UVM-SystemC Applications in the real world

Stephan Schulz, Thilo Vörtler, Karsten Einwich (Fraunhofer IIS/EAS)
Martin Barnasconi (NXP)
Outline

• Introduction and Motivation
  – Universal Verification Methodology (UVM) ... what is it?
  – Why UVM in SystemC/C++/SystemC-AMS?

• UVM-SystemC overview
  – UVM foundation elements
  – UVM test bench and test creation
  – Randomization and coverage

• Standardization within Accellera

• Applications and use cases of UVM-SystemC

• Summary and outlook
Outline

• Introduction and Motivation
  – Universal Verification Methodology (UVM) ... what is it?
  – Why UVM in SystemC/C++/SystemC-AMS?

• UVM-SystemC overview
  – UVM foundation elements
  – UVM test bench and test creation
  – Randomization and coverage

• Standardization within Accellera

• Applications and use cases of UVM-SystemC

• Summary and outlook
Introduction: UVM - what is it?

- Universal Verification Methodology facilitates the creation of **modular**, scalable, configurable and **reusable** test benches
  - Based on verification components with standardized interfaces
- **Class library** which provides a set of built-in features dedicated to simulation-based verification
  - **Utilities** for phasing, component overriding (factory), configuration, comparing, scoreboard, reporting, etc.
- Environment supporting migration from directed testing towards **Coverage Driven Verification (CDV)**
  - Introducing automated stimulus generation, independent result checking and coverage collection
Motivation

- No structured nor unified verification methodology available for **ESL design**
  - UVM (in SystemVerilog) primarily targeting block/IP level (RTL) verification, not system-level
- Porting UVM to SystemC/C++ enables
  - creation of more advanced **system-level test benches**
  - **reuse** of verification components between **system-level and block-level** verification

<table>
<thead>
<tr>
<th>Verification &amp; Validation Methodology</th>
</tr>
</thead>
<tbody>
<tr>
<td>UVM-SystemC*</td>
</tr>
<tr>
<td>SystemC-AMS</td>
</tr>
<tr>
<td>SCV</td>
</tr>
<tr>
<td>SystemC-AMS</td>
</tr>
<tr>
<td>TLM</td>
</tr>
<tr>
<td>C++</td>
</tr>
</tbody>
</table>

*UVM-SystemC = UVM implemented in SystemC/C++*
Why UVM in SystemC/C++ and SystemC-AMS?

• Strong need for a system-level verification methodology for embedded systems which include HW/SW and AMS functions
  – SystemC is the recognized standard for system-level design, and needs to be extended with advanced verification concepts
  – SystemC AMS available to cover the AMS verification needs

• Reuse tests and test benches across verification (simulation) and validation (HW-prototyping) platforms
  – This requires a portable language like C++ to run tests on HW prototypes and even measurement equipment
  – Enabling Hardware-in-the-Loop simulation or Rapid Control Prototyping
Why UVM in SystemC/C++ and SystemC-AMS?

- **Benefit from proven standards** and reference implementations
  - Leverage from existing methodology standards and reference implementations, aligned with best practices in verification
Outline

• Introduction and Motivation
  – Universal Verification Methodology (UVM) ... what is it?
  – Why UVM in SystemC/C++/SystemC-AMS?

• UVM-SystemC overview
  – UVM foundation elements
  – UVM test bench and test creation
  – Randomization and coverage

• Standardization within Accellera

• Applications and use cases of UVM-SystemC

• Summary and outlook
### UVM-SystemC overview

<table>
<thead>
<tr>
<th>UVM-SystemC functionality</th>
<th>Status</th>
</tr>
</thead>
<tbody>
<tr>
<td>Test bench creation with component classes: agent, sequencer, driver, monitor, scoreboard, etc.</td>
<td>✔</td>
</tr>
<tr>
<td>Test creation with test, (virtual) sequences, etc.</td>
<td>✔</td>
</tr>
<tr>
<td>Configuration and factory mechanism</td>
<td>✔</td>
</tr>
<tr>
<td>Phasing and objections</td>
<td>✔</td>
</tr>
<tr>
<td>Policies to print, compare, pack, unpack, etc.</td>
<td>✔</td>
</tr>
<tr>
<td>Messaging and reporting</td>
<td>✔</td>
</tr>
<tr>
<td>Register abstraction layer and callbacks</td>
<td>development</td>
</tr>
<tr>
<td>Coverage groups</td>
<td>development</td>
</tr>
<tr>
<td>Constrained randomization</td>
<td>SCV or CRAVE</td>
</tr>
</tbody>
</table>
UVM layered architecture

Spec

Test cases

Test

Scenario

Sequences

Verification environment (test bench)

Verification component

Functional

Sequencer

Scoreboard

Functional coverage

Command

Driver

Monitor

Driver

Monitor

Signal

Device under test

Device under test
UVM-SystemC phasing

- UVM phases are **mapped on the SystemC phases**
- UVM-SystemC supports the 9 common phases and the (optional) refined runtime phases
- **Completion** of a runtime phase **happens** as soon as there are **no objections** (anymore) to proceed to the next phase
Component responsible for **driving and monitoring** the DUT:
- Typically contains three components
  - Sequencer
  - Driver
  - Monitor
- Can contain analysis functionality for basic coverage and checking

Possible configurations
- **Active** agent: sequencer and driver are enabled
- **Passive** agent: only monitors signals (sequencer and driver are disabled)

C++ base class: **uvm_agent**
class dut_agent1: public uvm::uvm_agent{

public:
    dut_driver1<dut_trans1>* driver;
    dut_monitor1<dut_trans1>* monitor;
    dut_sequencer1<dut_trans1>* sequencer;

    UVM_COMPONENT_UTILS(dut_agent1);

dut_agent1::dut_agent1(uvm::uvm_name name) : uvm_agent(name), driver(0), monitor(0), sequencer(0) {}

void dut_agent1::build_phase(uvm::uvm_phase& phase) {
    uvm_agent::build_phase(phase);

    if (get_is_active() == uvm::UVM_ACTIVE) {
        sequencer = dut_sequencer1<dut_trans1>::type_id::create("sequencer", this);
        assert(sequencer);

        driver = dut_driver1<dut_trans1>::type_id::create("driver", this);
        assert(driver);
    }

    monitor = dut_monitor1<dut_trans1>::type_id::create("monitor", this);
    assert(monitor);
}

NOTE: UVM-SystemC API under review – subject to change
void dut_agent1::connect_phase(uvm::uvm_phase& phase) {
    if (get_is_active() == uvm::UVM_ACTIVE) {
        // connect driver and sequencer
        driver->seq_item_port(sequencer->seq_item_export);
    }
    ...
}

Only the connection between sequencer and driver is made here. Connection of driver and monitor to the DUT is done via the configuration mechanism.

NOTE: UVM-SystemC API under review – subject to change.
A UVM verification component (UVC) is an environment which consists of one or more cooperating agents.

UVCs or agents may set or get configuration parameters.

Each verification component is connected to the DUT using a dedicated interface.

C++ base class: uvm_env
class dut_uvc: public uvm::uvm_env
{
    public:
    UVM_COMPONENT_UTILS(dut_uvc);

    dut_agent1* agent1;

dut_uvc::dut_uvc(uvm::uvm_name name) :
    uvm_env(name), agent1(0) {
}
void dut_uvc::build_phase(uvm::uvm_phase& phase)
{
    uvm_env::build_phase(phase);
    // instantiate the agent
    agent1 = dut_agent1::type_id::create("agent1", this);
    assert(agent1);
}
};

A UVC is considered as a sub-environment in large system-level environments

In this example, the UVM verification component (UVC) contains only one agent

NOTE: UVM-SystemC API 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 test bench hierarchy, but are mapped onto one or more sequencers
• Sequences can be layered, hierarchical or virtual, and may contain multiple sequences or sequence items
• Sequences and transactions can be configured via the factory
class dut_trans1: public uvm::uvm_sequence_item
{
public:
    int data1;
    int data2;
    UVM_OBJECT_UTILS( dut_trans1);

    dut_trans1::dut_trans1(const std::string& name) :
        uvm_sequence_item(name), data1(0), data2(0){}

    virtual void
do_print(uvm::uvm_printer& printer) const;
    virtual void
do_pack(uvm::uvm_packer& packer) const;
    virtual void
do_unpack(uvm::uvm_packer& packer);
    virtual void
do_copy(const uvm::uvm_object& rhs);
    virtual bool
do_compare(const uvm::uvm_object& rhs) const;
};

User-defined data items (randomization can be done using SCV or CRAVE)

Transaction defined as sequence item

A sequence item should implement all elementary member functions to print, pack, unpack, copy and compare the data items (there are no field macros in UVM-SystemC )
class dut_sequence1: public uvm::uvm_sequence<> {
    public:
    UVM_OBJECT_UTILS(dut_sequence1);
    dut_sequence1(const std::string& name = "dut_sequence1");

    dut_sequence1::dut_sequence1(const std::string& name) :
    uvm_sequence<>(name) {}

    void dut_sequence1::pre_body() {
        if (starting_phase != NULL)
            starting_phase->raise_objection(this);
    }

    void dut_sequence1::body() {
        dut_trans1* req = dut_trans1::type_id::create("req");
        uvm::uvm_sequence_item* rsp;
        start_item(req);

        //Implement transaction contents
        finish_item(req);
        get_response(rsp);
    }

    void dut_sequence1::post_body() {
        if (starting_phase != NULL)
            starting_phase->drop_objection(this);
    }
};
UVM environment (test bench)

- A test bench is the environment which instantiates and configures the UVCs, scoreboard, and (optional) virtual sequencer.
- The test bench connects:
  - Agent sequencer(s) in each UVC with the virtual sequencer (if defined)
  - Monitor analysis port(s) in each UVC with the scoreboard subscriber(s)
  - Note: The driver and monitor in each agent connect to the DUT using the interface stored in the configuration database.
- C++ base class: uvm_env
class testbench : public uvm_env
{
public:
vip_uvc* uvc1;
vip_uvc* uvc2;
virt_sequencer* virtual_sequencer;
scoreboard* scoreboard1;

UVM_COMPONENT_UTILS(testbench);

testbench( uvm_name name ) : uvm_env( name ), uvc1(0), uvc2(0),
virtual_sequencer(0), scoreboard1(0) {}

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

    uvc1 = vip_uvc::type_id::create("uvc1", this);
    assert(uvc1);
    uvc2 = vip_uvc::type_id::create("uvc2", this);
    assert(uvc2);

    set_config_int("uvc1.*", "is_active", UVM_ACTIVE);
    set_config_int("uvc2.*", "is_active", UVM_PASSIVE);
...

All components in the test bench will be dynamically instan-tiated so they can be overridden by the test if needed

Definition of active or passive UVCs

NOTE: UVM-SystemC API under review – subject to change
... virtual_sequencer = virt_sequencer::type_id::create("virtual_sequencer", this);
assert(virtual_sequencer);

scoreboard1 = scoreboard::type_id::create("scoreboard1", this);
assert(scoreboard1);
}

virtual void connect_phase( uvm_phase& phase )
{
    virtual_sequencer->vip_seqr = uvc1->agent->sequencer;
    uvc1->agent->monitor->item_collected_port.connect(scoreboard1->xmt_listener_imp);
    uvc2->agent->monitor->item_collected_port.connect(scoreboard1->rcv_listener_imp);
}
;

NOTE: UVM-SystemC API under review – subject to change
Each UVM test is defined as a dedicated test class, which instantiates the test bench and defines the test sequence(s).

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

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

C++ base class: `uvm_test`
class test : public uvm_test {
    public:
        testbench* tb;
        bool test_pass;

    test( uvm_name name ) : uvm_test( name ),
        tb(0), test_pass(true) {} 

    UVM_COMPONENT_UTILS(test);

    virtual void build_phase( uvm_phase& phase )
    {
        uvm_test::build_phase(phase);
        tb = testbench::type_id::create("tb", this);
        assert(tb);

        uvm_config_db<uvm_object_wrapper*>::set( this,
            tb.uvc1.agent.sequencer.run_phase", "default_sequence",
            vip_sequence<vip_trans>::type_id::get());

        set_type_override_by_type( vip_driver<vip_trans>::get_type(),
            new_driver<vip_trans>::get_type() );

    ...

    Specific class to identify the test objects for execution in the sc_main program

    The test instantiates the required test bench

    Configuration of the default sequence, which will be executed on the sequencer of the agent in UVC1

    Factory method to override the original driver with a new driver

    NOTE: UVM-SystemC API under review – subject to change
virtual void run_phase( uvm_phase& phase )
{
    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 **");
}
The main program (top-level)

- 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`.

![Diagram of main program structure](image)
int sc_main(int, char*[]) 
{
    dut* my_dut = new dut("my_dut");
    vip_if* vif_uvc1 = new vip_if;
    vip_if* vif_uvc2 = new vip_if;

    uvm_config_db<vip_if*>::set(0, ".uvc1.*", "vif", vif_uvc1);
    uvm_config_db<vip_if*>::set(0, "*.uvc2.*", "vif", vif_uvc2);

    my_dut->in(vif_uvc1->sig_a);
    my_dut->out(vif_uvc2->sig_a);

    run_test("test");
    return 0;
}
Constrained randomization and functional coverage in UVM-SystemC

• Constrained randomization libraries for SystemC are available
  – SystemC Verification Library (SCV)
  – Constrained Random Verification Environment (CRAVE)

• No standardized functional coverage API in SystemC available
  – Proprietary/commercial SystemC coverage APIs available, but not offered (yet) for standardization

• Proposals for randomization and coverage APIs exist
Demo
UVM-SystemC-AMS

• The UVM-SystemC infrastructure can also handle AMS verification
• Transactions will program analog driver and monitors
• Drivers generate analog signals, Monitors analyze analog signals and extracting properties like amplitude, spectrum, ... and transfer them via transactions
• AMS verification requires continuous distribution function (and not PWC only)
• Randomization of DUT parameters is essential
UVC with AMS driver and monitor using SystemC-AMS

- Regular UVM-SystemC drivers and monitors are used in which SystemC-AMS Timed Data Flow (TDF) modules are instantiated.
- For the SystemC-AMS modules, TDF ports are necessary to allow read / write operations to the analog interface.
- The parent driver and monitor establish the connection from the TDF ports to the interface via the configuration mechanism.
Further UVM-SystemC-AMS extensions

• UVM AMS extensions will not break the existing UVM
• Time annotation to transaction
  – Decoupled sequence time
  – Data dependent synchronization
• Introducing of a pre-build phase under discussion
  – Is executed before the DUT is instantiated
  – Permits the setting of parameter, which influence the DUT creation
Demo
Outline

• Introduction and Motivation
  – Universal Verification Methodology (UVM) ... what is it?
  – Why UVM in SystemC/C++/SystemC-AMS?
• UVM-SystemC overview
  – UVM foundation elements
  – UVM test bench and test creation
  – Randomization and coverage
• Standardization within Accellera
• Applications and use cases of UVM-SystemC
• Summary and outlook
Standardization within Accellera

- UVM-SystemC Standardization within Accellera Verification WG is under way
  - UVM-SystemC Language Reference Manual (LRM) available
  - UVM-SystemC Proof-of-Concept implementation exists, released under Apache 2.0 license
Outline

• Introduction and Motivation
  – Universal Verification Methodology (UVM) ... what is it?
  – Why UVM in SystemC/C++/SystemC-AMS?

• UVM-SystemC overview
  – UVM foundation elements
  – UVM test bench and test creation
  – Randomization and coverage

• Standardization within Accellera

• Applications and use cases of UVM-SystemC

• Summary and outlook
Applications and use cases of UVM-SystemC

- Enables new use cases

- New re-use scenarios

- IP protected, language and simulator independent verification IP

- Enables System-level UVM based verification

- Simplifies development of UVM based verification methods for AMS systems
Re-use across Languages, Simulators, Abstraction Levels

Testbench (env)

- Test
- Test sequence

scoreboard

- Subscr 1
- ref model
- Subscr 2

UVC1 (env)

- agent
- Sqr
- conf
- Drv
- Mon

UVC2 (env)

- agent
- Sqr
- conf
- Drv
- Mon

DUT - VHDL

- in AMS
- out DIG SW

DUT - Verilog

- in AMS
- out DIG SW

DUT - SystemC

- in AMS
- out DIG SW

DUT - Matlab

- in AMS
- out DIG SW
Re-use for emulation and lab validation

Simulation - SystemC

Real Time Hardware

Real Time Hardware

Testbench (env)

Testbench (env)

Testbench (env)

DUT

DUT

DUT

SystemC - Behavioral

FPGA - Emulation

ASIC – 1st Silicon

Driver

Driver

Driver

Monitor

Monitor

Monitor

SystemC

Emulation

Lab equip

SystemC

Emulation

Lab equip

SystemC

Emulation

Lab equip

Download

Monitor

Integrate

Zynq Board (FPGA + ARM)

Source: ZedBoard.org
UVM for System-level / Functional verification

• Vision
  – Translating specifications (documents, standards) to readable – also for non verification experts - test scenarios, this should also include ranges and uncertainties
    • No separation between analog/digital, hard- and software
    • “real” system-level verification

• Main question:
  – Will the system work for the purposes for which it will be built?
Challenges for UVM System-level

• No executable reference model available

• Complex stimulation and expected sequences

• Coverage measure is different to implementation level

• How much of the possible application scenarios, input stimuli, operating conditions, specification items are verified?

→ UVM methodology/best practices have to be extended for system level!

→ UVM framework is generic enough to realize the required extensions needed for System level verification!
 Outline

• Introduction and Motivation
  – Universal Verification Methodology (UVM) ... what is it?
  – Why UVM in SystemC/C++/SystemC-AMS?
• UVM-SystemC overview
  – UVM foundation elements
  – UVM test bench and test creation
  – Randomization and coverage
• Standardization within Accellera
• Applications and use cases of UVM-SystemC
• Summary and outlook
Summary and outlook (1)

• Universal Verification Methodology created in **SystemC/C++**
  – Fully compliant with UVM standard
  – Target is to make all essential features of UVM available in **SystemC/C++**
  – UVM-SystemC language definition and proof-of-concept implementation contributed to Accellera Systems Initiative
  – SystemC-AMS is used for AMS system-level verification use cases
Summary and outlook (2)

• Ongoing developments
  – Extend UVM-SystemC with constrained randomization capabilities using SystemC Verification Library (SCV) or CRAVE
  – Introduction of randomization and functional coverage features
  – Add register abstraction layer and callback mechanism
  – Develop UVM based AMS and system-level verification methods
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 HETEROGENEOUS RELIABLE DESIGN AND INTEGRATION' (VERDI). The research leading to these results has received funding from the European Commission under grand agreement No 287562.
Resources

- SystemC, SystemC-AMS, UVM Standards
  - [www.accellera.org](http://www.accellera.org)
- SystemC proof-of-concept
- SystemC-AMS proof-of-concept
  - [www.coside.de/open_source.html](http://www.coside.de/open_source.html)
- Verdi project site (e.g. publications, tutorials for UVM SystemC)
  - [www.verdi-fp7.eu](http://www.verdi-fp7.eu)
- Crave randomization library
  - [www.systemc-verification.org/](http://www.systemc-verification.org/)
Questions
Guidelines (1)

• Please keep the default font size for main lines at 28pt (or 26pt)
  – And use 24pt (or 22pt) font size for the sub bullets
• Use the default bullet style and color scheme supplied by this template
• Limited the number of bullets per page.
• Use keywords, not full sentences
• Please do not overlay Accellera or DVCon logo’s
• Check the page numbering