Building a coherent ESL design and verification eco-system with SystemC, TLM, UVM-SystemC, and CCI

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

***Abstract*—In the last decade, new system-level design and verification methodologies, language standards and class libraries have been developed and deployed, addressing the design and verification challenges at both system as well as implementation level. For system-level design, SystemC and Transaction Level Modeling (TLM) have become an established design approach, which is being extended with concepts for Configuration, Control and Inspection (CCI). For functional verification, the Universal Verification Methodology (UVM) standard allows the creation of configurable and reusable verification environments. As UVM is also being standardized in SystemC, new opportunities arise to combine all these standards and methodologies to build a coherent SystemC-centric eco-system for system-level design and verification. As the SystemC standard and extensions were developed independently from the UVM-SystemVerilog standard, some conceptual and functional differences are noticeable related to transaction-based communication, configuration, and register interface and modeling. This paper is a first attempt to explain some of the differences and will give proposals how these standards could be used concurrently. These proposals are meant for further discussion to help in the evolution and alignment of these various standard developments, methodologies and their applications.**

Keywords—SystemC; Transaction-level modeling (TLM); Configuration, Control and Inspection (CCI); Universal Verification Methodology (UVM); UVM-SystemC; UVM-SystemVerilog; Electronic System Level (ESL)

I. INTRODUCTION

The creation of SystemC-based virtual prototypes has become a well-known approach in a modern Electronic System Level (ESL) design and verification flow, primarily to facilitate software development and architecture exploration. For this purpose, concepts like transaction-level modeling (TLM) are applied and new techniques are proposed by the Configuration, Control and Inspection (CCI) working group [1] for the configuration of system-level models. However, the main focus of virtual prototyping has always been to capture the design intent itself, by modeling the system architecture to understand the interplay between hardware and software components, and to a much lesser extent on the creation of a robust and reusable ESL verification environment which can be reused as soon as the design implementation becomes available. In December 2015, the Universal Verification Methodology (UVM) has been made available as class library in C++/SystemC [2], facilitating a standardized verification approach for stimuli creation and test bench assembly based on reusable and configurable verification components.

One of the main objectives has become to unify and apply these different language standards and class libraries to build a consistent SystemC-based eco-system for system-level design and verification. This should not only address ESL design and verification, but should also facilitate a way to include mixed TLM and RTL design descriptions to support hardware/software co-verification. The main challenge is to understand, and sometimes overcome, the conceptual and functional differences between the transaction-based communication, configuration, and register interface and modeling approaches as defined by these different standards. For example, the TLM API defined in UVM-SystemVerilog is not fully compatible with the TLM-1 and TLM-2.0 declarations specified in the SystemC standard. Furthermore, the configuration mechanism in UVM differs from the configuration parameter approach proposed by CCI. Lastly, the lack of a standardized register API in SystemC means that there is no common register modeling and interface in SystemC, which makes it difficult to establish a standardized register backdoor mechanism as defined in UVM.

This paper will explain some of the differences between these language standards and class libraries in terms of intended functionality and use model, and proposes ways to build a coherent and consistent eco-system for system-level design and verification using SystemC, TLM, UVM-SystemC, and CCI. The paper is organized as follows: section II will discuss transaction-level modeling differences between SystemC and UVM. Section III will discuss the configuration mechanisms of UVM and CCI. Section IV will look into the register modeling, interface communication between UVM and SystemC. In section V all the language standards and class libraries are brought together, demonstrating the overall SystemC-based design and verification eco-system.

II. Transaction level Modeling in SystemC and UVM

Abstraction of communication schemes, from cycle-based, signal-level communication to transaction-level communication, has been a well-established approach to create efficient and fast system-level models. Transaction-level communication is standardized in SystemC (IEEE Std 1666-2011 [3]), which defines TLM-1 for message passing interfaces and analysis ports, and TLM-2.0 for transport interfaces, direct memory interface (DMI) and debug transport interface. The concept of transaction-level communication is also introduced in the UVM-SystemVerilog standard [4]. Although the UVM-SystemVerilog standard uses similar naming for its TLM interfaces, called 'TLM1' and 'TLM2', some of the language constructs and semantics are quite different and some functionality is not available in UVM-SystemVerilog.

The UVM-SystemVerilog standard specifies TLM1 based interfaces for communication between verification components. Table I shows the difference in language constructs between SystemC TLM-1 and UVM-SystemVerilog TLM1. It shows that the main differences are in the naming of the non-blocking methods to put, get and peek transactions. Missing functionality in UVM-SystemVerilog TLM1 is a notification mechanism using methods similar to **ok_to_put**, **ok_to_get**, and **ok_to_peek**, which indicate that the callee becomes ready to accept or to return the next transaction.

Table I. Differences between SystemC TLM-1 and UVM (in SystemVerilog) TLM1 API

SystemC TLM-1 API	***UVM TLM 1 API (SystemVerilog)***
Blocking put / get / peek interface	
void **put**(const T &t) T **get**(**tlm**::**tlm_tag**<T> *t = 0) void **get**(T &t) T **peek**(**tlm**::**tlm_tag**<T> *t = 0) void **peek**(T &t) const	task **put**(input T t) N/A task **get**(output T t) N/A task **peek**(output T t)
Non-blocking put / get / peek interface	
bool **nb_put**(const T &t) bool **nb_can_put**(**tlm**::**tlm_tag**<T> *t = 0) const const **sc_core**::**sc_event** &**ok_to_put**(tlm_tag<T> *t = 0) const bool **nb_get**(T &t) bool **nb_can_get**(**tlm**::**tlm_tag**<T> *t = 0) const const **sc_core**::**sc_event** &**ok_to_get**(**tlm**::**tlm_tag**<T> *t = 0) const bool **nb_peek**(T &t) const bool **nb_can_peek**(**tlm**::**tlm_tag**<T> *t = 0) const **sc_core**::**sc_event** &**ok_to_peek**(**tlm**::**tlm_tag**<T> *t = 0)	function bit **try_put**(input T t) function bit **can_put**() N/A function bit **try_get**(output T t) function bit **can_get**() N/A function bit **try_peek**(output T t) function bit **can_peek**() N/A
Transport interface	
RSP **transport**(const REQ&) void **transport**(const REQ& req , RSP& rsp) N/A	N/A task **transport**(input REQ req , output RSP rsp) function bit **nb_transport**(input REQ req, output RSP rsp)
Analysis interface	
void **write**(const T&)	function void **write**(input T t)

Note: T, REQ and RSP represent type parameters for the request or response transaction.

For UVM in SystemC, the SystemC TLM-1 interface is used as foundation technology. To remain compatible with the UVM-SystemVerilog standard, the non-blocking put/get/peek interface methods (**try_*** and **can_***) have been added as an alias for the **nb_*** and **nb_can_*** methods. Furthermore, the UVM-SystemVerilog specific TLM1 ports, exports and implementations ('imps') for all blocking and non-blocking put/get/peek methods are implemented in UVM-SystemC by mapping them on the corresponding blocking and non-blocking put/get/peek TLM-1 interfaces. Not available in UVM-SystemC is the non-blocking transport method **nb_transport**, since this is not considered being part of the TLM-1 standard, see [3]. Instead, the non-blocking transport methods defined in TLM-2.0 should be used. No API changes were necessary for the analysis interface.

Besides these difference in TLM-1 language constructs, a more fundamental semantical issue has been found when comparing SystemC TLM-1 and UVM-SystemVerilog TLM1. As TLM-1 follows message passing semantics, the intention is that there should be no shared memory between caller and callee, which means that neither the caller nor the callee is permitted to modify the transaction object once it has been assigned by the sender. This effectively means that TLM-1 defines pass-by-value semantics. However, UVM-SystemVerilog follows pass-by-reference semantics, which means that the transaction object remains accessible (and thus modifiable) by the caller or the callee, since not the transaction content is passed, but the reference (memory location) of the transaction. Although this might be required to support (late) randomization of UVM transactions, it remains incompatible with the SystemC TLM-1 standard. As UVM-SystemC is based on SystemC TLM-1, the message passing semantics strictly follow a pass-by-value regime.

The inclusion of TLM2 interfaces in UVM serve a different purpose. The main reason to add these interfaces is to facilitate communication to a SystemC-based reference model (e.g., as part of a scoreboard) or to communicate to a SystemC-based TLM-2.0 design under test (DUT). As such, the UVM fabric itself does not rely on TLM2 interfaces and semantics, and one can argue why these interfaces were included in the class definition and library. In [5], an elaborate overview of differences between the SystemC TLM-2.0 standard and the UVM-SystemVerilog TLM1 API is given. UVM-SystemVerilog lacks a definition a direct memory interface, which prevents introduction of efficient methodologies to directly access DUT memory via a backdoor-like approach. Also there is no time quantum mechanism defined in UVM-SystemVerilog, which limits the use of more advanced time synchronization schemes for optimized temporal decoupling of communication between verification components or DUT. UVM-SystemVerilog only defines the TLM2 blocking and non-blocking transport interfaces, including class definitions for generic payload and the use of sockets. Unfortunately, these class definitions of UVM-SystemVerilog TLM2 for generic payload and sockets are structurally different from the SystemC TLM-2.0 classes. Due to all these differences and functional limitations, the UVM-SystemC reference implementation, which is using SystemC TLM-2.0 as the foundation, did not add a compatibility layer to support the UVM-SystemVerilog TLM2 class definitions. At this stage, it is proposed to use the SystemC TLM-2.0 API in the context of UVM-SystemC. Further alignment between the SystemC TLM and UVM-SystemVerilog standardization committees is encouraged to agree on a compatible API for SystemC TLM-2.0 and UVM TLM2.

III. Configuration in SystemC CCI and UVM

The configuration of a system-level or reference model, DUT or verification environment requires additional methods for parameterization of variables, to modify e.g. functionality (behavior) or architecture composition in terms of number of IP blocks, verification components or interfaces. The main objective of SystemC CCI is to standardize the interface between system-level models and tools. Although a test bench environment can also be seen as a model, the model-tool interface standard considered by CCI was mainly to configure virtual prototypes for software development or architecture exploration from the simulation tool side, not the test bench. The configuration elements proposed by CCI are based on a *parameter* and *broker* class. The parameter class (cci_param) defines the relation between the parameter name and its value, for an arbitrary C++ type. Each parameter is registered with a broker at construction time. The broker class (cci_broker_if) is in charge of both parameter registration and access control in terms of visibility (e.g., public or hidden parameters) and access type (e.g., read-only). Since CCI is also considered as a tool interface, direct access to the parameter and broker objects enable more advanced tool capacities related to introspection, authoring, debug, coverage, etc.

The UVM standard includes a configuration mechanism to enable to creation of reconfigurable verification environments, to make them reusable and scalable. The main concept is based on the creation of type-specific resource databases, where the parameter name (string) acts as a key to store and retrieve the parameter value. UVM supports two different flavors of configuration: a test bench hierarchy independent approach (so-called *resource database*, uvm_resource_db) and a hierarchy-dependent approach (called the *configuration database*, uvm_config_db). The latter is the promoted and commonly used approach, where also the hierarchical starting point (context) is stored in the database. This facilitates a basic filtering mechanism to make parameters only accessible for certain areas in the test bench hierarchy. A very powerful feature of UVM is the parameter lookup based on a regular expressions including wildcards.

Table II shows the main difference between the SystemC CCI and the UVM configuration capabilities. It summaries the requirements and features of both approaches, partly derived from [6].

Table II. Differences between SystemC CCI and UVM (in SystemVerilog and SystemC) configuration capabilities

Requirement / feature	*SystemC CCI*	*UVM configuration mechanism (in SystemVerilog and SystemC)*
Parameter can be any data type	Yes	Yes
Name-based parameter access	Yes	Yes
Type-based parameter access	No	Yes
Notification mechanism for parameter value change	Yes (callback mechanism)	Yes (event-based mechanism)
Support of different access types (RO, RW, etc.)	Yes	No
Parameter hiding	Yes	Partly (via context)
Parameter locking	Yes	No
Elaboration-time only parameters	Yes	No
Parameter look-up using regular expressions	No	Yes
Parameter authoring/tracing/debug	No (only via external tool)	Yes
Storage of other parameter info (e.g., documentation)	Yes	No
Precedence mechanism for parameters overrides	Yes	Yes
Hierarchy (context) aware parameters	Yes	Yes

The CCI concept of a parameter class offers additional functionality related to the encapsulation of parameters itself, such as parameter hiding, locking, access type, etc. The UVM configuration and resource database is, like the word says, more a database mechanism and does not offer features to manage individual parameters.

Since the CCI and UVM configuration mechanisms are conceptually different, and rather complementary in functionality and features, the application of both techniques is proposed, targeting its specific application domain: CCI primarily to be used for DUT configuration, and using UVM configuration for the test and test bench environment. UVM-SystemC adopted the complete configuration and resource API from UVM-SystemVerilog. Further study is necessary to evaluate if some configuration features of UVM-SystemC can be brought into CCI and vice versa.

IV. REGISTER INTERFACE IN SYSTEMC AND THE UVM REGISTER MODEL

None of the existing SystemC standards specify a register API to model registers or memory. In practice, proprietary, company- or tool-specific solutions are used for the modeling of registers in TLM models. Since the use cases to model registers in SystemC can be different (e.g. registers for SW development, high-level synthesis, or verification), different types of flavors of registers are created, all with different capabilities. In [7], a register introspection interface standard (scireg) is proposed to support register introspection, enabling tools to seamlessly display and update register values. Note that this proposal does not define the register implementation itself. In [8], a C++ Register Modeling Framework has been developed, offering capabilities similar to the UVM (uvm_reg) and Specman/*e* (vr_ad) register libraries.

In UVM, a register abstraction layer is introduced to create a register model representing the actual registers and memories in a DUT. Furthermore, it offers an interface for abstract register read/write operations, supported by specific register sequence and transaction classes. This register model closely follows the register layout of the design hierarchy, containing register bit fields, register blocks, maps, files, which form the full register map.

Although the UVM register model represents the DUT registers, it is not (and should not be) the same register object for the actual DUT. This means that both the UVM verification environment as well as the system-level model (acting as DUT) need their own abstract register model and interface to read/write operations. To guarantee that the DUT register and UVM register model are fully identical, the use of a register model generator is promoted, where the register specification in SystemRDL or IP-XACT acts as golden reference for the register generation flow.

Furthermore, register read/write access means that both the UVM register model as well as the DUT registers needs to be accessed. For this, UVM offers *front-door* and *back-door* access capabilities. When using the front-door, the DUT registers are accessed indirectly via the bus interface, where transactions are executed on the DUT peripherals which then read from or write to the registers. When using the back-door mechanism, the DUT registers are accessed directly using the UVM back-door mechanism, which bypasses the bus interface, and prevents the communication of transactions over the bus interface. However, to facilitate back-door access between a UVM-SystemC based environment and a SystemC-based DUT including registers, as well as having direct access to these registers from the simulation platform (tool), a standardized register read/write interface or register introspection API in SystemC is essential. As register modeling is rather use-case and often company-specific, the adoption of a register introspection API is favored, since it can be added to existing register implementations, without affecting existing modeling and automation flows.

V. THE SYSTEMC BASED ECO-SYSTEM

This section describes the overall eco-system for system-level design and verification by combining SystemC, TLM, UVM-SystemC, and CCI language standards and class libraries, which have been described in the previous sections. Figure 1 gives a graphical representation of the eco-system, where the UVM-SystemC-based verification environment is depicted at the left side and the SystemC-based system-level model as DUT at the right side. The layered structure of UVM defines dedicated components for tests, test bench environment and defines universal verification components (UVC) to realize the actual bus or signal-level interface to the DUT. In addition to the hierarchical composition of the verification environment, UVM defines sequences (stimuli) for register or bus read/write. The communication between the verification components follows the TLM-1 put/get semantics. UVM-SystemC monitors use TLM-1 analysis interfaces to pass the transactions received to other components in the verification environment, such as a coverage collector or scoreboard (not depicted).

The UVM-SystemC configuration mechanism is used to configure the functionality or composition of the test, test bench, and UVC(s) and selects the sequences which are being executed. It is not meant to configure the DUT. For that purpose, the CCI configuration is proposed.

In this simple example, the UVM-SystemC register model shows three registers, representing the registers of the bus peripheral in the DUT. For front-door access via the bus interface, the adapter translates the register R/W sequences into bus R/W transactions. For back-door access, a direct relation between the DUT register implementation and register model is specified. As explained in section IV, this requires a standardized interface to the register (introspection) interface in the DUT.

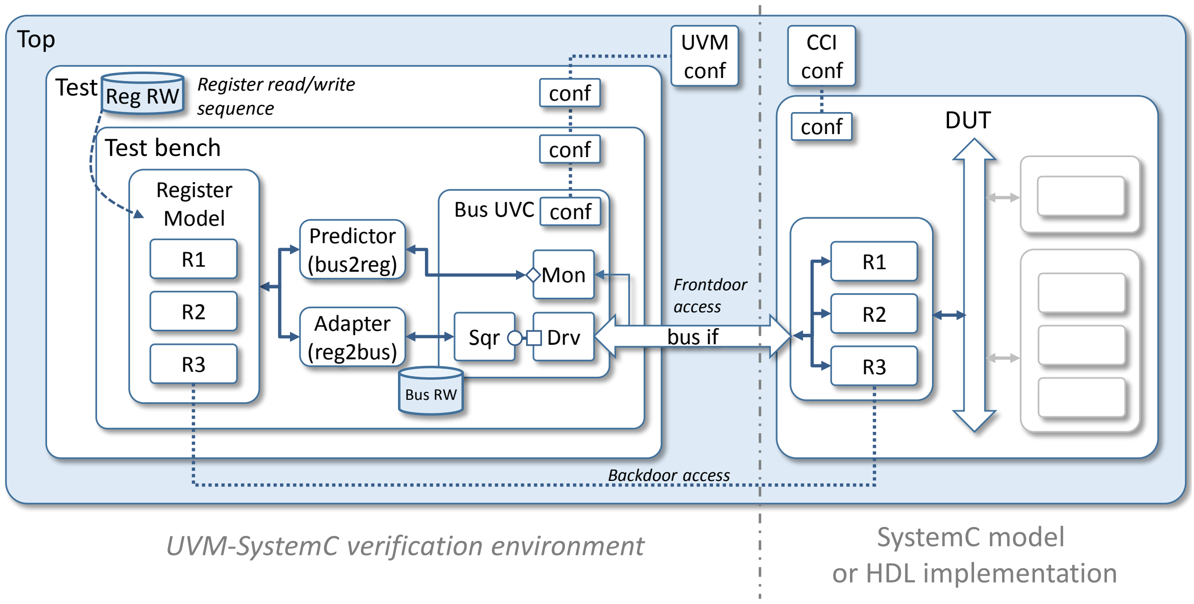

Figure 1. Design and verification eco-system using SystemC, TLM, UVM-SystemC, and CCI

VI. Conclusion and Summary

This paper gave an overview of the available system-level design and verification methodologies, language standards and class libraries based on SystemC, TLM, UVM-SystemC and CCI. It presented the functionality and differences between these standards and class libraries related to transaction-based communication, configuration, and register interface and modeling. It proposed solutions how these technologies can be combined, or how they could be extended, to create a consistent and coherent SystemC-based eco-system for system-level design and verification. As a summary, the following observations and proposals have been presented:

- *Transaction level modeling in SystemC and UVM*: The differences between the SystemC TLM-1 and the UVM TLM1 API have been resolved by introducing dedicated UVM TLM1 ports, exports, and port implementations in UVM-SystemC which are mapped on the SystemC TLM-1 interfaces. Unresolved topics are the difference in message passing semantics between SystemC TLM-1 versus UVM TLM1 and alignment of the SystemC TLM-2.0 and UVM TLM2 API. Alignment between the various standardization committees is encouraged to harmonize these TLM standards.

- *Configuration in SystemC CCI and UVM*: Due to the rather complementary nature of the configuration mechanisms in CCI and UVM, it is proposed to apply CCI for the configuration of the system-level model and to use the UVM configuration database to configure the verification environment and its components. Further study is recommended to evaluate if some configuration features of UVM-SystemC can be brought into CCI and vice versa.

- *Register interface in SystemC and UVM register model*: To facilitate back-door access between a UVM-SystemC based environment and a SystemC-based DUT including registers, as well as having direct access to these registers from a simulation platform (tool), the adoption of a standardized register introspection API is recommended. It facilitates an interface to existing register implementations, without the need to standardize a dedicated register modeling approach. It is encouraged to use the register abstraction layer defined in UVM for the creation of a UVM register model for verification purposes.

These proposals and recommendations are meant for further discussion, to help in the evolution and alignment of the various standard developments, methodologies and their applications.

Acknowledgment

The author would like to thank Philipp A Hartmann, chair of the Accellera SystemC language working group and Trevor Wieman, chair of the SystemC CCI working group for their valuable input and suggestions to further improve the completeness and correctness of this paper.

References

[1] Accellera Systems Initiative, CCI Working Group update, DVCon US 2013, http://www.accellera.org/images/community/systemc/about-systemc-cci/CCI_WG_Update_2-25-2013.pdf

[2] Accellera Systems Initiative, UVM-SystemC library and public review, December 2015, http://www.accellera.org/activities/working-groups/systemc-verification

[3] IEEE Stanadards Association, IEEE 1666-2011 - IEEE Standard for Standard SystemC Language Reference Manual, https://standards.ieee.org/findstds/standard/1666-2011.html

[4] Accellera Systems Initiative, Universal Verification Methodology (UVM) 1.2 Class Reference, June 2014, http://accellera.org/downloads/standards/uvm

[5] David Long, John Aynsley, A Beginner's Guide to Using SystemC TLM-2.0 IP with UVM, SNUG 2012

[6] Accellera Systems Initiative, SystemC CCI Configuration Requirements Specification, December 2009, http://accellera.org/downloads/drafts-review

[7] STMicroelectronics, ARM, and Cadence, Proposed Interfaces for Interrupt Modeling, Register Introspection and Modeling of Memory Maps, July 2013, http://forums.accellera.org/files/file/102-

[8] AMIQ, C++ Register Modeling Framework, December 2015, http://www.amiq.com/consulting/2015/12/09/c-register-modeling-framework/