Advancing system-level verification using UVM in SystemC

Martin Barnasconi, NXP Semiconductors, Eindhoven, The Netherlands (martin.barnasconi@nxp.com)
François Pêcheux, University Pierre and Marie Curie, LIP6, Paris, France (francois.pecheux@lip6.fr)
Thilo Vörtler, Fraunhofer IIS/EAS, Dresden, Germany (thilo.voertler@eas.iis.fraunhofer.de)

Abstract

This paper introduces the Universal Verification Methodology (UVM) using SystemC and C++ (UVM-SystemC), to advance system-level verification practices. UVM-SystemC enables the creation of a structured, modular, configurable and reusable test bench environment. Unlike other initiatives to create UVM in SystemC, the presented proof-of-concept class library uses identical constructs as defined in the UVM standard for test and sequence creation, verification component and test bench configuration and execution by means of simulation. Users familiar with either SystemC and/or with UVM will immediately feel comfortable to start using UVM-SystemC right away. The Universal Verification Methodology becomes universal, at last.

Keywords

Electronic System Level (ESL), Hardware-in-the-Loop (HiL), Rapid Control Prototyping (RCP), SystemC, SystemC Verification (SCV), Transaction Level Modeling (TLM), Universal Verification Methodology (UVM).

1 - Introduction

Electronic System Level (ESL) design has become a mature and proven practice to tackle the challenges in the concept and architecture design phases of complex embedded systems. Novel design technologies based on system-level language standards like SystemC [1] and the definition of transaction-level modeling (TLM) resulted in a variety of ESL methodologies, flows and tools to assist system architects and engineers to design and model these embedded systems. Examples are the creation of virtual prototypes to support use cases such as software development, architecture exploration and system verification [2].

However, the primary focus in ESL design has always been on the actual *system-level design* aspects, in terms of modeling the hardware and software components, and to a lesser extent on the creation of reusable *verification environments*. But as the verification effort starts dominating the total design effort, more advanced verification methodologies are needed to enable reuse and interoperability of SystemC-based test benches or verification components (developed internally or offered by 3rd parties) between the system-level and sub-system or RTL implementation phase.

This is where the Universal Verification Methodology standard [3] inspired us. UVM consolidates verification best practices by introducing a unified approach for test and sequence creation, building verification components, test bench configuration, and execution by means of simulation. To benefit from these concepts, essential UVM features are introduced in a SystemC/C++ class library, to avoid multi-language integration hassles and SystemVerilog [4] dependencies within a SystemC-centric ESL design and verification flow.

This paper gives an introduction to the verification methodology and class library in UVM-SystemC and demonstrates the strong resemblance with the UVM standard. The paper is organized as follows: the next section gives a historical perspective based on prior work and explains the main reasons for developing UVM in SystemC. In section 3, the basic concepts and features of UVM-SystemC are presented, followed by the foundation elements and examples presented in section 4. In section 5, the creation of a simple yet complete verification environment in UVM-SystemC is explained. The paper concludes with a summary and outlook in section 6.

2 – Historical perspective and motivation

The trend to start earlier with system-verification in the design cycle, in combination with the growing complexity of the actual design implementation (and its virtual prototype counterpart), emphasizes the need to apply a proven verification methodology. Therefore there have been many attempts in the past to create structured and reusable verification environments in SystemC/C++. In [5], the Open Verification Methodology (OVM) formed the basis for creating a SystemC-equivalent verification environment. The System Verification Methodology [6] was based on OVM-SystemC, donated by Cadence to the community [7]. Mentor Graphics released a SystemC version as part of their Advanced Verification Methodology (AVM) [8]. Also Synopsys introduced as part of their Verification Methodology Manual (VMM) a class library in SystemC [9, 10]. More recent donations to the community [11] address the need to support a true multi-language verification environment in SystemVerilog, SystemC and ***e*** [12].

However, all these initiatives do not fully comply with the methods defined in the UVM standard, primarily because they are built on the former AVM, OVM, and VMM technologies. The consolidation into a single UVM standard resulted in major changes. As a consequence, the user has to deal with the incompatibilities related to simulation semantics and language constructs. Especially the move from OVM to UVM significantly changed the way components deal with the phasing mechanism and how the end-of-test is managed. To avoid legacy concepts and constructs in modern test benches, migration to UVM standard compatible implementations should be encouraged.

Alternative solutions are proposed in [13, 14, 15] to address the multi-language integration challenges found in today's verification environments, by defining a set of coding guidelines centered around TLM communication. However, the creation of reusable verification components and integration in a test bench is much more than having an agreed communication method; additional elements like test bench configuration and reuse of test sequences do require a more holistic view on UVM and its principles, and justifies making these concepts available in other languages.

Therefore an up-to-date and UVM standard compliant language definition and reference implementation is needed in SystemC/C++, which not only gives the user community a semantically and syntactically correct implementation of UVM, but also the same user experience in terms of the UVM "look & feel". Especially the latter aspect would facilitate UVM users to start using SystemC for system-level and hardware/software co-verification, or make SystemC or software experts more familiar with the powerful UVM concepts to advance in the verification practice.

Ultimately, this will benefit the entire design community, where verification and system-level design practices come closer together. It will serve the *universal* objectives of the UVM, addressing the need for having a common verification platform, including hardware prototyping in which C-based test sequences or verification components in UVM-SystemC are reused in Hardware-in-the-Loop (HiL) simulation or Rapid Control Prototyping (RCP) [16].

3 – Overview of UVM-SystemC

UVM-SystemC defines all the essential features to create a UVM standard compliant verification environment. It contains many built-in capabilities dedicated to verification, such as test and test bench creation, configuration, phasing, comparing, scoreboarding, reporting, etc. Table 1 gives a summary of the available UVM-SystemC features and also lists the elements which are under development. In the following sections, the essential UVM concepts described in this table are introduced to create a structured, modular, configurable and reusable verification environment.

UVM-SystemC functionality	Status
Testbench creation with component classes: agent, sequencer, driver, monitor, scoreboard, etc.	☑
Test creation with test, (virtual) sequences, etc.	☑
Configuration and factory mechanism	☑
Phasing and objections	☑
Policies to print, compare, pack, unpack, etc.	☑
Messaging and reporting	☑
Register abstraction layer and callbacks	development
Coverage groups	development
Constrained randomization	SCV or CRAVE

Table 1: Overview of UVM-SystemC features

3A – Test and test bench architecture – a layered approach

UVM-SystemC follows the same layered architecture as defined in [9], where different levels of abstraction are introduced to clearly distinguish test case definition from test scenarios and the actual verification environment on which these sequences are executed, as shown in Figure 1.

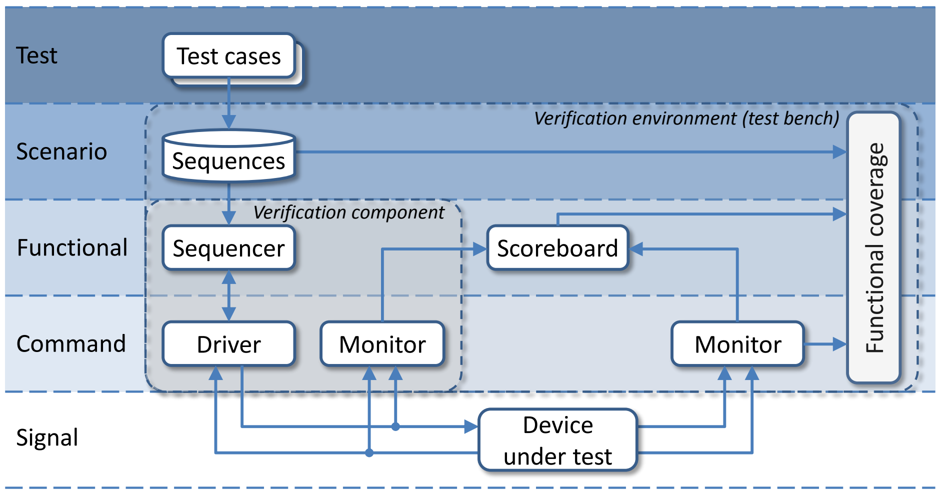


Figure 1: Layered verification environment architecture of UVM

The highest layer defines the actual tests, which consist of the selection of the verification environment (test bench) and sequences. The scenario layer will generate the actual test sequence(s), which consists of streams of transactions. The functional layer contains the sequencers, which deal with the ordering and arbitration of these transactions. In addition it also defines the self-checking infrastructure using a scoreboard and performs functional coverage and pass/fail reporting. The command layer contains the physical-level drivers, monitors and checkers. The signal layer, which is the lowest layer in the UVM test bench hierarchy, connects the test bench via (physical) signals with the Device-under-test (DUT).

With the overall objective to encourage reuse of verification environments and tests scenarios, UVM is built in a modular fashion with standardized communication interfaces between its components. A typical verification environment consists of multiple UVM verification components or agents, which typically contain a sequencer, driver and monitor. These verification components use TLM to interface with the higher levels in the test bench architecture and use a physical-level interface to communicate with the DUT.

For the creation of the test bench and its verification components, specific classes derived from **uvm_component** are available: **uvm_env**, **uvm_agent**, **uvm_monitor**, **uvm_driver**, **uvm_sequencer**, **uvm_scoreboard**, etc. These dedicated classes are introduced to be able to distinguish each component from another component and to introduce specific methods per component type. For the creation of tests, the class **uvm_test** is available, forming the container class to instantiate the test bench and to select the (virtual) sequence (**uvm_sequence**) which will be executed.

3B – Configurability and refactoring using the configuration and factory mechanism

To facilitate and promote reuse of existing verification components, entire environments or test sequences in different tests or even across different projects, a high degree of reconfiguration and refactoring capabilities are offered in UVM-SystemC.

The availability of a resource and configuration database (**uvm_resource_db** and **uvm_config_db**) is beneficial to provide access to configuration information from any place in the verification environment. Configuration properties are often stored at one of the higher layers in the UVM stack (e.g. in the test or test bench) and retrieved at the lower levels (e.g. in the agent, driver or monitor) to configure the components for specific verification tasks. For example, the interface object which is instantiated and connected to the DUT in the top-level, is also stored in the configuration database, so each verification component can be bound to the same interface by retrieving the handle to this interface object. The configuration database supports name-based and type-based storage and retrieval by introducing the static member functions **set** and **get**, respectively. The use of wildcards (e.g. * and ?) and regular expressions facilitate the configuration of multiple attributes with a single definition.

The UVM factory (**uvm_factory**) is based on the classical factory design pattern [17] to create objects without specifying the exact implementation for these objects that will be created. It offers a non-intrusive and flexible way of reconfiguration of the test sequence or test bench topology without changing the original code. For example, UVM component and object overrides for specific tests are specified at a high level in the test bench hierarchy. The configuration and factory in UVM-SystemC are object-type aware. This means that the UVM **type_id**-based configuration and creation functions in the factory are supported to substitute a predefined object or component type with another specialized type. In addition to type overrides, factory overrides by instance name are possible. The use of the UVM factory requires that each verification object and component is derived from class **uvm_object** or **uvm_component**, is registered in the factory by means of special utility macros (**UVM_*_UTILS**), and is instantiated using the static member function **type_id::create**. In section 4 the usage of the configuration and factory mechanism is demonstrated.

3C – Test execution using the phasing and objection mechanism

For the execution of tests, the UVM phasing mechanism follows a well-defined order of execution of pre-defined callback functions and processes called *phases*, which run either sequentially or concurrently. The primary objective of the phasing mechanism is to facilitate synchronization between the UVM components during execution. This section explains in detail how the UVM phases are mapped onto the SystemC phases.

UVM defines nine common phases, including four pre-run phases (**build_phase**, **connect_phase**, **end_of_elaboration_phase** and **start_of_simulation_phase**), a run-phase (**run_phase**) and four post-run phases (**extract_phase**, **check_phase**, **report_phase** and **final_phase**). These phases are mapped on the regular SystemC phases **before_end_of_elaboration**, **end_of_elaboration** and **start_of_simulation**, see Figure 2. In contrast to the UVM implementation in SystemVerilog, where all UVM phases are executed as part of the simulation phase, the UVM-SystemC implementation executes the pre-run phases as part of the SystemC elaboration phase. This means that the construction of the design hierarchy and connections for the (SystemC) DUT and (UVM-SystemC) test bench and its verification components all happen in the elaboration phase and thus prior to simulation.

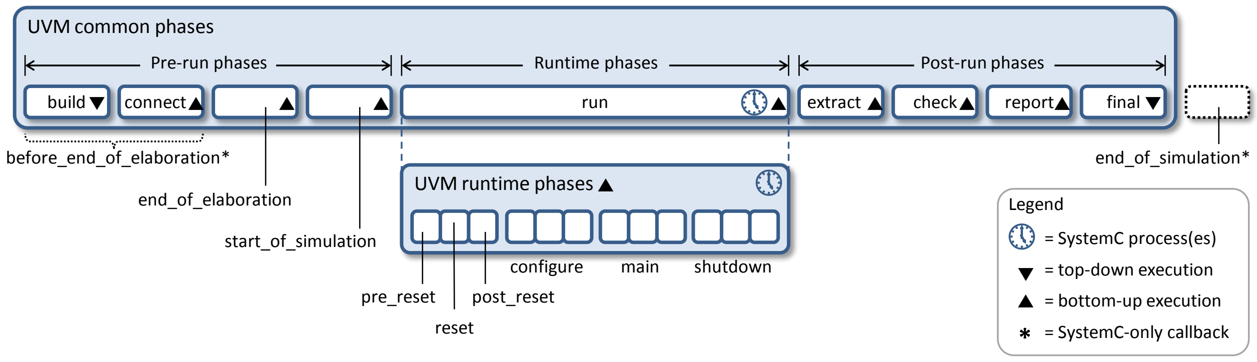


Figure 2: UVM common and runtime phases mapped on the SystemC phases

In addition to these common phases, a refined runtime phasing schedule is available, which executes in parallel with the existing run-phase. It offers a **reset_phase**, **configure_phase**, **main_phase** and **shutdown_phase** to encapsulate DUT specific actions. Each of these runtime phases offer additional callbacks for pre- and post-processing functions, and can be recognized with the **pre_** and **post_** prefix. The practical use of these refined runtime phases is a topic of discussion in the UVM standardization committee, and it remains unclear at this stage whether changes are made to these runtime phases. Although available in UVM-SystemC, the use of runtime phases is therefore not recommended. Besides the common and UVM runtime phases, user-defined phases could be added. UVM-SystemC also supports forward and backward jumping for the runtime phases.

The callbacks **build_phase** and **connect_phase** are executed sequentially within the SystemC **before_end_of_elaboration** phase. In the build phase, all UVM components are instantiated to construct the entire test bench topology using the factory. All components are connected in the connect phase. It is recommended to use the UVM build and connect phase, and not use the SystemC **before_end_of_elaboration** callback, to guarantee proper execution of the test bench construction.

The UVM callback **end_of_elaboration_phase** is available for post-elaboration activity, such as printing or hierarchy analysis. This callback is executed as part of the SystemC **end_of_elaboration** phase. The UVM callback **start_of_simulation_phase** is available to configure the verification components. It should not be used to configure the DUT. This callback is executed as part of the SystemC **start_of_simulation** phase.

The callback **run_phase** and (optional) runtime phases offer the placeholders to execute the actual test scenarios and to perform configurations for the DUT. These runtime phases are all spawned SystemC processes and can consume time, unlike the other phases which are untimed function calls.

Post-processing of the results is performed in the callbacks **extract_phase**, **check_phase** and **report_phase**, where the relevant results from the UVM verification components are extracted, checked and reported, respectively. The **final_phase** enables the application to clean-up and finalize the verification activity, such as closing files. Although not part of the UVM standard, the SystemC callback **end_of_simulation** could be used to perform additional tasks before the simulation is terminated.

Execution of these phases is done in a top-down (▼) or bottom-up order (▲), as depicted in Figure 2. A top-down and bottom-up phase is completed as soon as all callbacks for all applicable components in the hierarchy have been called and returned. UVM-SystemC also implements the objection mechanism to synchronize the runtime phases. A runtime phase should raise an objection (**raise_objection**) to block execution to the next phase and after completion of its tasks it should drop the objection (**drop_objection**) to allow proceeding to the next phase. A runtime phase will terminate as soon as all objections for that phase are dropped. In this case, all running (or waiting) spawned SystemC processes are killed.

3D – Other UVM-SystemC features

As most of the communication in the UVM test bench is transaction-oriented, additional functionality is offered to support printing, comparing, packing, and unpacking of these transactions and their internal properties. For example, the printer policy (**uvm_printer**) enables printing of the transaction properties to the standard output (stdout) in a table or tree-based manner. The comparer functionality (**uvm_comparer**) can be used in a scoreboard to perform the comparison of the reference and measured transactions. To serialize and de-serialize the transaction properties, the pack and unpack policies (**uvm_packer**) can be used, respectively

The messaging and reporting features offer the user the standard UVM functionality to report information, warnings, errors or fatal errors. It supports a component-level reporting mechanism by setting the severity level on a per-instance basis and messages can be filtered based on their verbosity settings. In addition, the convenience macros **UVM_INFO**, **UVM_WARNING**, **UVM_ERROR**, and **UVM_FATAL** are made available.

Currently under development are the UVM register abstraction layer and the callback interface. The UVM implementation in SystemVerilog makes use of 'native' verification features of this language, such as the constrained randomization and coverage groups. Therefore UVM-SystemC will define the necessary language constructs for randomization and coverage, but will rely on an external constrained random solver, for example based on the SystemC verification (SCV) [18] or CRAVE [19] library.

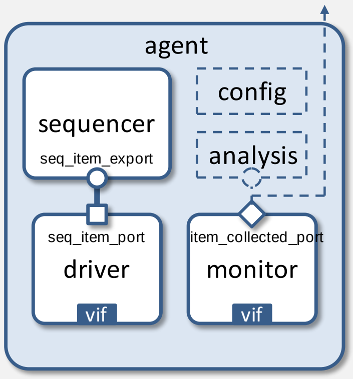
4 – Using the UVM-SystemC foundation elements

In this section, the foundation elements of UVM-SystemC are explained in detail, like the creation of a verification component, the application of the factory for registration and component instantiation, and the definition of sequence items, sequences and sequencers. To keep the examples in this paper concise, the code of the interface (.h) and the implementation (.cpp) are combined.

4A – The cornerstone of UVM: the agent component

Listing 1 below shows the creation of an UVM component with the user-defined name *vip_agent* in UVM-SystemC. An agent encapsulates the components which are necessary to drive and monitor the (physical) signals to (or from) the DUT. Typically, it contains three components: a sequencer, a driver and a monitor. The agent could also contain analysis functionality for basic coverage and checking, but this is not covered in this simple example.

```
class vip_agent : public uvm_agent
{
 public:
  vip_sequencer<vip_trans>* sequencer;
  vip_driver<vip_trans>*    driver;
  vip_monitor*              monitor;

  vip_agent( uvm_name name )
  : uvm_agent( name ), sequencer(0), driver(0), monitor(0) {}

  UVM_COMPONENT_UTILS(vip_agent);

  virtual void build_phase( uvm_phase& phase )
  {
    uvm_agent::build_phase(phase);

    if ( get_is_active() == UVM_ACTIVE )
    {
```



```
19        sequencer = vip_sequencer<vip_trans>::type_id::create("sequencer", this);
20        assert(sequencer);
21        driver = vip_driver<vip_trans>::type_id::create("driver", this);
22        assert(driver);
23      }
24      monitor = vip_monitor::type_id::create("monitor", this);
25      assert(monitor);
26    }
27
28    virtual void connect_phase( uvm_phase& phase )
29    {
30      if ( get_is_active() == UVM_ACTIVE )
31        driver->seq_item_port.connect(sequencer->seq_item_export);
32    }
33  };
```

Listing 1: Agent in UVM-SystemC

A UVM agent inherits its basic functionality from the base class **uvm_agent** (Line 1). In contrast with UVM-SystemVerilog, which offers the constructor called 'new', the constructor in C++ has the same name as the class name (Line 8). The argument with type **uvm_name** is the container for the string name and provides the mechanism for building the hierarchical names. In order to register this component in the UVM factory, the macro **UVM_COMPONENT_UTILS** is used (Line 11).

The agent implements two callbacks, namely the **build_phase** (Line 13) and the **connect_phase** (Line 28), which are called in the simulation build and connect phase, respectively. The user should always call the **build_phase** of the base class (Line 15), to make sure that the configuration variables for this component are retrieved correctly.

UVM agents can be active or passive. Active agents drive signals to the DUT and thus instantiates a driver and sequencer. Passive agents not do not drive the DUT and therefore do not instantiate a driver and sequencer. Independent on the active or passive mode of the agent, the monitor is always instantiated to collect the results. In Listing 1, the member function **get_is_active** is used (Line 17 and 30) to retrieve the configuration for this agent. This configuration item is defined in the test bench, which is discussed in section 5, Listing 8.

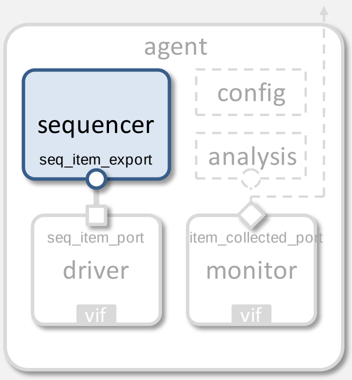
Instead of using the C++ operator **new** to instantiate the sequencer, driver and monitor, the UVM factory is used for instantiation, by means of the static member function **type_id::create** (Line 19, 21, and 24). In this way, the test bench or test definition itself is able to override these components for dedicated tests.

The connection between the sequencer and driver is done in Line 31. Both driver port (**seq_item_port**) and sequencer port (**seq_item_export**) are part of their respective component base classes. For compatibility reasons, the member function **connect** is made available for users familiar with UVM, but users familiar with SystemC can also use the member function **bind** or **operator()**, as this connection is made using the SystemC port binding infrastructure. The connection of the driver and monitor to the DUT is done via the configuration mechanism inside the driver or monitor, and not in the agent. This is presented in section 5.

4B – The UVM-SystemC sequencer

The creation of a sequencer is often a trivial task for the user, simply because all functionality, such as sequence ordering and arbitration, is inherited from the base class. Listing 2 below shows the implementation of the *vip_sequencer*, which is a template class derived from **uvm_sequence** (Line 2). The template argument REQ is the 'request', which defines the sequence item (Line 1).

```
1   template <class REQ>
2   class vip_sequencer : public uvm_sequencer<REQ>
3   {
4    public:
5     vip_sequencer( uvm_name name )
6      : uvm_sequencer<REQ>( name ) {}
7
8     UVM_COMPONENT_PARAM_UTILS(vip_sequencer<REQ>);
9
10  };
```



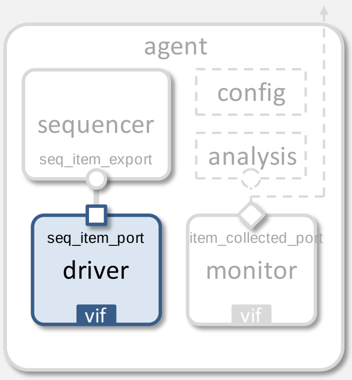
Listing 2: Sequencer in UVM-SystemC

Similar as in the agent, the constructor argument for the sequencer, **uvm_name**, defines the string name and stores the hierarchical name (Line 5). The sequencer is registered in the UVM factory using the macro **UVM_COMPONENT_PARAM_UTILS** (Line 8). Note that the template argument is added to the class name in this macro, offering a very flexible mechanism to deal with registration of template classes.

4C – The UVM-SystemC driver

The agent and sequencer presented in the previous sections did not reveal many fancy features of UVM-SystemC; it was mainly instantiation of components and factory registration. In Listing 3, showing the driver, more interesting UVM-SystemC features are introduced.

```
1   template <class REQ>
2   class vip_driver : public uvm_driver<REQ>
3   {
4    public:
5     vip_if* vif;
6
7     vip_driver( uvm_name name ) : uvm_driver<REQ>(name), vif(0) {}
8
9     UVM_COMPONENT_PARAM_UTILS(vip_driver<REQ>);
10
11    virtual void build_phase( uvm_phase& phase )
12    {
13     uvm_driver<REQ>::build_phase(phase);
14
15     if ( !uvm_config_db<vip_if*>::get(this, "*", "vif", vif) )
16        UVM_FATAL( this->get_name(), "Interface to DUT not defined! Simulation aborted!" );
17    }
18
19    virtual void run_phase( uvm_phase& phase )
20    {
21      REQ req, rsp;
22      while(true) // forever loop
23      {
24        this->seq_item_port->get_next_item(req);
25        drive_transfer(req);
26        rsp.set_id_info(req);
27        this->seq_item_port->item_done();
28        this->seq_item_port->put_response(rsp);
```



```
class vip_if
{
 public:
  sc_signal<int> sig_a;
  ...
  vip_if() : sig_a("sig_a"), ...
  {}
};
```

```
29      }
30    }
31
32    virtual void drive_transfer( const REQ& p )
33    {
34      ...
35      vif->sig_a.write(...);
36    }
37  };
```

Listing 3: Driver in UVM-SystemC

The UVM driver called *vip_driver* is a template class and inherits its basic functionality from the base class **uvm_driver** (Line 2). The template argument REQ is the 'request', which defines the actual sequence item (transaction) which is used to store the relevant properties for the creation of the physical signals to drive the DUT.

Line 5 defines the placeholder *vif* needed to store the handle to the interface object of type *vip_if*, which contains one or multiple channels (e.g. of type **sc_signal**, see code inlay). The interface is retrieved using the member function **get** of the configuration database (Line 15). The interface is defined (aka **set**) in the top-level, which is described in the next section. In case no interface is found, a fatal error is presented and the simulation will stop, by using the reporting macro **UVM_FATAL** (Line 16).

The driver's constructor argument **uvm_name** passes the string name to the base class (Line 7). As the driver is also a template class, it is registered using the macro **UVM_COMPONENT_PARAM_UTILS** where the class name and template argument are passed as argument of this macro (Line 9).

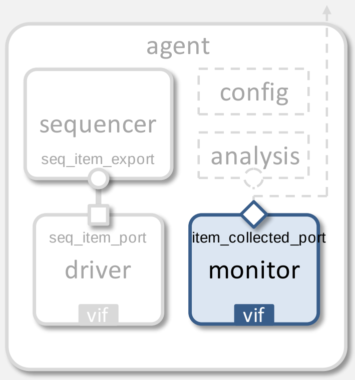
The actual behavior of the driver is implemented in the callback **run_phase** (Line 19). In this callback, the driver repeatedly requests sequence items (transactions), encapsulated in a sequence, via the sequencer, using the member function **get_next_item** (Line 24). The 'request' transaction is translated to one or more physical signal(s) in the user-defined member function *driver_transfer*. Note that in this function, the value is directly written to the channel, which is accessible via the interface *vip_if* (Line 35). In order to indicate to the sequencer that the transaction is completed, the driver calls the member function **item_done** (Line 27). Optionally, response information can be passed back to the sequencer using the member function **put_response** (Line 28). In this case, the identity information of the initial request transaction should be made available as well, and therefore the member function **set_id_info** is called beforehand to copy this information from the request into the response transaction (Line 26).

4D – The UVM-SystemC monitor

A UVM monitor collects signal information from the interface it is connected to. This information is combined into transactions, which are made available to the other components in the verification environment, such as scoreboard or coverage collection objects, using TLM analysis interfaces. Listing 4 shows an example of a monitor called *vip_monitor*, which is derived from class **uvm_monitor** (Line 1). The collected information is exported on an analysis port *item_collected_port*, containing transactions of type *vip_trans* (Line 4). Although not explained in detail in this simple example, a monitor can contain additional checking and coverage functionality, which can be enabled by using the configuration mechanism (Line 6, 7, 19, 20, 35, and 36). The end-to-end self-checking is done in scoreboards, and will be explained in section 5.

The constructor argument of type **uvm_name** passes the string name to the base class (Line 9). The macro **UVM_COMPONENT_UTILS** is used to register this object in the factory (Line 13).

```
 1  class vip_monitor : public uvm_monitor
 2  {
 3   public:
 4    uvm_analysis_port<vip_trans> item_collected_port;
 5    vip_if* vif;
 6    bool checks_enable;
 7    bool coverage_enable;
 8
 9    vip_monitor( uvm_name name ) : uvm_monitor( name ),
10      item_collected_port("item_collected_port"), vif(0),
11      checks_enable(false), coverage_enable(false) {}
12
13    UVM_COMPONENT_UTILS(vip_monitor);
14
15    virtual void build_phase( uvm_phase& phase )
16    {
17      uvm_monitor::build_phase(phase);
18
19      uvm_config_db<bool>::get(this, "*", "checks_enable", checks_enable);
20      uvm_config_db<bool>::get(this, "*", "coverage_enable", coverage_enable);
21      if ( !uvm_config_db<vip_if*>::get(this, "*", "vif", vif) )
22        UVM_FATAL( get_name(), "Interface to DUT not defined! Simulation aborted!" );
23    }
24
25    virtual void run_phase( uvm_phase& phase )
26    {
27      vip_trans p;
28      while (true) // forever loop
29      {
30        ... // Collect the data into transaction vip_trans
31        item_collected_port.write(p);
32
33        if ( checks_enable ) { ... }
34        if ( coverage_enable ) { ... }
35      }
36    }
37  };
```



Listing 4: Monitor in UVM-SystemC

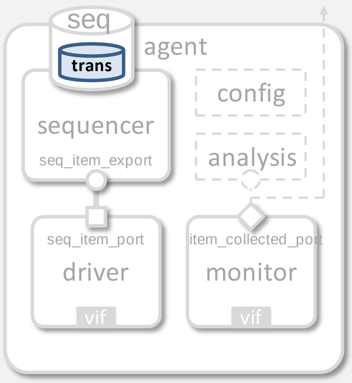
The actual signal detection and collection is done in an endless loop as part of the **run_phase** and is sent to the analysis port using the member function **write** (Line 31).

4E – Sequence items and sequences

In the previous section the actual signal to drive the DUT was encapsulated in a sequence, which was processed by the driver by requesting it from the sequencer. This section will show how sequence item and sequences are defined in UVM-SystemC.

Sequences encapsulate sequence items, also called transactions. The sequence item contains the actual properties (or attributes) of a transaction. Note that sequences are not part of the test bench hierarchy, but are UVM objects which are mapped on one or more sequencers. Similar as with verification components, sequences and sequence items can be configured via the factory. Sequences may contain other sequences to form a layered structure.

```
1   class vip_trans : public uvm_sequence_item
2   {
3    public:
4     int addr;
5     int data;
6     bus_op_t op;
7
8     vip_trans( const std::string& name = "vip_trans" )
9     : addr(0x0), data(0x0), op(BUS_READ) {}
10
11    UVM_OBJECT_UTILS(vip_trans);
12
13    virtual void do_print( uvm_printer& printer ) const { ... }
14    virtual void do_pack( uvm_packer& packer ) const { ... }
15    virtual void do_unpack( uvm_packer& packer ) { ... }
16    virtual void do_copy( const uvm_object* rhs ) { ... }
17    virtual bool do_compare( const uvm_object* rhs ) const { ... }
18  };
```

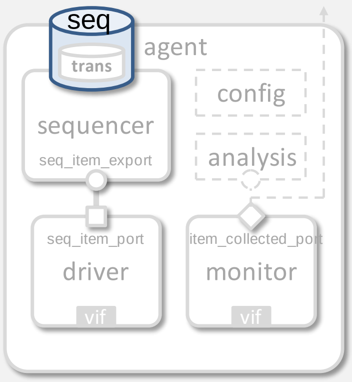


Listing 5: Sequence item in UVM-SystemC

Listing 5 shows the implementation of a sequence item called *vip_trans*. The transaction is derived from class **uvm_sequence_item** (Line 1). It contains three member variables address (*addr*), data (*data*) and an enumerator indicating *BUS_READ* or *BUS_WRITE* (Line 4-6). The macro **UVM_OBJECT_UTILS** is used to register the sequence item in the factory (Line 12). UVM-SystemC does not implement the field macros, due to the known limitations of runtime performance and debug transparency. Therefore a sequence item should implement all elementary member functions to print, pack, unpack, copy and compare the data items explicitly. For this, the member functions **do_print**, **do_pack**, **do_unpack**, **do_copy** and **do_compare** are available (Line 13-17). Note that the example does not show the constrained randomization features, as these are not implemented in UVM-SystemC. Instead, libraries like SCV or CRAVE can be used for this purpose.

The UVM sequence item is used as template argument for the creation of the actual sequence, as shown in Listing 6. A sequence is derived from template class **uvm_sequence** (Line 2). The macro **UVM_OBJECT_PARAM_UTILS** supports the registration of template classes with multiple arguments, which are derived from **uvm_object** (Line 8).

```
1   template <typename REQ = uvm_sequence_item, typename RSP = REQ>
2   class vip_sequence : public uvm_sequence<REQ,RSP>
3   {
4    public:
5     vip_sequence( const std::string& name )
6     : uvm_sequence<REQ,RSP>( name ) {}
7
8     UVM_OBJECT_PARAM_UTILS(vip_sequence<REQ,RSP>);
9
10    virtual void pre_body()
11    {
12      if ( starting_phase != NULL )
13        starting_phase->raise_objection(this);
14    }
15
16    virtual void body()
17    {
18      REQ* req = new REQ();
```



```
19      RSP* rsp = new RSP();
20      ...
21      start_item(req);
22      // req->randomize(); // randomization compatibility layer not yet available in UVM-SystemC
23      finish_item(req);
24      get_response(rsp);
25    }
26
27    virtual void post_body()
28    {
29      if ( starting_phase != NULL )
30        starting_phase->drop_objection(this);
31    }
32  };
```

Listing 6: Sequence in UVM-SystemC

The callback function **body** is used to implement the user-specific test scenario (Line 16). The member functions **start_item** and **finish_item** are called to negotiate and then send the sequence to the sequencer (Line 21 and 23). The member function **randomize**, defined as part of the compatibility layer to the SCV or CRAVE library, is not yet available (Line 22).

The member function **body** is automatically called from the higher level in the test bench, for example by explicitly calling the member function **start** in a sequence. Alternatively, a sequence can be started implicitly by defining a default sequence in a test along with specifying the component and phase where it should be executed. The latter approach is presented in the next section, see Listing 10.

The callback functions **pre_body** and **post_body** (Line 10 and 27) are called before and after the callback **body**, respectively, to raise and to drop an objection only when the sequence has no parent sequence. For this purpose, the data member **starting_phase** is used, offering a handle to the default sequence (Line 13 and 30).

5 – Creating test benches and executing tests in UVM-SystemC

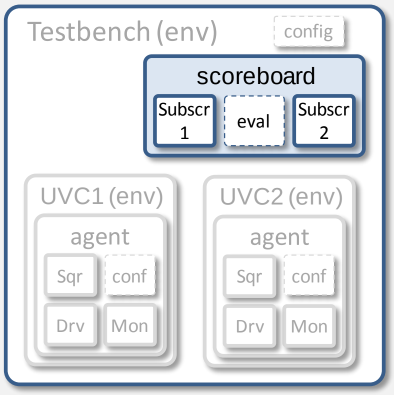
In the previous section the basic foundation elements of UVM-SystemC were presented to create verification components and sequences. In practice however, we expect that the focus in verification should be on the development of the actual tests, creation of the scoreboard, and execution of numerous test scenarios. The creation and configuration of the test bench should be a straight-forward exercise by reusing verification components (developed internally or offered by 3rd parties) and reusing predefined sequences from a sequence library.

Therefore we present in this section the process of assembly and configuration of the test bench and demonstrate how to define tests, which select and execute sequences on the test bench. It also shows the creation of a scoreboard to make self-checking test benches and the use of the type-based configuration database to connect the DUT with the verification components across the test bench hierarchy, all using SystemC and C++ constructs.

5A – Creating self-checking test benches using a scoreboard

A UVM scoreboard is responsible for the self-checking capabilities in a verification environment. Listing 7 shows a typical skeleton of a scoreboard, using the base class **uvm_scoreboard** (Line 1). It typically contains multiple subscribers (also called listeners), which are connected to the monitors using **uvm_analysis_export** interfaces (Line 4, 5, 7 and 8). With the use of this TLM analysis interface, a call to the member function **write** in a monitor is actually executed on each connected subscriber, which then calls the corresponding write method implemented in the scoreboard (Line 33 and 34). A subscriber can also act as a predictor, which consumes the input stimulus that is sent to the DUT and calculates the expected result based on an algorithm or reference model, and passes this to the scoreboard.

```
class scoreboard : public uvm_scoreboard
{
 public:
  uvm_analysis_export<vip_trans> xmt_listener_imp;
  uvm_analysis_export<vip_trans> rcv_listener_imp;
  bool error;
  xmt_subscriber* xmt_listener;
  rcv_subscriber* rcv_listener;

  scoreboard( uvm_name name ) : uvm_scoreboard( name ),
    xmt_listener_imp("xmt_listener_imp"),
    rcv_listener_imp("rcv_listener_imp"),
    xmt_listener(0), rcv_listener(0) {}

  UVM_COMPONENT_UTILS(scoreboard);

  virtual void build_phase( uvm_phase& phase )
  {
    uvm_scoreboard::build_phase(phase);
    ...
    xmt_listener = xmt_subscriber::type_id::create("xmt_listener", this);
    assert(xmt_listener);
    rcv_listener = rcv_subscriber::type_id::create("rcv_listener", this);
    assert(rcv_listener);
  }

  virtual void connect_phase( uvm_phase& phase )
  {
    xmt_listener_imp.connect( xmt_listener->analysis_export );
    rcv_listener_imp.connect( rcv_listener->analysis_export );
  }

  virtual void write_xmt( const vip_trans& p ) { ... } // store reference information
  virtual void write_rcv( const vip_trans& p ) { ... } // compare received data with reference data
};
```



Listing 7: Scoreboard in UVM-SystemC

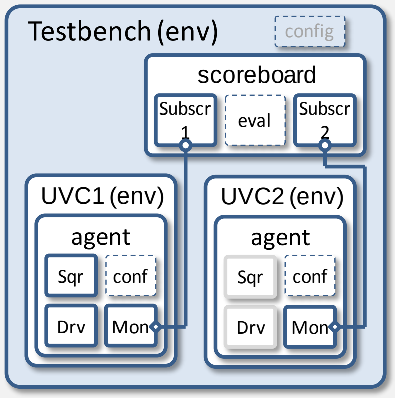
The actual evaluation functionality, which is the self-checking part in the scoreboard, is done by comparing the reference and received transactions. The implementation of the comparison depends on the type of application and therefore not presented in detail. In most cases, the transaction properties acting as golden reference are stored on a stack (e.g. FIFO or vector), in a write method called by the first subscriber (e.g. *write_xmt*). The comparison is done as soon as a transaction is received by the second subscriber, by calling another write method (e.g. *write_rcv*). The comparison result is stored in the public variable *error*, which is used in the test component to report the pass/fail result.

5B – The UVM-SystemC test bench

The test bench is defined as the complete verification environment which instantiates and configures the universal verification components (UVCs), scoreboard, and virtual sequencer if necessary. The UVCs are sub-environments in the test bench, which contain one or more agents.

Listing 8 shows the implementation of the test bench. It uses the base class **uvm_env** (Line 1). The UVCs and scoreboard are instantiated using the factory (Line 17-22). This facilitates component overriding from the test scenario, as discussed in the next sections. The configuration database is used to configure each agent in the UVC as being active or passive, by means of the global function **set_config_int** (Line 24 and 25).

```
 1  class testbench : public uvm_env
 2  {
 3   public:
 4    vip_uvc*    uvc1;
 5    vip_uvc*    uvc2;
 6    scoreboard* scoreboard1;
 7
 8    testbench( uvm_name name ) : uvm_env( name ), uvc1(0),
 9      uvc2(0), scoreboard1(0) {}
10
11    UVM_COMPONENT_UTILS(testbench);
12
13    virtual void build_phase( uvm_phase& phase )
14    {
15      uvm_env::build_phase(phase);
16
17      uvc1 = vip_uvc::type_id::create("uvc1", this);
18      assert(uvc1);
19      uvc2 = vip_uvc::type_id::create("uvc2", this);
20      assert(uvc2);
21      scoreboard1 = scoreboard::type_id::create("scoreboard1", this);
22      assert(scoreboard1);
23
24      set_config_int("uvc1.*", "is_active", UVM_ACTIVE);
25      set_config_int("uvc2.*", "is_active", UVM_PASSIVE);
26    }
27
28    virtual void connect_phase( uvm_phase& phase )
29    {
30      uvc1->agent->monitor->item_collected_port.connect(scoreboard1->xmt_listener_imp);
31      uvc2->agent->monitor->item_collected_port.connect(scoreboard1->rcv_listener_imp);
32    }
33  };
34
```



Listing 8: Test bench in UVM-SystemC

The test bench only makes the connections between the monitors of the UVCs and the listeners (subscribers) in the scoreboard (Line 37 and 38); the connections between the UVCs and the DUT are arranged within the driver and monitor of each UVC.

```
template <class REQ>
class new_driver : public vip_driver<REQ>
{
 public:
  new_driver( uvm_name name ) : vip_driver<REQ>( name ) {}
  UVM_COMPONENT_PARAM_UTILS(new_driver<REQ>);
  ...
};
```

Listing 9: New driver declaration to override existing driver component

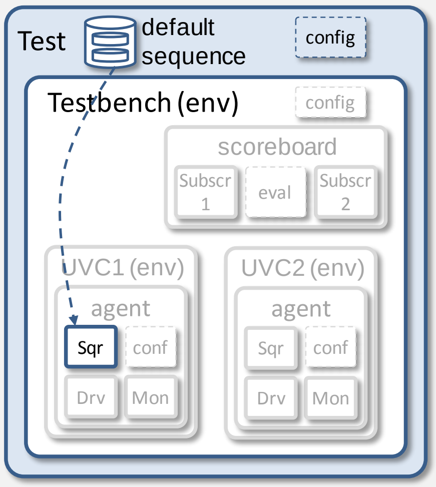
5C – Description of the test

Each UVM test is defined as a dedicated test class derived from class **uvm_test**, as shown in Listing 10 (Line 1). It instantiates the test bench (Line 16) and defines the default sequence which will be executed on the sequencer of one of the verification components (Line 19-21).

A new driver called *new_driver* is defined in Listing 9 (only partly shown), which will be used to override the original *vip_driver* instantiated in Listing 1. The factory member function **set_type_override_by_type** is used to override the original UVM driver in the agent by a new driver (Listing 10, line 23 and 24). The types of the original and new driver are obtained by calling the static member function **get_type**.

The result from the scoreboard checking is extracted in the **extract_phase** and updates the local data member *test_pass*. The actual pass/fail reporting is done in the callback **report_phase** using the available UVM macros for reporting information and errors.

```
1   class test : public uvm_test
2   {
3    public:
4     testbench* tb;
5     bool test_pass;
6
7     test( uvm_name name ) : uvm_test( name ),
8       tb(0), test_pass(true) {}
9
10    UVM_COMPONENT_UTILS(test);
11
12    virtual void build_phase( uvm_phase& phase )
13    {
14      uvm_test::build_phase(phase);
15
16      tb = testbench::type_id::create("tb",this);
17      assert(tb);
18
19      uvm_config_db<uvm_object_wrapper*>::set( this,
20        tb.uvc1.agent.sequencer.run_phase", "default_sequence",
21        vip_sequence<vip_trans>::type_id::get());
22
23      set_type_override_by_type( vip_driver<vip_trans>::get_type(),
24        new_driver<vip_trans>::get_type() );
25    }
26
27    virtual void run_phase( uvm_phase& phase )
28    {
```



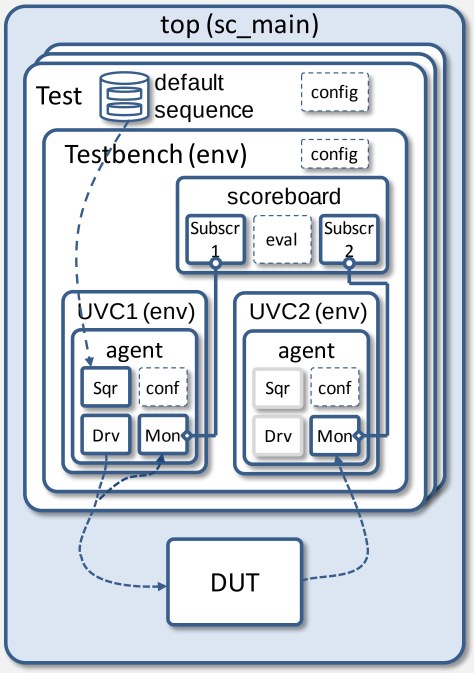
```
    UVM_INFO( get_name(), "** UVM TEST STARTED **", UVM_NONE );
  }

  virtual void extract_phase( uvm_phase& phase )
  {
    if ( tb->scoreboard1.error )
      test_pass = false;
  }

  virtual void report_phase( uvm_phase& phase )
  {
    if ( test_pass )
      UVM_INFO( get_name(), "** UVM TEST PASSED **", UVM_NONE );
      else
        UVM_ERROR( get_name(), "** UVM TEST FAILED **" );
  }
};
```

Listing 10: Test definition in UVM-SystemC

5D – The main program (Top-level)

The main program, also called top-level, uses the SystemC **sc_main** function and contains the DUT, the interfaces connected to the DUT, and the definition of the test, as shown in Listing 11. The interfaces are stored in the configuration database, so it can be used by the UVC drivers and monitors to connect to the DUT (Line 8-11).

```
 1  int sc_main(int, char*[])
 2  {
 3    dut* my_dut = new dut("my_dut");
 4
 5    vip_if* vif_uvc1 = new vip_if;
 6    vip_if* vif_uvc2 = new vip_if;
 7
 8    uvm_config_db<vip_if*>::set(0, "*.uvc1.*",
 9                                "vif", vif_uvc1);
10    uvm_config_db<vip_if*>::set(0, "*.uvc2.*",
11                                "vif", vif_uvc2);
12
13    my_dut->in(vif_uvc1->sig_a);
14    my_dut->out(vif_uvc2->sig_a);
15
16    run_test("test");
17
18    sc_start();
19
20    return 0;
21  }
```



Listing 11: The main program (top-level) in UVM-SystemC

The test to be executed is either defined by explicitly instantiating the test object (of type **uvm_test**) or implicitly by specifying the test as argument of the function **run_test** (Line 16). The latter method makes it possible to pass the test as string via the command line, available via the arguments of the function **sc_main**, and pass it to the **run_test** method.

The simulation is started using the SystemC function **sc_start**. It is recommended not to specify the simulation stop time, nor use the SystemC function **sc_stop**, as the end-of-test is automatically managed by the UVM-SystemC phasing mechanism.

6 – Summary and outlook

The Universal Verification Methodology standard offers the verification community good guiding principles to develop structured, modular, configurable and reusable verification environments. In order to introduce the same valuable concepts to the system-level design community, the UVM library is made available in SystemC/C++, offering them a powerful asset to further advance in system verification and hardware/software co-verification. Furthermore, it facilitates reuse of verification components, which are developed internally or offered by 3rd parties, between concept and architecture design and the IC implementation phase. As UVM-SystemC is compliant with the UVM standard and is using identical semantics and language constructs, it gives the user community the same “look & feel”, with the aim to bring verification and system-level design best practices closer together.

As a next step, the UVM-SystemC library will be extended with a UVM standard compatible register abstraction layer and callback mechanism. Furthermore, integration of constrained randomization capabilities using the SystemC Verification or CRAVE library is being developed, as well as the introduction of coverage groups.

In order to seek for standardization and industry adoption of UVM in SystemC, the objective is to contribute the initial language reference definition and associated proof-of-concept implementation of UVM-SystemC to Accellera Systems Initiative. It is therefore likely that the presented constructs are subject to change as part of the envisioned standardization process.

Ultimately, this initiative brings us one step closer in realizing the vision where the Universal Verification Methodology can be truly seen as language-independent methodology standard. By making UVM available in recognized languages such as SystemVerilog and SystemC/C++, new applications and use cases besides system verification, like hardware-in-the-loop simulation and rapid control prototyping will emerge. The Universal Verification Methodology becomes universal, at last.

7 – Acknowledgements

The development of the UVM-SystemC methodology and library has been supported by the European Commission as part of the Seventh Framework Programme (FP7) for Research and Technological Development in the project 'VERIFICATION FOR HETEROGENOUS RELIABLE DESIGN AND INTEGRATION' (VERDI) [20].
The research leading to these results has received funding from the European Commission under grand agreement No 287562.

8 – References

[1] IEEE Standards Association (IEEE-SA), IEEE Std 1666-2011 - IEEE Standard for Standard SystemC Language Reference Manual, http://standards.ieee.org/findstds/standard/1666-2011.html, accessed November 2013.

[2] Wolfgang Ecker, Volkan Esen, Robert Schwencker, Thomas Steininger, Michael Velten, TLM+ modeling of embedded HW/SW systems, Design, Automation & Test in Europe Conference & Exhibition (DATE), March 2010.

[3] Accellera Systems Initiative, Universal Verification Methodology standard, http://www.accellera.org/downloads/standards/uvm, accessed November 2013.

[4] IEEE Standards Association (IEEE-SA), IEEE Std 1800-2012 - IEEE Standard for SystemVerilog--Unified Hardware Design, Specification, and Verification Language, http://standards.ieee.org/findstds/standard/1800-2012.html, accessed November 2013.

[5] Arkadiusz Koczor, Wojciech Sakowski, SystemC library supporting OVM compliant verification methodology, IP Embedded System Conference and Exhibition (IP-SoC), December 2011.

[6] Marcio F. S. Oliveira, Christoph Kuznik, Wolfgang Mueller, Wolfgang Ecker, Volkan Esen, A SystemC Library for Advanced TLM Verification, Design and Verification Conference & Exhibition (DVCon), February 2012.

[7] Cadence Extends the Open Verification Methodology Beyond SystemVerilog to Include SystemC and e Language Support, http://www.cadence.com/community/blogs/fv/archive/2009/02/27/ovm-multi-language-libraries-a-closer-look.aspx, February 2009, accessed November 2013.

[8] Mentor Graphics, Advanced Verification Methodology Cookbook version 3.0, May 2007.

[9] Synopsys, Verification Methodology Manual User Guide, July 2011, http://vmmcentral.org, accessed November 2013.

[10] Ambar Sarkar, Verification in the trenches: A SystemC implementation of VMM1.2, http://www.vmmcentral.org/vmartialarts/2010/12/verification-in-the-trenches-a-systemc-implementation-of-vmm1-2/, accessed November 2013.

[11] Gabi Leshem, Vitaly Yankelevich, Bryan Sniderman, UVM-ML Whitepaper - A Modular Approach for Integrating Verification Frameworks, March 2013.

[12] IEEE Standards Association (IEEE-SA), IEEE Std 1647-2011 - IEEE Standard for the Functional Verification Language e, http://standards.ieee.org/findstds/standard/1647-2011.html, accessed November 2013.

[13] John Aynsley, SystemVerilog Meets C++: Re-use of Existing C/C++ Models Just Got Easier, Design and Verification Conference & Exhibition (DVCon), February 2010.

[14] Mentor Graphics, UVM Connect - a SystemC TLM interface for UVM/OVM - v2.2, http://forums.accellera.org/files/file/92-uvm-connect-a-systemc-tlm-interface-for-uvmovm-v22, accessed November 2013.

[15] Asif Jafri, Nasib Naser, Interoperable testbenches using VMM TLM, Synopsys User Group, April 2010.

[16] Karsten Einwich, Thilo Vörtler, Thomas Arndt, New technological opportunities coming along with SystemC / SystemC AMS for AMS IP Handling and Simulation, European Nanoelectronics Design Technology Conference (DTC), June 2012.

[17] Alan Shalloway, James R. Trott, Design Patterns Explained: A New Perspective on Object-Oriented Design, 2nd edition, Pearson Education, October 2004.

[18] Accellera Systems Initiative, SystemC Verification Library (SVC), http://www.accellera.org/activities/committees/systemc-verification/, accessed November 2013.

[19] Finn Haedicke, Hoang M. Le, Daniel Große, Rolf Drechsler, CRAVE: An Advanced Constrained Random Verification Environment for SystemC, International Symposium on System on Chip (SoC), October 2012.

[20] The VERDI project, part of Seventh Framework Programme (FP7), http://verdi-fp7.eu, accessed November 2013.