A SystemC Library for Advanced TLM Verification

Marcio F. S. Oliveira, Christoph Kuznik, Wolfgang Mueller
C-Lab, University of Paderborn
Paderborn, Germany
marcio@c-lab.de
christoph.kuznik@c-lab.de
wolfgang@acm.org

Wolfgang Ecker, Volkan Esen
Infineon Technologies
Munich, Germany
Wolfgang.Ecker@infineon.com
Volkan.Esen@infineon.com

Abstract— **This paper introduces the System Verification Methodology (SVM) Library as an advanced TLM library for SystemC, which is based on the OVM-SC library, a SystemC implementation of an Open Verification Methodology (OVM) subset. SVM integrates with a functional coverage library and comes as a significant extension of the OVM-SC library, by providing domain specific components (drivers, monitors, scoreboards and others) and facilities for generation and management of stimuli.**

Keywords- SystemC; OVM; UVM; Verification Library; Electronic System Level

I. INTRODUCTION

The introduction of Electronic System Level (ESL) supports the modeling and verification of complex mixed hardware and software systems. As systems grow in complexity, verification activities started to exceed the design effort. Today, it may account to more than 70% of the activities of the entire development cycle [12] with an increasing exponential growth [7].

As such, the verification community and EDA vendors have spent a large effort to improve the productivity in the verification process, and dedicated Hardware Verification Languages (HVL) were introduced, such as *e* [11] and SystemVerilog [10]. Although their constructs support verification, the verification process productivity was subject of further investigations in development and standardization, and several verification methodologies and libraries for the development of reusable verification components emerged. Examples are Verisity/Cadence's *e* Reuse Methodology (eRM) [24] and Synopsys' Reference Verification Methodology (RVM) [21], which are based on the *e* language and OpenVera [22] respectively. With the outcome of SystemC [4], the SystemC Verification Library (SCV) [17] was introduced, in order to support elemental constrained-random stimuli techniques for RTL verification. However, SystemC lacks an interoperable verification methodology. In contrast, several verification methodologies were developed as SystemVerilog implementations, such as the Universal Reuse Methodology (URM) [5] from Cadence, the Advanced Verification Methodology (AVM) [13] from Mentor Graphics, and the Verification Methodology Manual (VMM) [3] from Synopsys. Later more advanced libraries were introduced, namely the Open Verification Methodology (OVM) [18], as a joint effort between Cadence and Mentor Graphics, which combined best practices from URM and AVM. Finally, OVM was enhanced to the Universal Verification Methodology (UVM) [23], with contributions from Synopsys based on VMM.

Though SystemC was widely accepted for development at higher abstraction levels, e.g. by using TLM 2.0 [16], its verification capabilities are limited in comparison to other languages such as SystemVerilog. Moreover, verification capabilities of the SCV are quite limited and mainly support RTL verification. In February 2009 the multiple-languages release of OVM (OVM-ML) was donated by Cadence Inc. to ovm-world.org. OVM-ML provides the OVM for SystemC (OVM-SC) library, which is a SystemC implementation of OVM subset. However, OVM-SC lacks various necessary features, such as domain specific components, stimuli generation facilities, sequence library management, sequence arbitration, response to request routing, commando-line processor and a register abstraction layer.

This paper presents the System Verification Methodology (SVM) library for SystemC. SVM Library is based on OVM-SC implementation and incorporates verification best practices from OVM-SC, such as factory and configuration facilities and simulation phasing, as well as implements missing features, such as the domain specific components, stimuli sequence generation and management, call-back facilities, response to request routing, transaction recording and a command-line processor for dynamic test bench loading.

The remainder of this paper is organized as follows. The next section presents related work. Section 3 introduces main features and some details of our SVM library. In Section 4 we provide an application example. Finally, Section 5 closes with conclusions and future directions.

II. Related Works

In the area of (add-on) verification libraries, the Open Verification Library (OVL) [1] is maintained by Accellera and provides checkers for assertion, assumption or coverage point checkering. The most recent version, namely 2.5 supports SystemVerilog, Verilog and VHDL. Unfortunately, there is currently no support for SystemC. Additionally, except SystemVerilog the supported languages are limited to RTL level, impeding verification on higher levels of abstraction on more abstract data types.

Other works try to extend the SCV library, improving or providing missing features for verification in the SystemC language, such as coverage [19][20], assertions [9][14], and randomization/constraint solver [8]. Although basic verification extensions for standard SystemC are meaningful, a library offering components for test bench development, like in UVM, is an even more crucial building block for efficient, qualified and interoperable test bench development. As such, OVM-SC became available, implementing a limited OVM subset, mainly part of the OVM base package, in SystemC. UVM was introduced as a SystemVerilog implementation with many improvements compared to OVM. It presents a more dedicated API, introduces more control over the simulation phases, adds a command line processor, and a register abstraction layer. However, the UVM multiple- languages package does not support all those improvements for SystemC.

Our SVM SystemC library is based on the OVM multiple-languages - SystemC package v2.1.1 and OVM for SystemVerilog v2.1.1. The base package is being refactored, to reflect the improvements from the transition of OVM to the UVM standard. Moreover, the base package was improved by call backs, transaction routing and recording facilities. We also included a package with the structural components to build the verification environment. We implemented stimuli sequences, and sequence scheduler facilities, as well as a command-line processor. A coverage library [25], which implements the functional coverage part of the IEEE 1800-2009 SystemVerilog standard, was integrated into SVM to improve basic SystemC verification features.

III. An Advanced SystemC Library for TLM Verification

Our SVM SystemC library for TLM verification implements building blocks to ease the development of verification components and test environment. It includes base classes, utilities and macros, which support the engineer to construct disciplined artifacts, improving the reuse of verification components and stimuli. However, taking advantages from SystemC abstract modeling and refinement features, SVM follows the principles of OVM and UVM. In this sense, our library is developed in compliance with OVM and UVM keeping the interoperability between these libraries as best it is possible. However, the differences between SystemVerilog and SystemC impose some adaptations to keep the conformance with the OSCI SystemC simulator or to exploit SystemC advantages.

A. SVM Library Structure

SVM was implemented to be compatible with IEEE standard SystemC simulators. Its packages are defined for a seamless integration of the library into different verification flow and legacy verification components. Assertions, Randomization/Constraint Solver and Coverage packages, implement dedicated TLM verification features. Actually, they are packages intended to provide missing features required to perform adequate verification in the SystemC ecosystem. Therefore, the packages providing Assertions, Randomization / Constraint Solver and Coverage are comparable to the OSCI SCV library in contrast to a verification methodology library, such as UVM or the one presented in this paper. Figure 1 gives an overview of the SVM library packages, in the illustration the dashed boxes indicate packages provided by our research partners [26].

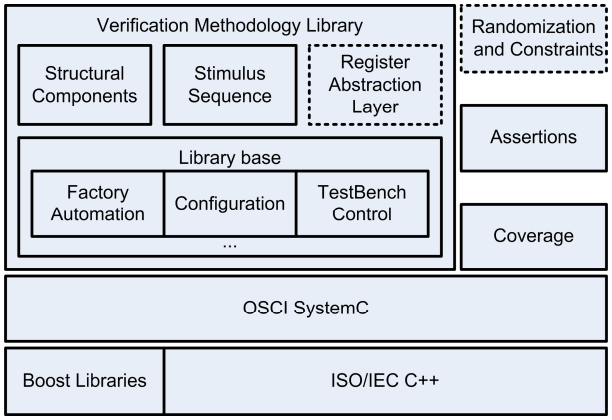

Figure 1. SVM Library Structure

B. Library Basis

This package was inherited from the OVM multiple-languages 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 component, which is the base for all other verification component. Into this package we include call-back facility, command-line processor, and a transaction routing and recording feature. We also move the base component to the

component package, reflecting our alignment with the improvement from OVM to UVM.

1) Factory automation:

The SVM library 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 an 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 stimuli 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 level. 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.

2) Environment Configuration

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 – or 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 file it should use to read stimuli or the number and type of components, which should be instantiated and bound to a communication bus. Although automatic configuration is provided by the latest OVM SystemVerilog version, due performance and reusability problems [6] we decided not to support this feature in the first SVM release.

3) Transaction

We include a transaction facility because transactions represent the flow of data in a system, such as flow of instructions, pixels and data items. The transaction facilities allow user to record transactions, route response to specific requests and control timing information for a transaction.

4) Call Backs

We implement also call backs facility. Call back is an extension mechanism, which allows changing the behavior of components without change 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 some function.

5) Simulation Control

The simulation kernels of SystemC and SystemVerilog perform different execution phases. They must be harmonized in order to change the environment, the configuration of objects, start multiple sections, etc. The OVM methodology 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 known. Although there is a basic alignment in OVM-SC between the phases of OVM and SystemC some further adaptation is required. Figure 2 compares OVM, which is aligned to SystemVerilog, and the SystemC simulation phases.

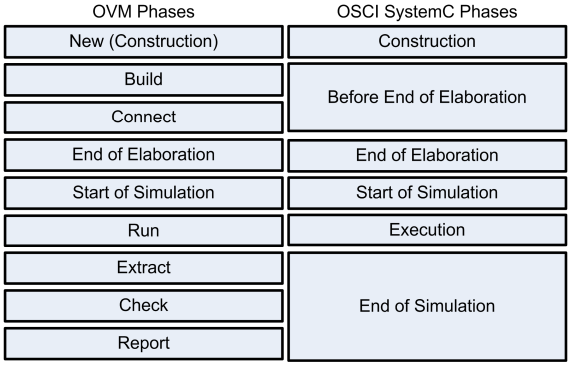

Figure 2. Comparison of OVM and SystemC Simulation phases.

Construction: This first OVM phase compares to the first phase of SystemC. It executes the constructors of components from top to bottom in the topology. However, some coding styles can improve the configuration at further steps. For instance, ports and exports of a component, which are typically member fields, need to be declared as pointers, as well as their child components. This coding style is present in [4]. Nevertheless, it had to be adapted to exploit the OVM phases and configuration/factory facilities.

Build: It compares to the *Before End of Elaboration* phase and, consistent with the SystemC OSCI LRM, performs creation of the bulk of components. The *Build* phase exploits the configuration API provided in the configuration manager to change the type and number of instances before creating the child using the factory facilities. Note that, as the bulk of creation is performed at the *Build* phase, the specific binding with full hierarchical path names of ports/exports is not available because the topology is not fully constructed at this time. However, this issue must be addressed in future, as we want to keep the compatibility with OSCI SystemC.

Connection: Although available in the API, this phase-callback is not automatically called by SystemC kernel, so that binding has to be performed inside of *Build* phase. However, in order to improve the conformance to OVM, the *Connection* phase is called automatically after the execution of the Build method of each component. Notice that it still executing in the *Build* phase and full hierarchical name cannot be used in this phase. However, by using this two distinguished phases, *Build* and *Connection*, the code for connect components is easily identified and ready to be used in a real connection phase, in the case of OSCI adopt such phasing organization.

End of Elaboration: In this phase it is possible to make final adjustments and, as the complete environment is created and connected, it is possible to analyze the system's net list.

Start of Simulation: This phase executes some pre-run activities, such as reporting the topology and configuration, printing information, initialization of channels and ports.

Run: It compares to the SystemC Execution phase. The operation at this phase can consume simulation time. The behavior of a verification component, as described in the following sections, is described in a callback named run and it implemented as `SC_THREAD` of the base component.

Extract: The *End of Simulation* SystemC phase was extended for a more specialized phase control. The *Extract* phase is the first of three phases the SystemC phase *End of Simulation* has to be divided into. In this phase the user can extract information about coverage, assertions, or internal data from the simulated components. The motivation to create a different phase for extraction is to ensure that all data from different sources are available prior to the *Check* phase.

Check: This phase is used to analyze and validate the simulation results, extracted in the previous phase.

Report: This phase is used to write the results to an output.

C. SVM 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 modules allow the construction of a topology easy to use, to understand, and reuse. They reduce some implementation details, improve automation and are the base for future improvements. Figure 3 illustrates a sample topology constructed with the base classes provided in our SVM library.

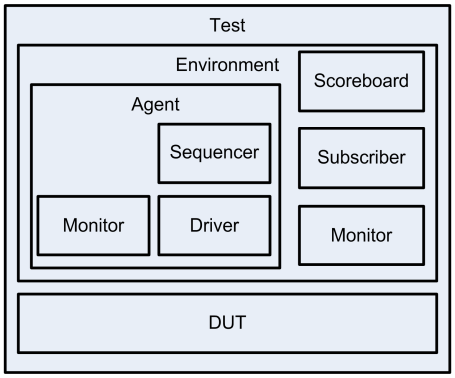

Figure 3. General verification modules structure.

Test: This module has to be extended by the user in order to generate a self-contained test for DUT verification. 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 (Env): This module encapsulates the configuration and instantiation of the topology of verification components. It may contain Agents, Monitors and Scoreboards, etc., which 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 DUT and passive Agents are used to monitor DUT activity. Figure 4 illustrates a partial code for the agent used in section 5. It contains a Sequencer, a Monitor and a Driver. In the code it is possible to notice macros for register the component within a factory. In the Build member the subcomponents are created by using the factory features.

```
#include <systemc.h>
#include <tlm.h>
#include <svm.h>

class ActorAgent : public svm_agent {
public:
   tlm::tlm_analysis_port<tlm::tlm_generic_payload > aport;
   ActorDriver *pDriver;
   ActorMonitor *pMonitor;
   ActorSequencer *pSequencer;
   ActorAgent(sc_core::sc_module_name name);

   SVM_COMPONENT_UTILS(ActorAgent)

  void ActorAgent::build(){
    svm_agent::build();
    get_config_int("debug", debug);
    pSequencer = DCAST<ActorSequencer*>(
      svm_factory::create_component("ActorSequencer", "",
         "pSequencer") );
 …
  }
…
};
SVM_COMPONENT_REGISTER(ActorAgent);
```

Figure 4. Example of a Agent based on svm_agent.

Driver: This module drives the signal to the DUT ports. Drivers receive Sequence Items (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 Sequence Items 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.

D. Stimuli Sequence

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 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 sequence of stimuli, which can be hierarchically or sequentially organized in libraries. Moreover, different arbitration modes are available to provide the right sequence distribution.

Sequence Item: Sequence Item 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 Sequence Item may be randomized to generate different stimuli in different runs. Figure 5 shows a Sequence Item used in the Section 5. It contains three fields, which are randomized following the constraint during the construction.

```
class IfxCommandItem : public svm_sequence_item, public rand_obj
{
public:
   randv<IfxCommandValT> command;
   randv<unsigned int> degree;
   randv<unsigned int> percent;

   void create_constraints();
   IfxCommandItem(const std::string& name)  :
         svm_sequence_item(name) { create_constraints(); }
           …
          SVM_OBJECT_UTILS(IfxCommandItem);
};
```

Figure 5. Sequence Item example

Sequence: Sequence implements the procedure to create Sequence Items. 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.

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 provide different arbitration modes to select the next Sequence in the library: First-In-First-Out (FIFO), Weighted Priority Distribution, Random, Strict FIFO, Strict Random, and a user implemented arbitration comparison.

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 Virtual Sequences, which are associated to Virtual Sequencers, containing subsequences to coordinate the flow of stimuli. This feature allows the user to generate complex stimuli, combining Sequences from a library.

E. Comparisom between OVM, UVM and SVM

This section provides a comparison overview of OVM, UVM, UVM for SystemC and SVM. The base for comparison is the OVM version 2.2.1, as OVM is the first standardized verification methodology. Late, OVM incorporates some features from its successor, UVM 1.1. The most prominent is the UVM Register Kit for OVM p which originates from Synopsys VMM and joined work from Synopsys and Mentor Graphics. The UVM provides additional features, and improves other features already present in OVM.

In February 2009 Cadence donated the OVM-ML package to ovm-world.org, containing the OVM-SC version. At the same time, a functionally similar UVM for SystemC version (UVM-SC), with minor changes, was donated by Cadence to

uvmworld.org under the UVM multiple-languages package umbrella.

The SVM Library improves the OVM/UVM for SystemC, by adding features based on the OVM for SystemVerilog version 2.2.1. Additionally, we integrate libraries to provide Assertion, Randomization/Constraints and Coverage, in order to support advanced RTL/TLM for SystemC. The Table 1 provides the feature comparison between those methodologies.

TABLE I. FFEATURES 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-Creation / Register	Yes	Yes	Yes	Yes
Methodology Components	Yes	Yes	Yes	Yes[1]
Objection	Yes	Yes	No	No
Packing	Yes	Yes	Yes	Yes
Phasing	Yes	I	Yes	Yes
Polices	Yes	Yes	No	No
Recording	Yes	Yes	SCV	I
Register Abstraction Layer	C[2]	Yes	No	Yes[3]
Reporting	Yes	Yes	SC	SC
Routing	Yes	Yes	No	Yes
Sequenceing / Stimuli	Yes	Yes	No	Yes
Synchonization	Yes	Yes	SC	SC
Assertion	SV	SV	No	I
Coverage	SV	SV	SCV	I
Randomization / Constraints	SV	SV	SCV	I[3]

Yes - The library provides the feature; No - The library does not provide the feature; C - There is a contribution package that implements the feature; I - Improved feature when compared to OVM 2.2.1; SC - Feature from the SystemC Language; SV - Feature from the SystemVerilog Language; SCV - Feature from the SystemC Verification Library; 1- The components are tailored for RTL/TLM DUTs; 2 - It is a UVM package adapted to OVM, which requires UVM components too; 3 - Provided by research partners and is to be integrated into the SVM.

Because the first release of UVM was a full version of OVM, on comparing the UVM and OVM one notices they are quite similar. Further releases included new features, mainly a command-line processor and register abstraction layer. They also improved the simulation control and the API. The table highlights the big difference between the UVM-SC to UVM, the original version for SystemVerilog. UVM-SC lacks most of the features provided by UVM. SVM provide most of all missing features in UVM-SC, in particular the sequence package and the methodology components. Moreover, it improves some features, in order to provide advanced TLM verification methodology library, such as some methodological components, recording and coverage features.

IV. EXAMPLE

We applied the SVM library to create an interface verification component for a sensor/actor subsystem as part of the overall verification infrastructure of this system.

One feature of the sensor/actor system is to receive commands, which immediately affect the actor part of this system. We use the interface verification component to create random sequences of commands and apply them to the subsystem. The overall structure of this subsystem is outlined in Figure 6. The following explanations focus on the overall verification component structure, rather than the formulation of constraints and coverage, as details on the constrained randomization, provided by our research partners, and the coverage library are found respectively in [28] and [25].

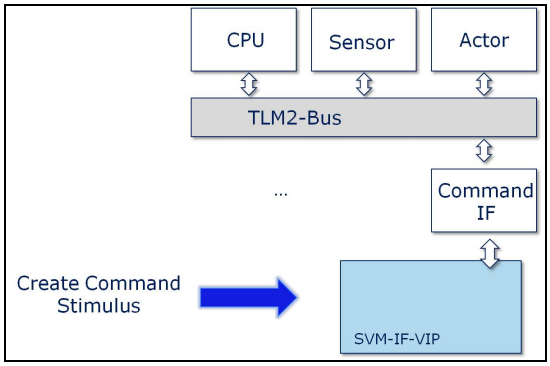

Figure 6. Overall Structure

Figure 7. outlines the structure of the verification component. The interface verification component (SVM-IF-VIP) consists of one agent, which is composed of one TLM2 Driver and a monitor. Furthermore, it contains a set of sequences which generate particular commands and a sequencer which handles the execution of sequences and their interaction with the driver. The command that needs to be pushed into the subsystem is formed by two transactions. The first transaction holds the command value and the second transaction holds the command parameter. Hence, we defined a sequence item, which contains two variables holding the command and parameter value. One sequence generates one item while performing randomization and applying constraints. This item is then pushed to / or pulled from the driver. Following that, the driver interprets the item and executes two

transactions – one holding the command and one holding the parameter.

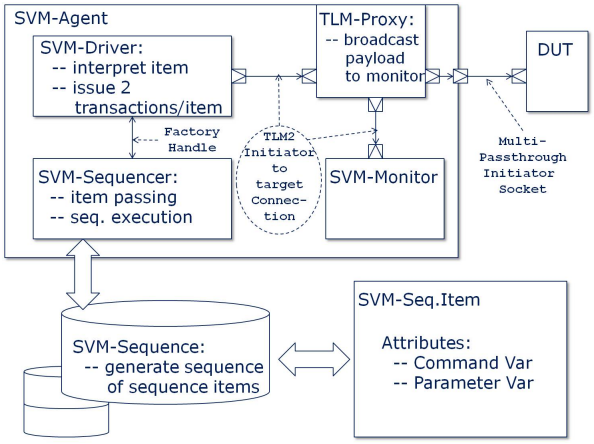

Figure 7. Verification Component Structure

When it comes to stimulating TLM2 interfaces the question of how to organize a driver and a monitor arises. In RTL verification this split is fairly simple, since the driver is connected via signals to the DUT. The monitor hence, needs to be connected to the same signals for observing the behavior. In 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 a TLM model, which uses performance optimization techniques such as the Quantum Keeping mechanism as suggested by the TLM2 standard.

While developing this example we also compared this setup to a similar SystemVerilog setup, which however was created for the RTL version of the DUT. While the languages induces some differences, the main structure, as well as the verification methodology has stayed the same. The development time of the SVM-based verification component regarding the parts, which are independent from the DUT abstraction was comparable to the development time of a similar SystemVerilog structure. However, the attempt to apply a SystemVerilog structure to a SystemC TLM-design showed drawbacks in the setup time, because a simulator-dependent solution is needed to bridge both languages at the TLM-level. In most cases these solutions also induce delta-cycles between the verification component and the DUT, which makes it much more difficult to verify higher level concepts which are based on timing abstractions such as TLM+ [27] or the quantum keeping.

V. CONCLUSIONS

We presented SVM as an advanced SystemC library for TLM verification. The SVM SystemC library is based on OVM-SC. SVM provides advanced features for TLM verification, such as the factory and configuration facilities and extended it by almost all OVM features. As such, we added structural verification components, such as Agents, Monitors, Scoreboards, and so on. They are defined as a set of base classes provided to support a well-structured structural implementation of a test bench, which improve the reuse of verification components. Moreover, we implemented classes to support hierarchical stimuli generation, sequence library management and arbitration, allowing the construction of complex behavior, such as hierarchical protocols. This was combined with a functional coverage library [25] and is currently integrated with an improved Randomization / Constraint Solver [26] .

The described example has demonstrated that the overall methodology using the SVM library does adhere to the methodology defined by OVM and that the utilization of SystemC as the "language" for the verification component has reduced the effort for connection a verification environment to SystemC TLM DUTs.

The future work will cover extensions for UVM compliance, the inclusion of a register abstraction layer, assertions, and better configuration capabilities. This work will be done in cooperation with other research partners from the SANITAS project [26].

ACKNOWLEDGMENT

This work was partly funded by the DFG SFB 614 and the German Ministry of Education and Research (BMBF) through the project SANITAS (01M3088I) and the ITEA2 projects VERDE (01S09012H) and TIMMO-2-USE (01IS10034A). We greatly appreciate the cooperation with the project partners.

REFERENCES

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

[2] Accellera Verification IP Technical Subcommittee (UVM Development Website); http://www.accellera.org/apps/org/workgroup/vip

[3] Bergeron, J., Cerny, E., Hunter, A., Nightingale, A.. Verification Methodology Manual for SystemVerilog. Springer, 2006.

[4] Black, D. C.; Donovan, J.; Bunton, B.; Keist, A.. SystemC From the Ground Up. Springer, 2010.

[5] Cadence Inc. Universal Reuse Methodology (URM).

[6] Erickson, A.. Are OVM & UVM Macros Evil? A Cost-Benefit Analysis. In Design and Verification Conference (DVCon), 2011.

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

[8] Große, D.; Ebendt, R.; Drechsler, R.. Improvements for Constraint Solving in the SystemC Verification Library. In: 17th ACM Great Lakes Symposium on VLSI(GLSVLSI), New York, NY, USA: ACM, 2007, p. 493-496.

[9] Habibi, A.; Tahar, S.; Towards an efficient assertion based verification of SystemC designs. In Ninth IEEE International High-Level Design Validation and Test Workshop, 2004., vol., no., p. 19- 22, Nov. 2004

[10] IEEE Std 1800-2009 IEEE Standard for System Verilog-Unified Hardware Design, Specification, and Verification Language. Available at: http://dx.doi.org/10.1109/IEEESTD.2009.5354441

[11] Iman, S.; Joshi, S. The e Hardware Verification Language. Springer. 2004.

[12] Lam, W. K.. Hardware Design Verification. 2005.

[13] Mentor Graphics - Advanced Verification Methodology

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

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

[16] Open SystemC Initiative, TLM-2.0 Language Reference Manual, Open SystemC Initiative Std., 2007.

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

[18] Open Verification Methodology. Available: http://www.ovmworld.org/

[19] Schwartz, K.. A technique for adding functional coverage to SystemC. In Design and Verification Conference (DVCon), 2007.

[20] Silva K. R. G. da, E. U. K. Melcher, G. Araujo, and V. A. Pimenta. An automatic testbench generation tool for a SystemC functional verification methodology. in 17th Symposium on Integrated Circuits and System Design (SBCCI). New York, NY, USA: ACM, 2004, p. 66–70.

[21] Synopsys. Reference Verification Methodology User Guide. 2005.

[22] Synopsys. OpenVera. Available at: http://www.open-vera.com/

[23] Universal Verification Methodology. Available at: http://www.uvmworld.org/

[24] Verisity Design. e Reuse Methodology Developer Manual, 2002-2004.

[25] Kuznik, Christoph; Müller, Wolfgang. Functional Coverage-driven Verification with SystemC on Multiple Level of Abstraction. In Design and Verification Conference (DVCon), 2011.

[26] 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

[27] Wolfgang Ecker, Volkan Esen, Robert Schwencker, Thomas Steininger, Michael Velten: TLM+ modeling of embedded HW/SW systems. DATE 2010: 75-80

[28] Finn Haedicke, Hoang M. Le, Daniel Große und Rolf Drechsler: CRAVE: An Advanced Constrained RAndom Verification Environment for SystemC. MBMV 2012