Functional coverage-driven verification with SystemC on multiple level of abstraction

Christoph Kuznik, Wolfgang Müller

Faculty of Electrical Engineering,
Computer Science and Mathematics
University of Paderborn/C-LAB
D-33102 Paderborn, Germany
christoph.kuznik@c-lab.de
wolfgang@acm.org

***Abstract*—SystemC is a versatile C++ based design and verification language, offering various mechanisms and constructs required for embedded systems modeling. Using the add-on SystemC Verification Library (SCV) elemental constrained-random stimuli techniques may be used for verification. However, SCV has several drawbacks such as lack of a functional coverage facility supporting coverage collection on RTL and TLM models. In this article we present a functional coverage library which implements parts of the IEEE 1800-2005 SystemVerilog standard capturing functional coverage throughout the design and verification process, and allows to facilitate coverage-driven verification in SystemC.**

I. Introduction

In any domain incorporating embedded systems the verification of the functional and non-functional properties for integrated system behaviors is essential. Moreover, the verification process itself will remain the main bottleneck of every design flow, preventing the industry from better numbers in first silicon success. For example, in [1] it is estimated that the verification effort grows at a double-exponential rate with respect to the Moore's Law curve. Hence, if the number of transistors per chip increases by 10X between 2008 and 2018, then the verification effort would increase by 1024X. Verifying such complex heterogeneous embedded systems is a time consuming and tedious task. To cope with always more complexity and to boost productivity, more efficient verification techniques and technologies were introduced through the last years like the notion of functional verification [2] which supports features such as verification by assertions, constraint-based random test pattern generation, and functional coverage.

In this article we present a functional coverage library to enable coverage-driven verification of SystemC designs on multiple levels of abstraction, which is continuation of work conducted in [3]. While specialized hardware design and verification languages (HDLV) such as IEEE-1800 SystemVerilog [4] and IEEE-1647 *e* incorporate functional coverage language features, these functionalities are neither available in the IEEE-1666 SystemC standard [5], the SCV addon-library [6] nor complete compared to the aforementioned in any publicly available SystemC library. Moreover, to our knowledge there is no particular activity of the SystemC working groups to add these functionalities to the next versions of SystemC or SCV.

The remainder of this article is as follows. In section II we will briefly summarize the SystemVerilog `covergroup` concept and also discuss existing shortcomings. In section III we will introduce the functional coverage library for SystemC in detail. In section IV we will illustrate the handling of the SystemC functional coverage library and the API via several examples. In section V we propose how to integrate the library with a verification methodology. Related work will be discussed in section VI before we conclude and give an outlook on further development and research activities in sections VII and VIII.

II. Coverage Driven Verification

Functional coverage is a user defined metric intended to investigate to which extent the functionality of a given design under test (DUT) has been verified by the stimuli generated from previous simulation runs. As such, value coverage keeps track of value assignments and changes of expressions and conditions within the code. Thus, it is not verified if the DUT is working properly rather than just gives information of the quality of the test patterns with respect to the user-defined metrics [7]. So, functional coverage can tell if a property was executed at the right time, in the right order and in the correct context and is a valuable metric for verification closure. The IEEE-1800 SystemVerilog standard implements a metric for value and transition coverage collection by means of it's **`covergroup`** keyword. A **`covergroup`** is a hierarchical element that group coverpoints. A **`coverpoint`** is associated with a value source, for example a `sc_signal` in SystemC RTL designs. A **`coverpoint`** contains **`bins`**, which actually represent counters, and increment when the value of the associated value source fits in any of the bins assigned integer intervals or value transitions during a sampling event. The existing coverage driven verification scenarios of SystemVerilog and *e* are mainly targeted at RTL-level designs and RTL related signal types. Coverage analysis of system models at higher levels of abstraction is a promising approach to cope with the complexity and performance requirements for verification of ever increasing design sizes. Apart from that, a mandatory requirement for true TLM capable coverage collection is the possibility of multiple samplings per simulator delta cycle.

In general the coverage functionalities have to be independent from the model abstraction level and coding styles, for example TLM-2. Moreover, the option to explicitly sample coverpoints with data, in order to minimize the data collected and to maximize its information content, is meaningful.

III. A SystemC functional coverage library

In this section we will introduce the functional coverage library for SystemC in detail. Therefore, we will explain our assumptions, highlight supported features and name unsupported features of the current implementation.

A. Design Constraints

During the conception phase potential verification use-cases were discussed and several requirements and assumptions were taken which influenced the later prototype implementation of the functional coverage library. Among others requirements, the library

- shall implement a metric according to the SystemVerilog-2005 concept of covergroups, coverpoints and bins etc.
- may be used on the standard OSCI SystemC kernel.
- may use header only functionalities of Boost libraries.
- shall not rely on the SystemC Verification Library (SCV).
- shall allow coverage collection and sampling on RTL as well as TLM abstraction levels.
- shall be designed as an addon library.

B. Library Overview

The SystemC functional coverage library was designed as a singleton factory class, which is the main facility of the library, providing all necessary setup and management API calls for the creation and administration of every element of the implemented SystemVerilog coverage metric. It may be used upon the standard OSCI SystemC kernel on any C++ framework as it was designed as an add-on library for C++/SystemC environments. This also eases the addition of functional coverage collection and evaluation to existing testbenches. Moreover, the factory allows the administration of the coverage database. The database stores collected functional coverage information and has to fulfill two requirements. It has **(i)** to capture already sampled coverage information prior to the next run and **(ii)** to save this data after the test, which is simplified by a set of convenience API functions. This allows temporal merging of coverage results from independent simulation runs of the same coverage metric.

The structure of the library is depicted in figure 1. Via the connection level the library allows connection of coverpoints to `sc_signals`, variables or callback functions. If implicit sampling is used, each coverpoint has to be bound to a signal source. If a coverpoint is not bound to a value source upon simulation start the library will notify about that and halt the simulation. Connections of value sources to coverpoints are registered and saved prior to the `init_factory()` method of the factory, as can be seen in listing 1 and 2. Once a coverpoint is connected to a source it may be sampled during simulation runs. This can be done via implicit sampling or explicit sampling with an integer argument. Moreover, both ways can be performed simultaneously along the entire hierarchy of covergroups, coverpoints and bins. For example, the statement **`coverpoint`**`->sample()` invokes an one-time value retrieval from the coverpoints connected source and invokes `sample(`*arg*`)` on all its `bins`. Within the evaluation level, all coverpoints and their bins are examined if the sampled value fits in one of the specified intervals. In this case, the hit counter of the specific bin will increase. Moreover, during construction of the coverage metric, the verification engineer may set coverage goals for each coverpoint such as minimal hits, targeted hit count and may associate weights. The API level provides API functions and macros to instantiate

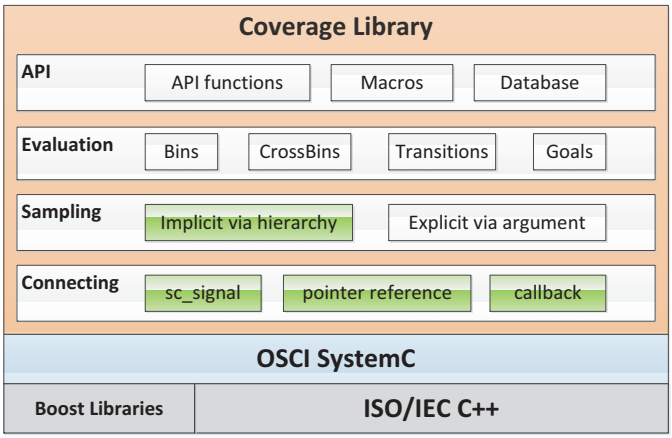

Fig. 1. Structure of the SystemC functional coverage library.

the covergroup structure, to connect sources to the coverpoints and to control and evaluate the coverage collection. Apart from that, the coverage results can be written to a simple database format.

C. Features

Our SystemC functional coverage implementation allows the definition and instantiation of covergroups, coverpoints as well as cross-coverpoints. Each coverpoint may contain an arbitrary number of (normal) bins, illegal bins, ignore bins and one optional default bin. Each bin may have numerous integer intervals assigned. Moreover, a coverpoint can contain transition bins. Each transition bin can be assigned with an arbitrary number of integer sequences to implement a simple successional value transition coverage. Each defined transition sequence is assigned with a vector matching class instance. The relation of the library elements is shown in figure 2. A default bin contains all non-specified intervals of a data type. Default bins, one per coverpoint, and their corresponding intervals can be generated for integer data types. Once an illegal bin is hit the library notifies and halts the simulation. The hits of ignore bins will be counted but ignored for the overall coverage percentage calculation of bins and coverpoints respectively. Moreover, the library allows the definition of cross-coverpoints. In detail, the two selection expressions **`binsof`** and **`intersect`** from IEEE-1800 SystemVerilog

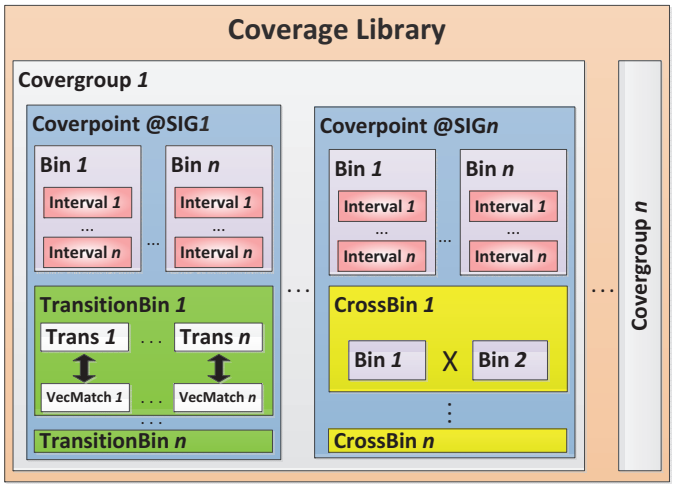

Fig. 2. Elements according to IEEE-1800 **covergroup** concept.

are implemented to calculate and limit the cross product of bins. To allow multiple samplings within one SystemC simulator delta cycle we do not rely on SC_MODULEs and clock sensitive SC_METHODs within the implementation. The sampling process can be triggered from the outside in a method call fashion. All sources are connected via pointer references or callback functions so their values may be read multiple times per delta cycle. Apart from simple value coverage boolean expression coverage can be implemented via callbacks functions.

Compared to IEEE-1800 SystemVerilog, the current implementation does not support open value ranges, clocking block signals, conditional guards to avoid sampling, wildcards specification as well as repeat ranges in transition bins. Moreover, several coverage type options are not evaluated during the sampling process and the coverage percentage calculation. Examples are `detect_overlap`, `cross_num_print_missing` as well as the `strobe` option. Moreover, the library functionality is restricted to functional coverage collection only, so other parts of SystemVerilog such as assertions, randomization and constraint-solving are not considered for implementation. It is assumed that the used SystemC testbench environment provides these features, e.g. with help of the SCV library or another library, to allow true coverage driven verification closure.

D. Simulation Performance

If implicit sampling is used all value sources are referenced via pointers or Boost callback functions, so the glueing and value retrieval itself is fast. The execution tradeoff when applying functional coverage analysis mainly depends on the granularity and amount of bins and associated intervals, due to the fact that the library internally has to check if a sampled value fits into any interval of the bins interval vector. So the implemented sampling intervals and hierarchy levels are crucial for simulation performance. Moreover, the end-user has the option to choose the appropriate level of sampling granularity and when to trigger coverpoints. This is the key for using functional coverage successfully; minimize the data collected but maximize its information content [7]. Metric-depending constants such as default bins intervals are calculated just once during finalization of the coverage metric construction. If default bins are defined, they automatically increase if all other bin types of the specific coverpoint were checked but did not hit for the specific value.

IV. Examples

In the following, several examples will be shown to illustrate the SystemC functional coverage library usage as well as the corresponding API calls.

A. Usage in a Testbench

Within a SystemC testbench the coverage library may be included in the testbench infrastructure just by including the library header file and instantiation of the coverage factory. Once the factory is instantiated within the testbench the user-defined functional coverage metric can be defined via the creation of covergroups, coverpoints and bins. For example, in a RTL testbench a SC_MODULE is created to act as a coverage monitor and is then connected to the DUT signals. During the connection phase of coverpoints the identifiers must be known to derive pointers for them. For RTL designs the definition of a SC_METHOD which invokes the users sample function on posedge events could be done. Within the sample function the verification engineer could sample the whole coverage metric from top-level, or just certain coverpoints or bins, depending on environment conditions. On the other hand, in TLM designs it is meaningful to make use of analysis ports and interfaces and to use explicit sampling with an integer argument (also see section V).

B. Instantiation

In listing 1 the instantiation of the coverage library singleton factory is shown. Using the API functions `set_coverage_db_name(`*name*`)` the verification engineer specifies where to store the collected coverage data. By `load_coverage_db(`*name*`)`, this data may be loaded and evaluated at later steps, e.g. for temporal merging of coverage results from independent simulation runs of the same coverage metric.

```
// Include the functional coverage library header
#include <SCFC_Factory.h>

// Instantiates the functional coverage factory
// ensures singleton behavior
pFac = SCFC_Factory::init();

// set external coverage database
pFac->set_coverage_db_name("smallTest.db");

// coverage metric definition
...

// finishes the coverage definition
// instantiates all objects
pFac->init_factory();
```

Listing 1. Example for instantiation of the coverage factory.

C. Coverage Metrics - Value Coverage

In listing 2 the API calls for the creation of covergroups, coverpoints and bins with their integer intervals are shown. First a new type of covergroup is created, called `CG_1` and an instance is created. Using the created reference `cg_one` it is now possible to add coverpoints to this covergroup. Each coverpoint also has a `std::string` identifier. Using the overloaded `connect` method one can connect three types of value source to the coverpoint:

- `sc_signal`s of types that can be casted to integer
- pointers to variables that can be casted to integer
- functions with return type integer

The respective identifier has to be known at runtime in the respective namespace. The API call to create bins makes use of the C++ `va_arg` macro for variadic functions to specify the amount of intervals. The third parameter of `new_bins` specifies how much bins to create for this interval(s).

```
// define a new covergroup type and create an instance
CG* cg_one = pFac->new_covergroup(this, "CG_1", "CG_1_inst");

// create a new coverpoint
CP* cp_one = pFac->new_coverpoint(cg_one, "CP_1_SystemC")

// Multiple overlays for connect method, here
// connect a sc_signal<int> to this coverpoint
cp_one->connect(Addr);

// Declare a bin for with intervals 20:30 and 50:60
pFac->new_bins(cp_one,"20to30&50to60",1,4,20,30,50,60);

// Declare an ignore bin with range 50-80
pFac->new_ignore_bins(cp_one,"Ign50-80",1,2,50,80);

// Declare an illegal bin for value 99
pFac->new_illegal_bins(cp_one,"99_BIN",AUTOBINS,2,99,99);

// Add all other ranges of Addr to a default bin
pFac->add_default_bins("DEFAULT_BIN");
```

Listing 2. Example for **`covergroup`**, **`coverpoint`** and **`bin`** definition.

Here the constant `AUTOBINS` is the equivalent to the `bins a[]` notation in SystemVerilog, and creates as many bins as intervals. The fourth parameter of `new_bins` is a `va_arg` argument that specifies how many interval tuples follow. The methods `new_ignore_bins` and `new_illegal_bins` share the same parameter intent. Using `add_default_bins` a default bin is created. Internally the library now calculates all non-covered intervals of this coverpoint and assigns them to this bin.

D. Coverage Metrics - Goals and Weights

The SystemC functional coverage library implements various `type_option` members as declared in the SystemVerilog metric. For example, `set_at_least(`*int*`)` defines the minimum number of times a bin needs to hit before it is declared as hit and `set_goal(`*int*`)` specifies the target goal for a covergroup instance, a coverpoint or a cross-coverpoint of an instance. The usage is depicted in listing 3. Using the `set_weight(`*int*`)` method it is possible to specify the weight of this covergroup instance which is considered while computing the overall instance coverage.

```
// set the minimum hit count for the coverpoint to
// be covered according to your metric
cp_one->set_comment("We use this CP to identify ...");
cp_one->set_weight(1);
cp_one->set_at_least(2);
cp_one->set_goal(90);

// all option values can be read via the API
cout << cp_one->get_goal() << endl;
cout << cp_one->get_at_least() << endl;
```

Listing 3. Usage of goals and weights within the simulation.

E. Coverage Metrics - Transition Coverage

In listing 4 the definition of transition bins is shown. First a new transition bin container has to be created via the method `new_trans_bin` and has to be assigned to a coverpoint. Once this transition bin is created, the verification engineer may add an arbitrary number of transitions. This can be done in two ways. First, using the library's transition class where new values can be added to the transition in a stream syntax. Second, it is possible to fill a `std::vector<int>` and pass it to the transition bin.

```
// define a new transition bin
Bins* transBin = pFac->new_trans_bin(cp_one, "TRANS");

// define a new transition from integer 23 downto 20
transition<int> vec23_20;
vec23_20 << 23 << 22 << 21 << 20;

// assign this transition to the transition bin
transBin->set_trans(vec23_20);

// create another transition
std::vector<int> vec15_13;
vec15_13.push_back(15);
vec15_13.push_back(14);
vec15_13.push_back(13);

// assign this transition to the transition bin
transBin->set_trans(vec15_13);
```

Listing 4. Example for transition bins.

For every transition that is defined an instance of a vector matching class is created. During sampling, each vector matching class compares the sample value with the stored transition sequence. If the value appears in the right order and the end of the sequence is reached, the hit counter of the transition bin will increase.

F. Coverage Metrics - Cross Coverage

In listing 5 the definition of cross bins for cross coverage is shown. First a cross coverpoint has to be created and assigned to a covergroup. Then coverpoints can be added to this cross coverpoint via the `add_cp2ccp` method. Now a cross bin can be defined that evaluates its value with the help of a select expression. This select expression may be composed of logical AND, OR, NOT and the functions `binsof` and `intersect`. The expression `binsof` refers to the bins of an existing coverpoint. Using `intersect` this bins may be limited by enumerations of values or intervals. The notation of intervals is again based on `va_arg`, so the second parameter of `intersect` states the amount of intervals that follow.

```
// add a cross coverpoint to the covergroup cg_one
CrossCoverpoint* CCP = pFac->new_CCP(cg_one,"CP1_x_CP2");

// add the coverpoints cp_one and cp_two
// to the cross coverpoint
pFac->add_cp2ccp(CCP, cp_one);
pFac->add_cp2ccp(CCP, cp_two);

// define a cross bin, the bins value is evaluated with
// help of the defined select expression
pFac->new_CrossBins("X1",
  pFac->binsof(CP_one, bin1) && pFac->binsof(CP_two, bin2));

// define another cross bin, limit range
pFac->new_CrossBins("X2",
  pFac->binsof_intersect(cp_one, 2, 50, 100) &&
  pFac->binsof_intersect(cp_two, 2, 0xA000, 0xB000));
```

Listing 5. Example for cross coverage using **`cross-coverpoints`**.

G. Implicit and Explicit Sampling

The functional coverage metric can be sampled in the following ways. First, it is possible to explicitly sample a certain coverpoint or bin with an argument using the `sample(`*arg*`)` method. Internally, it is checked if any bin of the coverpoint or any single bin has matching intervals defined. In case, the hit counter for the specific bin increases. Second, a sample method without any argument can be called on coverpoints and bins. In this case, every coverpoint must be assigned to a value source, using a supported overload of the `connect(`*source*`)` methods in the metric definition phase. Of course each coverpoint can be connected to a different value source. During sampling, the library dereferences the saved pointer or invokes the saved callback respectively. The result value is then used to perform a `sample(`*arg*`)` call on the specific coverpoint bins or single bin.

```
// a SC_METHOD is assigned with a sample function,
// that invokes sample(arg) on the functional
// coverage metric coverpoints, bins.
SC_METHOD(sample_func);

// clk is used as trigger
sensitive_pos << *clk;
```

Listing 6. Sampling with clocking event.

The sampling can be performed in relation to a clock event, similar to SystemVerilog `@posedge clk`, as depicted in listing 6. Moreover, the verification engineer may specify different sampling methods, each evaluating other parameters of the actual simulation run and focusing coverage collection on the interesting activities only. The API provides several convenience functions to retrieve defined covergroups, coverpoints, bins etc. by name or to iterate of vectors of them as shown in listing 7. Moreover, by means of the explicit `sample(`*arg*`)` method the coverage metric sampling can be manually triggered by the start or end of the execution of a specific function or class method. This allows to implement behavior similar to that of SystemVerilog block event expressions. Coverage collection on abstract modeling levels such as TLM-2 is typically achieved using the analysis port interface. In doing so, an analysis port is attached to the interesting model and a coverage monitor is bound to it as a subscriber. Within the subscriber the payload can be decoded and functional coverage can be collected.

```
// Code within sample_func function:

std::vector<SCFC_Coverpoint*> cp_vec;
std::vector<SCFC_Coverpoint*>::iterator cp_it;

// retrieve the vector of defined coverpoints
// from covergroup cg_one
cp_vec = cg_one->get_SCFC_Coverpoint_vector();

// iterate over every coverpoint
// call sample without argument
for (cp_it = cp_vec.begin(); cp_it != cp_vec.end(); cp_it++)
{
  (*cp_it)->sample();
}
```

Listing 7. Implicit sampling on coverpoints, bins.

H. Callbacks and Expression Coverage

The library allows to make use of a boost function pointer such as `boost::function <int (`*void*`)>` to connect a value source to a coverpoint. Within the callback function calculations can be done or boolean expressions can be evaluated. The return value of the function is then used to sample the specific coverpoint or bin. Listing 8 depicts a very simple implementation of a callback function. During the creation of the functional coverage metric this `callback_fct()` can be connected to a coverpoint via the connect method `coverpoint->connect(callback_fct)`. Note that in this example all three variables x, y and z must be accessible in the callback function namespace. C++ does not support nested functions. However, by means of inner classes a structured approach is possible.

```
int callback_fct(){

// evaluate boolean expression
return ( (x < z) || (y < z) );
}
```

Listing 8. Callback functions and expression coverage.

I. Evaluation

The factory provides several API methods, such as `get_coverage()`, to calculate the functional coverage of all covergroups as well as single covergroups or coverpoints. Moreover, it is also possible to just retrieve the hit counter for every individual bin. With help of the API methods the verification engineer can build an evaluation metric. If covergroup type member options were used, such as weights, they will be considered during all calculations.

```
// write collected coverage data in database file
// which was specified during initialization
pFac->write_db();

// the coverage information can be queried
// per factory, covergroup, coverpoint
cout << pFac->get_coverage() << endl;
cout << cg_one->get_coverage() << endl;
cout << cp_one->get_coverage() << endl;

// also the hits can be retrieved
cout << transBin->get_hits() << endl;
```

Listing 9. Coverage evaluation at the end of simulation.

Besides, the evaluations can also be done *during* the simulation. This allows to alter the simulation specific conditions, such as the constraint solving, depending on the actual coverage results and to steer the stimuli generation into uncovered areas. In listing 9 a simple coverage evaluation is shown that prints the coverage of the functional coverage metric at the end of the simulation.

V. Methodology

In order to fully leverage the presented functional coverage functionalities in SystemC testbenches we propose the usage of the Open Verification Methodology (OVM) [8]. By means of the OVM multi-language release, an efficient and standardized structure of the testbench and its components such as monitors, drivers, scoreboard etc. is introduced into the SystemC ecosystem. Here the coverage monitor can be modeled as a subscriber of an analysis port, decoding all transaction and performing proper sampling. Concerning sampling we consider implicit sampling (so sample without argument, the value is derived from the registered value source) useful on RTL-like SystemC designs where maybe sampling also has to occur in relation to clock events. On the other hand, when simulating more abstract models, e.g. with TLM-2 interfaces, we suggest to make use of explicit sampling, as the nature of payloads etc. can be so general and abstract, the verification engineer should have the possibility to define when, where and how often to sample what piece of data.

VI. Related Work

An earlier version of our SystemC functional coverage library was used to conduct a case study on a CAN bus network in [3]. Despite the connection level was limited to RTL `SC_SIGNAL`s, using the coverage analysis we could identify coverage holes in the stimuli generation as well as unforeseen corner cases.

A. Standards and Industry

In the area of (add-on) verification libraries, the Open Verification Library (OVL) [9] is maintained by Accellera and provides checkers that may work as assertion, assumption or coverage point checkers. The most recent versions such as v2.5 support SystemVerilog, Verilog and VHDL. Unfortunately, there is currently no support for SystemC. Additionally, except SystemVerilog the supported languages are working on RTL level, impeding verification on higher levels of abstraction on more abstract data types. In the area of functional coverage implementations for SystemC, [10] introduces a functional coverage prototype. Unfortunately, just a list of functions without any details is available. In contrast, our library will be open with all details as open source. Apart from that, JEDA Technologies provides commercial products for SystemC code, functional and transition coverage [11]. NextOP Software mainly targets on assertion synthesis and assertion-driven coverage collection and does not focus on SystemC [12].

B. Academic

In [13] the author uses the callback facility of the SystemC SCV library to achieve functional equivalent of SystemVerilog value and (simple) transition coverpoints. Besides the mentioned data structures the author uses the SCVs introspection facility and smart pointer callbacks to tie variables to coverpoints. This leads to automatic sampling of coverpoint which is sometimes not intended. Besides we decided in section III-A not to rely on the SCV facility, the possibility to tie the sampling to custom events (similar to block event expressions in SystemVerilog) or to sample custom values (e.g. string coverage) to coverpoints is not supported. Moreover, unfortunately this approach does not include advanced features such as cross coverage with select operators, illegal bins or default bin declaration. In [14] the authors introduce a verification framework also based on the SystemC Verification Library (SCV) providing a coverage monitor library for functional coverage modeling. The implemented basic coverage operators range from logical `OR`, `equal`, `non-equal`, `greater`, etc. but are not as powerful as the IEEE-1800 SystemVerilog coverage features. In [15] the authors propose a coverage-driven verification methodology approach that uses the their own `bve_cover` class. This class has also been referenced in [16] and [17]. It is reported that the approach allows definition of (illegal) buckets (similar to bins) and cross-coverage. Unfortunately, the authors do not introduce implementation details. In [18] a coverage driven testing policy is proposed whereas Property Specification Language (PSL) expressions which are converted to C++ are used to gather and inspect function coverage information.

Concerning the related work, we can conclude that to our knowledge there is no other free of charge functional coverage library for the OSCI SystemC ecosystem that incorporates value, expression and transition coverage as well as cross coverage. Moreover, to our knowledge there is no particular activity of the OSCI Verification Working Group (VWG) to add functional coverage functionalities to the SCV library.

VII. Outlook

For further development and research activities we see a lot of interesting topics. First of all, missing features of the current implementation in comparison with IEEE-1800 SystemVerilog such as per-bin conditional guards could be implemented easily via vectors of callbacks with return type boolean per bin. Another topic could be improved transition coverage with open ranges and more advanced features. Here we have to find a tradeoff between implementing a heavily C++ meta-programming based facility or defining the desired value sequence as a property and use an assertion language. Whenever the assertion fires we could use this as a covergroup sample or count event. Moreover, as C++ does not have the declarative syntax feel of SystemVerilog the API syntax needs to be more compacted in general. Here more efficient usage of variadic macros techniques and template (meta)-programming is necessary. Unfortunately, when building C/C++ libraries, usually a tradeoff between optimal compilation footprint - with

help of extensive usage of generic programming and template meta-programming - and the necessary amount of time for (re-)compilation is meaningful. As a lot of iteration steps and recompilations are likely for testbench engineering this is an important timing issue. Therefore, the decoupling of management and verification tasks of the testbench environment in, e.g. SystemC/C++ and an interpreter language such as Python is another interesting point of research. In detail, it should be considered to which extend the coupling can be made without SystemC kernel modification, to allow the overall verification library still to run on standard OSCI SystemC installations. Moreover, the database interface should support the upcoming Accellera Unified Coverage Interoperability Standard (UCIS) which is planned to be first released in 2011 [19], [20].

VIII. Conclusion

In this article we presented an approach to implement functional coverage for the SystemC ecosystem, capable of running on the standard OSCI SystemC kernel. We briefly introduced the available functionalities ranging from value coverage, expression coverage, cross coverage and simple transition coverage, by means of multiple example code listings. All introduced functional coverage functionalities are implementations of the IEEE-1800 SystemVerilog covergroup metric. Overall, we see big potential for reduction of the verification effort for coverage-driven verification with SystemC when moving to higher-levels of abstraction. Enabling the coverage-driven verification paradigm for verification closure in SystemC design flows using a functional coverage library will boost SystemC's role as high-level design and verification language (HLDV), in doing so, still being a fully free of charge and open source ecosystem. In the broader scope this library will be combined with the results of other work packages of the BMBF founded SANITAS project, e.g. an enhanced OVM for SystemC, enabling early verification across the entire value-creation chain.

Acknowledgment

This work was partly funded by the DFG Collaborative Research Centre 614 and by the German Ministry of Education and Research (BMBF) through the BMBF project SANITAS (01M3088I). We greatly appreciate the cooperation with the project partners [21].

References

[1] H. Foster. Redefining Verification Performance (Part 2). (2010, August) [Online]. Available: http://blogs.mentor.com/verificationhorizons/blog/2010/08/08/redefining-verification-performance-part-2/

[2] J. Bergeron, "Writing Testbenches: Functional Verification of HDL models. Kluwer Academic Publishers," 2003.

[3] C. Kuznik, G. B. Defo, and W. Müller, "Verification of a CAN bus model in SystemC with functional coverage," in *SIES 2010 Proceedings*, 2010.

[4] IEEE Standard for System Verilog-Unified Hardware Design, Specification, and Verification Language, *IEEE STD 1800-2009*, pp. C1 –1285, 2009.

[5] Open SystemC Initiative, *IEEE Standard SystemC Language Reference Manual*, Open SystemC Initiative Std., 2006.

[6] Open SystemC Initiative, *SystemC Verification Library v1.0p2*, 2006. [Online]. Available: http://www.systemc.org/downloads/standards/

[7] P. Marriott. The What, When, and How Much of functional coverage. EDA Tech Forum Journal. (2006, September) [Online]. Available: http://www.edatechforum.com/volumes/volume-3/september-2006/the-what-when-and-how-much-of-functional-coverage/

[8] Cadence Design Systems, Inc. Open Verification Methodology Multi-Language. [Online]. Available: http://www.ovmworld.org/

[9] Accellera Organization Inc. Open Verification Library (OVL). (2009, May) [Online]. Available: http://www.accellera.org/activities/ovl/

[10] R. Siegmund, U. Hensel, A. Herrholz, and I. Volt. Functional coverage prototype for SystemC-based verification of chipset designs. AMD Dresden Design Center. (2004) [Online]. Available: http://www-ti.informatik.uni-tuebingen.de/~systemc/Documents/Presentation-9-UP_1_siegmund.pdf

[11] JEDA Technologies Inc. JEDA ESL Validation Solution. [Online]. Available: http://www.jedatechnologies.net/base/?q=Products

[12] NextOp Software, Inc. NextOp assertion-based verification. [Online]. Available: http://www.nextopsoftware.com/

[13] K. Schwartz, "A technique for adding functional coverage to SystemC," in *DVCON 2007*. Willamette HDL Inc., 2007.

[14] S. Park and S.-I. Chae, "A C/C++-based functional verification framework using the SystemC verification library," *Rapid System Prototyping, IEEE International Workshop on*, vol. 0, pp. 237–239, 2005.

[15] K. R. G. da Silva, E. U. K. Melcher, G. Araujo, and V. A. Pimenta, "An automatic testbench generation tool for a SystemC functional verification methodology," in *SBCCI '04: Proceedings of the 17th symposium on Integrated circuits and system design*. New York, NY, USA: ACM, 2004, pp. 66–70.

[16] G. S. Silveira, K. R. G. da Silva, and E. U. K. Melcher, "Functional verification of an MPEG-4 decoder design using a random constrained movie generator," in *SBCCI '07: Proceedings of the 20th annual conference on Integrated circuits and systems design*. New York, NY, USA: ACM, 2007, pp. 360–364.

[17] C. L. Rodrigues, K. R. G. da Silva, and H. N. Cunha, "Improving functional verification of embedded systems using hierarchical composition and set theory," in *SAC '09: Proceedings of the 2009 ACM symposium on Applied Computing*. New York, NY, USA: ACM, 2009, pp. 1632–1636.

[18] Y. Lahbib, O. Missaoui, M. Hechkel, D. Lahbib, B. Mohamed-Yosri, and R. Tourki, "Verification flow optimization using an automatic coverage driven testing policy," in *Design and Test of Integrated Systems in Nanoscale Technology, 2006. DTIS 2006. International Conference on*, sept. 2006, pp. 94 –99.

[19] R. Goering, "An inside look at the Unified Coverage Interoperability Standard," *Industry Insight Blog, Cadence Design Systems, Inc.*, March 2010.

[20] R. Ranjan, M. Burns, A. Sarkar, and R. Ho, "What can be expected from the Accellera Unified Coverage Interoperability Standard?" *electronicdesign.com, Penton Media Inc.*, October 2010.

[21] Collaborative verification along the entire value-added chain; "SANITAS" research project launched under management of Infineon. [Online]. Available: http://www.infineon.com/cms/en/corporate/press/news/releases/2009/INFXX200912-018.html