The System Verification Methodology for Advanced TLM Verification

Marcio F. S. Oliveira[1] Finn Haedicke[2] Rolf Drechsler[2,3]
Christoph Kuznik[1] Hoang M. Le[2] Wolfgang Ecker[4]
Wolfgang Mueller[1] Daniel Große[2] Volkan Esen[4]

[1]C-LAB, University of Paderborn, 33102 Paderborn, Germany
{marcio, kuznik, wolfgang}@c-lab.de
[2]Institute of Computer Science, University of Bremen, 28359 Bremen, Germany
{finn, hle, grosse, drechsle}@informatik.uni-bremen.de
[3]Cyber-Physical Systems, DFKI GmbH, 28359 Bremen, Germany
[4]Infineon Technologies AG, 85579 Neubiberg, Germany
{Wolfgang.Ecker, Volkan.Esen}@infineon.com

ABSTRACT

The IEEE-1800 SystemVerilog [20] system description and verification language integrates dedicated verification features, like constraint random stimulus generation and functional coverage, which are the building blocks of the *Universal Verification Methodology* (UVM) [3], the emerging standard for electronic systems verification. In this article, we introduce our *System Verification Methodology* (SVM) as a SystemC library for advanced *Transaction Level Modeling* (TLM) testbench implementation. As such, we first present SystemC libraries for the support of verification features like functional coverage and constrained random stimulus generation. Thereafter, we introduce the SVM with advanced TLM support based on SystemC and compare it to UVM and related approaches. Finally, we demonstrate the application of our SVM by means of a testbench for a two wheel self-balancing electric vehicle.

Categories and Subject Descriptors

J.6 [**Computer-Aided Engineering**]: [Computer-Aided Design (CAD)]

General Terms

Verification, Standardization

Keywords

UVM, SystemVerilog, SystemC, Functional Coverage, Constrained Random Stimulus Generation

CODES+ISSS'12, October 7-12, 2012, Tampere, Finland.

1. INTRODUCTION

As electronic systems grow in complexity, verification activities have started to dominate the design effort. Today, verification accounts for more than 70% of the total development effort [27] with the trend to an exponential growth [15]. To cope with that increasing complexity and to boost productivity at *Electronic System Level* (ESL), more efficient verification languages and technologies have been introduced. Today, advanced verification languages like IEEE-1800 SystemVerilog [20] and IEEE-1647 *e* [22] support features like assertion specification to define behavioral properties, constrained random stimulus generation to generate directed randomized testbench input, and functional coverage definitions to determine the verification closure.

For the structured implementation of testbenches, the *Universal Verification Methodology* (UVM) [3] with the *Open Verification Methodology* (OVM) [8] as a predecessor is well accepted for ESL testbench implementation. UVM is a methodology with a complementary library for RTL and ESL verification, at which the full feature set comes as a SystemVerilog and *e* implementation only.

On the other hand, IEEE 1666 SystemC is widely accepted for description and simulation of mixed HW/SW systems at higher levels of abstraction. For instance, the TLM 2.0 library has already been introduced in 2007 [4] and is meanwhile integrated in the IEEE 1666-2011 SystemC standard [21]. However, in contrast to SystemVerilog, the support of dedicated verification features is very limited in SystemC. This is an obstacle for adapting verification library concepts like UVM to SystemC. For instance, SystemC has no built-in language construct for functional coverage specification such as the `covergroup` metric of SystemVerilog. Therefore, there exists an OVM implementation in SystemC (OVM-SC) [10], which is limited to an OVM subset and lacks various important modeling and verification features like domain specific components, random stimulus generation facilities, sequence library management, sequence arbitration, response to request routing, command-line processor, and a register abstraction layer. Nevertheless, the

SystemC Verification (SCV) library [31] can be used as an alternative for random stimulus generation and constraint solving. However, it has several drawbacks: poor dynamic constraint support, no constraint specification for dynamic data-structures, low constraints specification usability for composed data structures, insufficient information in case of over-constraining, and substantial limits in constraints complexity.

This paper presents the *System Verification Methodology* (SVM) for SystemC, which is based on the OVM-SC [10] implementation and incorporates verification best practices and standards from OVM and partly UVM, such as factory and configuration facilities and stimulus sequence generation and management. The SVM library integrates functional coverage and advanced constraint solving and constrained random stimulus generation as mandatory features to SystemC for true testbench support.

The remainder of this paper is organized as follows. The next section presents related work in the areas of verification tools and methodologies for the SystemC ecosystem. Section 3 introduces our contributions to improve the SystemC verification feature set. In Section 4 we present the SVM, which integrates the two previous contribution in a feature rich library to boost the verification in SystemC environments. In Section 5 we apply our SVM library to a high level control model of a two wheel self-balancing electric vehicle. Finally, Section 6 closes with conclusions and future directions.

2. RELATED WORK

For testbench implementation, several verification methodologies were developed such as the *Universal Reuse Methodology* (URM) [9] from Cadence, the *Advanced Verification Methodology* (AVM) [28] from Mentor Graphics, and the *Verification Methodology Manual* (VMM) [5] from Synopsys. Cadence and Mentor Graphics first joined efforts to the *Open Verification Methodology* (OVM) [8] with a later unification to the Universal Verification Methodology (UVM) [3] with Synopsys. Additionally, the *Open Verification Library* (OVL) [2] was developed by Accellera. OVL provides means that may work as assertion, assumption, or coverage point checkers. All verification methodologies came with a handbook and a complementary SystemVerilog library for testbench implementation. There is no SystemC support except an OVM implementation published by Accellera which implemented a very limited OVM subset in SystemC, i.e., OVM- SC [10], which was the starting point for our SVM implementation.

More precisely, our SVM SystemC library is based on the concepts of OVM multiple languages SystemC package v2.1.1 and OVM for SystemVerilog v2.1.1 where its base package was refactored to reflect the improvements from the transition of OVM to UVM additionally improved by callbacks, transaction routing, and recording facilities. We also included a package with crucial structural components to build the verification environment. For example, we implemented stimulus sequences, sequence scheduler facilities, as well as a command-line processor.

All of those libraries are based on dedicated verification features from languages like IEEE-1800 SystemVerilog [20] and IEEE-1647 *e* [22]. Therefore, a direct conversion of those verification libraries to SystemC is not possible. Thus, we first had to extend SystemC by functional coverage and advanced constrained random stimulus generation, which we presented in [26] and [19].

For SystemC functional coverage, [34] presents a vendor-specific functional coverage implementation in SystemC. The author of [33] uses the callback facility of the SCV library to achieve functional equivalent of SystemVerilog value and (simple) transition coverpoints. Moreover, the SCV introspection facility and smart pointer callbacks are used to tie variables to coverpoints. This leads to automatic sampling of coverpoints, which is often not intended. Furthermore, this approach does not include advanced features such as cross coverage with select operators, illegal bins or default bin declaration. Moreover, various vendors provide commercial products for SystemC code, and limited functional and transition coverage such as [23, 29].

For constrained random stimulus generation, the SCV library [31] provides a SystemC implementation. However, it has limits in dynamic constraint support, dynamic data structures, composed data structures, over-constraining, and the complexity of constraints. Therefore, several improvements for the SCV library have been introduced. In [16] bit vector operators have been added and the uniform distribution among all constraint solutions is ensured in all cases. An approach to determine the exact reasons in case of over-constraining has been presented in [17]. In [37] the BDD-based constraint-solver is replaced by a method which uses a generalization of *Boolean Satisfiability* (SAT).

In contrast to our implementation, all these approaches compensate only some of the SCV weaknesses. In particular, no constraints on dynamic data structures can be specified, constraints cannot be controlled dynamically during run-time, references to the state of constraints are not available, and no inline constraints are possible limiting the usability. In addition, the integration of different constraint solvers working in parallel is mandatory to reduce the time for stimulus generation to a minimum.

3. SYSTEMC EXTENSIONS FOR VERIFICATION

Before we introduce our *System Verification Methodology* (SVM) library, we first present SystemC verification extensions, which are required as a basis for SVM, i.e., functional coverage and advanced constrained random stimulus generation.

3.1 Functional Coverage

Functional coverage is a user-defined metric that measures how many percentages of the verification objectives are met by the test plan. It can be, for instance, used to measure if interesting scenarios, corner cases or specification invariants have been observed, validated, and tested by means of covergroups and coverpoints. The coverage of a group C_g is the weighted average of the coverage of all items i defined in the group. It is computed by the following equation:

$$C_g = \frac{\sum_i W_i \cdot C_i}{\sum_i W_i},$$

whereas W_i is the weight associated with item i and C_i is the coverage of item i. The remainder of this section outlines details of the functional coverage implementation of the SVM as a SystemC library, which is based on the following assumptions:

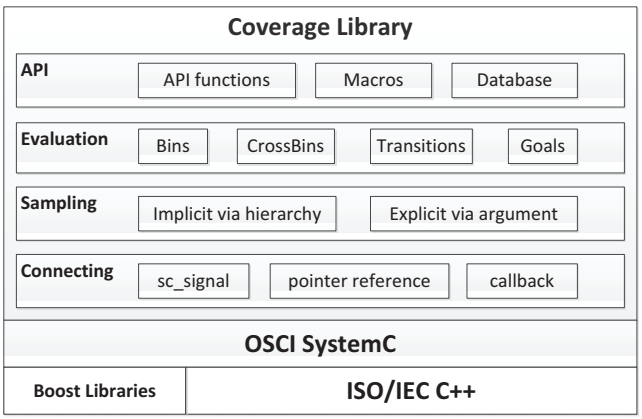

Figure 1: Elements according to IEEE-1800 **covergroup** concept.

- Metric Expressiveness
  A structured hierarchical composition of a functional coverage metric shall be enabled. The metric elements shall allow the capturing of coverage information in fine-grained fashion, e.g., like the concept of covergroups, coverpoints and bins in SystemVerilog.
- Interoperability
  To collect and evaluate functional coverage no modifications of the SystemC kernel are permitted.
- Build Environment Dependencies
  The functional coverage facility shall not rely on the SystemC Verification Library (SCV). Moreover, it may only use header file includes of third-party libraries such as the boost library.
- Advanced TLM Verification
  Multiple explicit samplings per delta cycle shall be supported to allow coverage of pre and post transaction event coverage.

The main facility of the functional coverage library is a single factory class providing all necessary setup and management API calls for the creation and administration of the implemented coverage metric elements. Moreover, the factory handles the administration of the coverage database, which stores collected functional coverage information **(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.

3.1.1 Functional Coverage Metric

The library is organized in different layers, as given in Figure 1. Besides explicit sampling, a connection layer allows the binding of coverpoints to `sc_signals`, variables or callback functions of the DUT (Design under Test). The evaluation layers provide the implementation of the SystemVerilog metric elements such as covergroups, coverpoints, and bins as well as the definition of associated goals and weights for the overall functional coverage computation.

3.1.2 Coverage Features

As such, the coverage library implements covergroups, coverpoints, and coverpoints. Each coverpoint may contain an arbitrary number of (normal) bins, illegal bins, ignore bins, and an optional default bin. 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. Each bin may also have numerous integer intervals assigned. A coverpoint can contain transition bins, which may be associated with an arbitrary number of integer sequences to implement simple successional value transition coverage. Allowing multiple samplings within one SystemC simulator delta cycle, we do not rely on `SC_MODULE`s and clock sensitive `SC_METHOD`s within the implementation. The sampling process can be triggered from the outside in a method call fashion.

3.2 Advanced Constrained Random Stimulus Generation

Constrained random verification applies stimuli to the DUT that are solutions of constraints. These solutions are determined by a constraint solver. Thereby, the generated stimuli are much more likely to hit corner cases. This is due to the fact that scenarios are simulated which the verification engineer might have not thought of. Furthermore, the stimulus generation process is automated and hence a huge set of scenarios can be executed leading to higher coverage. Hence, for large and complex systems the confidence in the correct functionality significantly increases.

In the following we describe our developed constrained random stimulus generation library.

3.2.1 Features

Our constrained random stimulus generation is based on CRAVE [19], an advanced *C*onstrained *RA*ndom *V*erification *E*nvironment for SystemC. CRAVE provides the following features:

- New Constraint Specification API
  An intuitive and user-friendly API to specify random variables and random objects has been developed.
- Dynamic Constraints and Data Structures
  Constraints can be controlled dynamically at run-time. Moreover, constraints for elements of dynamic data structures like STL vectors can be specified.
- Improved Usability
  Inline constraints can be formulated and changed incrementally at run-time. Furthermore, automatic debugging of unsatisfiable constraints is supported.
- Parallel Constraint Solving
  BDD-based and SAT/SMT-based techniques have been integrated for constraint solving. A portfolio approach is used to enable very fast generation of constraint solutions.

Those features are supported as follows, where we take the example in Figure 2 for further outlines.

```
1  struct packet : public rand_obj {
2    randv< unsigned int > src_addr;
3    randv< sc_uint<16> > dest_addr;
4    rand_vec< char > data;
5
6    packet(int &expected_max_size) : src_addr(this),
         dest_addr(this), data(this) {
7     constraint(src_addr() <= 0xFFFF);
8     constraint("diff", src_addr() != dest_addr());
9     soft_constraint(dest_addr() % 4 == 0);
10
11    constraint( data().size() % 4 == 0 &&
         data().size() < reference(expected_max_size) );
12    constraint.foreach( data, _i, IF_THEN( _i == 0,
         'A' <= data()[_i] && data()[_i] <= 'Z') );
13    constraint.foreach( data, _i, IF_THEN( _i != 0,
         'a' <= data()[_i] && data()[_i] <= 'z') );
14    constraint.soft_foreach( data, _i, data()[_i] +
         data()[_i-1] > 'a' + 'b');
15   }
16 };
```

Figure 2: Constrained Packet.

3.2.2 Constraint Specification

We first introduce the APIs for the creation of the elementary entities: random variables and constrained random objects.

Random Variable.

The template class *randv<T>* represents a random variable of the C/C++ or SystemC built-in type *T*. All standard applicable operators (arithmetic, comparison, logical, etc.) are overloaded so that an instance *x* of *randv<T>* behaves as if it were a variable of type *T*. A call *x.next()* assigns a random value in the range of *T* to *x*. While the focus of CRAVE is on complex constraints with many variables, it also supports simple constraints on a single variable. Two member functions *addRange* and *addWeightedRange* can be used to refine the distribution of *x.next()*. Furthermore, *x()* returns a symbolic link to the value of *x* to be used to specify constraints in conjunction with other instances of *randv<T>* as shown hereafter.

Random Object and Constraint Inheritance.

Complex constrained random objects can also be specified. They must inherit from the class *rand_obj* provided by CRAVE. Such an object can contain several instances of *randv<T>* and *rand_obj*. Constraints for each of these instances as well as constraints between them can be specified in a constructor of the object. For the instance *x* of *rand_obj*, *x.next()* randomizes all belonging instances of *randv<T>* and *rand_obj*, respecting the specified constraints. The inheritance/reuse of constraints in CRAVE is straightforward. The user can add more fields and constraints to an existing random constrained object by using C++ class inheritance.

The following example outlines the application of CRAVE features. The constrained packet in Figure 2 inherits from *rand_obj* and consists of a source address as *randv<unsigned int>*, a destination address as *randv<sc_uint<16>>*, a data field as a randomized vector (see Section 3.2.3), and a set of constraints. The source address is constrained to be in the range [0x0, 0xFFFF] (Line 7) and the both addresses must not be the same (Line 8). Both constraints are so-called hard constraints, i.e., they must be satisfied otherwise *next()* fails. The second constraint is also a named constraint, which enables dynamic management of constraints as described later in Section 3.2.3. Line 9 shows a soft constraint stating that the destination address should be a multiple of four. Soft constraints can be ignored by the constraint-solver if they cannot be satisfied in conjunction with the specified hard constraints. As it can be seen in all constraints, the symbolic links to the actual instances of *randv* are used. Line 11-14 declare constraints related to the randomized vector, which will be explained in the next section.

3.2.3 Dynamic Data Structures and Constraints

This section introduces vector constraints, references, and dynamic constraint management as three distinctive features of CRAVE, which are not supported by the SCV library.

Vector Constraints.

The SCV library offers no direct support for the constrained randomization of dynamic data structures such as vectors. The user must mimic dynamic data structures by using arrays of fixed-size. This is inconvenient and not memory-efficient. Furthermore, the upper bound on size of dynamic data structures might not be known at the time of constraint specification. CRAVE offers a template class *rand_vec<T>* for the constrained randomization of vectors, where currently C/C++ and SystemC built-in data types are supported as template parameter *T*.

The class *rand_vec<T>* also implements the APIs of the STL class *vector* and thus behaves as if it is an STL vector. Similar to *randv<T>*, for an instance *v* of *rand_vec<T>*, *v* refers to the actual vector while *v()* is the symbolic vector used to specify constraints. For the symbolic vector *v()*, *v().size()* refers to the size, *v()[_i]* to a symbolic vector element, and *v()[_i - c]* to a previous element relative to *v()[_i]* (*_i* is a predefined constant in CRAVE and *c* is a positive constant). The symbolic elements *v()[_i]* and *v()[_i - c]* are used in a *foreach* constraint.

Back to the example shown in Figure 2. The vector constraints can now be explained: Line 12 and 13 ensure that the first element is capitalized and the remainder is in lower case letters, respectively. Both are hard constraints. The last constraint is a soft *foreach* constraint: two consecutive elements cannot be *aa*, *ab* or *ba*. The first one (Line 11) forces the size of the vector to be a multiple of four and strictly less than an upper bound. This upper bound can change dynamically depending on *max_expected_size*. This is captured by the concept of *references* introduced next.

References.

In many cases, the randomization depends on the dynamically changing state of the verification environment. To include the state in the constraints using the SCV library, the user must use additional variables to save the state and to update them manually whenever the state is changed. For this purpose, CRAVE provides references as a convenient shortcut. A reference in CRAVE basically links a "real" variable with a symbolic variable, which can be used during constraint specification. Before the constraints are solved, the value of this symbolic variable is fixed to the actual value of the linked variable.

Dynamic Constraint Management.

During the verification process, it is very useful that the user can enable/disable specific constraints of a random object. With the SCV library, the user has to mimic the feature by adding an auxiliary variable and constrain this variable in an implication with the constraints to be enabled/disabled, which is inconvenient and inefficient. CRAVE's constraint

management APIs of *rand_obj*: *enable_constraint(name)* and *disable_constraint(name)* allow to directly enable/disable named constraints. Note that the vector constraint *foreach* can also be named and intentionally soft constraints cannot be enabled/disabled.

Where the previous features demonstrated different use cases where CRAVE offers clear advantages over the SCV library, the next paragraphs discuss important usability enhancements of CRAVE compared to the SCV library.

3.2.4 Usability

Inline and Incremental Constraints.

Constraints in CRAVE can be specified without a formal constraint class. A standalone constrained random generator can be created anywhere and used with arbitrary variables and constraints. In practice, this reduces the effort when coding non-trivial testbench environments. A generator can use all features of CRAVE except for inheritance. However, incremental constraint specification is supported. This feature is very helpful for dynamic testbenches. After the generator has been executed for a certain set of constraints, new constraints can arbitrarily be added, e.g., to generate more general values first and more specific ones later. The use of inline constraints will be demonstrated later in the case study.

Debugging Constraint Contradictions.

The composition of constraints to large sets can easily give contradiction(s) in the global constraint. The manual debugging of such a contradiction is very time-consuming. Therefore, CRAVE can automatically identify which named constraints are part of a conflict. This is completely done on a formal level. Therefore, all minimal subsets of the constraints that form a conflict will be reported.

3.2.5 Parallel Constraint Solving

Various alternatives to BDD-based constraint solving have been studied, see e.g. [25]. Approaches based on *Boolean Satisfiability* (SAT) [32, 24] or *Satisfiability Modulo Theories* (SMT) [37] gave very good results for constraints, which are hard to be solved by BDDs. However, in general it is not possible to know in advance which type of constraint solver will show the best performance. Therefore, CRAVE uses a portfolio approach. Instead of running a specific constraint solver, an SMT-based constraint solver as well as a BDD-based constraint-solver are executed in parallel for the same set of constraints. We use *metaSMT* [18] for implementing the constraint solving in CRAVE. Essentially, *metaSMT* allows engine independent programming by providing a unified interface to different solvers, e.g., CUDD [35], Z3 [11], Boolector [7], MiniSAT [12], PicoSAT [6], SWORD [36], and AIGER [1]).

Overall, solutions for complex constraints can be generated very fast.

4. SYSTEM VERIFICATION METHODOLOGY

For the seamless integration of the presented functional coverage and advanced constrained random stimulus generation techniques an appropriate integration within a verification methodology is essential. For this, we introduce the *System Verification Methodology* (SVM) library for RTL and TLM [30]. The SVM library provides the building blocks for efficient testbench modeling with integrated functional coverage and constrained random stimulus libraries for SystemC verification extension as introduced before. The building blocks of the SVM library include base classes, utilities, and macros, which support the engineer to construct disciplined artefacts improving the reuse of verification components and stimuli.

Taking advantages from SystemC abstract modeling and refinement features, the methodology used in SVM follows the principles of OVM and UVM. In this sense, SVM is developed in compliance with OVM and UVM, thus keeping the best possible interoperability between these libraries.

4.1 Verification Methodology for SystemC

SVM was designed to be compatible with the standard IEEE SystemC simulators. Its packages are defined for a seamless integration of the library into different verification flows, as well as legacy verification environments. Assertions, randomization/constraint solver, and coverage packages implement dedicated TLM verification features. The SVM base, components, and sequences packages will be further outlined in the next subsections.

4.1.1 Base Package

This package is an extension to the OVM multiple languages release v2.1.1, a donation of Cadence to the OVM community, which includes a SystemC implementation. Originally, this package contained elements for factory automation, environment configuration, simulation control, and a root component, which is the base for all other verification components. We additionally included a callback facility, a command-line processor, and a transaction routing and recording feature and moved the base component to the component package, which reflects the alignment to UVM.

The verification methodology implementation follows the factory design pattern, which introduces higher abstraction in the process of instantiating components/objects. In this context, it is possible to change object behavior by providing different implementations with same interface without changes in the object itself that applies that interface. Examples of the application of the factory in the verification process are when it is required to change stimuli, e.g., using a stimulus generator with more constraints, or providing a different driver to adapt the way data is sent to the DUT, e.g., considering a refinement from TLM to RTL. The factory implementation provides facilities to overwrite types and to control the effect of object creation in the entire environment or to a specific object.

The library also provides a configuration facility, which allows the registration of a configuration to affect the entire environment or a specific object. By registering a configuration, a verification component/object queries for an existing configuration that applies to it and performs the required adaptation. The configuration can adapt the component topology - the types and number of subcomponents, and its fields. For example, one can consider a component reading from its configuration table, the name of the input file to read stimuli, or the number and type of components, which are instantiated and bound to a communication bus. Although automatic configuration is provided by the latest OVM SystemVerilog version, due performance and reusability problems [13], we decided not to support this feature in the current SVM release.

On top of that, SVM supports a transaction facility as transactions represent important flows of data object, such as instructions, pixels and data items. The transaction facilities allow users to record transactions, route response to specific requests, and to control timing information for a transaction.

SVM also comes with a callback facility. A callback is an extension mechanism used in complement to the factory, which allows changing the behavior of components without change of the component itself. It can be used to modify the component parameter definition during generation of a testbench or to provide flexible mechanism to allow execution of personalized behavior before or after executing an arbitrary function. One obstacle in the direct conversion of testbenches between SystemC and SystemVerilog is that their simulation kernels perform different execution phases. They must be harmonized in order to change the environment, the configuration of objects, start multiple sections, etc. On top of that, OVM defines multiple phases, improving the simulation phases as given by the SystemVerilog simulation standard. We started with the phase implementation from OVM-SC as it is aligned with the OSCI SystemC simulator and already widely accepted. Although there is a basic alignment in OVM-SC between the phases of OVM and SystemC, some further adaptation is still required due the different between construction and connection performed in SystemC and OVM. Since most of the construction of the OVM hierarchy is performed in the Build Phase, the Connection Phase is required so that all components are created in the time of connection. Although the Connect Phase callback is available in the OVM-SC, it is not automatically called by SystemC kernel, so that a binding has to be performed inside of the Build Phase. However, in order to improve the conformance to OVM, we implemented the automatic call to the Connection Phase after the execution of the Build method of each component. Note that it still executes in the Build Phase and full hierarchical names cannot be used in this phase. However, by using this two distinguished phases, i.e., Build and Connection, the code for connect components can be easily identified and is ready to be used in an SystemC Connection Phase, in the case of potential future SystemC extensions.

4.1.2 Components

OVM-SC provides one verification component, which must be used as base class for all other components. In comparison to OVM-SC, we add an additional package with structural components, which support the development of verification environments and tests in a well-structured way. It includes classes, such as Agents, Drivers, Monitors, etc. These components allow the construction of a topology easy to use, to understand, and to reuse. They reduce some implementation details, improve automation and are the base for future improvements. Currently, SVM provides the following components:

Test: This module has to be extended by the user in order to generate a self-contained test for the DUT. Instances of different Test modules can be used to perform a set of tests, which can be executed in batch mode. Each test can contain one or more Environments in order to verify multiple properties or views.

Environment: This module encapsulates the configuration and instantiation of the topology of verification components. It may contain Agents, Monitors, Scoreboards, etc. that are configured for different environments.

Agent: This module is an abstract container for Driver, Monitor and Sequencer. It is used to emulate the DUT or a functional behavior of components that must be connected to the DUT. Active Agents emulate devices connected to the DUT and passive Agents are used to monitor DUT activity.

Driver: This module drives the signal to the DUT ports. Drivers receive SequenceItems (transaction data) and pass them to DUT. It has detailed information about the DUT interface and its logic and can be used to refine or adapt SequenceItems to a DUT interface.

Monitor: This module extracts transactions, signals and other information from DUTs and makes them available to other components. Typically, a monitor is a subcomponent of an Agent, so that it checks only data relevant for the parent Agent.

Subscriber: This module is used to perform coverage analysis and check the information from DUT provided by Monitors. Multiple Subscribers can be connected to a Monitor. Each Subscriber is responsible to encapsulate different coverage and verification logic.

Scoreboard: Scoreboards may receive different pieces of information from different Monitors for self-checking Environments. Additionally, it can provide coverage information and verify the design at the functional level.

4.1.3 Sequences

The central task in the verification process is to generate and coordinate the stimuli for the DUT. Beyond standard stimuli generation technologies, such as constrained random stimulus generation, the management and arbitration of generated stimuli require special attention to create reusable stimuli. For this purpose, we add a package in our SVM library that contains classes, which support the definition of stimuli and sequences of stimuli. These classes encapsulate the procedure to generate data for the DUT and allow the organization of different data in sequences of stimuli, which can be hierarchically or sequentially organized in libraries. Moreover, different arbitration modes are available to provide the right sequence distribution.

SequenceItem: SequenceItem represents data for stimulus and response of the DUT. It may represent a command, a bus transaction, or a protocol package. The fields in a SequenceItem may be randomized to generate different stimuli in different runs.

Sequence: Sequence implements the procedure to create SequenceItems. Sequences can be reused or combined hierarchically to generate complex stimuli. When Sequences are used sequentially they can represent the different phases of a stimulus, such as configuration phase prior to a communication phase. Additionally, Sequences may be combined, in order to create a hierarchy of stimuli or to generate stimuli in parallel to multiple interfaces of a DUT. They are denoted as Virtual Sequences, which are associated to Virtual Sequencers, containing subsequences to coordinate the flow of stimuli. This feature allows users to generate complex stimuli combining Sequences from a library.

Sequencer: Sequencers are used to generate and to coordinate the Sequences submitted to the Driver or the response to it. Using Sequencers, the user may model time in different scenarios and call the randomization mechanism in Sequences and Sequence Items to generate stimuli. They pro-

vide different arbitration modes to select the next Sequence in the library: FIFO, Weighted Priority Distribution, Random, Strict FIFO, Strict Random, and a user implemented arbitration comparison.

4.2 OVM, UVM, and SVM Comparison

The status of our current SVM library (including functional coverage and constraint random stimulus generation) can be classified best when we compare it with OVM and UVM as given in Table 1.

The basis for comparison with OVM is the OVM version 2.2.1, as OVM is the first standardized verification methodology library. After its publication, OVM incorporated some features from UVM 1.1. The most prominent is the UVM Register Abstraction Layer, supported by the UVM Register Kit for OVM, which originates from Synopsys VMM and joined work from Synopsys and Mentor Graphics. However, to use this feature some UVM components must be used together with the OVM library.

As the first UVM release was comparable the latest version of OVM, they show not much difference. Later UVM releases included new features, mainly a command-line processor, a Register Abstraction Layer, an improved simulation control (Phasing), and with general improvements of the API.

In February 2009, Cadence donated the OVM-ML package to ovm-world.org including the OVM-SC. At the same time, a functionally equivalent UVM for SystemC (UVM-SC) version, with minor changes, was donated by Cadence to uvmworld.org under the UVM multiple-languages package umbrella.

Table 1 highlights the main differences between UVM-SC and UVM, the original version for SystemVerilog. It can easily be seen there that UVM-SC lacks most of the features provided by UVM, which are mandatory to perform coverage-driven and constrained-random verification.

The SVM library supports almost all missing features missing of OVM/UVM for SystemC, by adding features based on the OVM for SystemVerilog version 2.2.1. Additionally, we integrated the libraries for the functional coverage and stimulus generation features of SystemC. Moreover, SVM is complemented with a powerful assertion language for SystemC, which was first introduced in [14].

Therefore the set of libraries tailored for advanced TLM verification, which integrates SVM provides powerful tools to develop CDV and CRV testbenches for SystemC environments using TLM and RTL models.

5. CASE STUDY

Our case study implements a high-level control model of a two wheeled self-balancing electric vehicle. The closed-loop control model, a classic inverted pendulum control, receives its driving commands from the user's balance and through a vertical steering axis with immediate impact of the driving motors.

We created a virtual platform, which runs the control algorithms wrapped in SystemC components as given in Figure 3. The virtual platform is composed of one CPU, sensors and actuators, communicating through a TLM2 bus. A command interface receives commands from the steering axis and forwards it to the CPU component using the TLM2 bus.

The testbench consists of a `SVM Test` component that is

Table 1: Comparison of OVM, UVM and SVM.

Feature	OVM	UVM	UVM-SC	SVM
Call-backs	Yes	Yes	No	Yes
Comparison	Yes	Yes	No	No
Command-Line Processor	No	Yes	No	Yes
Configuration	Yes	Yes	Yes	Yes
Factory	Yes	Yes	Yes	Yes
Methodological Components	Yes	Yes	Yes	Yes
Objection	Yes	Yes	No	No
Packing	Yes	Yes	Yes	Yes
Simulation Control and Phasing	Yes	I	Yes	Yes
Polices	Yes	Yes	No	No
Recording	Yes	Yes	SCV	Yes
Register Abstraction Layer	Yes	Yes	No	YES
Reporting	Yes	Yes	SC	SC
Routing	Yes	Yes	No	Yes
Sequencing	Yes	Yes	No	Yes
Synchronization	Yes	Yes	SC	SC
Assertion	SV	SV	No	I
Coverage	SV	SV	SCV	I
Constrained Random Stimuli	SV	SV	SCV	I

Yes - The library provides the feature; No - The library does not provide the feature; I - Improved feature; SC - Feature from the SystemC Language; SV - Feature provided by SystemVerilog language; SCV - Feature provided by SystemC Verification Library.

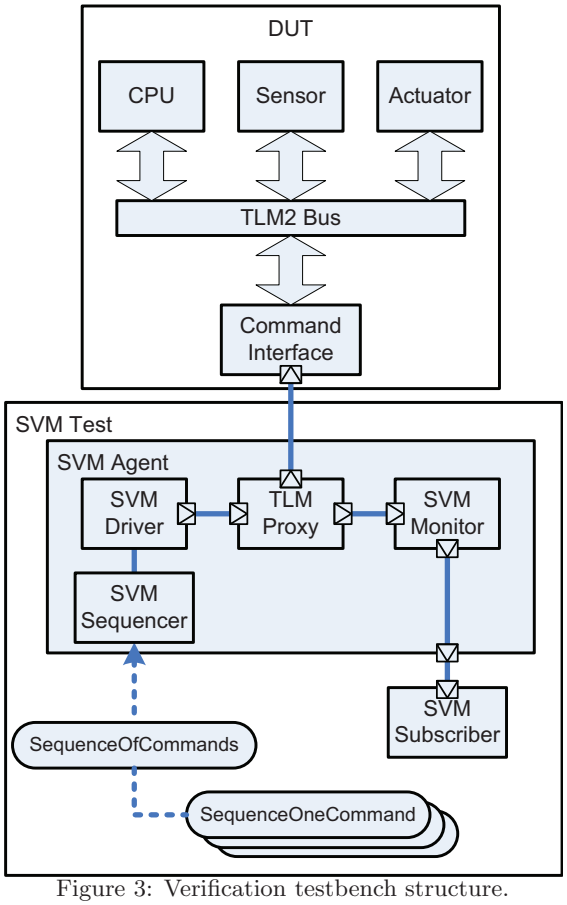

Figure 3: Verification testbench structure.

```
1  class SVMAgent : public svm_agent {
2  public:
3    tlm::tlm_analysis_port<tlm::tlm_generic_payload > aport;
4    SVMDriver *pDriver;
5    SVMMonitor *pMonitor;
6    SVMSequencer *pSequencer;
7
8    SVMAgent(sc_core::sc_module_name name);
9
10   SVM_COMPONENT_UTILS(SVMAgent)
11
12   void SVMAgent::build(){
13     svm_agent::build();
14     get_config_int("debug", debug);
15     pSequencer = DCAST<SVMSequencer*>(
           svm_factory::create_component( "SVMSequencer",
           "","pSequencer" ) );
16   }
17 };
18 SVM_COMPONENT_REGISTER(ActorAgent);
```

Figure 4: Partial code showing the SVMAgent structure.

the main container for the testbench. The `SVM Test` contains one `SVM Agent` that replaces the steering axis and is composed by one TLM2 `SVM Driver` and one TLM2 `SVM Monitor`. The `SVM Agent` contains a `SVM Sequencer`, which receives sequences of stimuli from the `SVM Test` and schedules it to the `SVM Driver`, which drives the stimuli for the DUT interface. Figure 4 shows an excerpt of the `SVM Agent` code. At Line 18 SVM defined macros are used to register components within the factory, the creation of component using the factory at line 15 as well as the acquisition of configuration values at Line 14.

When it comes to the stimulation of TLM2 interfaces, the question of how to organize a driver and a monitor is important. In RTL verification, this separation is fairly simple since a RTL driver is connected via signals to the DUT. A RTL monitor, hence, needs to be connected to the same signals for observing the behavior. At TLM, however, a connection is usually done on a port-to-port basis, i.e., initiator to target connections. Hence, it was necessary to incorporate a so called "`TLM-Proxy`". This proxy broadcasts any incoming transaction to both the monitor and towards the DUT. However, the broadcast to the monitor happens twice - once at the beginning of the transaction and once when the transaction call has returned from the DUT. This allows a monitor to observe pre- and post-conditions of a transaction.

The connection to the DUT is established through a regular SystemC TLM2 initiator to target binding. Hence, a delta-free connection could be established between the DUT and the verification component. This also enables the verification environment to perform tests, which check the behavior of the TLM model by performance optimization techniques such as the Quantum Keeping mechanism.

Extending the `svm_sequence_item` class we defined the `Command Item` class, which holds the stimuli generated for the DUT command interface. Each `Command Item` to the DUT can either make it turn left/right by a given angle or increase/decrease its speed by a given percentage, or simply stop the vehicle. Therefore, the `Command Item` contains three fields: the *command*, the given *degree*, and the given *percent*. The constraints for each `Command Item` are shown in Figure 5. As can be seen at Line 3 and 4, *degree* and *percent* are constrained to be zero if *command* does not indicate a turn or a change of speed, respectively.

```
1  constraint(0 <= degree() && degree() <= 36);
2  constraint(0 <= percent() && percent() <= 100);
3  constraint(IF_THEN(command() != ifx::IFX_TURN_LEFT
       && command() != ifx::IFX_TURN_RIGHT, degree() ==
       0));
4  constraint(IF_THEN(command() !=
       ifx::IFX_INCREASE_SPEED && command() !=
       ifx::IFX_DECREASE_SPEED, percent() == 0));
```

Figure 5: Constraints for One Command Item.

```
1  BIN: IFX_ACTOR_CMD:TURN_LEFT::: 2014 Hits
2  BIN: IFX_ACTOR_CMD:TURN_RIGHT::: 1965 Hits
3  BIN: IFX_ACTOR_CMD:INCREASE_SPEED::: 1985 Hits
4  BIN: IFX_ACTOR_CMD:DECREASE_SPEED::: 2068 Hits
5  BIN: IFX_ACTOR_CMD:STOP::: 1968 Hits
6  BIN: IFX_ACTOR_DEGREE:DEGREE::: 3979 Hits
7  BIN: IFX_ACTOR_PERCENT:PERCENT::: 4053 Hits
8
9  BIN: SEQ_LENGTH:0-15::: 1908 Hits
10 BIN: SEQ_LENGTH:16-30::: 57 Hits
11 BIN: SEQ_LENGTH:31-..::: 3 Hits
```

Figure 6: `IXF_ACTOR` transaction coverage.

The DUT is stimulated by two hierarchical sequences, which define how the `Command Items` are driven to the command interface. At high level, we define one `Sequence of Commands`, which is the main container for the generated sequences. It creates sets of `Sequence One Command`, so that pattern of commands can be generated and grouped, building a library of stimuli. Finally, each `Sequence One Command` holds an `Command Item`, which is randomized and is ready to be driven to the DUT. For this, one `Command Item` is pushed by the `SVM Sequencer` to the driver. Following that, the `SVM Driver` interprets the `Command Item` and executes two transactions in the command interface - one for the command and one for the parameter. In order to investigate the verification closure of the simulation runs, we monitor the received transactions within the `SVM Monitor` component. A `SVM Subscriber` is connected to `SVM Monitor` and implements a functional coverage metric with four coverpoints as follow:

`IFX_ACTOR_CMD` has bins corresponding to `command` values. This allows checking the distribution of the generated commands during the simulation.

`IFX_ACTOR_DEGREE` which implements bins for `degree` values, therefore keeping track of the path.

`IFX_ACTOR_PERCENT` which collects coverage of the altering of the speed, expressed in percentage ranging from 0 to 100.

`SEQ_LENGTH` which covers the number of consecutive commands before a STOP command occurs.

In the first run of stimuli, we send 10,000 random `Command Items` to the DUT. As shown by the first seven lines of the collected coverage. in Figure 6, the first run examines the design thoroughly. However, the main shortcoming is that the STOP command was executed too often. That means, that long sequences of turns and speed changes before stop are not sufficiently considered. This is visible in the last three lines in Figure 6: most considered sequences have only between 0 and 15 commands.

Therefore, in the second run, we send several constrained sequences of commands to the DUT. The constraints for the

```
1  Generator<Context> inline_constraints;
2  rand_vec<IfxCommandValT> commands;
3  randv<int> tmp;
4
5  inline_constraints(25 <= tmp() && tmp() <= 40);
6  inline_constraints(commands().size() == tmp());
7  inline_constraints.foreach(commands, _i, IF_THEN(_i == 0,
       commands()[_i] == ifx::IFX_INCREASE_SPEED));
8  inline_constraints.foreach(commands, _i, IF_THEN_ELSE(_i <
       reference(tmp) - 1, commands()[_i] != ifx::IFX_STOP,
       commands()[_i] == ifx::IFX_STOP));
9  inline_constraints.foreach(commands, _i, (commands()[_i] !=
       commands()[_i - 1]) || (commands()[_i] !=
       commands()[_i - 2]) || (commands()[_i] !=
       commands()[_i - 3]));
10
11 while (sequenceCount-- > 0) {
12   inline_constraints.next();
13   // generate each individual command item and deliver it
         to the DUT
14   ...
15 }
```

Figure 7: Constraints for One Command Sequence.

sequences are captured using inline constraints on a random vector of CRAVE as depicted in Figure 7. The first two constraints ensure that each sequence contains between 25-40 commands. The third constraint forces the first command to be an increase of speed. The fourth constraint specifies that all commands should not be a stop command with the exception of the last command. Finally, no four consecutive commands in a sequence should be equal. After the generation of a vector of commands (Line 12), each individual command item is generated respecting the constraints in Figure 5 with the *command* field being fixed to the corresponding value in the generated vector.

We defined also a similar set of constraints in the SCV library using fixed size array of commands. However, even for only 25 commands, the SCV library has not been able to generate a single sequence after several hours. This result is consistent with the experimental comparison of CRAVE and SCV reported in [19].

6. CONCLUSIONS

In this paper we presented the *System Verification Methodology* (SVM) as a set of SystemC libraries for TLM testbench implementation. SVM combines missing building blocks for advanced functional verification with SystemC. The combination of advanced constraint solving and random stimulus generation capabilities with a functional coverage facility under the umbrella of a verification methodology allows building highly modular testbenches in a standardized, structured, efficient, and reusable way. The SVM is completely compliant to existing standards as it is based on the concepts of OVM Multiple-Languages - SystemC package v2.1.1 and OVM for SystemVerilog v2.1.1 that has been refactored for UVM.

We successfully applied our methodology to a high level control model of a two wheel self-balancing electric vehicle and performed regression tests within larger case studies. At this point, a pretty stable implementation of SVM is apparently achieved as it has also been applied to more complex industrial designs. The next step will be the completion towards complete UVM. This is mainly the support of the register abstraction layer.

7. ACKNOWLEDGMENTS

This work was partly funded by the German Ministry of Education and Research (BMBF) through the project SANITAS (01M3088), the DFG SFB 614, the ITEA2 projects VERDE (01S09012) and TIMMO-2-USE (01IS10034), and the DFG Reinhart Koselleck project DR 287/23-1. We greatly appreciate the cooperation with the project partners.

8. REFERENCES

[1] Aiger. `http://fmv.jku.at/aiger/`.

[2] Accellera Organization, Inc. Open Verification Library (OVL), May 2009.

[3] Accellera Organization, Inc. Universal Verification Methodology (UVM), May 2012.

[4] J. Aynsley. *OSCI TLM-2.0 LANGUAGE REFERENCE MANUAL*. Open SystemC Initiative (OSCI), 2009.

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

[6] A. Biere. Picosat essentials. *JSAT*, 4(2-4):75–97, 2008.

[7] R. Brummayer and A. Biere. Boolector: An efficient SMT solver for bit-vectors and arrays. In *Tools and Algorithms for the Construction and Analysis of Systems*, pages 174–177, 2009.

[8] Cadence Design Systems, Inc. Open Verification Methodology Multi-Language (OVM-ML).

[9] Cadence Design Systems, Inc. Universal Reuse Methodology (URM).

[10] Cadence Design Systems, Inc. OVM-SC Library Reference Version 2.0.1, February, 2009.

[11] L. de Moura and N. Bjørner. Z3: An efficient SMT solver. In *TACAS*, pages 337–340, 2008.

[12] N. Eén and N. Sörensson. An extensible SAT-solver. In *SAT*, pages 502–518, 2003.

[13] A. Erickson. Are OVM & UVM Macros Evil? A Cost-Benefit Analysis. In *Proceeding of Design and Verification Conference (DVCON)*, Mar. 2011.

[14] V. Esen. *A new assertion language covering multiple levels of abstraction*. PhD thesis, 2008.

[15] H. Foster. Redefining Verification Performance (part 2), August 2010.

[16] D. Große, R. Ebendt, and R. Drechsler. Improvements for constraint solving in the SystemC verification library. In *ACM Great Lakes Symposium on VLSI*, pages 493–496, 2007.

[17] D. Große, R. Wille, R. Siegmund, and R. Drechsler. Contradiction analysis for constraint-based random simulation. In *Forum on Specification and Design Languages*, pages 130–135, 2008.

[18] F. Haedicke, S. Frehse, G. Fey, D. Große, and R. Drechsler. metaSMT: Focus on your application not on solver integration. In *DIFTS'11: 1st International workshop on design and implementation of formal tools and systems*, pages 22–29, 2011.

[19] F. Haedicke, H. M. Le, D. Große, and R. Drechsler. CRAVE: An advanced constrained random verification environment for SystemC. In *International Symposium on System-on-Chip (SoC)*, 2012. Available at www.systemc-verification.org.

[20] IEEE Computer Society. IEEE Standard for System Verilog-Unified Hardware Design, Specification, and Verification Language - IEEE 1800-2009. 2009.

[21] IEEE Computer Society. IEEE 1666-2011 Standard SystemC Language Reference Manual. *IEEE Std 1666-2011*, 2011.

[22] IEEE Computer Society. Standard for the Functional Verification Language e. *IEEE Std 1647-2011 (Revision of IEEE Std 1647-2008)*, pages 1 –495, 26 2011.

[23] JEDA Technologies, Inc. JEDA ESL Validation Solution.

[24] H. Kim, H. Jin, K. Ravi, P. Spacek, J. Pierce, B. Kurshan, and F. Somenzi. Application of formal word-level analysis to constrained random simulation. In *Computer Aided Verification*, 2008.

[25] N. Kitchen and A. Kuehlmann. Stimulus generation for constrainted random simulation. In *International Conference on Computer-Aided Design*, pages 258–265, 2007.

[26] C. Kuznik and W. Müller. Functional Coverage-driven Verification with SystemC on Multiple Level of Abstraction. *Proceedings of DVCON*, 2011.

[27] W. K. Lam. *Hardware Design Verification: Simulation and Formal Method-Based Approaches (Prentice Hall Modern Semiconductor Design Series)*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 2005.

[28] Mentor Graphics, Inc. Advanced Verification Methodology (AVM).

[29] NextOp Software, Inc. NextOp assertion-based verification.

[30] M. F. S. Oliveira, C. Kuznik, W. Mueller, W. Ecker, and V. Esen. A SystemC Library for Advanced TLM Verification. In *Proceeding of Design and Verification Conference (DVCON)*, Mar. 2012.

[31] Open SystemC Initiative. SystemC Verification Library v1.0p2, 2006.

[32] S. M. Plaza, I. L. Markov, and V. Bertacco. Random stimulus generation using entropy and XOR constraints. In *Design, Automation and Test in Europe*, pages 664–669, 2008.

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

[34] R. Siegmund, U. Hensel, A. Herrholz, and I. Volt. A functional coverage prototype for SystemC-based verification of chipset designs. In *9th European SystemC User Group Meeting at Design, Automation and Test in Europe*, 2004.

[35] F. Somenzi. *CUDD: CU Decision Diagram Package Release 2.4.1*. University of Colorado at Boulder, 2009.

[36] R. Wille, G. Fey, D. Große, S. Eggersglüß, and R. Drechsler. Sword: A SAT like prover using word level information. In *VLSI of System-on-Chip*, pages 88–93, 2007.

[37] R. Wille, D. Große, F. Haedicke, and R. Drechsler. SMT-based stimuli generation in the SystemC verification library. In *Forum on Specification and Design Languages*, pages 1–6, 2009.