Introducing the Universal Verification Methodology (UVM) in SystemC and SystemC-AMS

Karsten Einwich  Fraunhofer IIS/EAS
Martin Barnasconi  NXP Semiconductors
Thilo Vörler  Fraunhofer IIS/EAS
Thomas Klotz  Fraunhofer IIS/EAS

NASCUG 2014

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
- Contribution to Accellera
- Applications and use cases of UVM-SystemC
- Summary and outlook

NASCUG 2014
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
- Contribution to 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, scoreboardng, 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
- Target to make UVM truly universal, and not tied to a particular language

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
- 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
- Contribution to 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:</td>
<td>✔️</td>
</tr>
<tr>
<td>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
  - Test cases
- Scenario
  - Sequences
- Functional
  - Sequencer
  - Driver
  - Monitor
- Command
  - Sequencer
  - Driver
  - Monitor
- Signal
  - Device under test
  - Verification component
  - Verification environment (test bench)
  - Functional coverage

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
**UVM agent**

- 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`

**UVM-SystemC agent (1)**

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

  vip_agent( uvm_name name )
    : uvm_agent( name ), sequencer(0), driver(0), monitor(0) {}
  virtual void build_phase( uvm_phase& phase )
  {
    uvm_agent::build_phase(phase);
    if ( get_is_active() == UVM_ACTIVE )
      {
        sequencer = vip_sequencer<vip_trans>::type_id::create("sequencer", this);
        assert(sequencer);
        driver = vip_driver<vip_trans>::type_id::create("driver", this);
        assert(driver);
      }
    monitor = vip_monitor::type_id::create("monitor", this);
    assert(monitor);
  }
};
```

- **Dedicated base class** to distinguish agents from other component types
- **Registers the object in the factory**
- **Children are instantiated in the build phase**
- **Essential call to base class to access properties of the agent**
- **Call to the factory which creates and instantiates this component dynamically**
UVM-SystemC agent (2)

```c++
virtual void connect_phase(uvm_phase& phase)
{
    if (get_is_active() == UVM_ACTIVE)
    {
        // connect sequencer to driver
        driver->seq_item_port.connect(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.

UVM verification component

- 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
- An independent test sequence is processed by the driver via a sequencer
- Each verification component is connected to the DUT using a dedicated interface
- C++ base class: `uvm_env`
In this example, the UVM verification component (UVC) contains only one agent. In practice, more agents are likely to be instantiated.

<table>
<thead>
<tr>
<th>UVM-SystemC verification component</th>
</tr>
</thead>
<tbody>
<tr>
<td>class vip_uvc : public uvm_env</td>
</tr>
<tr>
<td>{ public:</td>
</tr>
<tr>
<td>vip_agent* agent;</td>
</tr>
<tr>
<td>UVM_COMPONENT_UTILS(vip_uvc);</td>
</tr>
<tr>
<td>vip_uvc( uvm_name name ) : uvm_env( name ), agent(0) {}</td>
</tr>
<tr>
<td>virtual void build_phase( uvm_phase&amp; phase )</td>
</tr>
<tr>
<td>{ uvm_env::build_phase(phase);</td>
</tr>
<tr>
<td>agent = vip_agent::type_id::create(&quot;agent&quot;, this);</td>
</tr>
<tr>
<td>assert(agent);</td>
</tr>
<tr>
<td>}</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>UVC with AMS driver and monitor using SystemC-AMS</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Regular UVM-SystemC drivers</strong> and monitors are used in which SystemC-AMS Timed Data Flow (TDF) modules are instantiated.</td>
</tr>
<tr>
<td>Factory overrides enable configuration of the UVC’s driver or monitor for specific AMS use cases.</td>
</tr>
<tr>
<td>For the SystemC-AMS modules, TDF ports are necessary to allow read/write operations to the analog interface.</td>
</tr>
<tr>
<td>The parent driver and monitor establish the connection from the TDF ports to the interface via the configuration mechanism.</td>
</tr>
</tbody>
</table>
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.

---

**UVM-SystemC sequence item**

```c++
class vip_trans : public uvm_sequence_item {
    public:
    int addr;
    int data;
    bus_op_t op;
    UVM_OBJECT_UTILS(vip_trans);

    vip_trans(const std::string& name = "vip_trans")
    : addr(0x0), data(0x0), op(BUS_READ) {}

    virtual void do_print( uvm_printer& printer ) const { ... }
    virtual void do_pack( uvm_packer& packer ) const { ... }
    virtual void do_unpack( uvm_packer& packer ) const { ... }
    virtual void do_copy( const uvm_object* rhs ) const { ... }
    virtual bool do_compare( const uvm_object* rhs ) const { ... }
};
```

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.)
**UVM-SystemC sequence**

```cpp
template <typename REQ = uvm_sequence_item, typename RSP = REQ>
class sequence : public uvm_sequence<REQ,RSP> {
  public:
    sequence(const std::string& name) : uvm_sequence<REQ,RSP>(name) {};

    UVM_OBJECT_PARAM_UTILS(sequence<REQ,RSP>);

    virtual void pre_body() {
      if (starting_phase != NULL)
        starting_phase->raise_object(this);
    }

    virtual void body() {
      REQ* req;
      RSP* rsp;
      ...
      start_item(req);
      finish_item(req);
      get_response(rsp);
    }

    virtual void post_body() {
      if (starting_phase != NULL)
        starting_phase->drop_object(this);
    }
};
```

- A sequence contains a request and (optional) response, both defined as sequence item
- Factory registration supports template classes
- Raise objection if there is no parent sequence
- Compatibility layer to SCV or CRAVE not yet available
- Optional: get response

**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**
UVM-SystemC test bench (1)

class testbench : public uvm_env
{
  public:
    vip_uvc* uvc1;
    vip_uvc* uvc2;
    virtual_sequence* virtual_sequence;
    scoreboard* scoreboard1;
    UVM_COMPONENT_UTILS(testbench);

testbench(uvm_name name)
  : uvm_env(name), uvc1(0), uvc2(0),
    virtual_sequence(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);

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

virtual void connect_phase(uvm_phase& phase)
{
  virtual_sequence->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);
}
};

All components in the test bench will be dynamically instantiated so they can be overridden by the test if needed.

UVM-SystemC test bench (2)

virtual sequencer = virt_sequencer::type_id::create(  
  "virtual_sequence", this);
assert(virtual_sequence);

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

UVM-SystemC API under review – subject to change.
UVM test

- Each UVM test is defined as a dedicated C++ 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`

UVM-SystemC test (1)

```cpp
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<vip_driver<vip_trans>::get_type(),
        vip_sequence<vip_trans>::type_id::get()>::set_type_override_by_type(vip_driver<vip_trans>::get_type(),
        new_driver<vip_trans>::get_type());

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

        ...}

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

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

Specific class to identify the test objects for execution in the `sc_main` program
UVM-SystemC test (2)

```c
... 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 **");
}
};
```

Get result of the scoreboard in the extract phase

Report results in the report phase

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
UVM-SystemC main program

```c
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");
    sc_start();
    return 0;
}
```

Constrained randomization in UVM-SystemC

- **Available** constrained randomization libraries for SystemC
  - SystemC Verification Library (SCV)
  - Constrained Random Verification Environment (CRAVE)
- Both APIs differ from the SystemVerilog constrained randomization API
  - Preference to have aligned API across these different languages
- UVM-SystemC therefore introduces a “compatibility layer” for constrained randomization with UVM/SystemVerilog “Look & Feel”
  - Introduction of dedicated randomization variables and constraint objects, randomize() method, etc.
  - To be proposed as SCV language and library extension to Accellera
  - Proof-of-concept implemented using CRAVE library
Example: Constrained randomization in UVM-SystemC

```cpp
class simplesum : public scvx_rand_object
{
public:
  scvx_rand< int > x, y, z;
  scvx_constraint c1;

  simplesum( scvx_name name ) :
    x(name + "x"),
    y(name + "y"),
    z(name + "z"),
    c1(name + "c1")
  { c1( z() == x() + y() ); }

  void print_result() const
  { cout << name() << "x": " << z << "y": " << x << "y": "y" << endl; }
};
```

```cpp
int sc_main(int, char*[])
{
  bool result;

  simplesum s("simplesum");
  result = s.randomize();
  if (result)
    s.print_result();
  else
    cout << "No solution found." << endl;

  result = s.randomize_with(s.y() > 10 && s.x() == 8 );
  if (result)
    s.print_result();
  else
    cout << "No solution found." << endl;
}
```

---

Functional coverage in UVM-SystemC

- No standardized functional coverage API in SystemC available
- Proprietary/commercial SystemC coverage APIs available, but not offered (yet) for standardization
- Introduction of a functional coverage API for UVM-SystemC with UVM/SystemVerilog “Look & Feel”
- To be proposed as SCV language and library extension to Accellera
- Proof-of-concept implementation offers:
  - Coverage of variables (supports data types sc_bv, integer, and bool)
  - Automatic as well as user-defined coverage bins
  - Associate bins with sets of values (called multi-value bins)
  - Specialized bins (e.g. to ignore values, illegal values, etc)
  - Optional directives to control coverage collection and reporting
Example: Functional coverage in UVM-SystemC (1)

```cpp
class cg : public scvx_covergroup
{
  public:
    scvx_coverpoint cp_m;
    scvx_coverpoint cp_n;
  
  cg(const scvx_name name, int8 m, int8 n)
  {
    cp_m("cp_m", m);
    cp_n("cp_n", n);
    
    option.auto_bin_max = 16;
    cp_m.bins("bin_a") = list_of(4, 0, 1, 2, 3);
    cp_m.bins("bin_b") = list_of(4, 4, 5, 6, 7);
    cp_m.ignore_bins("ignore_bins_m") = 6;
    cp_n.ignore_bins("ignore_bins_n") = 13;
  }
};
```

```cpp
int sc_main(int, char*[])
{
  int m;
  int n;
  int stimuli_m[10] = {3, 5, 6, 6, 5, 5, 3, 3, 3, 3};
  int stimuli_n[10] = {13, 1, 6, 3, 16, 12,
                        8, 3, 13, 3};
  cg cg_inst("cg_inst", m, n);
  for (int i = 0; i < 10; i++)
  {
    m = stimuli_m[i];
    n = stimuli_n[i];
    cg_inst.sample();
  }
  cg_inst.report();
  return 0;
}
```

Example: Functional coverage in UVM-SystemC (2)

```bash
c . / test.exe
Covergroup: cg_inst
----------------------------------------------------------
VARIABLE  Expected  Covered  Percent
----------------------------------------------------------
cp_m   7        2    28.57
cp_n   15       5    33.33
----------------------------------------------------------
TOTAL:   22       7    31.82
----------------------------------------------------------
coverpoint: cp_m
----------------------------------------------------------
Name  Percent  Hitrate
----------------------------------------------------------
bin_a[0] 0       0
bin_a[1] 0       0
bin_a[2] 0       0
bin_a[3] 100      4
bin_b[4] 0       0
bin_b[5] 100      4
bin_b[7] 0       0
----------------------------------------------------------

coverpoint: cp_n
----------------------------------------------------------
Name  Percent  Hitrate
----------------------------------------------------------
auto[0] 100      0
auto[1] 100      1
auto[2] 0       0
auto[3] 100      3
auto[4] 0       0
auto[5] 0       0
auto[6] 100      1
auto[7] 0       0
auto[8] 100      1
auto[9] 0       0
auto[10] 0      0
auto[11] 0      0
auto[12] 100      1
auto[14] 0      0
auto[15] 0      0
----------------------------------------------------------
```
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
- Contribution to Accellera
- Applications and use cases of UVM-SystemC
- Summary and outlook

Contribution to Accellera

- Objective: seek further industry support and standardization of UVM-SystemC
- UVM-SystemC contribution to Accellera Verification WG
  - UVM-SystemC Language Reference Manual (LRM)
  - UVM-SystemC Proof-of-Concept implementation, released under Apache 2.0 license
- Align with SCV and Multi-Language requirements and future developments
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
- Contribution to 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

Re-use for emulation and lab validation
UVM-SystemC Codegeneration

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

    UVM_COMPONENT_UTILS(vip_agent)

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

    virtual void build_phase( uvm_phase& phase )
    {
        uvm_agent::build_phase(phase);
        if ( get_is_active() == UVM_ACTIVE )
        {
            sequencer = vip_sequencer<vip_trans>::type_id::create("sequencer",this);
            assert(sequencer);
            driver = vip_driver<vip_trans>::type_id::create("driver", this);
            assert(driver);
        }
        monitor = vip_monitor::type_id::create("monitor", this);
        assert(monitor);
    }
};
```

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 a tighter binding/synchronization to the time
- Non-causalities due continuous signal behavior
- AMS verification requires continuous distribution function (and not PWC only)
- Randomization of DUT parameters is essential
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
  - Is executed before the DUT is instantiated
  - Permits the setting of parameter, which influence the DUT creation
- Constraint randomization with continuous distribution function can be hardly realized
  - Constraints for AMS are usually much simpler

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!

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
- Contribution to Accellera
- Applications and use cases of UVM-SystemC
- Summary and outlook
Summary and outlook

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

Ongoing developments
- Extend UVM-SystemC with constrained randomization capabilities using SystemC Verification Library (SCV) or CRAVE
- Introduction of assertions 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 HETEROGENOUS 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

- SystemC proof-of-concept
  - www.accellera.org/downloads/standards/systemc

- SystemC-AMS proof-of-concept
  - www.coside.de/open_source.html

- Verdi project site (e.g. publications, tutorials for UVM SystemC)
  - www.verdi-fp7.eu

- Crave randomization library
  - www.systemc-verification.org/