UVM-SystemC in COSIDE®

Stephan Schulz (FhG IIS/EAS),
Martin Barnasconi (NXP)
UVM what is it?

• Universal Verification Methodology to create **modular, scalable, configurable and reusable testbenches** based on verification components with standardized interfaces.

• **Class library** which provides a set of built-in features dedicated to verification, e.g., phasing, component overriding (factory), configuration, comparing, scoreboard, reporting, etc.

• Environment supporting migration from **directed testing** towards **Coverage Driven Verification (CDV)** which consists of automated stimulus generation, independent result checking and coverage collection.
UVM what is it not...

• Infrastructure offering tests or scenario’s *out-of-the-box*: all behaviour has to be *implemented by user*

• Coverage-based verification templates: application is responsible for coverage and randomization definition; UVM only offers the hooks and technology

• Verification management of requirements, test items or scenario’s

• Test item execution and regression – automation via e.g. the command line interface or “regression cockpit” is a shell around UVM
Outline

• Part A - Introduction
• Part B – UVM Elements and Applications
• Part C – Further steps & Outlook
Main concepts of UVM (1)

• Clear separation of test stimuli (sequences) and test bench
  – Sequences are treated as ‘transient objects’ and thus independent from the test bench construction and composition
  – In this way, sequences can be developed and reused independently

• Introducing test bench abstraction levels
  – Communication between test bench components based on transaction level modeling (TLM)
  – Register abstraction layer (RAL) using register model, adapters, and predictors

• Reusable verification components based on standardized interfaces and responsibilities
  – Universal Verification Components (UVCs) offer sequencer, driver and monitor functionality with clearly defined (TLM) interfaces
Main concepts of UVM (2)

• Non-intrusive test bench configuration and customization
  – Hierarchy independent configuration and resource database to store and retrieve properties everywhere in the environment
  – Factory design pattern introduced to easily replace UVM components or objects for specific tests
  – User-defined callbacks to extend or customize UVC functionality

• Well defined execution and synchronization process
  – Simulation based on phasing concept: build, connect, run, extract, check and report. UVM offers additional refined run-time phases
  – Objection and event mechanism to manage phase transitions

• Independent result checking
  – Coverage collection, signal monitoring and independent result checking in scoreboard are running autonomously
Verification stack:
tools, language and methodology

- Verification management
- Universal Verification Methodology
- UVM (-SC / -AMS) Class library
- SystemC(-AMS) Language
- SystemC(-AMS) compliant simulator

Addition tool layer like “verification cockpit”
(e.g. vManager, vPlan)

UVM-SystemC scope

UVM technology elements:
- Methodology = what
- Class library = how

Language and modeling technology elements:
Tool / simulator

© Accellera Systems Initiative 2015
UVM Layered Architecture

- The top-level (e.g. `sc_main`) contains the test(s), the DUT and its interfaces.
- The DUT interfaces are stored in a configuration database, so it can be used by the UVCs to connect to the DUT.
- The test bench contains the UVCs, register model, adapter, scoreboard and (virtual) sequencer to execute the stimuli and check the result.
- The test to be executed is either defined by the test class instantiation or by the member function `run_test`.

© Accellera Systems Initiative
UVM layered architecture

Spec

Test

Test cases

Scenario

Sequences

Verification environment (test bench)

Functional

Verification component

Sequencer

Scoreboard

Command

Driver

Monitor

Monitor

Signal

Device under test

© Accellera Systems Initiative
Why UVM in SystemC?

• Elevate verification beyond block-level towards system-level
  – System verification and Software-driven verification are executed by teams not familiar with SystemVerilog and its simulation environment
  – Trend: Tests coded in C or C++. System and SW engineers use an (open source) tool-suite for embedded system design and SW dev.

• Structured ESL verification environment
  – The verification environment to develop Virtual Platforms and Virtual Prototypes is currently ad-hoc and not well architected
  – Beneficial if the first system-level verification environment is UVM compliant and can be reused later by the IC verification team

• Extendable, fully open source, and future proof
  – Based on Accellera’s Open Source SystemC simulator
  – As SystemC is C++, a rich set of C++ libraries can be integrated easily
Why UVM in SystemC?

- Support analogue DUTs with SystemC AMS
- Reuse tests and test benches across verification (simulation) and validation (HW-prototyping) platforms
  - requires portable language like C++ to run tests on HW prototypes, measurement equipment, ...
  - Enables Hardware-in-the-Loop simulation and Rapid Control Prototyping
UVM in SystemC versus UV in SystemVerilog

• UVM-SystemC follows the UVM 1.1 standard where possible and/or applicable
  – Equivalent UVM base classes and member functions implemented in SystemC/C++
  – Use of existing SystemC functionality where applicable
    • TLM interfaces and communication
    • Reporting mechanism
  – Only a limited set of UVM macros is implemented
    • usage of some UVM macros is not encouraged and thus not introduced

• UVM-SystemC does not cover the ‘native’ verification features of SystemVerilog, but considers them as (SCV) extensions
  – Constrained randomization
  – Coverage groups (not part of SCV yet)
Outline

• Part B – UVM Elements and Applications
  – Components and Classes
  – Register Model
  – Abstraction re-use
  – Generator
  – Visualization
UVM Testbench setup

• Required minimum
  – Test
  – Testbench
  – Agent
  – Sequencer
  – Driver
  – Monitor
  – DUT
  – Scoreboard

• Optional
  – More Agents
  – Virtual Sequencers
  – Register Model
  – Extensive configuration on every element
UVM agent

- Component responsible to drive and monitor the DUT
- Typically contains three components
  - Sequencer
  - Driver
  - Monitor
- Could contain analysis functionality for basic coverage and checking

© Accellera Systems Initiative

NOTE: UVM-SystemC API and LRM under review – subject to change
UVM agent

- Possible configurations
  - Active agent: sequencer and driver are enabled
  - Passive agent: only monitors signals (sequencer and driver are disabled)
  - Master or slave configuration
- Base class: uvm_agent
UVM sequencer

- The sequencer controls and delivers transaction data items upon request of the driver*
- This allows to react to the current state of the DUT for every data item generated
- The UVM standard describes an interface between sequencer and driver that follows TLM (1.0) communication
- The sequencer serves as an arbiter for controlling transactions from multiple stimulus generators
- Base class: uvm_sequencer

* Alternatively, there is a UVM push sequencer (class uvm_push_sequencer) which pushes the sequence items to the driver, but this is not yet available in UVM-SystemC

© Accellera Systems Initiative

NOTE: UVM-SystemC API and LRM under review – subject to change
UVM driver

- The driver is responsible to create the physical signals to drive the DUT.
- For this, the driver repeatedly requests transactions, encapsulated in a sequence, via the sequencer, and translates these to one or more physical signal(s).
- Connection between the driver and the DUT is established by using a dedicated channel, which is made available via the configuration mechanism.
- Base class: `uvm_driver`
UVM monitor

• The monitor is a passive element that ‘only’ captures the DUT signals
• It extracts signal information from the interface and translates this information to abstract transactions
• It will distribute this transaction to all connected elements for e.g. coverage collection and checking
• Connection between the monitor and the DUT is established by using a dedicated channel, which is made available via the configuration mechanism
• Base class: uvm_monitor
UVM verification component (UVC)

- A reusable verification component (UVC) is a (sub-) environment which consists of one or more agents.
- The verification component or agents may set or get configuration parameters.
- An independent sequence, which contains the actual transaction data, is processed by the driver via a sequencer.
- Each verification component is connected to the DUT using a dedicated interface.
- Base class: uvm_env. 

NOTE: UVM-SystemC API and LRM under review – subject to change.
UVM sequences

- Sequences are part of the test scenario and define streams of *transactions*.
- The properties (or attributes) of a transaction are captured in a *sequence item*.
- Sequences are *not* part of the testbench hierarchy, but are mapped onto one or more sequences.
- Sequences can be layered, hierarchical or virtual, and may contain multiple sequences or sequence items.
- Sequences and transactions can be configured via the factory.
UVM virtual sequence

- A virtual sequence encapsulates one or more sequences, which are executed on the sub-sequencers in each UVC agent, which are all connected to the parent virtual sequencer.
- A virtual sequence can be configured as default sequence in a test, to facilitate automatic execution on a virtual sequencer or a sequencer which belongs to a UVC agent.
- Base class: `uvm_sequence` (same as ‘normal’ sequences)
A virtual sequencer contains references to its subsequencers such as UVC sequencers or other virtual sequencers.

Virtual sequencers process virtual sequences which encapsulate sequences for multiple verification components.

Virtual sequencers do not execute transactions on themselves but ‘offload’ this to its subsequencers.

Base class: `uvm_sequencer` (same as ‘normal’ sequencers)
UVM scoreboard

- The scoreboard performs *end-to-end* checking by comparing expected and processed transactions.
- These transactions are retrieved by dedicated *subscribers* or *listeners*, which implement the *write* method of the analysis ports of each monitor, to which these subscribers are connected.
- A scoreboard may contain a predictor, which acts as reference or golden model. Alternatively, the scoreboard may contain an algorithm to calculate the expected transaction.
- Base class: *uvm_scoreboard*
Each UVM test is defined as a dedicated test class, which instantiates the testbench and defines the test sequence(s)

Reuse of tests and topologies is possible by deriving tests from a test base class

The configuration and factory concept can be used to configure or override UVM components, sequences or sequence items

Tests can be selected (passed) as command line option*

Base class: `uvm_test`

* Not yet available in UVM-SystemC
UVM testbench

- A testbench is defined as the complete environment which *instantiates* and *configures* the UVCs, scoreboard, and virtual sequencer if available.
- The UVCs are sub-environments in a testbench.
- The testbench only makes the connections between the scoreboard and virtual sequencer to each UVC; the connection between UVCs and the DUT is arranged within the UVCs.

© Accellera Systems Initiative

NOTE: UVM-SystemC API and LRM under review – subject to change
UVM configuration mechanism

- Central resource database to store and retrieve any type specific information of UVM and non-UVM objects at any place in the verification environment.
- Configuration is facilitated during the build process and/or run time.
- Information can be accessed by name (string) or arbitrary type.
- Scope (context) of accessibility of information can be defined by the application.
- Easy access to resource database via the configuration mechanism `uvm_config_db`.
- Base class: `uvm_resource`.

© Accellera Systems Initiative

NOTE: UVM-SystemC API and LRM under review – subject to change.
Top, Tests and Testbench

- The top-level (e.g. `sc_main`) contains the test(s) and the DUT
- The interface to which the DUT is connected is stored in the configuration database, so it can be used by the UVCs to connect to the DUT
- The test to be executed is either defined by the test class instantiation or by the argument of the member function `run_test`

© Accellera Systems Initiative

NOTE: UVM-SystemC API and LRM under review – subject to change
## Work-in-Progress: Register Abstraction Layer

<table>
<thead>
<tr>
<th>Register Abstraction Layer</th>
<th>Status</th>
</tr>
</thead>
<tbody>
<tr>
<td>Register model containing registers, fields, blocks, etc.</td>
<td>testing</td>
</tr>
<tr>
<td>Register callbacks</td>
<td>testing</td>
</tr>
<tr>
<td>Register adapter, predictor, sequences and transaction items</td>
<td>testing</td>
</tr>
<tr>
<td>Register front-door access</td>
<td>testing</td>
</tr>
<tr>
<td>Build-in register test sequencers</td>
<td>development</td>
</tr>
<tr>
<td>Memory and memory allocation manager</td>
<td>development</td>
</tr>
<tr>
<td>Virtual registers and fields</td>
<td>development</td>
</tr>
<tr>
<td>Register back-door access (hdl_path)</td>
<td>study</td>
</tr>
<tr>
<td>Randomization of registers</td>
<td>study</td>
</tr>
</tbody>
</table>
Application Examples
UVM-SystemC Generator

• Generator is based on *easier uvm code generator for SystemVerilog* from Doulos (http://www.doulos.com/knowhow/sysverilog/uvm/easier_uvm_generator/)

• Generator uses template files as input, which are similar to the Doulos generator

• Generates complete running UVM-SystemC environment
UVM-SystemC Generator

• Generated UVM objects and files:
  – UVM_Agent
  – UVM_Scoreboard
  – UVM_Driver
  – UVM_Monitor
  – UVM_Sequencer
  – UVM_Environment
  – UVM_Config
  – UVM_Subscriber
  – UVM_Test
  – Makefile to compile the generated UVM project
  – Instantiation and DUT connection
UVM-SystemC Generator

- Input file for generating a complete agent
  - Transaction items
  - Interface ports

- General Config File

```
#DUT directory
dut_source_path = mydut
#Additional includes
inc_path = include
#DUT toplevel name
dut_top = mydut
#Pin connection file
dut_pfile = pinlist
```

- DUT connection to agent interfaces (DUT port <-> agent port))

```
!clkndata_if
clk clk
reset_n reset_n
rw_master1 rw_master
scl1 scl
sda1 sda
!agent2_if
...
```

#agent name
agent_name = clkndata

#transaction item
trans_item = data_tx

#transaction variables
trans_var = int data

#interface ports
if_port = sc_core::sc_signal<bool> clk
if_port = sc_core::sc_signal<bool> reset_n
if_port = sc_core::sc_signal<bool> scl
if_port = sc_core::sc_signal<bool> sda
if_port = sc_core::sc_signal<bool> rw_master
if_clock = clk
if_reset = reset_n

#agent mode
agent_is_active = UVM_ACTIVE
Hands-on example (Generator)

- DUT is a minimalistic ALU
- Tests checks basic arithmetic with static operands
- Plain SystemC Testbench as reference
- Re-implementation with UVM-SystemC
Hands-on example (Visualizer)
Benefits

• Avoidance of boilerplate code copy & paste disasters
• Manual input amount as in hand-crafted testbench
  – DUT setup
  – Test sequence
  – Driver implementation for DUT driving
  – Monitor implementation for DUT interpreting
• UVM conformity
• Re-Usage because of modularity more likely
Re-use across abstraction levels (1)

- Design of a complex system within a SystemC environment
  - One-time verification setup with UVM-SystemC
  - Behavioral model for concept phase
  - Detailed model for further implementation require additional tests
Re-use across abstraction levels (2)

- Continued use of previous verification setup by running the verification environment as a real-time model on a HiL platform
  - Exchange of UVM driver verification components suitable for the board
  - Additional tests specific to new model details
Re-use across abstraction levels (3)

- Continued use of previous verification setup by running the verification environment as a real-time model on lab-test equipment
  - Exchange of UVM driver verification components necessary
  - Re-use of all tests possible
Re-use across abstraction levels (4)
Outline

• Part C – Further steps & Outlook
  – Standardization in Accellera
  – Next steps
  – Summary and outlook
Standardization in Accellera

- Standardization in SystemC Verification WG ongoing
  - UVM-SystemC Language Reference Manual (LRM) completed
  - Improving the UVM-SystemC Proof-of-Concept (PoC) implementation
  - Creation of a UVM-SystemC regression suite started
- Draft release of UVM-SystemC planned for CW48/49 2015
  - Both LRM and PoC available under the Apache 2.0 license

NOTE: UVM-SystemC API and LRM under review – subject to change
Next steps in VWG

• Main focus this year:
  – Further mature and test the proof-of-concept implementation
  – Extend the regression suite with unit tests and more complex (application) examples

• Next year...
  – Finalize upgrade to UVM 1.2 (upgrade to UVM 1.2 already started)
  – Add constrained randomization capabilities (e.g. SCV, CRAVE)
  – Introduction of assertions and functional coverage features

• ...and beyond: IEEE standardization
  – Alignment with IEEE P1800.2 (UVM-SystemVerilog) necessary
Summary and outlook

• Good progress with UVM-SystemC standardization in Accellera
• UVM foundation elements are implemented
• Register Abstraction Layer currently under development
• Review of Language Reference Manual finished and Proof-of-concept implementation ongoing
• Draft release of UVM-SystemC planned for CW48/49 2015
  – Updates of LRM and PoC implementation afterwards