPOINT(cg, name, bins, th)`

Add a coverpoint to a covergroup. This is an executable statement and must go in the class constructor. `Cg` is the variable name of the covergroup as supplied to the `SC_COVERGROUP` macro instance. `Name` is a string name for the coverpoint. It must be unique among all the coverpoints and crosses in the same covergroup. `Bins` is an integer value greater than zero which specifies the number of bins in the coverpoint.

`SC_ADD_CROSS(cg, name, cpname)`

Add a cross to a covergroup. `Cg` is covergroup as supplied to an instance of `SC_COVERGROUP`, `name` is the string name of the new cross, and `cpname` is the name of a coverpoint to be added to the cross. Since it doesn't make sense to have a cross with only one coverpoint, this macro can be instantiated multiple times with the same `cg` and `name` arguments, and different `cpname` argument. The cross `name` must be unique within scope of a single covergroup.

`SC_COVERPOINT_BIN_NAMES(cg, name, names, count)`

Specify a list of names for the bins of a coverpoint or cross. Using this macro is optional, default names will be assigned to each bin for reporting purposes. Like previous macros, `cg` is the identifier of the covergroup as specified to an instance of the `SC_COVERGROUP` macro and `name` is the string name of a coverpoint. In this case the coverpoint must have been previously defined. The `names` argument is an array of C-style strings ('\0' terminated) and `count` is the number of strings in the array. Ideally, the number of strings supplied is the same as the number of bins specified in the coverpoint. If the number of strings is less than the number of bins, the first `count` strings will be assigned to the first `count` bins and the rest will have default names. If there are more strings supplied than bins, then the first `count` strings will be assigned to bins and the rest ignored.

This feature of the API is very useful for improving the readability of the generated reports. As an example, consider an instruction register that is being covered. In each instruction cycle a coverpoint is updated to indicate which instruction is being executed. It is much easier to read a coverage report where the bins are named ADD, SUB, MUL, MOV, JMP, JMPZ, etc. rather than bin0, bin1, bin2, bin3, etc.

`SC_COVERPOINT_IGNORE_BIN(cg, name, bin)`

Define a bin as an IGNORE bin. An IGNORE bin will be ignored when computing the coverage metrics.

`SC_COVERPOINT_ERROR_BIN(cg, name, bin)`

Define a bin as an ERROR bin. This bin will be treated as an error bin for the purposes of computing coverage metrics.

`SC_COVERPOINT_DEFAULT_BIN(cg, name, bin)`

Define a bin as a DEFAULT bin. This can be used if a bin has previously been defined as an IGNORE or ERROR bin.

5.2 Coverage sampling and data collection

The coverage API contains three macros for sampled coverpoints, and three macros for inline coverpoints. A *sampled coverpoint* is one that is evaluated when a specified sampling event occurs. An *inline coverpoint* resides within procedural code and is sampled when the locus of control passes through the coverpoint.

Below is a table of the data collection macros, followed by a detailed description of each.

Table 3. Data Collection Macros

Macro	Description
`SC_EXPR_COVERPOINT(cg, name, expr, bin)`	Define an expression coverpoint.
`SC_RANGE_COVERPOINT(cg, name, expr, lo, hi, bin)`	Define a range coverpoint.
`SC_VAR_COVERPOINT(cg, name, var)`	Define a variable coverpoint.
`SC_SAMPLE_COVERPOINT(name, sample_event)`	Define a sample event.
`SC_INLINE_EXPR_COVERPOINT(cg, name, expr, bin)`	Define an inline expression coverpoint.

SC_INLINE_RANGE_COVERPOINT(cg, name, expr, lo, hi, bin)	Define an inline range coverpoint.
SC_INLINE_VAR_COVERPOINT(cg, name, var)	Define an inline variable coverpoint.

5.2.1 Sampled Coverpoints

SC_EXPR_COVERPOINT(cg, name, expr, bin)

Define a sampled expression coverpoint. The bin identified by the bin argument is incremented when expr evaluates to "true," that is, it's non-zero.

Here is the implementation of the SC_EXPR_COVERPOINT macro:

```
void coverpoint_##name()                                    \
{                                                           \
  bool cov = (expr);                                        \
  if(cov)                                                   \
    cg.incr_bin(sc_str(name), bin);                         \
}
```

The macro defines a function whose name is the name of the coverpoint prepended with "coverpoint_". The body of the function simply evaluates the expression and, if it evaluates to "true," then the appropriate bin is incremented. The SC_SAMPLE_COVERPOINT macro (below) assigns an event to this so that it can be executed.

SC_RANGE_COVERPOINT(cg, name, expr, lo, hi, bin)

Define a sampled coverpoint that updates a bin when expr is within a specified range. The range is defined by the lo and hi arguments. The bin identified by the bin argument is incremented when expr is greater or equal to lo and less than or equal to hi. The implementation of this macro is similar to SC_EXPR_COVERPOINT.

SC_VAR_COVERPOINT(cg, name, var)

Define a sampled variable coverpoint. A variable coverpoint does not take a bin argument as expression and range coverpoints do. Instead the var argument specifies the bin to increment. This macro is also implemented like SC_EXPR_COVERPOINT.

SC_SAMPLE_COVERPOINT(name, sample_event)

Define a sample event for a coverpoint. This macro maps an event to a sampled coverpoint. The mapping is done by name. Once mapped, the coverpoint is evaluated when the event occurs. This macro establishes the static sensitivity of the coverpoint function. Here is the implementation of the macro:

```
SC_METHOD(coverpoint_##name);                               \
sensitive << sample_event;
```

The macro identifies the coverpoint function as an SC_METHOD and defines its static sensitivity. This macro must be instantiated within the declaration of the containing SC_MODULE.

5.2.2 Inline Coverpoints

Inline coverpoints do not have an associated sampling event, unlike sampling coverpoints. Instead, the sampling event is the point in time when the locus of control passes through the coverpoint instance. Inline coverpoints must be coded in procedural code – i.e. in functions.

`SC_INLINE_EXPR_COVERPOINT(cg, name, expr, bin)`

Define an inline expression coverpoint. When the locus of control passes through an instance of this macro, the bin identified by the bin argument is incremented if expr evaluates to "true," a non-zero value. Here is the implementation of the SC_INLINE_EXPR_COVERPOINT.

```
{                                                          \
  bool cov = (expr);                                       \
  if(cov)                                                  \
  {                                                        \
    cg.incr_bin(sc_str(name), bin);                        \
  }                                                        \
}
```

It is very similar to the implementation of `SC_EXPR_COVERPOINT` except that the body code is not encapsulated in a function. It performs the same operation as `SC_EXPR_COVERPOINT` as the locus of control passes through the macro.

`SC_INLINE_RANGE_COVERPOINT(cg, name, expr, lo, hi, bin)`

Define an inline range coverpoint. The `expr` argument is evaluated when the locus of control passes through an instance of this macro. If the value of `expr` is greater or equal to `lo` and less than or equal to `hi` then the bin identified by the `bin` argument is incremented.

`SC_INLINE_VAR_COVERPOINT(cg, name, var)`

Define an inline variable coverpoint. The `var` argument is evaluated when the locus of control passes through an instance of this macro. If the value of `var` is within the range of defined bins, then the bin identified by `var` is incremented. For example, if a variable coverpoint is defined with ten bins and `var` evaluates to between zero and nine (inclusive) then that bin will be incremented. If `var` evaluates to 10 or greater than no bin will be incremented.

5.3 Coverage Collector

It is not always convenient or desirable to clutter a functional model with coverpoints and crosses. We can isolate them into another entity called a *coverage collector*. A coverage collector is a module that is connected to our module of interest via an analysis port. The module whose coverage we are collecting supplies information via the analysis interface. The coverage collector contains a covergroup, coverpoints, and crosses.

The coverage collector is a simple subscriber model defined as follows:

```
template <typename T>
class sc_cov_collector : public tlm_analysis_if<T>
                       , public sc_object
{
```

```
 public:

  sc_cov_collector(string nm)
    : sc_object(nm.c_str())
  {
    string cg_name(nm + "_cg");
    SC_ADD_COVERGROUP(cg, cg_name);
  }

  virtual void write (const T& t) = 0;

 protected:
  SC_COVERGROUP(cg); //!< Built-in covergroup
};
```

The class is derived from both `sc_object` and `tlm_analysis_if`, the analysis interface. The analysis interface has only a single function, `write()`. The module will call `write()` when there is data available to be analyzed for coverage. The coverage collector provides an implementation of `write()` that contains inline coverpoints. There is no need for a sample event; the call to `write()` is effectively the event that defines when it is time to sample coverage information. The argument to `write()` is a transaction object that contains information to be counted in bins. The inline coverpoints evaluate the members of the transaction object to determine which, if any, bins are to be incremented.

5.4 Turning Coverage Collection On and Off

Coverage collection consumes some overhead. Evaluating expressions, triggering events, updating bins, all consume time and memory. There are times when you want to run the models in simulation without consuming the overhead of coverage collection. It would be difficult to comment out the coverage macros or ifdef them out. The SystemC coverage facility provides a means for turning coverage collection on and off. An alternate set of definitions are available for the macros that form the API. The alternate definitions are empty or otherwise trivial. When the macro NO_SC_COV is defined then the empty/trivial implementations are defined, otherwise the normal definitions are in place. By default, the coverage facility is on, meaning the normal definitions are active. To turn off coverage collection put `-DNO_SC_COV` on the compile command line or `#define NO_SC_COV` in a position before including the header file that contains the macro definitions.

5.5 Usage

The API is designed around the idea that each `sc_module` that has interesting things to cover will have its own covergroup. The covergroup is declared using `SC_COVERGROUP()` in the class header. The construction of the covergroup and assignment of coverpoints and crosses to the covergroup using `SC_ADD_COVERPOINT` and `SC_ADD_CROSS` are done procedurally in the class constructor.

Sampled coverpoints are declared in the class header using `SC_EXPR_COVERPOINT`, `SC_RANGE_COVERPOINT`, and `SC_VAR_COVERPOINT`. The sampling event is also declared in the class header using `SC_SAMPLE_COVERPOINT`. This macro establishes static sensitivity for the coverpoint.

Inline coverpoints do not have to be declared in the class header. They are procedural and are placed in-line in any function that serves as a good place to observe a particular behavior.

6. Examples and Use Models

As a running example we'll use a transaction-level floating point unit constructed entirely in SystemC. The FPU has a set of floating point operations that it can perform on registers containing floating point values. The details of the model are beyond the scope of this paper. We will limit our examples to code required to define and capture coverage information. To maintain clarity, we'll use ellipses to indicate the location of code that is irrelevant to the example at hand.

6.1 Basic Coverage Collection

The primary unit of construction in SystemC is the module (`sc_module`). Typically, each module where there are observation points for interesting behaviors contains a covergroup and a set of coverpoints and crosses. Although it's possible to do so, it's best to keep covergroups confined to a single module.

In the constructor of our floating point unit we instantiate the covergroup and assign it to the coverage model. We also create the coverpoints and assign names to the bins.

```
const char * fpu::op_names[] =
{
    "OP_NOP",
    "OP_FLOAT",
    "OP_ADD",
    "OP_SUB",
    "OP_MULT",
    "OP_DIV",
    "OP_SQRT"
};

fpu::fpu(sc_module_name name)
  : sc_module(name)
  , target_socket("target_socket")
{
  …

  SC_ADD_COVERGROUP(cg, "fpu_cov");

  SC_ADD_COVERPOINT(cg, "fpu_op", 7, 1);
  SC_COVERPOINT_BIN_NAMES(cg, "fpu_op", op_names, 7);
}
```

The coverpoint has seven bins. Names are assigned to the bins via an array of strings that contains the names of the floating point operations.

The FPU operation arrives at the FPU unit via a TLM2 socket. Within the `b_transport()` implementation is an inline coverpoint to count each of the operations.

```
SC_INLINE_VAR_COVERPOINT(cg, fpu_op, op);
void
fpu::b_transport(tlm_generic_payload& gp,
                 sc_core::sc_time& t)
{
  …
```

```
SC_INLINE_VAR_COVERPOINT(cg, fpu_op, op);
switch(op)
{
  … // perform the specified operation
}
```

As the flow of control proceeds through this coverpoint the bin representing the operation will be incremented.

6.2 Cross Coverage

A cross is the cartesian product of two or more bins. In the SystemC coverage facility a cross is a reference to two or more coverpoints in the same covergroup along with the number of bins required to hold the cross coverage data. For example, if coverpoint A has 2 bins, coverpoint B has 5 bins, and coverpoint C has 3 bins, then a cross of AxBxC has 2x5x3 = 30 bins. The cross bins are essentially an N-dimensional array of counters organize internally in a row-major fashion. An N-ary index is used to access individual bins.

Each coverpoint has an update flag. When bins are updated in a coverpoint , the update flag is set. When a cross that contains the coverpoint is sampled then the cross bins are updated for the coverpoints that have changed since the last update. Then the update flag is cleared.

7. Differences with SystemVerilog Functional Coverage

7.1 Coverpoint Definition

SystemVerilog has a very expressive syntax for coverpoint expressions. Our SystemC coverage facility is limited to C++ expression syntax.

7.1.1 Complex bin definitions

SystemVerilog syntax allows for complex specification of bins. Bins can be distributed across ranges of values in non-uniform ways. The same effect can be achieved in SystemC using multiple coverpoints (either inline or sampled). E.g.:

```
SC_EXPR_COVERPOINT(cg, "a", (x >= 0 && x < 12), 0)
SC_EXPR_COVERPOINT(cg, "a", (x == 12), 1)
SC_EXPR_COVERPOINT(cg, "a", (x > 12), 2)
```

These three coverpoint expressions cover the cases where x is less than twelve, x is equal to 12, and x is greater than 12.

7.1.2 Coverpoint bin `with` covergroup expressions

The `with` syntax in SystemVerilog coverpoint definitions provides a means for further filtering the data as it is sampled. Here is an example from [2]:

```
a: coverpoint x
{
  bins mod3[] = {[0:255]} with (item % 3 == 0);
}
```

This states that bin mod3 in coverpoint a samples values of x in the range of 0-255 that are multiples of 3. The same semantics can be achieved in SystemC using normal C++ expression syntax in a coverpoint. E.g.

```
SC_INLINE_EXPR_COVERPOINT(cg, "a", (x >= 0 & x < 255 & x %3 == 0), 5)
```

(where 5 is the index of the bin of interest)

7.1.3 Transition Coverage

Transition coverage is currently not supported in our SystemC coverage facility. This is an area of active research. In the current system transition coverage can be achieve manually. That is a set of coverpoints would have to be defined that matched previous and next states of a state variable. E.g.

```
SC_EXPR_COVERPOINT(cg, "state", (prev_state == A && state == B), 1)
SC_EXPR_COVERPOINT(cg, "state", (prev_state == A && state == C), 2)
```

These two coverpoints would cover transitions A->B and A->C. The other transitions would have to be hand coded.

7.2 Crosses

The SystemC coverage facility provides the means for creating crosses. Two or more coverpoints can be associated together in a cross. The SystemC facility does not provide `binsof/intersect` semantics for creating fine-grained cross definitions. You can manually set individual bins to be IGNORE or ERROR bins.

7.3 Sampling

The SystemC coverage facility provides two ways to sample coverpoints – via an event or inline. SystemVerilog has some more exotic ways to sample coverage data. For example, SystemVerilog clocking blocks can be used as the sampling event for covergroups. This ensures that the data is up-to-date on any particular clock edge (or other synchronizing event).

SystemVerilog covergroups support a `sample()` method which initiates coverage sampling. This functionality is not directly supported in the SystemC coverage facility, but can easily be emulated. One way is to code a set of inline coverpoints into a function. Calling the function causes the coverpoints to be evaluated and coverage updated.

8. Conclusion

Functional coverage is a powerful tool for managing a verification project and has long been used effectively in SystemVerilog testbenches. We have demonstrated a similar facility in SystemC which is very useful for managing SystemC validation projects. The SystemC coverage facility is written entirely in C++/SystemC. It dumps the collected coverage data into a UCIS compliant XML file which allows existing tools to analyze and report the data. Now SystemC model developers can build coverage models for managing their validation projects and ensuring the quality of their models.

9. References

[1] IEEE Computer Society, *IEEE Standard for Standard SystemC® Language Reference Manual*, IEEE-1666-2011, January 2012

[2] IEEE Computer Society and IEEE Standards Association Corporate Advisory Group, *IEEE Standard for SystemVerilog -- Unified Hardware Design, Specification, and Verification Language*, IEEE-1800-2012, February 2013

[3] Christoph Kuznik, Wolfgang Müller, *Functional coverage-driven verification with SystemC on multiple level of abstraction*, DVCon 2011

[4] Accellera Systems Initiative, *Unified Coverage Interoperability Standard (UCIS)*, version 1.0, June 2, 2012.