DESIGN & VERIFICATION DVCON 2014 CONFERENCE & EXHIBITION

Advancing system-level verification using UVM in SystemC

Martin Barnasconi, NXP Semiconductors
François Pêcheux, University Pierre and Marie Curie
Thilo Vörtler, Fraunhofer IIS/EAS

NXP UPMC SORBONNE UNIVERSITÉS Fraunhofer IIS

Outline

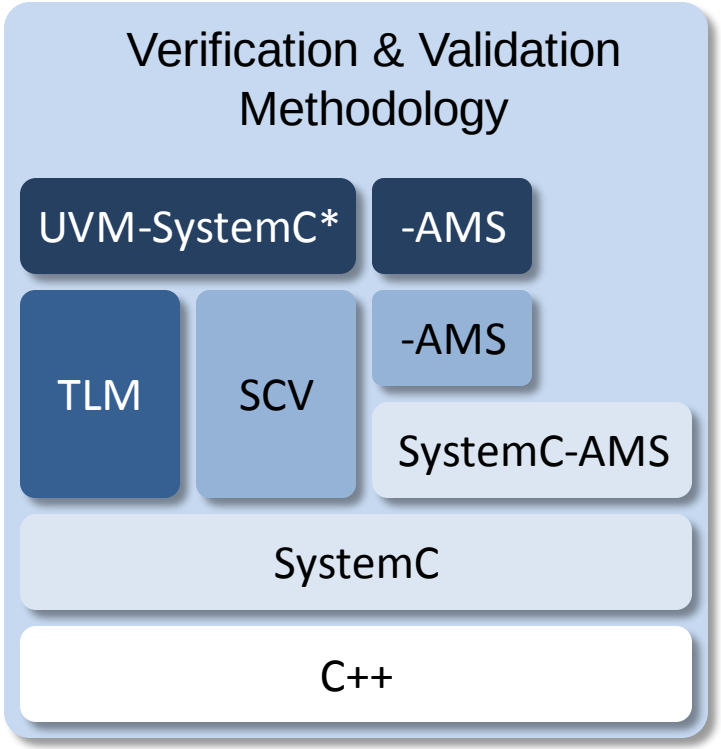
- Introduction
  - Universal Verification Methodology (UVM) … what is it?
- Motivation
- Why UVM in SystemC?
- UVM-SystemC overview
  - UVM foundation elements
  - UVM test bench and test creation
- Contribution to Accellera
- Summary and outlook
- Acknowledgements

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, scoreboarding, 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



*UVM-SystemC = UVM implemented in SystemC/C++

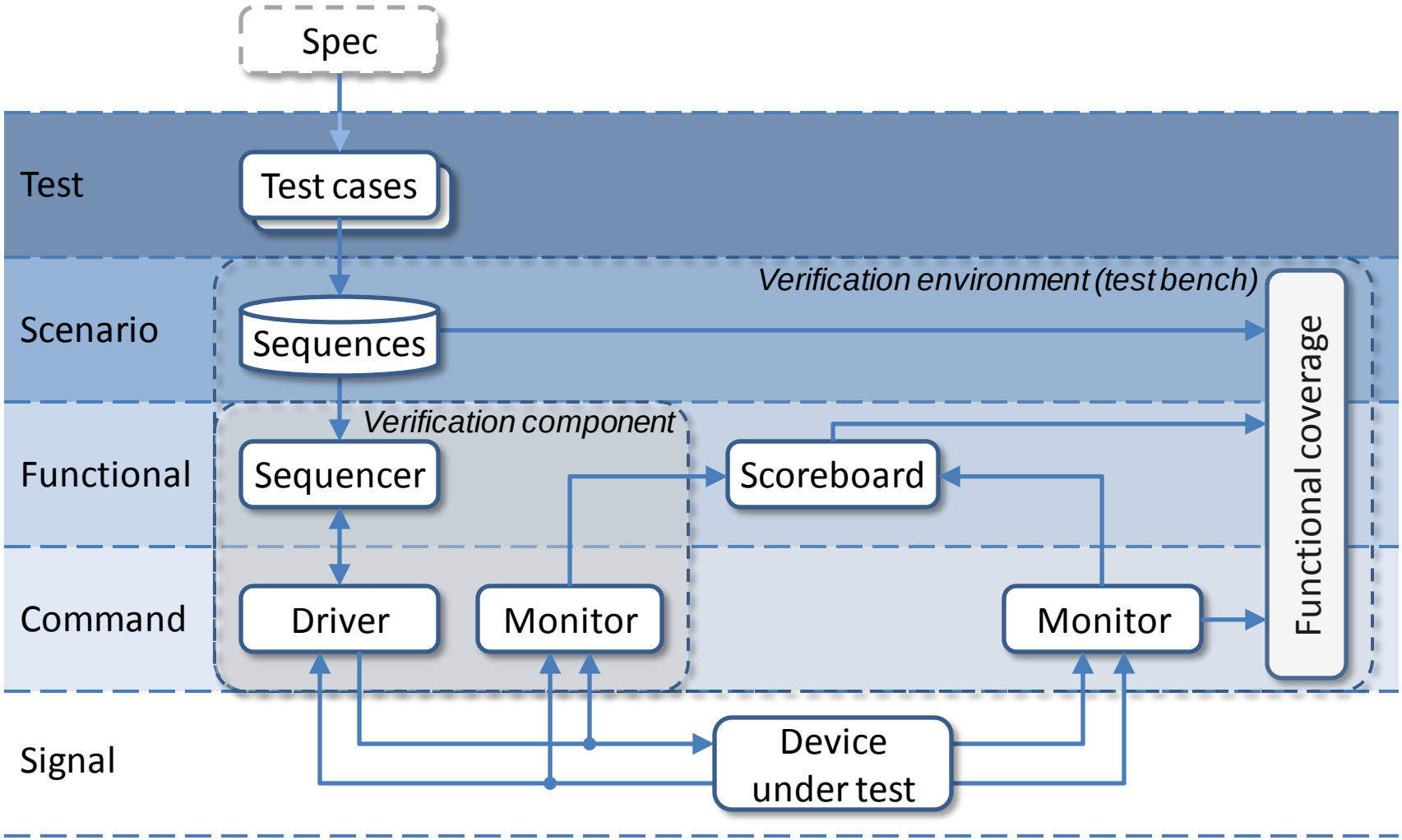
Why UVM in SystemC/C++ ?

- 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
- Vision: 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** (HiL) simulation or Rapid Control Prototyping (RCP)
- Benefit from proven standards and reference implementations
  - Leverage from existing methodology standards and reference implementations, aligned with best practices in verification

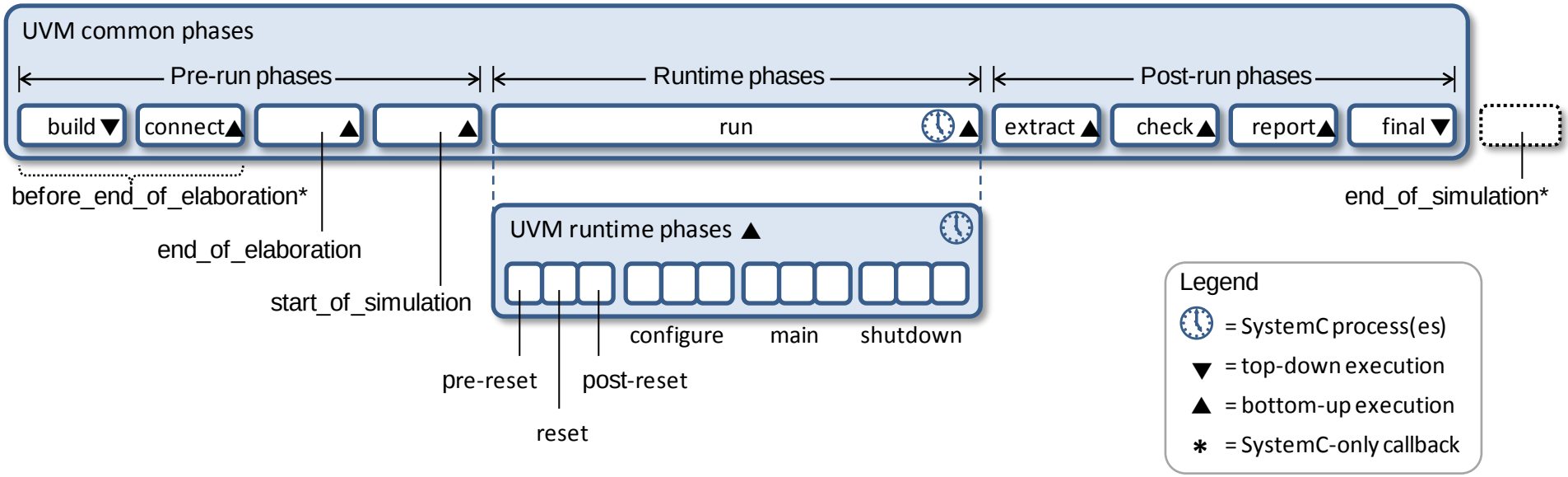
UVM-SystemC overview

UVM-SystemC functionality	Status
Test bench creation with component classes: agent, sequencer, driver, monitor, scoreboard, etc.	☑
Test creation with test, (virtual) sequences, etc.	☑
Configuration and factory mechanism	☑
Phasing and objections	☑
Policies to print, compare, pack, unpack, etc.	☑
Messaging and reporting	☑
Register abstraction layer and callbacks	development
Coverage groups	development
Constrained randomization	SCV or CRAVE

UVM layered architecture



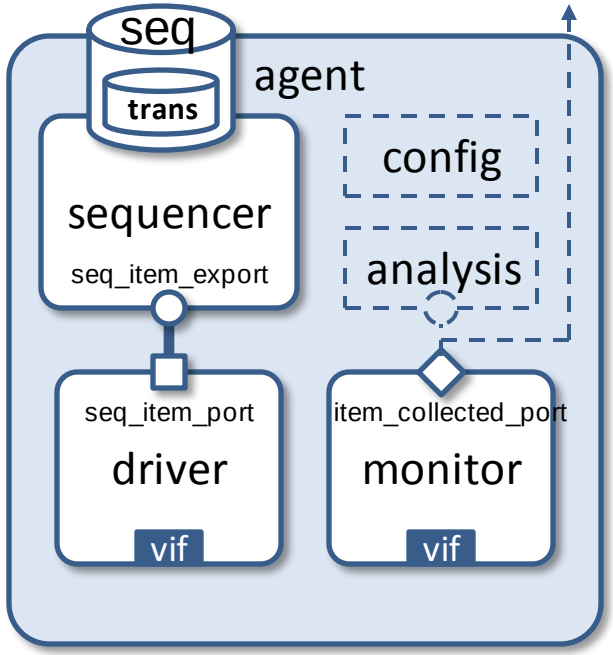
UVM-SystemC phasing



- 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

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

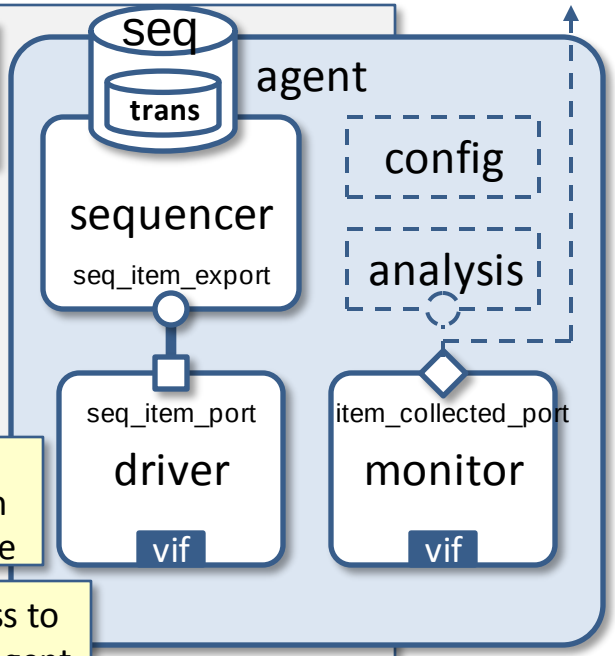
Dedicated base class to distinguish agents from other component types

Registers the object in the factory

Children are instantiated in the build phase

Essential call to base class to access properties of the agent

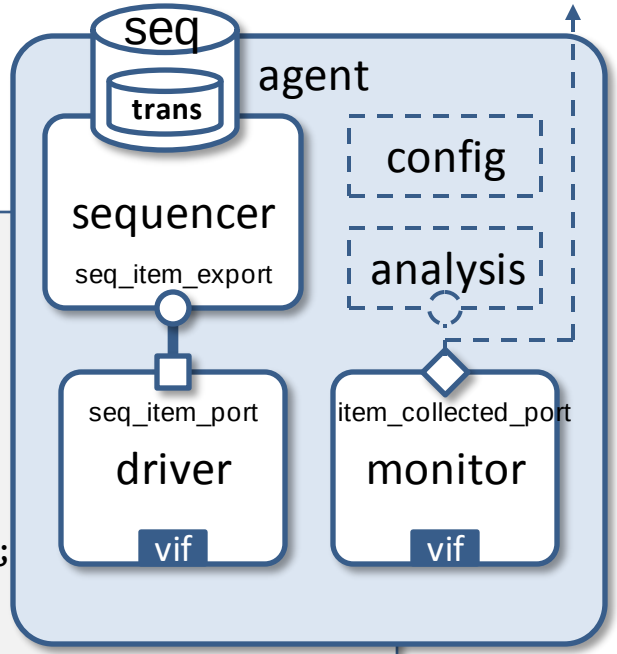
Call to the factory which creates and instantiates this component dynamically



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

UVM-SystemC agent

```
  ...

  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);
    }

  }
};
```

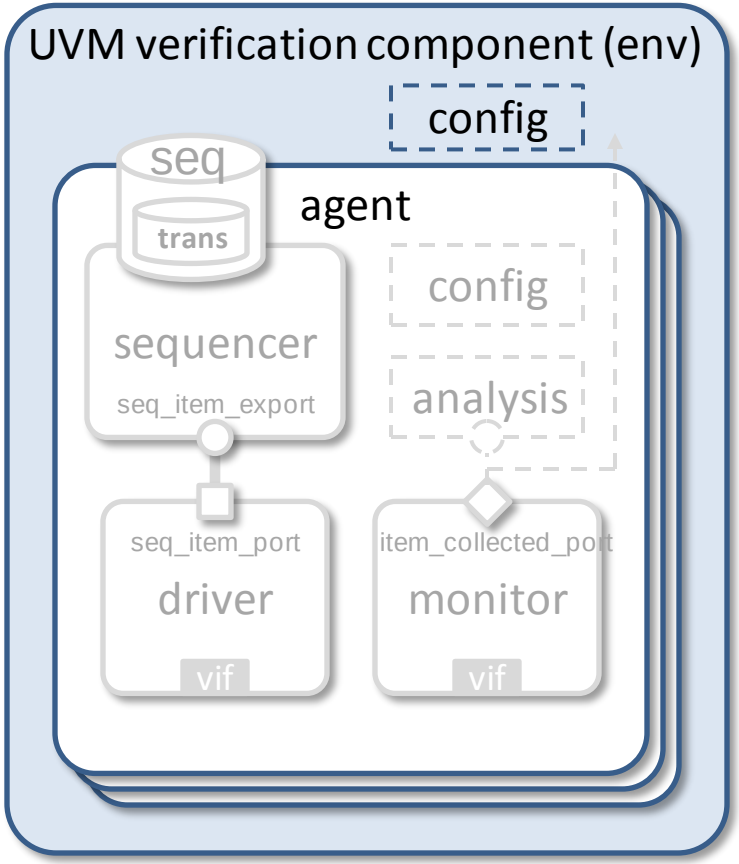


Only the connection between sequencer and driver is made here. Connection of driver and monitor to the DUT is done via the configuration mechanism

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

UVM verification component

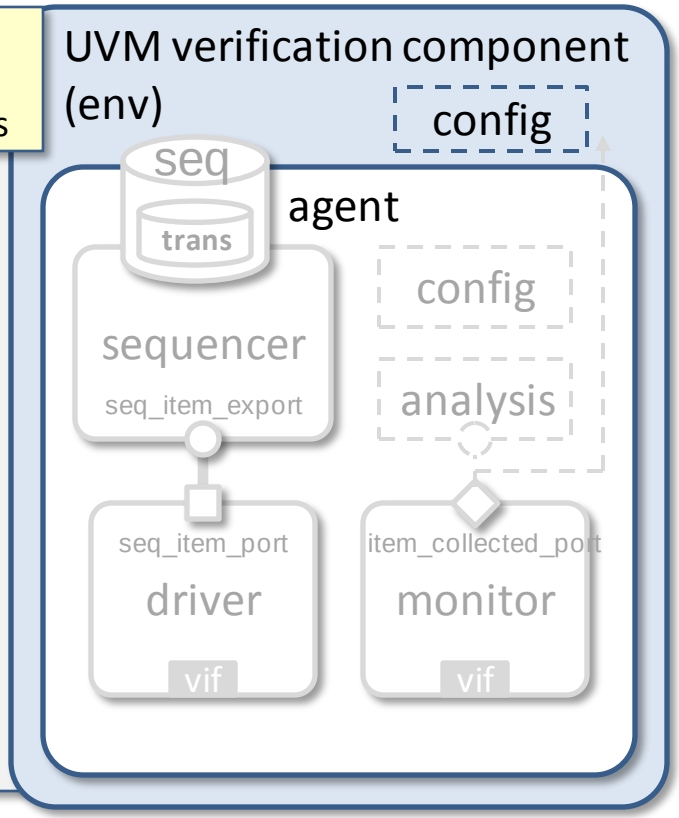
- A UVM verification component (UVC) is an environment which consists of one or more cooperating agents
- UVCs or agents may set or get configuration parameters
- An independent test sequence is processed by the driver via a sequencer
- Each verification component is connected to the DUT using a dedicated interface
- C++ base class: `uvm_env`



UVM verification component

```
class vip_uvc : public uvm_env
{
 public:
  vip_agent* agent;

  UVM_COMPONENT_UTILS(vip_uvc);

  vip_uvc( uvm_name name )
  : uvm_env( name ), agent(0) {}

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

    agent = vip_agent::type_id::create("agent", this);
    assert(agent);
  }

};
```

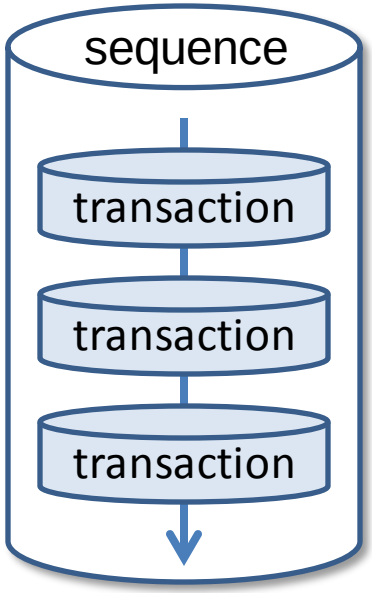
A UVC is considered as a sub-environment in large system-level environments



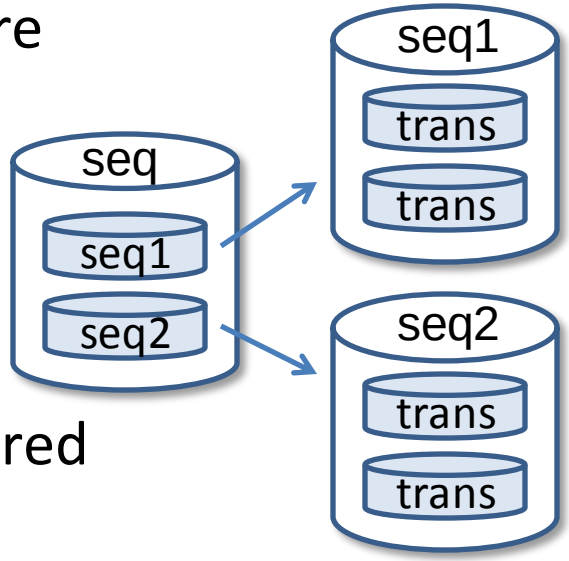
- In this example, the UVM verification component (UVC) contains only one agent. In practice, more agents are likely to be instantiated

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



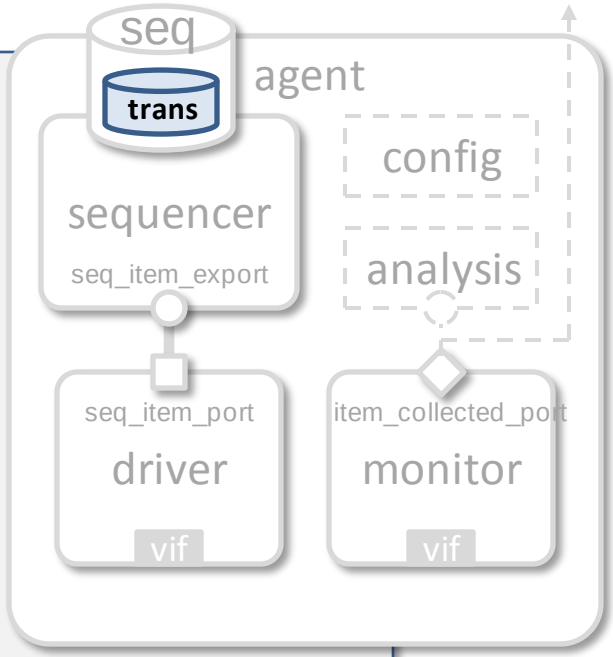
UVM-SystemC sequence item

```
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 ) { ... }
  virtual void do_copy( const uvm_object* rhs ) { ... }
  virtual bool do_compare( const uvm_object* rhs ) const { ... }

};
```

Transaction defined as sequence item

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

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)



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

UVM-SystemC sequence

```
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_objection(this);
  }

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

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

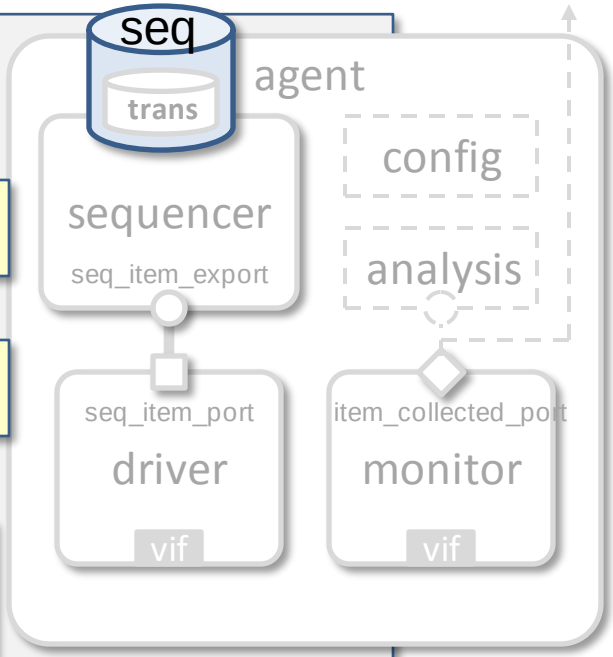
Factory registration supports template classes

Raise objection if there is no parent sequence

A sequence contains a request and (optional) response, both defined as sequence item

Compatibility layer to SCV or CRAVE not yet available

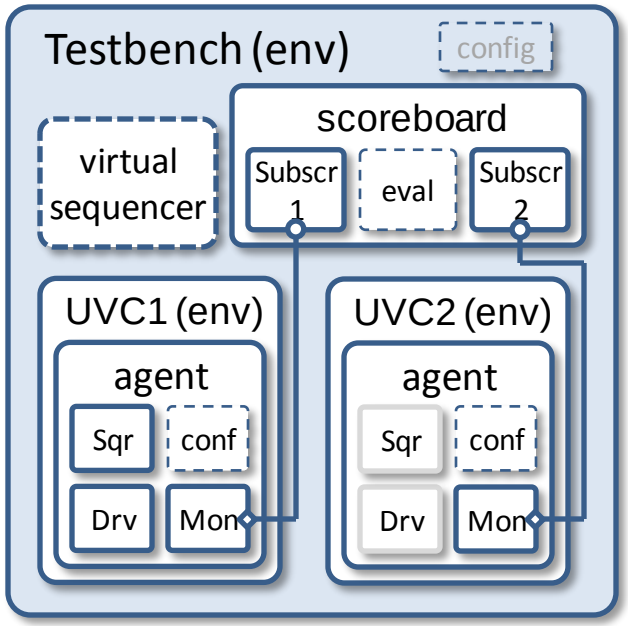
Optional: get response



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

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`

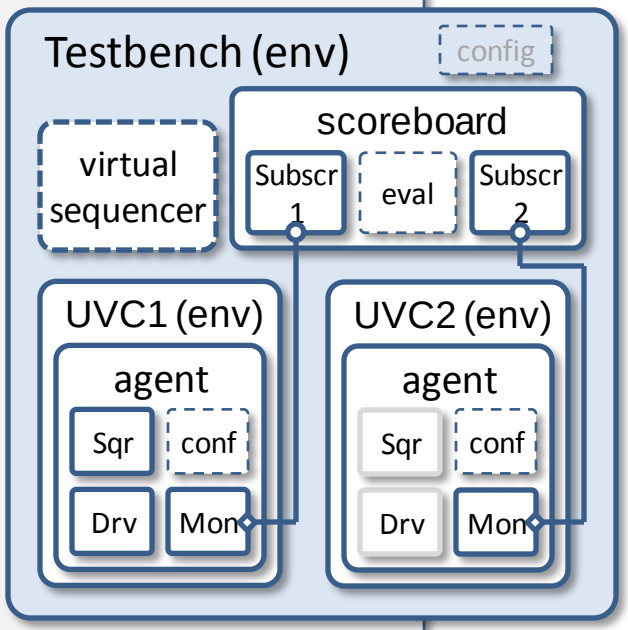


UVM-SystemC test bench

```
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 instantiated so they can be overidden by the test if needed

Definition of active or passive UVCs



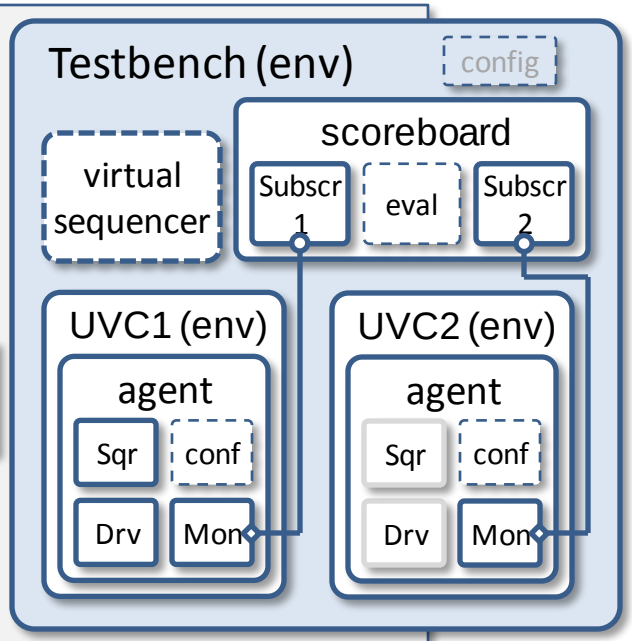
NOTE: UVM-SystemC API under review – subject to change

UVM-SystemC test bench

```
    ...
    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);
  }

};
```

Virtual sequencer points to UVC sequencer

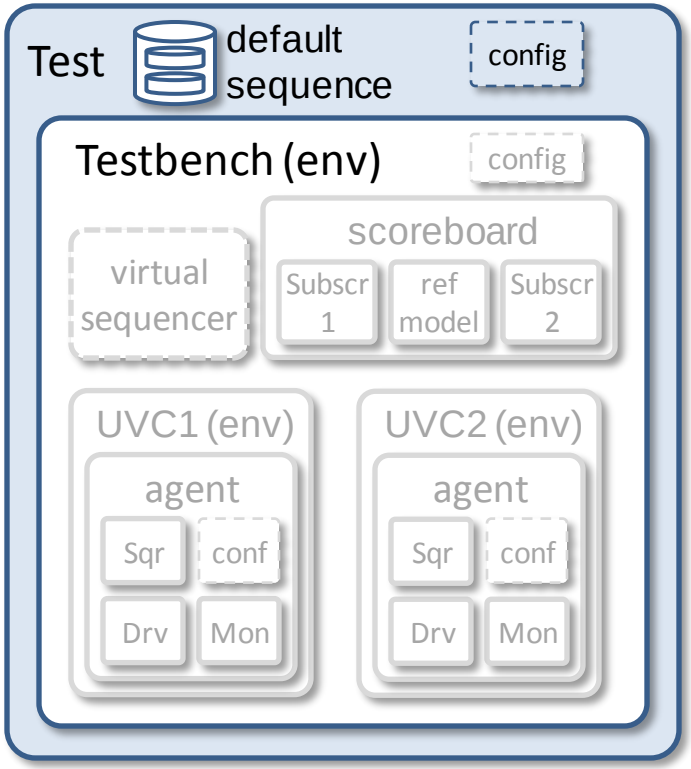
Analysis ports of the monitors are connected to the scoreboard subscribers (listeners)



NOTE: 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

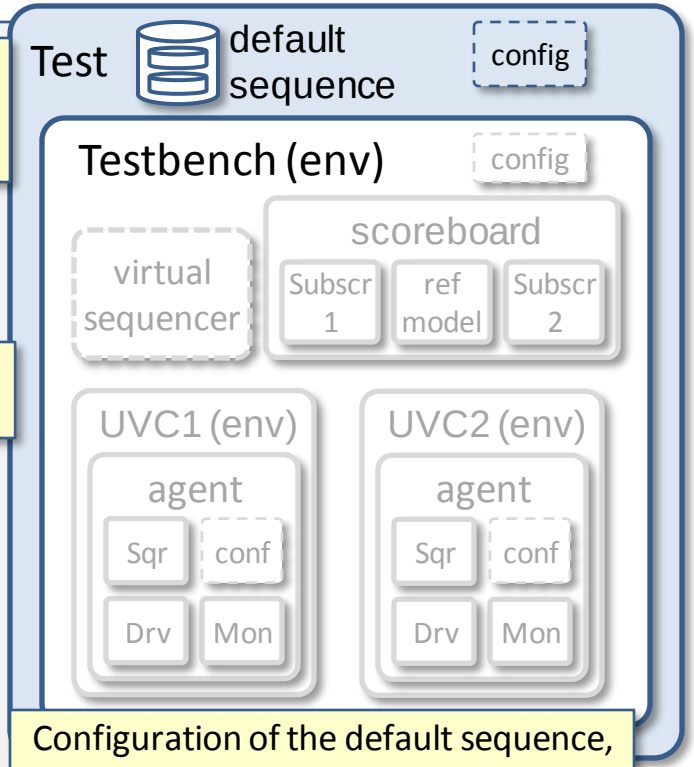
```
class test : public uvm_test
{
 public:
  testbench* tb;
  bool test_pass;

  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



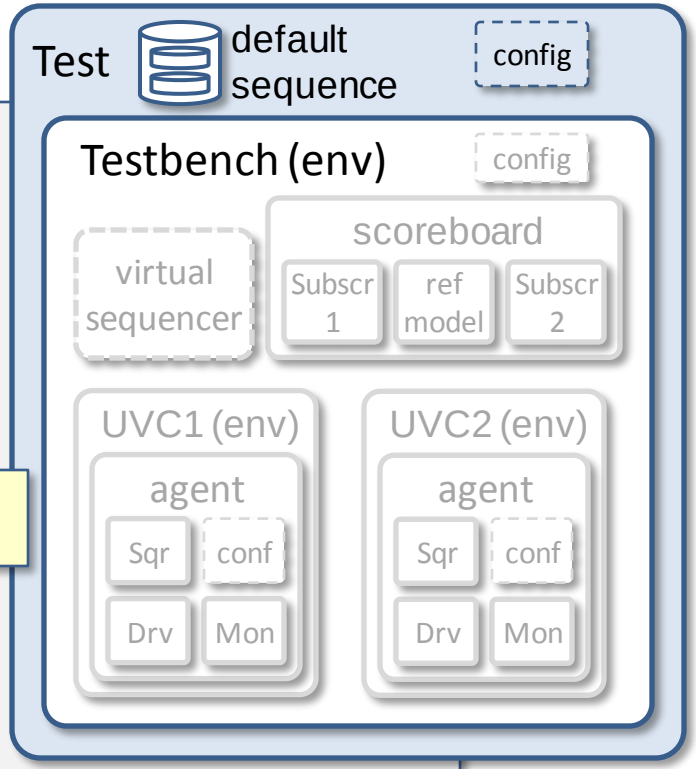
NOTE: UVM-SystemC API under review – subject to change

UVM-SystemC test

```
  ...

  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

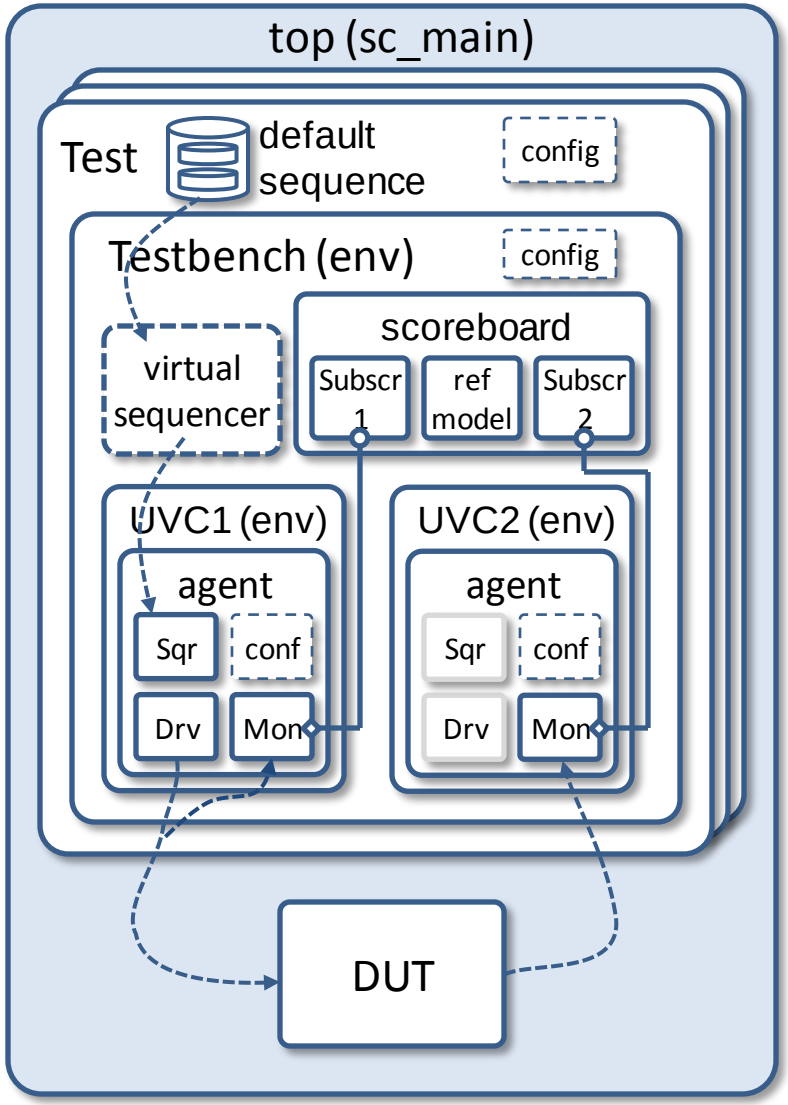
Report results in the report phase



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

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



The main program

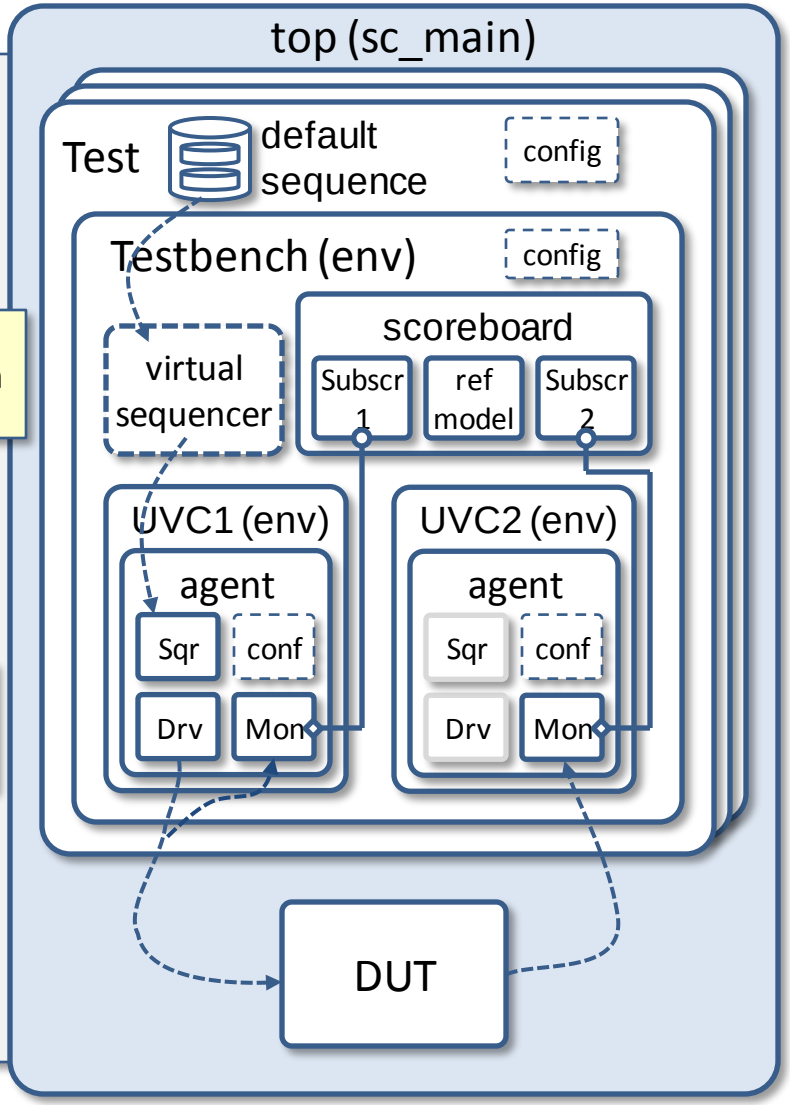
```
int sc_main(int, char*[])
{

  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;
}
```

Instantiate the DUT and interfaces

register interface using the configuration database

Connect DUT to the interface

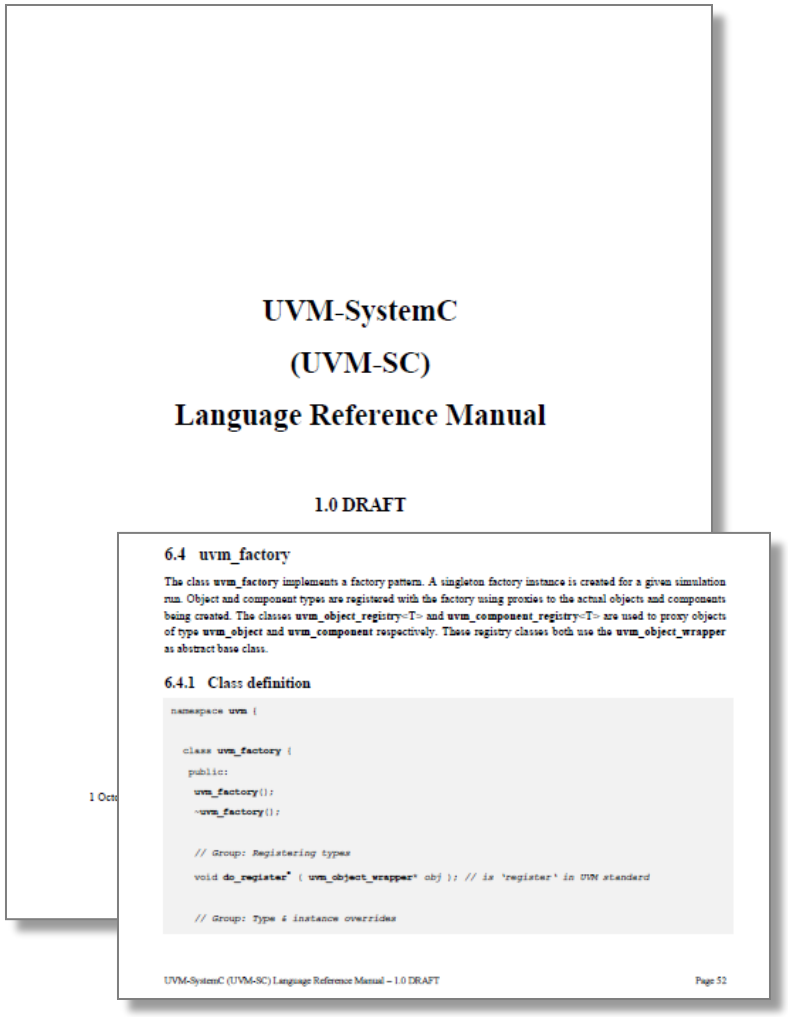
Register the test to be executed. This function also dynamically instantiates the test if given as argument



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

Contribution to Accellera

- Objective: seek further industry support and standardization of UVM in 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

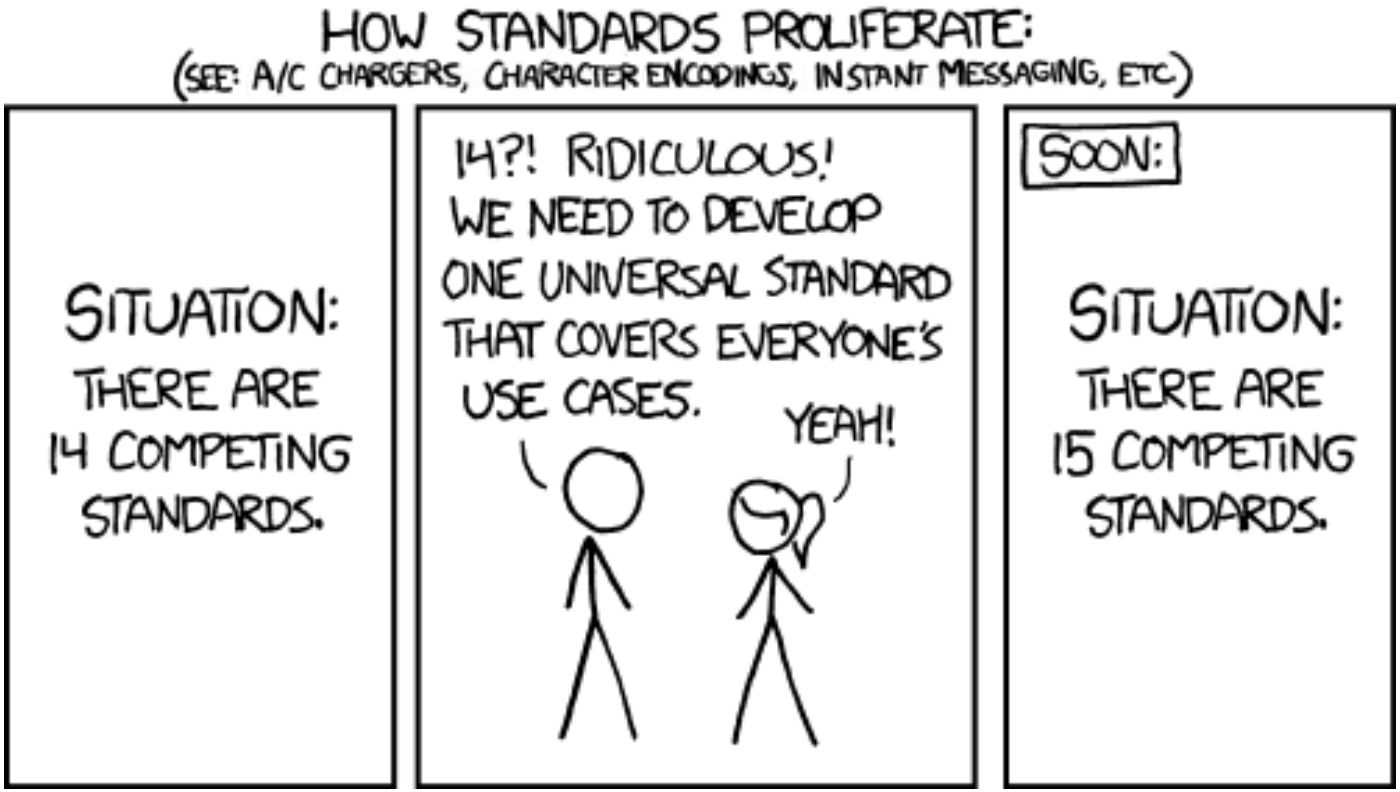


Summary and outlook

- 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
- 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
  - Introduce SystemC-AMS to support AMS system-level verification

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.


HOW STANDARDS PROLIFERATE:
(SEE: A/C CHARGERS, CHARACTER ENCODINGS, INSTANT MESSAGING, ETC)
SITUATION: THERE ARE 14 COMPETING STANDARDS.
14?! RIDICULOUS! WE NEED TO DEVELOP ONE UNIVERSAL STANDARD THAT COVERS EVERYONE'S USE CASES.
YEAH!
SOON:
SITUATION: THERE ARE 15 COMPETING STANDARDS.
xkcd.com