Get Ready for

UVM-SystemC

Martin Barnasconi - NXP Semiconductors

Presented by: Anupam Bakshi - Agnisys

accellera SYSTEMS INITIATIVE™

2015 DESIGN AND VERIFICATION™ DVCON CONFERENCE AND EXHIBITION INDIA

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

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, 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

Main concepts of UVM (1)

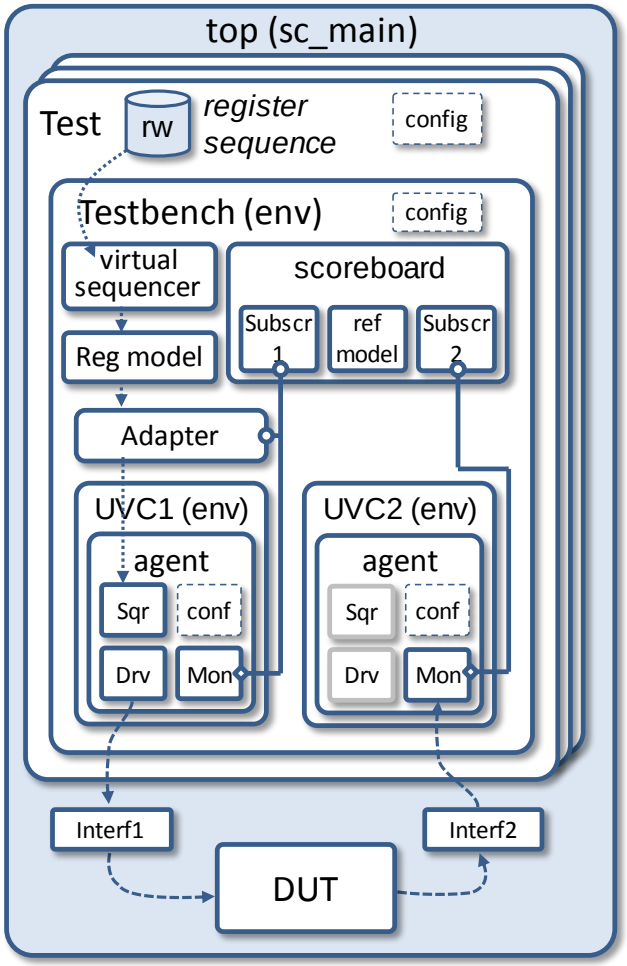
- Clear **separation** of test stimuli (sequences) and test bench
  - Sequences are treated as ‘transient objects’ and thus independent from the test bench construction and composition
  - In this way, sequences can be developed and reused independently
- Introducing test bench **abstraction levels**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



Advantages of UVM-SystemC

- UVM-SystemC library features
  - UVM components are 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

LRM definition

```
namespace uvm {

  class uvm_component : public sc_core::sc_module,
                        public uvm_report_object
  { ... };

} // namespace uvm
```

Application

```
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**

LRM definition

```
namespace uvm {

  template <typename REQ, typename RSP = REQ>
  class uvm_sqr_if_base
  : public virtual sc_core::sc_interface
  {
   public:
    virtual void get_next_item( REQ& req ) = 0;
    virtual bool try_next_item( REQ& req ) = 0;
    virtual void item_done( const RSP& item ) = 0;
    virtual void item_done() = 0;
    virtual void put( const RSP& rsp ) = 0;
    virtual void get( REQ& req ) = 0;
    virtual void peek( REQ& req ) = 0;
    ...
  }; // class uvm_sqr_if_base

} // namespace uvm
```

LRM definition

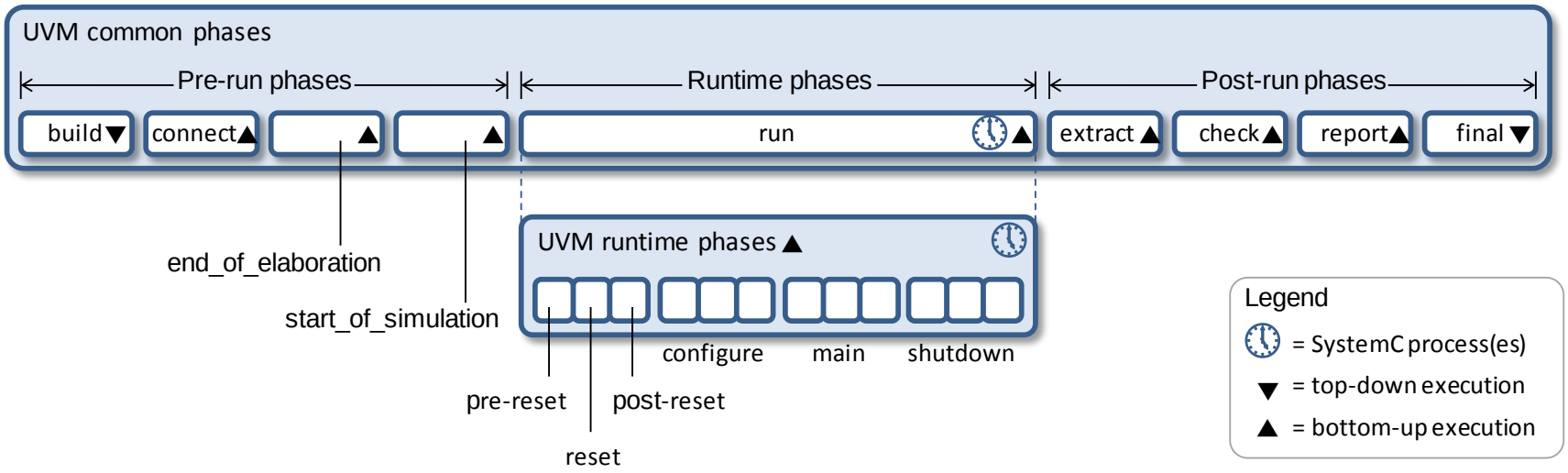
```
namespace uvm {

  template <typename T>
  class uvm_analysis_port : public tlm::tlm_analysis_port<T>
  {
   public:
    uvm_analysis_port();
    uvm_analysis_port( const std::string& name );

    virtual const std::string get_type_name();
    virtual void connect( tlm::tlm_analysis_if<T>& _if );
    ...
```

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

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 **==**)

Application

```
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);
  }
  ...
};
```

Disadvantage: type-specific methods

Application

```
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;
  }
  ...
};
```

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 RES/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

Application

```
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);
     ...
  }
};
```

Template class

UTILS macro supports template arguments

Template argument defines request type

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**

LRM definition

```
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…

Work-in-Progress: Register Abstraction Layer

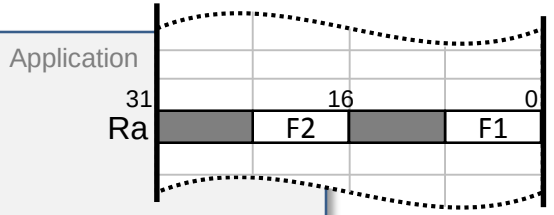
Register Abstraction Layer	Status
Register model containing registers, fields, blocks, etc.	testing
Register callbacks	testing
Register adapter, predictor, sequences and transaction items	testing
Register front-door access	testing
Build-in register test sequencers	development
Memory and memory allocation manager	development
Virtual registers and fields	development
Register back-door access (hdl_path)	study
Randomization of registers	study

Register Model example (1)

```
class reg_Ra : public uvm_reg
{
 public:
  uvm_reg_field* F1;
  uvm_reg_field* F2;

  UVM_OBJECT_UTILS(reg_Ra);

  reg_Ra( uvm_object_name name = "Ra" ) : uvm_reg(name, 32, UVM_NO_COVERAGE) {}

  void build()
  {
    F1 = uvm_reg_field::type_id::create("F1");
    F1->configure(this, 8, 0, "RW", false, 0x0, true, false, true);
    F2 = uvm_reg_field::type_id::create("F2");
    F2->configure(this, 8, 16, "RO", false, 0x0, true, false, true);
  }

}; // class reg_Ra
```

UVM register class

Register "Ra" contains two fields, F1 and F2



Register contains 32 bits and contains no coverage model

Use of the UVM factory to instantiate the register fields

Register field 'F1' configuration:
- Size: 8 bits
- LSB position in register: bit 0
- Access policy: Read/Write (RW)
- Volatile register: no (false)
- Reset value (if applicable): 0x0
- Reset possible: yes (true)
- Can be randomized: no (false)
- Is individually accessible: yes (true)

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

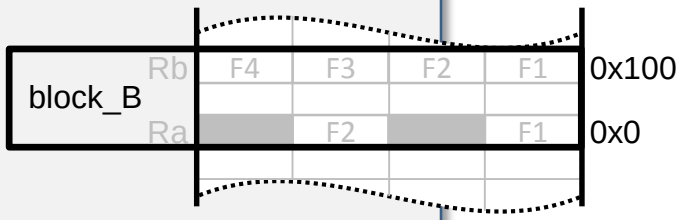
- Although the user can create a register model manually, the recommended use model is to generate this register model from an IP-XACT register description

Register Model example (2)

```
class block_B : public uvm_reg_block
{
 public:
  reg_Ra* Ra;
  reg_Rb* Rb;

  UVM_OBJECT_UTILS(block_B);

  block_B( uvm_object_name name = "B") : uvm_reg_block( name, UVM_NO_COVERAGE ) {}

  void build()
  {
    uvm_reg_addr_t base_addr = 0x0000;
    unsigned int n_bytes = 4;

    default_map = create_map("default_map", base_addr, n_bytes, UVM_BIG_ENDIAN);

    Ra = reg_Ra::type_id::create("Ra");
    Ra->configure(this, NULL);
    Ra->build();
    ...
    default_map->add_reg(Ra, 0x0,   "RW");
    default_map->add_reg(Rb, 0x100, "RW");
    ...
  }
}; // class block_B
```

UVM register block class

Register block B contains two registers: Ra and Rb



Create the address map

Use of the UVM factory to instantiate the registers

Add register to the register map, specifying the offset and access rights

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

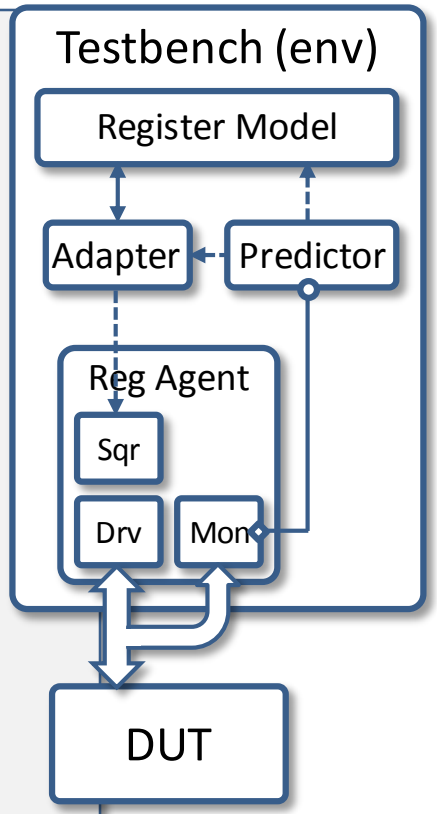
Test Bench including Register Model (1)

```
class tb_env : public uvm_env
{
 public:
  UVM_COMPONENT_UTILS(tb_env);

  block_B* regmodel;
  reg_agent<dut>* bus;
  uvm_reg_predictor<reg_rw>* predict;
  reg2rw_adapter* reg2rw;

  tb_env( uvm_component_name name = "tb_env" )
  : uvm_env(name), regmodel(NULL), bus(NULL),
    predict(NULL), reg2rw(NULL) {}

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

    bus = reg_agent<dut>::type_id::create("bus");

    regmodel = block_B::type_id::create("regmodel");
    regmodel->build();
    regmodel->lock_model();

    predict = uvm_reg_predictor<reg_rw>::type_id::create("predict");
  }
  ...
```

The test bench (env) contains the register model, agent, adapter and predictor

Instantiate components and build the register map

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



Test Bench including Register Model (2)

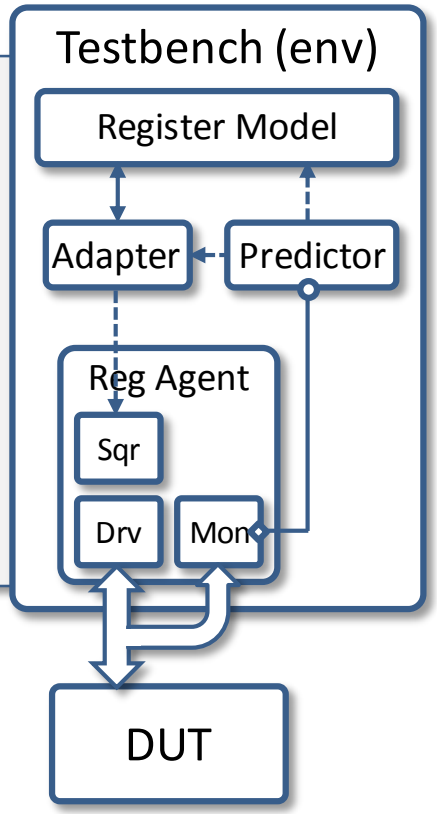
```
  ...
  void connect_phase( uvm_phase& phase )
  {
    reg2rw = reg2rw_adapter::type_id::create("reg2rw");

    regmodel->default_map->set_sequencer(bus->sqr, reg2rw);
    regmodel->default_map->set_auto_predict(false);

    predict->map = regmodel->default_map;
    predict->adapter = reg2rw;

    bus->mon->ap.connect(predict->bus_in);
  }

}; // class tb_env
```

Set the sequencer and adapter associated with this map.

Associate predictor with the register map and adapter

Connect monitor analysis port with predictor

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



Execute Build-in Register Test (1)

```
class test : public uvm_test
{
 public:
  tb_env* env;
  uvm_reg_sequence<>* seq;

  test( uvm_component_name name = "test" )
  : uvm_test(name), env(NULL), seq(NULL) {}

  UVM_COMPONENT_UTILS(test);

  void build_phase( uvm_phase& phase )
  {
    uvm_test::build_phase(phase);

    env = tb_env::type_id::create("tb_env");
    seq = uvm_reg_bit_bash_seq::type_id::create("seq");
  }

  void run_phase( uvm_phase& phase )
  {
    phase.raise_objection(this);
    env->regmodel->reset();
    seq->model = env->regmodel;
    seq->start(env->bus->sqr);
    seq->wait_for_sequence_state(UVM_FINISHED);
    phase.drop_objection(this);
  }
}; // class test
```

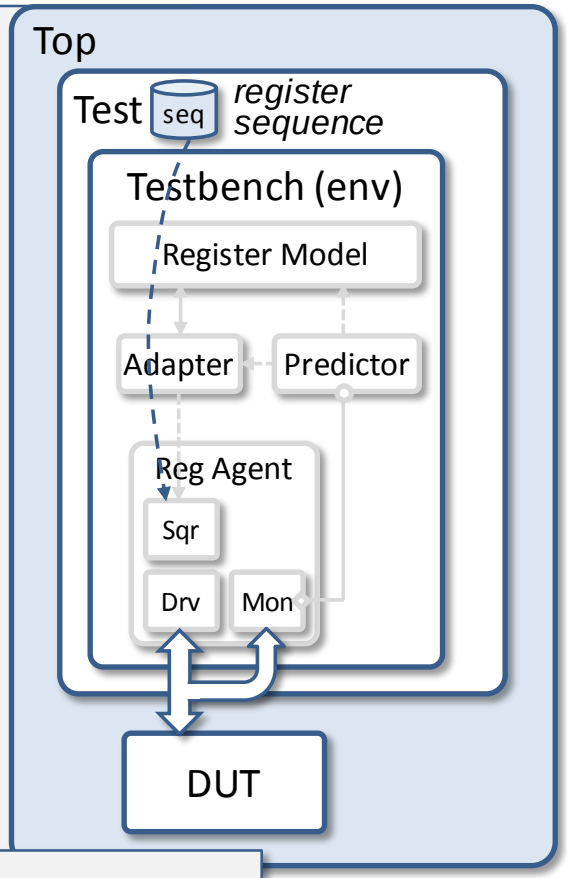
Application

Instantiate the test bench

Select the build-in "bit bashing" sequence

Start the register sequence

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



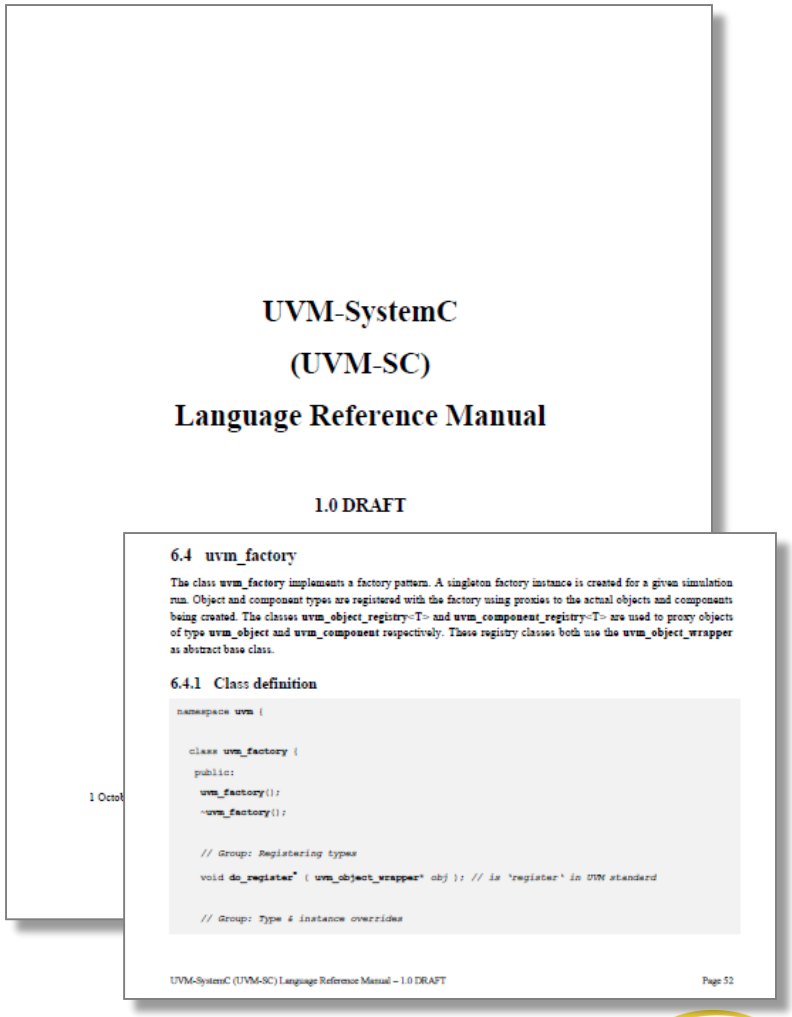

```
// top-level
int sc_main(int, char*[])
{
  ... // instantiate DUT
         and interfaces
  run_test("test");
  return 0;
}
```

Execute Build-in Register Test (2)

```
        SystemC 2.3.1-Accellera --- Dec 29 2014 13:55:54
        Copyright (c) 1996-2014 by all Contributors,
        ALL RIGHTS RESERVED

   Universal Verification Methodology in SystemC (UVM-SystemC)
        Version: 1.0-alpha1  Build: 510  Date: 2015-09-01
          Copyright (c) 2006 - 2015 by all Contributors
            See NOTICE file for all Contributors
                   ALL RIGHTS RESERVED
                 http://www.verdi-fp7.eu/
         Licensed under the Apache License, Version 2.0

UVM_INFO @ 0 s: reporter [RNTST] Running test 'test'...
UVM_INFO @ 0 s: reporter [STARTING_SEQ]
UVM_INFO @ 0 s: reporter [uvm_reg_bit_bash_seq] Verifying bits in register regmodel.Ra  in map 'regmodel.default_map'...
UVM_INFO @ 0 s: reporter [uvm_reg_bit_bash_seq] ...Bashing RW bit #0
UVM_INFO @ 0 s: reporter [uvm_reg_map] Writing 0x0000000000000001 at address 0x0 via map 'regmodel.default_map'...
UVM_INFO @ 0 s: reporter [REG_PREDICT] Observed WRITE transaction to register regmodel.Ra: value = 0x1 : updated value = 0x1
UVM_INFO @ 0 s: reporter [uvm_reg_map] Wrote 0x0000000000000001 at address 0x0 via map 'regmodel.default_map': UVM_IS_OK...
UVM_INFO @ 0 s: reporter [RegModel] Wrote register via map regmodel.default_map: regmodel.Ra = 0x1
UVM_INFO @ 0 s: reporter [uvm_reg_map] Reading address 0x0 via map 'regmodel.default_map'...
UVM_INFO @ 0 s: reporter [REG_PREDICT] Observed READ transaction to register regmodel.Ra: value= 0x1
UVM_INFO @ 0 s: reporter [uvm_reg_map] Read 0x0000000000000001 at address 0x0 via map 'regmodel.default_map': UVM_IS_OK...
UVM_INFO @ 0 s: reporter [RegModel] Read register via map regmodel.default_map: regmodel.Ra = 0x1
                :
                :
--- UVM Report Summary ---
Quit count :     0 of    10
** Report counts by severity
UVM_INFO      : 836
UVM_WARNING   :   1
UVM_ERROR     :   0
UVM_FATAL     :   0
** Report counts by id
[RNTST]                 1
[RegModel]            256
[STARTING_SEQ]          1
[TPRGED]                1
[uvm_reg_bit_bash_seq] 66
[uvm_reg_map]         512
UVM_INFO @ 0 s: reporter [FINISH] UVM-SystemC phasing completed; simulation finished
```

Standardization in Accellera

- Growing industry interest for UVM in SystemC
- Standardization in SystemC Verification WG ongoing
  - Writing and review of UVM-SystemC Language Reference Manual (LRM)
  - Improving the UVM-SystemC Proof-of-Concept (PoC) implementation
  - Creation of a UVM-SystemC regression suite
- 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:
  - UVM-SystemC API documented in the Language Reference Manual
  - 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
  - Review of Language Reference Manual and Proof-of-concept implementation ongoing
  - 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

UVM-SystemC at DVCon Europe

- DVCon Europe hosts a tutorial on UVM-SystemC:

  "UVM Goes Universal - Introducing UVM in SystemC"

- Contents
  - Introduction: Basics and key mechanisms of UVM
  - Verification examples: demonstrating the applications of UVM-SystemC
  - Standardization perspective: Presents the ongoing development of the proof-of-concept implementation and the language reference manual
- Program and registration: www.dvcon-europe.org

2015 DESIGN AND VERIFICATION™ DVCON CONFERENCE AND EXHIBITION EUROPE

Nov 11-12, 2015
Munich, Germany

accellera SYSTEMS INITIATIVE™

2015 DESIGN AND VERIFICATION™ DVCON CONFERENCE AND EXHIBITION INDIA

Questions