Enriching UVM in SystemC with AMS extensions for randomization and functional coverage*

Thilo Vörtler, Thomas Klotz, Karsten Einwich, Fraunhofer IIS - Design Automation Division - EAS, Dresden, Germany (*thilo.voertler@eas.iis.fraunhofer.de*)

Yao Li, Zhi Wang, Marie-Minerve Louërat, Jean-Paul Chaput, François Pêcheux, Ramy Iskander, LIP6, UMR 7606 SU-UPMC/CNRS, Paris, France (*marie-minerve.louerat@lip6.fr*)

Martin Barnasconi, NXP Semiconductors, Eindhoven, The Netherlands, (*martin.barnasconi@nxp.com*)

***Abstract*—The Universal Verification Methodology (UVM) is a coverage driven verification approach, which has become the standard for the verification of digital systems. The framework provided by UVM makes it possible to create structured test environments, which facilitates the reuse of verification components and scenarios. However, the UVM library is only available for SystemVerilog, limiting the verification of designs at the register transfer level. Recently, UVM has been made available in SystemC/C++, shifting the focus to system-level verification including analog/mixed-signal functions by using SystemC-AMS. However, UVM itself fully relies on features built directly into the SystemVerilog language necessary for constrained randomization and functional coverage. In this paper we propose an API similar to SystemVerilog that enables randomization and coverage in UVM for SystemC. A special focus is the introduction of continuous distribution functions for the randomization of real-value data types and means to capture these real values for functional coverage. These extensions will allow the creation of coverage-based test environments in SystemC and SystemC-AMS, enabling verification of heterogeneous analog/mixed-signal systems.**

Keywords*— Electronic System Level (ESL), SystemC, SystemC-AMS, SystemC Verification (SCV), Transaction Level Modeling (TLM), Universal Verification Methodology (UVM), Constrained Random Stimulus, Functional Coverage

I. INTRODUCTION

Today's embedded systems interact more and more tightly with the analog physical environment. Digital hardware/software (HW/SW) subsystems become functionally interwoven with analog/mixed-signal (AMS) blocks such as RF interfaces, power electronics, or sensors and actuators to form truly heterogeneous systems. Examples are software-defined radios, sensor networks, automotive applications or systems for image sensing. This requires new means to model and simulate the interaction between AMS subsystems and HW/SW subsystems at functional and architecture level. Especially for this purpose, the SystemC [1] language standard has been extended with powerful AMS [2] modeling capabilities to tackle the challenges in heterogeneous electronic system-level architecture-exploration and design phases.

Yet, as great effort was made to mature system-level design and modeling technologies, less was made to improve the system-level verification approaches in SystemC. Coverage-driven verification of complex digital IP has become more mature since the introduction of the UVM standard, implemented in SystemVerilog [3]. The UVM principle is to build a test bench using reusable verification components, and introducing a structured way for constraint randomization and functional coverage. When applying UVM, the test bench is designed in a hierarchical and modular way, using similar abstraction concepts as applied in the device under test (DUT). This includes techniques such as transaction level modeling (TLM) for the test sequences, combined with cycle accurate, signal-level interfaces to the DUT. To support system-level verification of SystemC-centric HW/SW systems, UVM has been made available in SystemC [4].

In this paper we propose new APIs dedicated to verification for coverage and randomization, as extension for the UVM-SystemC library. Our API supports the random generation of real values, following continuous distribution functions, which are subject to constraints. Furthermore, a functional coverage API is introduced, based on covergroups, coverpoints, and coverbins, enabling coverage collection using SystemC. This API also supports coverage of real values.

* This work was funded by the project Verification for Heterogeneous Reliable Design and Integration (VERDI) [13] , which is supported by the European Commission within the 7th Framework Programme for Research and Technological Development (FP7/ICT 287562).

In the following section we describe how randomization and coverage are applied in UVM for SystemC and SystemC-AMS. The API for randomization is described in Section III. As a backend for randomization of integer- and SystemC-based data types, we use the CRAVE library. For constrained randomization of real values, the randomization features and distribution functions of C++11 are used. The proposed functional coverage API is described afterwards in Section IV.

II. UVM FOR SYSTEMC AND SYSTEMC-AMS

A. *UVM-SystemC randomization and coverage concepts*

UVM is a verification framework that allows the creation of test benches based on a constrained random stimulus principle. Instead of testing the DUT with directed test sequences, random stimulus is applied, which is shaped by constraints so that the randomly generated values are valid stimulus. As the input stimulus is randomly generated, it is very important to observe which data has been sent to the DUT, to make sure that all design corners have been tested during a verification regression run. Therefore, functional coverage can be used which allows to define own coverage goals.

The UVM standard and associated class library implementation in SystemVerilog does not define the constructs for randomization and functional coverage, because these concepts are intrinsically part of the SystemVerilog standard, defined in IEEE Std. 1800 [5] . In a similar way, UVM in SystemC will not introduce such constructs for randomization and coverage, but will make use of dedicated libraries for this purpose. Several good attempts have been made to support constrained randomization for SystemC, such as the SystemC Verification library (SCV) [6] and the Constrained Random Verification Environment for SystemC (CRAVE) [7]. Also different libraries that add functional coverage to SystemC have been proposed in [8][9][10]. Furthermore, various commercial, proprietary or vendor-specific solutions are available.

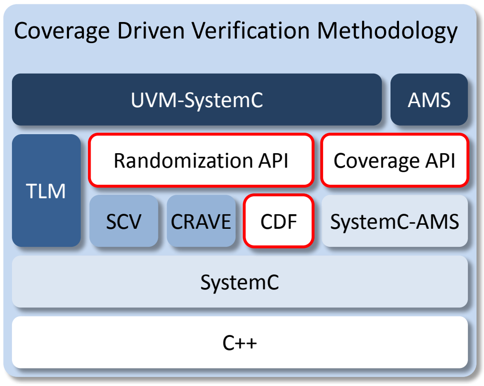

Figure 1: Overview on languages and libraries used for UVM-SystemC

In Figure 1 it is shown how UVM-SystemC defines the upper methodology layer which unifies various powerful SystemC extension libraries such as TLM, SCV, CRAVE and SystemC-AMS. Unfortunately, there has been no attempt so far to standardize the randomization and coverage API for SystemC. Due to the growing interest in UVM, we propose an API for randomization and coverage, indicated in red, which is clearly recognized by the UVM community, as it offers a *look & feel* similar to the SystemVerilog language. The randomization API is proposed as compatibility layer that can either use CRAVE or SCV as a backend solver. In addition, continuous distribution functions (CDF) are introduced to generate random real values, which are essential in an AMS verification environment.

B. *Application of randomization and coverage in UVM-SystemC*

Figure 2 shows a typical verification environment implemented in UVM-SystemC and SystemC-AMS. It contains a top level environment with two separate Verification IPs (VIPs), a scoreboard and a virtual sequencer. All components can be configured using the UVM configuration database. A virtual sequence ① running on a virtual sequencer coordinates the execution of the lower level sequences ② running on sequencers which are part of the agents. These sequences generate a stream of sequence items (transactions) that are translated into pin level signals, which are sent to the DUT via a interface[1] via a driver ③. Throughout the generation of sequences, from virtual sequences down to sequence items sent to a driver, randomization can be applied (shown in red). For example, sequences can be randomly selected to be run or data fields can be randomized.

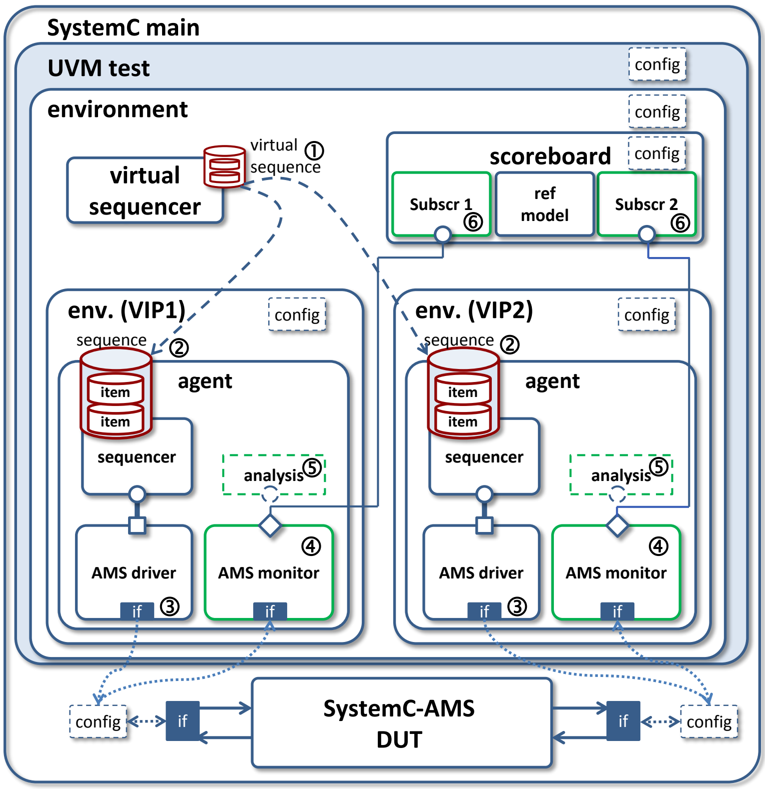

Figure 2: UVM-SystemC test environment and use of randomization and coverage

Functional coverage in UVM can be collected by adding coverage models at different levels of abstractions (shown in green). Coverage that needs explicit access to signals is collected in a monitor ④. More abstract coverage information is collected based on transactions, which a monitor provides through an analysis port. Reusable coverage that is related to an interface, which an VIP implements can be collected in an optional analysis component ⑤. If the functional coverage is related to the overall checking of the verification goals it is collected as part of the overall correctness checks in a scoreboard ⑥.

III. RANDOMIZATION FOR UVM-SYSTEMC

A. *UVM-SystemC compatibility layer definition*

The UVM-SystemC compatibility layer defines constructs for randomization and constraints which can be mapped onto the SCV or CRAVE library implementation. Table I lists the basic language constructs which are introduced, and their equivalence with the SystemVerilog API. Especially in the context of UVM-SystemC, it is preferred to use similar language constructs and functionalities, to ease the introduction of UVM in different languages.

Table I: Basic language constructs as part of the UVM-SystemC compatibility layer for randomization (**scvx**)

Functionality	**SystemVerilog**	**UVM-SystemC (scvx)**
Random variable declaration	**rand T**	**scvx_rand<T>**
Enable or disable random variable	**rand_mode(…)**	**rand_mode(…)**
Constraint block declaration	**constraint**	**scvx_constraint**

[1] As the SystemC language doesn't rquires virtual interfaces as in SystemVerilog, an interface is a pointer to a class, which is retrieved via the UVM configuration database similar to UVM for SystemVerilog.

Functionality	SystemVerilog	UVM-SystemC (scvx)
Enable or disable constraint	**constraint_mode(…)**	**constraint_mode(…)**
Randomization container object	-	**scvx_rand_object**
Randomize method	**randomize()**	**randomize()**
Randomize method with inline constraint	**randomize() with …**	**randomize_with(…)**

These language constructs are proposed as an extension to the SCV API, and therefore use the initial prefix and namespace **scvx**. If accepted, these constructs might become integral part of the SCV library and its **scv** namespace.

Random variables are declared using the template class **scvx_rand<T>**, in which **T** represents a C, C++ or SystemC data type. The member function **rand_mode**, which is part of this class, accepts the arguments *true* or *false* to respectively activate or inactivate the randomization of the variable. The constraint declaration uses the class **scvx_constraint**. Also constraints can be made active and inactive, by means of the member function **constraint_mode**.

In contrast to SystemVerilog, where any class can contain random variables and constraints, a dedicated randomization container base class called **scvx_rand_object** is introduced. For the UVM-SystemC class library implementation, it is proposed to derive class **uvm_object** from class **scvx_rand_object**. In this case, all UVM objects derived from **uvm_object** can be used to encapsulate randomized variables and constraints. As the UVM class **uvm_sequence_item** is also derived from class **uvm_object**, there is no need for the user to explicitly use the base class **scvx_rand_object** to create randomized objects.

So any object derived, or indirectly derived from base class **scvx_rand_object** may contain multiple random variables and constraints, which then belong together, meaning that the declared variables in this class can be used in the constraint definitions and randomization process. The member function **randomize**, which is part of the base class, assigns the random value to the declared variables in the derived class, taking into account the constraints, if any. Alternatively, the member function **randomize_with** can be used, which facilitates the declaration of additional in-line constraints, these are only valid for that particular randomization call. The member functions **randomize** and **randomize_with** return *true* if randomization was successful, otherwise they will return *false*.

B. *UVM-SystemC compatibility layer implementation on top of CRAVE*

Listing 1 demonstrates the use of the randomization API using the CRAVE library. This library has been selected due to its improved capabilities compared to the SCV library, for example the inline and incremental constraint definition and the more powerful parallel constraint solving. For simplicity, the example does not show the use of UVM-SystemC, but solely the randomization capabilities of the compatibility layer. Line 1 defines the class *simplesum*, which is derived from class **scvx_rand_object**. The variables to be randomized are declared in line 4. The constraints are defined on line 5. As part of the constructor initialization, line 8 and 9, all variables and constraints get a name. If initialization is omitted by the user, default names will be assigned to these elements. The constraint definitions are part of the constructor implementation, at line 11 to 13. In this example, constraint *c1* defines the equation ‘$z = x + y$’. A helper function at line 16 is defined to print the result.

A special class **scvx_name** is introduced, as argument in the constructor (line 7), which acts as a container to store the string name of the current instance and provides the mechanism for building the hierarchical names between parent and child objects.

The implementation of the main program is given from line 23 and onwards. The instantiation of the randomization object *simplesum* is done in line 25. The actual randomization of the variables, taking into account the constraints, is executed on line 27. If randomization was successful, the result is printed (line 29). Otherwise the message is printed that there was no solution found. At line 32, constraint *c2* of randomization object *s* is disabled. Line 34 shows the use of the member function **randomize_with**, which starts randomization using an additional in-line constraint, being ‘x == 10’. Note that the randomization object instance name should be explicitly added to the constraint definitions, because these variables reside in the object itself. Line 39 shows how variables can be excluded from randomization. In this case, the actual value of variable *y* after the second randomization request is kept the same.

```
 1  class simplesum : public scvx::scvx_rand_object
 2  {
 3   public :
 4    scvx::scvx_rand< int > x, y, z ;
 5    scvx::scvx_constraint c1, c2, c3 ;
 6
 7    simplesum( scvx::scvx_name name )
 8    : x("x"), y("y"), z("z"),
 9      c1("c1"), c2("c2"), c3("c3")
10    {
11      c1( z() == x() + y() );
12      c2( x() == 5 );
13      c3( y() > 0 && y() < 10 );
14    }
15
16    void print_result() const
17    {
18      cout << name() << "  : "  << z << "  == "
19           << x << "  + " << y << endl ;
20    }
21
22  }; // class simplesum
```

```
23  int sc_main(int, char*[])
24  {
25    simplesum s("simplesum");
26
27    bool result = s.randomize();
28
29    if (result) s.print_result();
30    else cout << "No solution found." << endl;
31
32    s.c2.constraint_mode( false );
33
34    result = s.randomize_with( s.x() == 10 );
35
36    if (result) s.print_result();
37    else cout << "No solution found." << endl;
38
39    s.y.rand_mode( false );
40
41    result = s.randomize();
42
43    if (result) s.print_result();
44    else cout << "No solution found." << endl;
45
46    return 0;
47  }
```

Listing 1: Example of UVM-SystemC compatibility layer for constrained randomization

The generated output is shown in Listing 2.

```
$ ./simplesum.exe

constraint c1 registered.
constraint c2 registered.
constraint c3 registered.
simplesum: 13 == 5 + 8
constraint c2 disabled.
in-line constraint ci_0 applied (disabled after use)
simplesum: 13 == 10 + 3
random variable 'y' made inactive (value remains 3).
simplesum: 33556493 == 33556490 + 3
```

Listing 2: Output of constrained randomization example

C. *Randomization for Analog/Mixed-Signal systems*

In addition to the discretized weighted values, a set of continuous distribution functions for supporting real values have been introduced. The randomization features and distribution functions of C++11 [11] are used to build this API. The continuous distribution functions made available are listed in Table II.

Table II: Continuous Distribution Functions available for real values in UVM-SystemC (**scvx**)

Distribution function	**UVM-SystemC (scvx)**
Normal distribution	**scvx_normal_distribution**
Uniform distribution	**scvx_uniform_real_distribution**
Bernoulli distribution	**scvx_bernoulli_distribution**
Piece-wise linear probability distribution function	**scvx_piecewise_linear_probability_distribution**
Discretized probability distribution function	**scvx_discrete_probability_distribution**

The distribution function is set as a parameter of the random variable, as shown in Listing 3.

```
scvx::scvx_rand<real> a
a.set_distribution( dist_type(dist_params) );
```

Listing 3: Setting a distribution function for a random variable

The member function **set_distribution** defines the distribution *dist_type*, requiring the parameters *dist_params*, for random variable *a*. The seed used to generate the random variables may be set globally or per **scvx_rand_object**. Therefore the execution of some test scenario can be reproduced identically several times. As an example, Figure 3 presents the random samples, sorted by value generated for a uniform distribution, showing the real values, compared with integer values that would be generated by CRAVE.

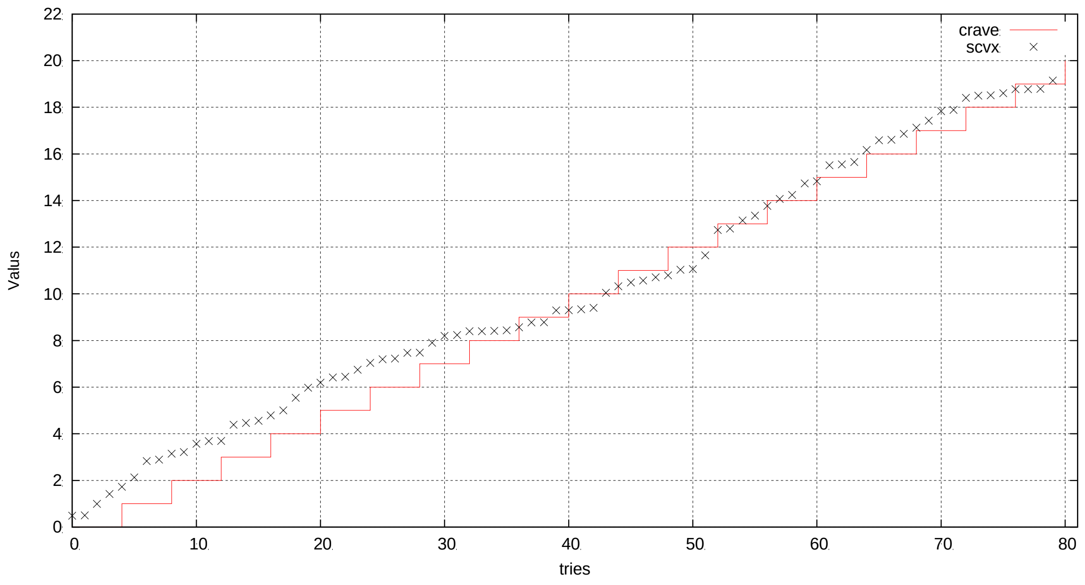

Figure 3: Uniform distribution example using **scvx_uniform_real_distribution** (cross points) compared to discretized uniform distribution provided by CRAVE (red ladder).

Constraints can be set on randomized variables, using the same operators as supported in SystemVerilog. Table III presents these operators.

Table III: Operator supported by randomized real values in UVM-SystemC (**scvx**)

Operator	**Function**	**Operator**	**Function**
= =	Equality	%	Modulo
!=	Inequality	&&	Logical AND
>	Greater-than	||	Logical OR
<	Less-than	!	Logical negation
>=	Greater-than-or-equal-to	&	Bitwise AND
<=	Less-than-or-equal-to	|	Bitwise OR
+	Addition	^	Bitwise XOR
-	Subtraction	<<	Shift-left
*	Multiplication	>>	Shift-right
/	Division	~	Bitwise negation

This API has been used to update the *simplesum* use case (resulting in *realsimplesum*) with real randomized values *a* and *b*. The distribution function is set uniform. The following three constraints are set on the variables: $a + b < 36$, $a > 18$ and $b < 16$. The generated output is shown in Listing 4.

```
Sequence starts here...
-------randomization with constraints--------
*  Definition of random variables and constraints :
* a: uniform real distribution from 15.3 to 40.2
* b: uniform real distribution from 2.5 to 20.5
* the additional constraint on variable a is: a > 18
* the additional constraint on variable b is: b < 16
* the constraint on variable a and b is: a + b < 36

*--- 1st sequence is randomized here ---*
      1| v[0,"a"]:39.2135
      1| v[1,"b"]:19.7869
scvx_constraint <b_less_16> has not been met.
scvx_constraint <a_plus_b_less_36> has not been met.
      2| v[0,"a"]:29.8232
      2| v[1,"b"]:12.9987
scvx_constraint <a_plus_b_less_36> has not been met.
      3| v[0,"a"]:18.628
      3| v[1,"b"]:4.90581
Sequence finished.
0 s: test.tb.uvc0.agent.monitor changed DUT inputs op_a = 18.628 op_b = 4.90581
0 s: test.tb.monitor0 received  result 23.5338
0 s: test.tb.scoreboard0 Successfully compared adder output 23.5338
```

Listing 4: Output of constrained randomization example *realsimplesum* with real values

IV. FUNCTIONAL COVERAGE EXTENSIONS FOR UVM-SYSTEMC

This section describes the proposed functional coverage API for UVM-SystemC. Table IV defines the language constructs which are proposed as extension to the SystemC Verification library, and therefore use the initial prefix and namespace **scvx**.

Table IV: Basic language constructs for functional coverage in UVM-SystemC (**scvx**)

Functionality	**SystemVerilog**	**UVM-SystemC (scvx)**
Coverage model	**covergroup**	**scvx_covergroup**
Coverage points	**coverpoint**	**scvx_coverpoint**
Coverage state bins	**bins**	**bins()**
Illegal bins	**illegal_bins**	**illegal_bins()**
Ignore bins	**ignore_bins**	**ignore_bins()**
Triggers sampling of the covergroup	**sample()**	**sample()**
Option to specify the maximum of automatically created bins	**option.auto_bin_max**	**option.auto_bin_max**
Coverage option to specify an additional comment	**option.comment**	**option.comment**
Option to specify the weight of the covergroup instance	**option.weight**	**option.weight**
Option to specify the name of the covergroup instance	**option.name**	**option.name**

In Listing 5 the use of the functional coverage API is demonstrated.

```
class cg: public scvx::scvx_covergroup
{
 public:
  scvx::scvx_coverpoint cp_m;
  scvx::scvx_coverpoint cp_n;

  cg( scvx::scvx_name name, int& m, int& n )
  : cp_m( "cp_m", m ),
    cp_n( "cp_n", n )
  {
    option.auto_bin_max = 16;

    cp_m.bins("bin_a") =
      scvx::list_of(4,  0, 1, 2, 3 );

    cp_m.bins("bin_b", scvx::SINGLE_BIN) =
      scvx::list_of(4,  4, 5, 6, 7 );
```

```
int sc_main(int, char*[])
{
  int m; // variable to be covered
  int n; // variable to be covered

  int stimuli_m[] =
    { 3, 5, 6, 5, 3, 6, 5, 5, 3, 3 };

  int stimuli_n[] =
    { 13, 1, 6, 3, 16, 12, 8, 3, 13, 3 };

  cg cg_inst("cg_inst", m, n);

  for ( int i = 0; i < 10; i++ )
  {
    m = stimuli_m[i];
    n = stimuli_n[i];
18
19      cp_m.ignore_bins("ignore_bins") = 6;
20      cp_n.ignore_bins("ignore_bins") = 13;
21    }
22  };
```

```
40      cg_inst.sample();
41    }
42    cg_inst.report();
43    return 0;
44  }
```

Listing 5: Example of functional coverage API for UVM-SystemC

The coverage model defined in class *cg* makes use of the base class **scvx_covergroup**, see line 1. The coverpoints of type **scvx_coverpoint** are children in this covergroup object (line 4 and 5). As part of the constructor initialization list, the names for the coverpoints are specified, and also they are bound to their associated data members (line 8 and 9). In the implementation of the constructor, coverage bins can be defined. For this purpose, the member function **bins** of class **scvx_coverpoint** is introduced. The name of the coverage bin is specified as first argument of this member function. If the second argument is not given, or explicitly defined as **SINGLE_BIN**, then it will create a bin which can hold a single value. Alternatively, the argument **MULTI_BIN** can be used to assign multiple values per bin.

The number of individual single bins and the value it can collect is specified as a standard vector of values, by means of the helper function **list_of**, which is also part of the SCV extension library. In this example, the following bins are created: bin_a[0], bin_a[1], bin_a[2], bin_a[3], bin_a[4], bin_b[4], bin_b[5], and bin_b[7]. In addition to the **list_of** helper function, a function **range_of** is supported that defines bins which can capture real values between an upper and lower real-value bound.

The type and number of coverage bin(s) can be decided per coverpoint. In the example in Listing 5, only bins are explicitly specified for coverpoint *cp_m* only. If there are no bins specified by the user, like for coverpoint *cp_n*, a default set of bins is automatically created (so called 'autobins'). The number of default coverage bins is determined by the size of the data type which is covered. When using integers as data type, the number of default bins would explode; therefore the coverage option **auto_bin_max** (line 16) can be specified to limit the number of coverage bins. By default, **auto_bin_max** is set to 64.

The member function **ignore_bins** is introduced, which specifies one or more values to be explicitly excluded from coverage. In this example, the value 6 is ignored for coverpoint *cp_m*, and thus excluded in the coverage calculation (line 19). For coverpoint *cp_n*, where default bins are created, the value 13 is ignored, see line 21. In a similar fashion, the member function **illegal_bins** could be used. When a value is stored in an illegal bin, a run-time error is generated.

The bin is said to be *covered* (100%) as soon as at least one of the specified values is stored in the coverage bin. In case multiple hits per bin occur, the property 'hitrate' will increase. In this coverage model, there is no tracking of the individual values for multi-value bins.

```
$ ./test.exe

Covergroup: cg_inst
----------------------------------------
VARIABLE      Expected  Covered  Percent
----------------------------------------
cp_m                 7        2    28.57
cp_n                15        5    33.33
----------------------------------------
TOTAL:              22        7    31.82
----------------------------------------

coverpoint: cp_m
---------------------------
Name     Percent     Hitrate
---------------------------
bin_a[0]       0           0
bin_a[1]       0           0
bin_a[2]       0           0
bin_a[3]     100           4
bin_b[4]       0           0
bin_b[5]     100           4
bin_b[7]       0           0
---------------------------
```

```
coverpoint: cp_n
---------------------------
Name     Percent     Hitrate
---------------------------
auto[0]      100           0
auto[1]      100           1
auto[2]        0           0
auto[3]      100           3
auto[4]        0           0
auto[5]        0           0
auto[6]      100           1
auto[7]        0           0
auto[8]      100           1
auto[9]        0           0
auto[10]       0           0
auto[11]       0           0
auto[12]     100           1
auto[14]       0           0
auto[15]       0           0
---------------------------
```

Listing 6: Output of functional coverage example

The member function **sample** of the coverage group *cg_inst* is called to perform the actual coverage analysis (line 40). The member function **report** of the coverage group *cg_inst* has been created to print the coverage results to the console (stdout). In the future, it is expected that the coverage results are written to a coverage database following the Accellera Unified Coverage Interoperability Standard (UCIS) [12].

The generated functional coverage output is shown in Listing 6. Note the missing bins bin_b[6] and auto[13], which are defined as **ignore_bins**.

V. Conclusions

In this paper we have presented constrained randomization and functional coverage extensions for UVM in SystemC. New features dedicated to AMS verification have been introduced, namely the random generation of real values, which can be subjected to constraints, supported by the use of continuous distribution functions. These randomization extensions use a syntax similar to the SystemVerilog language standard and are implemented in a compatibility layer on top of existing constrained randomomization libraries such as SCV or CRAVE. Furthermore, a functional coverage API is presented for UVM-SystemC, introducing covergroups and coverpoints, enabling coverage collection of the results also in SystemC.

These concepts are being contributed for further standardization to the Accellera Systems Initiative, as an extension to the SystemC Verification (SCV) library.

References

[1] IEEE Computer Society, IEEE Std. 1666-2005, IEEE Standard SystemC Language Reference Manual, http://standards.ieee.org/findstds/standard/1666-2011.html

[2] Accellera Systems Initiative, SystemC AMS 2.0 Standard, http://www.accellera.org/downloads/standards/systemc

[3] Accellera Systems Initiative, Standard Universal Verification Methodology (UVM), http://www.accellera.org/downloads/standards/uvm/

[4] M. Barnasconi, F. Pêcheux, T. Vörtler, K. Einwich, Advancing System-Level Verification Using UVM in SystemC, DVCON 2014, March 2014, San Jose, California, USA

[5] IEEE Computer Society, IEEE Std. 1800, IEEE SystemVerilog Unified Hardware Design, Specification, and Verification Language, http://standards.ieee.org/findstds/standard/1800-2012.html

[6] Accellera Systems Initiative, SystemC Verification Library, http://www.accellera.org/downloads/standards/systemc

[7] F. Haedicke, H. M. Le, D. Große, R. Drechsler, CRAVE: An Advanced Constrained Random Verification Environment for SystemC, International Symposium on System on Chip (SoC), October 2012, Tampere, Finland

[8] R. Siegmund, U. Hensel, A. Herrholz, and I. Volt. A functional coverage prototype for SystemC-based verification of chipset designs, 9th European SystemC User Group Meeting, Design Automation and Test in Europe (DATE) conference, February 2004, Paris, France.

[9] K. Schwartz, A technique for adding functional coverage to SystemC, DVCON 2007, February 2007, San Jose, California, USA.

[10] M. F. S. Oliveira, C. Kuznik, W. Mueller, W. Ecker, V. Esen, A SystemC Library for Advanced TLM Verification, DVCON 2012, March 2012, San Jose, California, USA.

[11] The C++ Standards Committee, 2011 C++11, http://www.open-std.org/jtc1/sc22/wg21/

[12] Accellera Systems Initiative, Unified Coverage Interoperability Standard (UCIS), http://www.accellera.org/downloads/standards/ucis

[13] The VERDI project, part of Seventh Framework Programme (FP7), http://verdi-fp7.eu