UVM goes Universal -
Introducing UVM in SystemC

Stephan Schulz (FhG IIS/EAS), Thilo Vörtler (FhG IIS/EAS), Martin Barnasconi (NXP)

Fraunhofer & NXP Logo goes here
Outline

• A bit of history...
• Why UVM in SystemC?
• Main concepts of UVM
• Advantages of UVM-SystemC
• Work-in-Progress: Register Abstraction Layer
• Register Model examples
• Standardization in Accellera
• Next steps
• Summary and outlook
• UVM-SystemC tutorial at DVCon Europe
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, scoreboardng, 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 (classes)
- Verification management of requirements, test items or scenario’s is outside the scope of UVM
- Test item execution and regression – automation via e.g. the command line interface or “regression cockpit” is a shell around UVM
A bit of history...

• In the pre-UVM era, various EDA vendors offered a verification methodology in SystemC
  – OVM-SC (Cadence), AVM-SC (Mentor), VMM-SC (Synopsys)
• Unfortunately, consolidation towards UVM focused on a SystemVerilog standardization and implementation only
• Non-standard methods and libraries exist to bridge the UVM and SystemC world
  – Cadence’s UVM Multi Language library: offers a ‘minimalistic’ UVM-SC
  – Mentor’s UVM-Connect: Mainly TLM communication and configuration
• In 2011, a European consortium started building a UVM standard compliant version based on SystemC / C++
  – Initiators: NXP, Infineon, Fraunhofer IIS/EAS, Magillem, Continental, and UPMC
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?

• 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
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-SC scope**

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

Language and modeling technology elements:
- Tool / simulator
UVM-SC versus UVM-SV

- 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)
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
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`
UVM layered architecture

- Spec
  - Test cases
    - Sequences
      - Sequencer
      - Driver
      - Monitor
    - Scoreboard
      - Monitor
      - Monitor
  - Device under test
    - Functional coverage
      - Verification environment (test bench)
Advantages of UVM-SystemC

• UVM-SystemC library features
  – UVM components based on SystemC modules
  – TLM communication API based on SystemC
  – Phases of elaboration and simulation aligned with SystemC
  – Packing / Unpacking using stream operators
  – Template classes to assign RES/RSP types
  – Standard C++ container classes for data storage and retrieval
  – Other C++ benefits (exception handling, constness, multiple inheritance, etc.)
UVM components are SystemC modules

- The UVM component class (**uvm_component**) is derived from the SystemC module class (**sc_module**)
  - It inherits the execution semantics and all features from SystemC
  - Parent-child relations automatically managed by **uvm_component_name** (alias of **sc_module_name**); no need to pass ugly *this*-pointers
  - Enables creation of spawned SystemC processes and introduce concurrency (**SC_FORK**, **SC_JOIN**); beneficial to launch runtime phases
  - No need for SV-like “virtual” interfaces; regular SystemC channels (derived from **sc_signal**) between UVC and DUT can be applied

```cpp
namespace uvm {
  class uvm_component : public sc_core::sc_module,
                        public uvm_report_object {
    { ... };
  } // namespace uvm

  class my_uvc : public uvm_env {
    public:
      my_uvc( uvm_component_name name ) : uvm_env( name )
    { }
    ... 
  }
}
```

NOTE: UVM-SystemC API under review – subject to change
SystemC TLM communication (1)

• TLM-1 put/get/peek interface
  – put/get/peek directly mapped on SystemC methods
  – UVM methods get_next_item and try_next_item mapped on SystemC
  – TLM-1 primarily used for sequencer-driver communication

• TLM-1 analysis interface
  – UVM analysis port, export and imp using SystemC tlm_analysis_if
  – Used for monitor-subscriber (scoreboard) communication
  – UVM method connect mapped on SystemC bind
SystemC TLM communication (2)

• As the UVM TLM2 definitions are inconsistent with the SystemC TLM-2.0 standard, these are not implemented in UVM-SystemC

• Furthermore, UVM only defines TLM2-like transport interfaces, and does not support the Direct Memory Interface (DMI) nor debug interface

• Therefore, a user is recommended to directly use the SystemC TLM-2.0 interface classes in UVM-SystemC

• Hopefully, the UVM SystemVerilog Standardization Working Group in IEEE (P1800.2) is willing to resolve this inconsistency and align with SystemC (IEEE Std 1666-2011)
Phases of elaboration and simulation

- UVM-SystemC phases made consistent with SystemC phases
- UVM-SystemC supports the 9 common phases and the (optional) refined runtime phases
- Objection mechanism supported to manage phase transitions
- Multiple domains can be created to facilitate execution of different concurrent runtime phase schedules
(Un)packing using stream operators

- Thanks to C++, stream operators (<<, >>) can be overloaded to enable elegant type-specific packing and unpacking.
- Similar operator overloading technique also applied for transaction comparison (using equality operator ==)

```cpp
class packet : public uvm_sequence_item
{
    public:
        int a, b;

    UVM_OBJECT_UTILS(packet);

    packet( uvm_object_name name = "packet" )
        : uvm_sequence_item(name), a(0), b(0) {}

    virtual void do_pack( uvm_packer& p ) const
    {
        p.pack_field_int(a, 64);
        p.pack_field_int(b, 64);
    }

    virtual void do_unpack( uvm_packer& p )
    {
        a = p.unpack_field_int(64);
        b = p.unpack_field_int(64);
    }
};
```

```cpp
class packet : public uvm_sequence_item
{
    public:
        int a, b;

    UVM_OBJECT_UTILS(packet);

    packet( uvm_object_name name = "packet" )
        : uvm_sequence_item(name), a(0), b(0) {}

    virtual void do_pack( uvm_packer& p ) const
    {
        p << a << b;
    }

    virtual void do_unpack( uvm_packer& p )
    {
        p >> a >> b;
    }
};
```

Disadvantage: type-specific methods
Elegant packing using stream operators

NOTE: UVM-SystemC API under review – subject to change
C++ Template classes

• Template classes enable elegant way to deal with special types such as REQ/RSP

• UVM-SystemC supports template classes using macros
  UVM_COMPONENT_UTILS or UVM_COMPONENT_PARAM_UTILS (no difference)

• More advanced template techniques using explicit specialization or partial specialization are possible

```cpp
template <typename REQ>
class vip_driver : public uvm_driver<REQ>
{
    public:
        vip_if* vif;

    vip_driver( uvm_component_name name )
    : uvm_driver<REQ>(name), vif(NULL) {}

    UVM_COMPONENT_PARAM_UTILS(vip_driver<REQ>);

    void build_phase( uvm_phase& phase )
    {
        uvm_driver<REQ>::build_phase(phase);

        if (!uvm_config_db<vip_if*>::get(this, "*", "vif", vif))
            UVM_FATAL(this->get_name(),
                "Interface not defined! Simulation aborted!");
    }

    void run_phase( uvm_phase& phase )
    {
        REQ req;

        while(true) // execute all sequences
            {
                this->seq_item_port->get_next_item(req);
                drive_transfer(req);
                rsp.set_id_info(req);
                this->seq_item_port->item_done();
            }

    void drive_transfer( const REQ& p )
    {
        vif->sig_data.write(p.data);
        ...
    }
};
```

NOTE: UVM-SystemC API under review – subject to change
Standard C++ container classes

- Standard C++ containers can be used for efficient data storage using push/pop mechanisms and retrieval using iterators and operators.
- Examples: dynamic arrays (std::vector), queues (std::queue), stacks (std::stack), heaps (std::priority_queue), linked lists (std::list), trees (std::set), associative arrays (std::map).
- Therefore UVM-SystemC will not define `uvm_queue` nor `uvm_pool`.

```cpp
namespace uvm {
    class uvm_object : public uvm_void {
        public:
            ...
        // Group: Packing
            int pack( std::vector<bool>& bitstream, uvm_packer* packer = NULL );
            int pack_bytes( std::vector<unsigned char>& bytestream, uvm_packer* packer = NULL );
            int pack_ints( std::vector<unsigned int>& intstream, uvm_packer* packer = NULL );
            ...  
    } // namespace uvm
```

NOTE: UVM-SystemC API under review – subject to change.
Other benefits

• **Exception handling:**
The standard C++ exception handler mechanism is beneficial to catch serious runtime errors (which are not explicitly managed or found using `UVM_FATAL`) and enables a graceful exit of the simulation

• **Constness:**
Ability to specify explicitly that a variable, function argument, method or class/object state cannot be altered

• **Multiple inheritance:**
Ability to derive a new class from two ‘origins’ or base classes.

• ...and much more C++ features...
Re-use across languages & simulators
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)
UVM-SystemC Generator

• Generator is based on easier uvm code generator for SystemVerilog from Doulos
  – 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

```
#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
```

- 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
...
```
Standardization in Accellera

• Growing industry interest for UVM in SystemC
• 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 end 2015
  – Both LRM and PoC made available under the Apache 2.0 license
  – Exact timing dependents on progress (and issues we might find)
Next steps

• 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
  – Multi-language verification usage (UVM-SystemVerilog ↔ UVM-SystemC)

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

• Good progress with UVM-SystemC standardization in Accellera
  – UVM-SystemC foundation elements are implemented
  – Register Abstraction Layer currently under development
  – First draft release of UVM-SystemC planned for end 2015

• Next steps
  – Make UVM-SystemC fully compliant with UVM 1.2
  – Introduce new features: e.g. randomization, functional coverage

• How you can contribute
  – Join Accellera and participate in this standardization initiative
  – Development of unit tests, examples and applications
Questions