# UVM goes Universal - Introducing UVM in SystemC

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

Fraunhofer IIS

accellera SYSTEMS INITIATIVE

2015 DESIGN AND VERIFICATION™ DVCON CONFERENCE AND EXHIBITION 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, scoreboarding, 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

# Outline

- Part A – Introduction
- Part B – Examples and Applications
- Part C – Further steps & Outlook

# Outline

- Part A - Introduction
  - A bit of history…
  - Why UVM in SystemC?
  - Main concepts of UVM
  - Advantages of UVM-SystemC

# A bit of history…

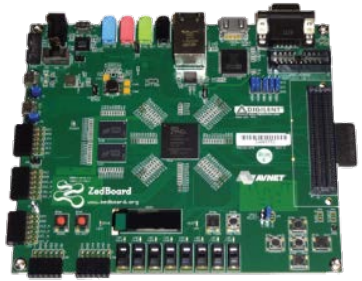
- In the pre-UVM era, various EDA vendors offered a verification methodology in SystemC
  - OVM-SC (Cadence), AVM-SC (Mentor), VMM-SC (Synopsys)
- Unfortunately, consolidation towards UVM focused on a SystemVerilog standardization and implementation only
- Non-standard methods and libraries exist to bridge the UVM and SystemC world
  - Cadence's UVM Multi Language library: offers a 'minimalistic' UVM-SystemC
  - 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?

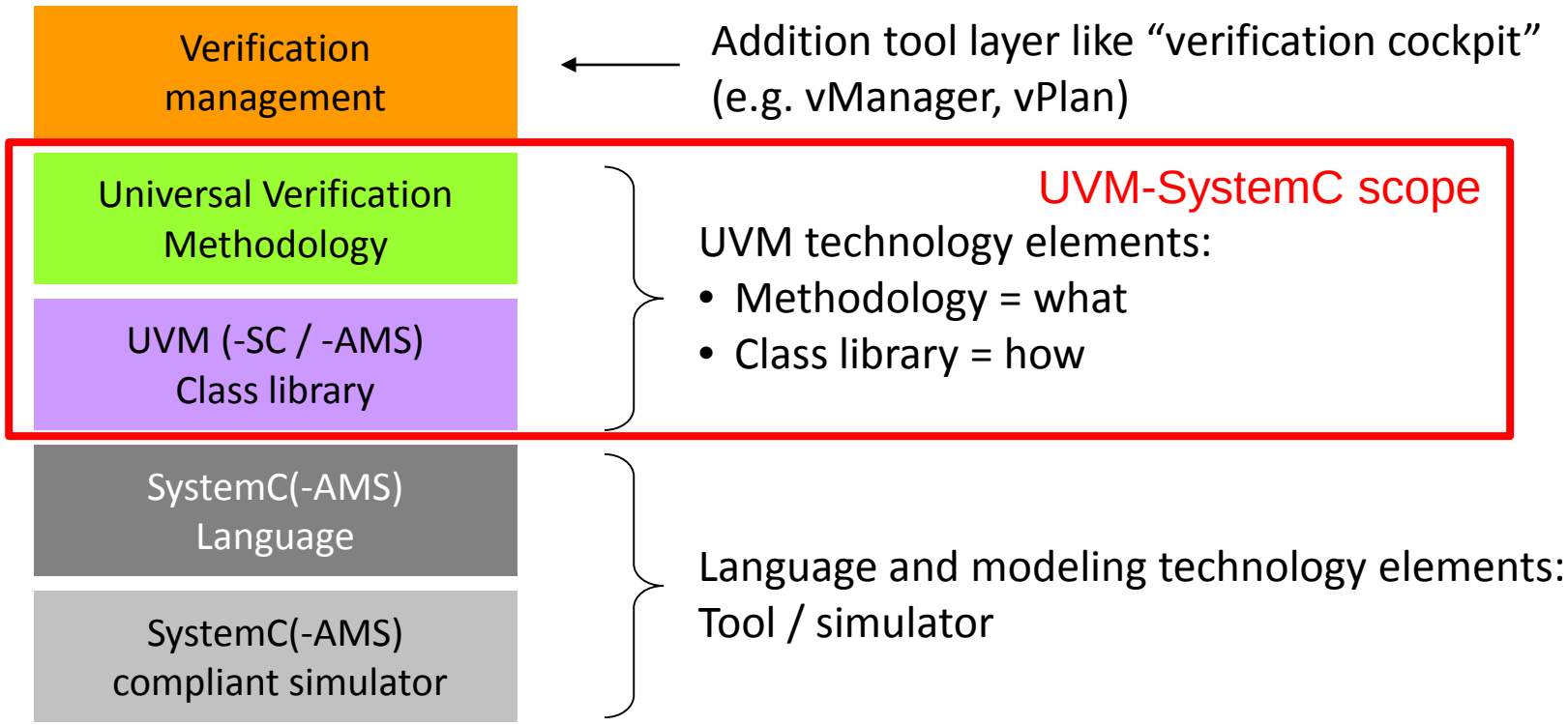
- Elevate verification **beyond block-level** towards **system-level**
  - **System verification** and **Software-driven verification** are executed by teams not familiar with SystemVerilog and its simulation environment
  - Trend: Tests coded in C or C++. System and SW engineers use an (open source) 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?

- Support analogue DUTs with SystemC AMS
- 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



# UVM in SystemC versus UV in SystemVerilog

- 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)

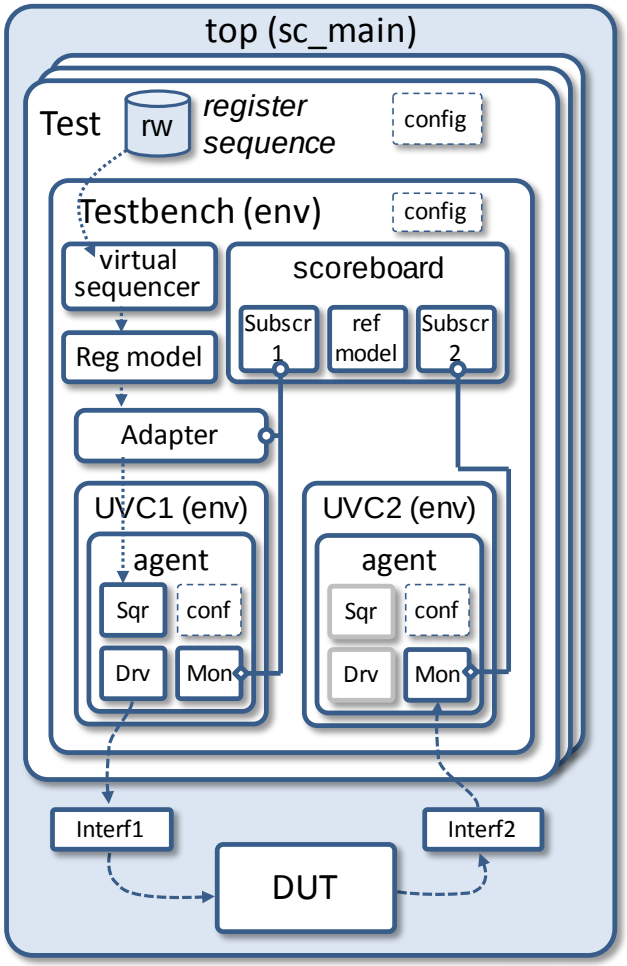
- Clear **separation** of test stimuli (sequences) and test bench
  - Sequences are treated as ‘transient objects’ and thus independent from the test bench construction and composition
  - In this way, sequences can be developed and reused independently
- Introducing test bench **abstraction levels**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

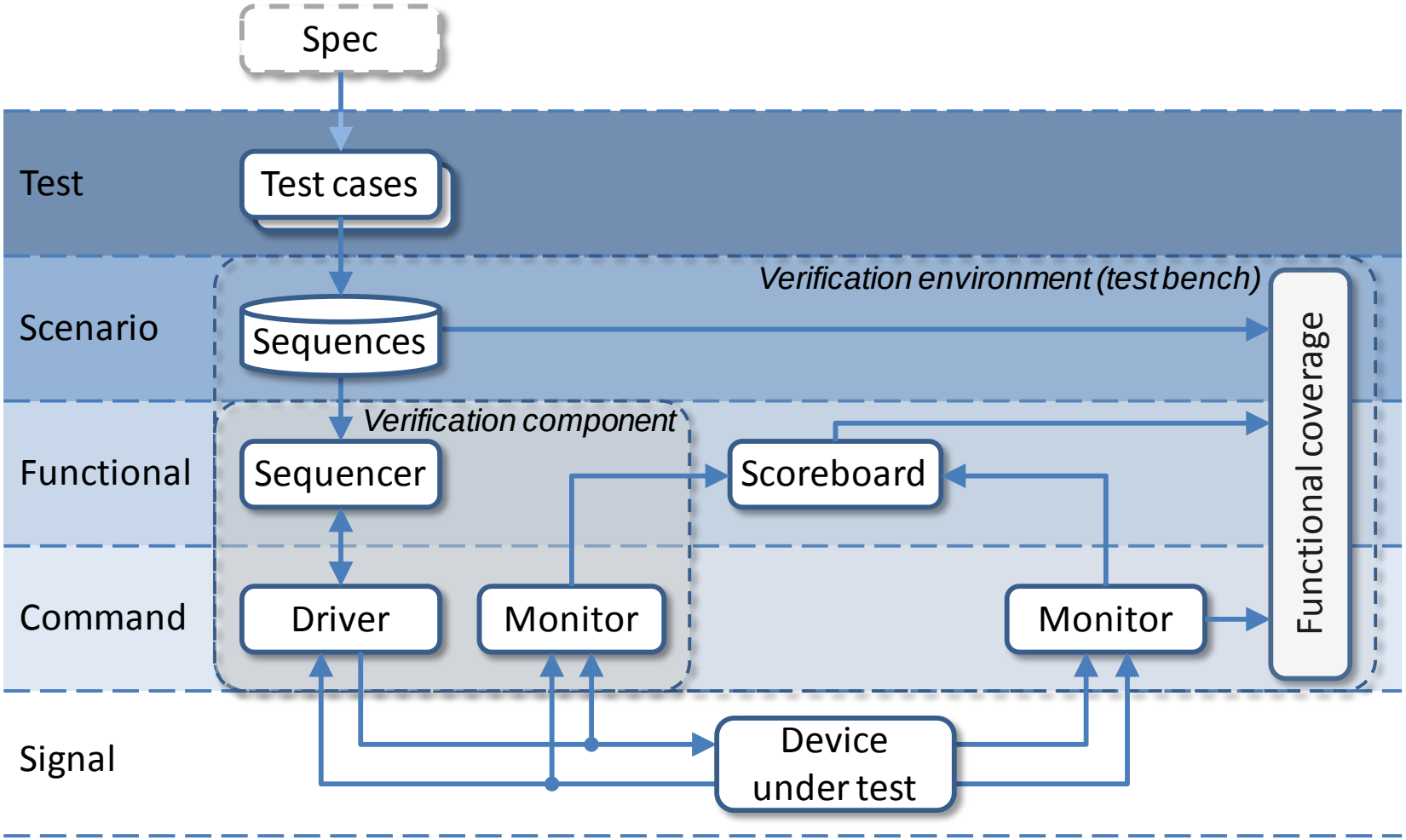
- Non-intrusive test bench **configuration** and **customization**
  - Hierarchy independent configuration and resource database to store and retrieve properties everywhere in the environment
  - Factory design pattern introduced to easily replace UVM components or objects for specific tests
  - User-defined 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



# 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

LRM definition

```
namespace uvm {
  class uvm_component : public sc_core::sc_module,
                        public uvm_report_object
  {
    uvm_component( uvm_component_name name);
    ...
  };
} // namespace uvm
```

Application

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




NOTE: UVM-SystemC API and LRM 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 import 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() const;
    virtual void connect( tlm::tlm_analysis_if<T>& _if );
    void write( const T& t );
  }; // class uvm_analysis_port

} // namespace uvm
```




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

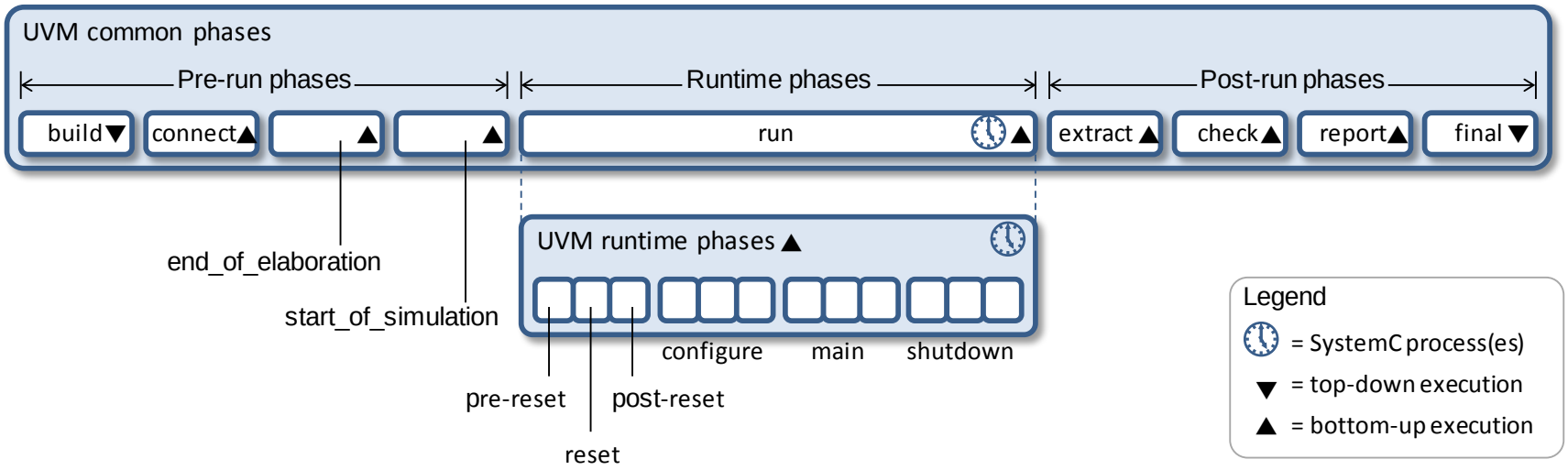
# SystemC TLM communication (2)

- 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)




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

# 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




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

# (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 and LRM 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

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 and LRM 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 uvm_report_object  {
    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 );
    ...
  }; // class uvm_object
} // namespace uvm
```

# 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…

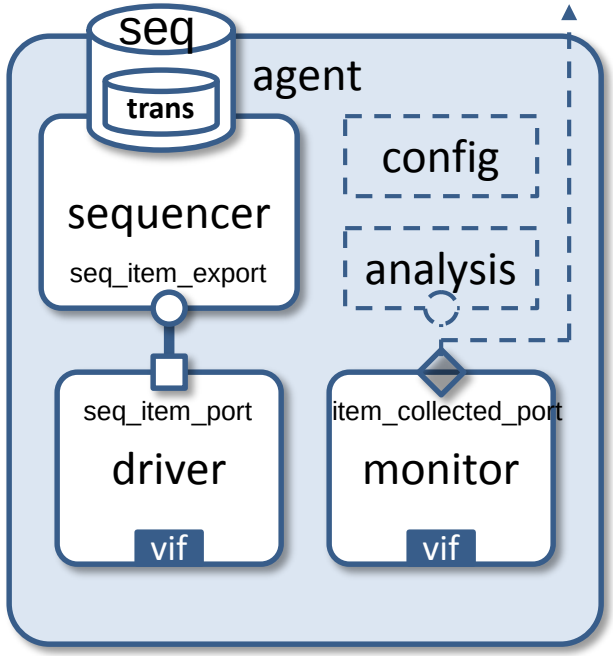
# Outline

- Part B – Examples and Applications
  - Components and Classes
  - Register Model
  - Abstraction re-use
  - Generator
  - Visualization

# UVM agents, drivers, and monitors

# UVM agent

- Component responsible to drive and monitor the DUT
- Typically contains three components
  - Sequencer
  - Driver
  - Monitor
- Could contain analysis functionality for basic coverage and checking



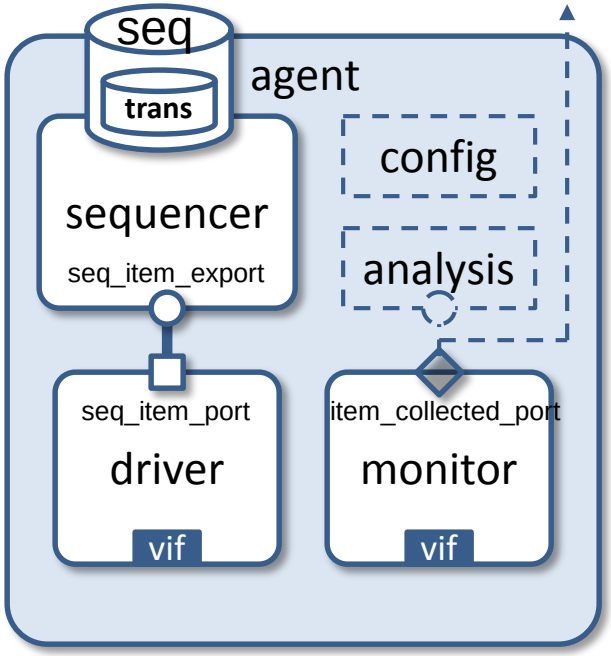




NOTE: UVM-SystemC API and LRM under review – subject to change

# UVM agent

- Possible configurations
  - Active agent: sequencer and driver are enabled
  - Passive agent: only monitors signals (sequencer and driver are disabled)
  - Master or slave configuration
- Base class: `uvm_agent`



# UVM agent – example -1

```
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_component_name name )
  : uvm_agent( name ), sequencer(0),
    driver(0), monitor(0) {}

  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);
  }
```

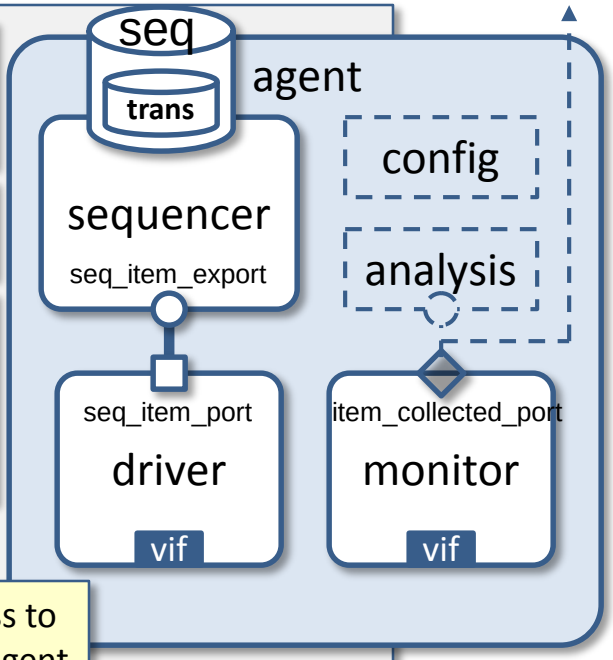
Dedicated base class to distinguish agents from other component types

Registers the object in the factory

Container for the string name and provides the mechanism for building the hierarchical names

Essential call to base class to access properties of the agent

Call to the factory which creates and instantiates this component dynamically



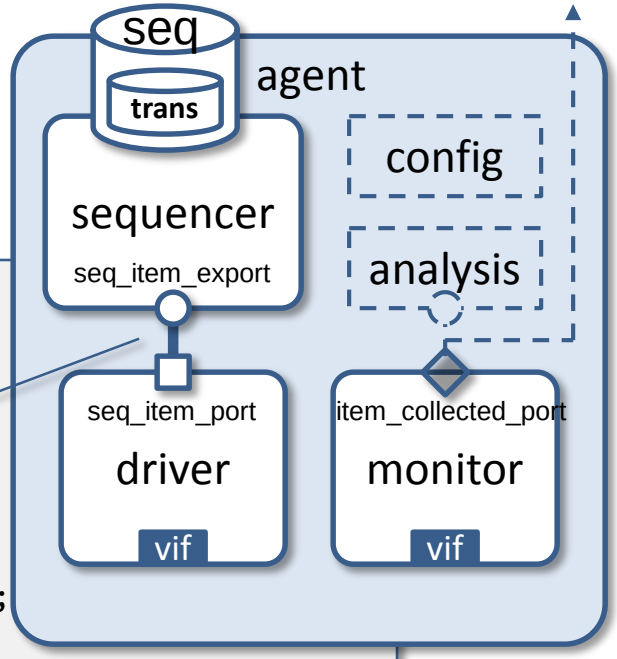




NOTE: UVM-SystemC API and LRM under review – subject to change

# UVM agent – example -2

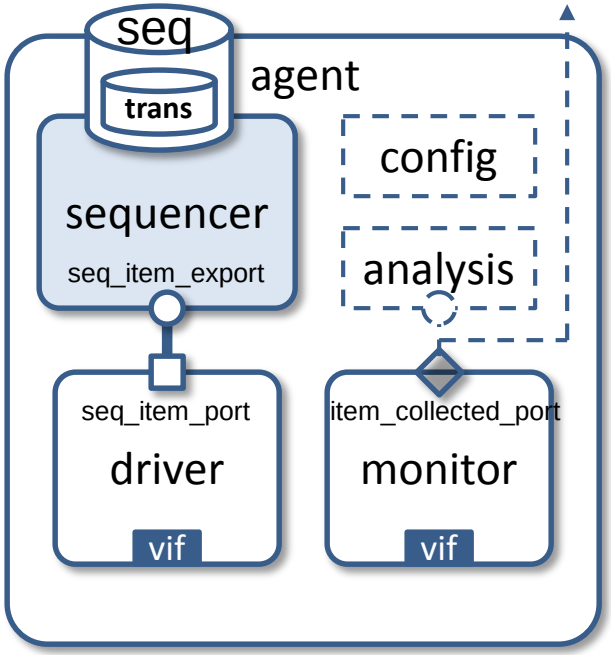
```
  ...

  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);
    }

  }
}; // class vip_agent
```




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

# UVM sequencer

- The sequencer controls and delivers transaction data items upon request of the driver*
- This allows to react to the current state of the DUT for every data item generated
- The UVM standard describes an interface between sequencer and driver that follows TLM (1.0) communication
- The sequencer serves as an arbiter for controlling transactions from multiple stimulus generators
- Base class: `uvm_sequencer`



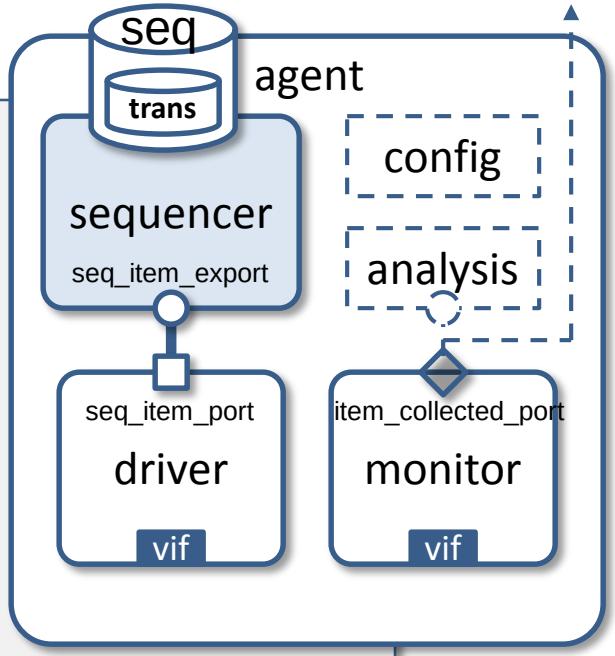

* Alternatively, there is a UVM push sequencer (class `uvm_push_sequencer`) which pushes the sequence items to the driver, but this is not yet available in UVM-SystemC

# UVM sequencer – example

Template parameter will be used for transaction type

```
template <class REQ>
class vip_sequencer : public uvm_sequencer<REQ>
{
 public:

  vip_sequencer( uvm_component_name name )
   : uvm_sequencer<REQ>( name ) {}

  UVM_COMPONENT_PARAM_UTILS(vip_sequencer<REQ>)

}; // class vip_sequencer
```

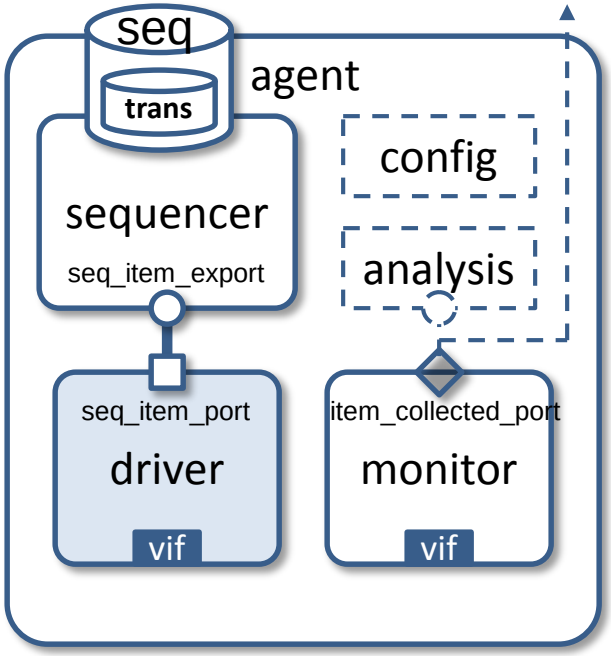
Factory registration: use '_PARAM_' macro in case template parameters are used. Note that the macro supports template classes




- The communication between sequence and sequencer is implemented in the base classes of the respective sequence and sequencer; the user / application does not have to deal with this

# UVM driver

- The driver is responsible to create the physical signals to drive the DUT
- For this, the driver repeatedly requests transactions, encapsulated in a sequence, via the sequencer, and translates these to one or more physical signal(s)
- Connection between the driver and the DUT is established by using a dedicated channel, which is made available via the configuration mechanism
- Base class: `uvm_driver`



# UVM driver – example -1

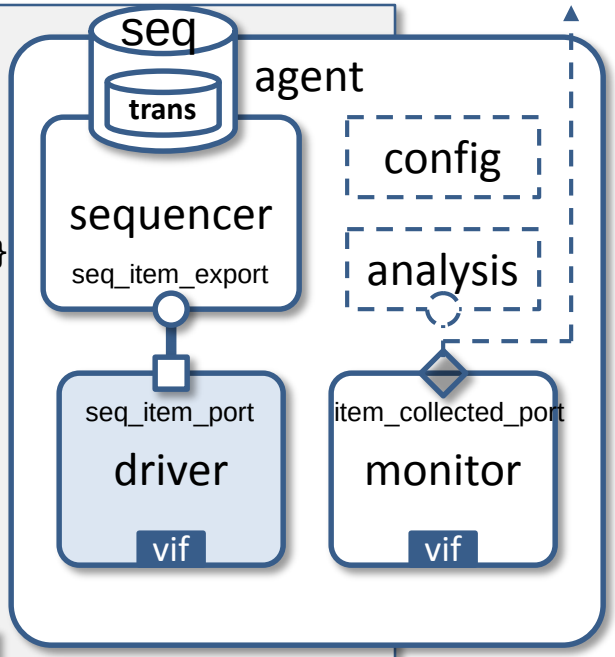
```
template <class REQ>
class vip_driver : public uvm_driver<REQ>
{
 public:
  vip_if* vif;

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

  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(),
        "Virtual interface not defined! Simulation aborted!")
  }

  void run_phase( uvm_phase& phase )
  {
    REQ req, rsp;
    while(true) // forever loop
    {
      this->seq_item_port->get_next_item(req);
      drive_transfer(req);
      rsp.set_id_info(req);
      this->seq_item_port->item_done();
      this->seq_item_port->put_response(rsp);
    }
  }
```

Placeholder needed to store the handle to the interface object

Registration of the template class

Get interface object using the configuration mechanism

Process transactions via the sequencer interface





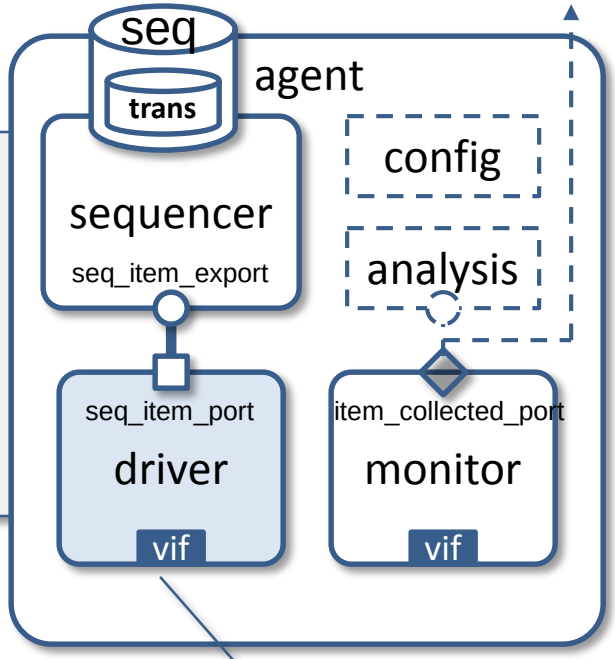


NOTE: UVM-SystemC API and LRM under review – subject to change

# UVM driver – example -2

```
  ...

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

}; // class vip_driver
```

Driver output signals are written to the interface directly



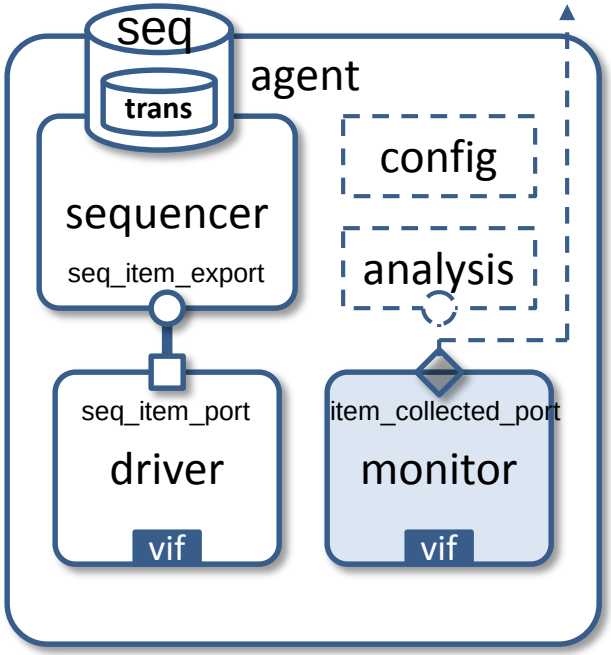

Connection to the DUT is established via the configuration mechanism

```
class vip_if
{
 public:
  sc_signal<int> sig_a;
  vip_if() : sig_a("sig_a") {}
};
```


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

# UVM monitor

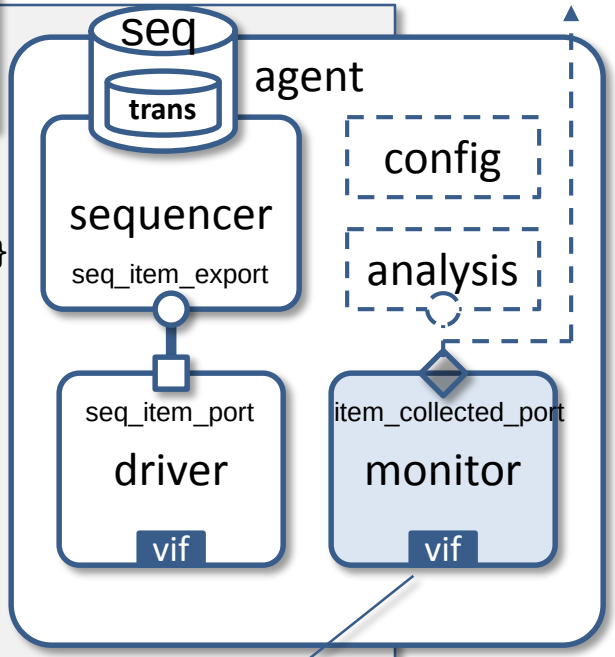
- The monitor is a passive element that ‘only’ captures the DUT signals
- It extracts signal information from the interface and translates this information to abstract transactions
- It will distribute this transaction to all connected elements for e.g. coverage collection and checking
- Connection between the monitor and the DUT is established by using a dedicated channel, which is made available via the configuration mechanism
- Base class: `uvm_monitor`



# UVM monitor – example

```
class vip_monitor : public uvm_monitor
{
 public:
  uvm_analysis_port<vip_trans> item_collected_port;
  vip_if* vif;

  vip_monitor( uvm_component_name name ): uvm_monitor( name ) {}

  UVM_COMPONENT_UTILS(vip_monitor)

  void build_phase( uvm_phase& phase )
  {
     uvm_monitor::build_phase(phase);
     if (!uvm_config_db<vip_if*>::get(this, "*", "vif", vif))
      UVM_FATAL(this->get_name(),
        "Virtual interface not defined! Simulation aborted!")
  }

  void run_phase( uvm_phase& phase )
  {
    vip_trans t;
    while(true) // monitor f
    {
      wait( vif->sig_data.default_event() );
      ...
      t.data = vif->sig_a.read();
      item_collected_port.write(t);
    }
  }
}; // class vip_monitor
```

Analysis port used to pass collected data to attached components




Example: wait for input changes in case there is no clock

Connection to the DUT is established via the configuration mechanism

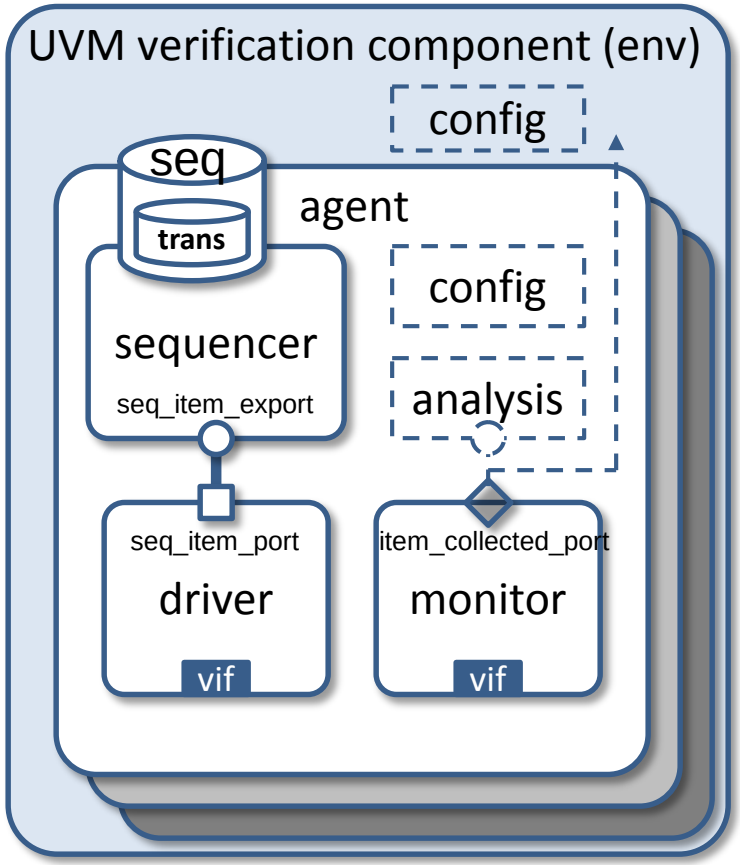
```
class vip_if
{
 public:
  sc_signal<int> sig_a;
  vip_if() : sig_a("sig_a") {}
};
```




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

# UVM verification component (UVC)

- A reusable verification component (UVC) is a (sub-) environment which consists of one or more agents
- The verification component or agents may set or get configuration parameters
- An independent sequence, which contains the actual transaction data, is processed by the driver via a sequencer
- Each verification component is connected to the DUT using a dedicated interface
- Base class: `uvm_env`







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

# UVC – example

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

  UVM_COMPONENT_UTILS(vip_uvc);

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

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

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

}; // class vip_uvc
```

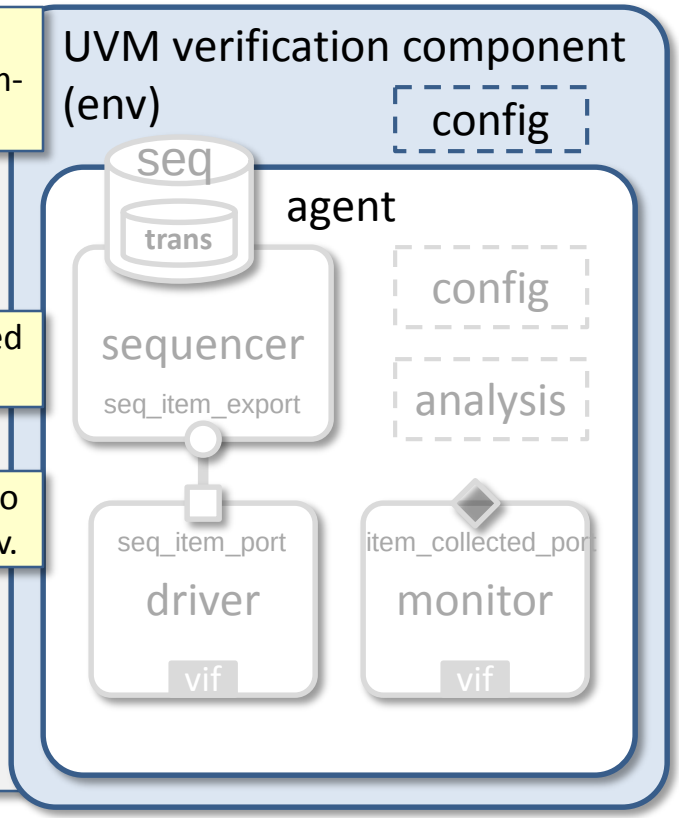
A UVC is treated as a sub-environment in large system-level environments

Registers the object in the factory

Components are instantiated in the build phase

Essential call to base class to access properties of the env.

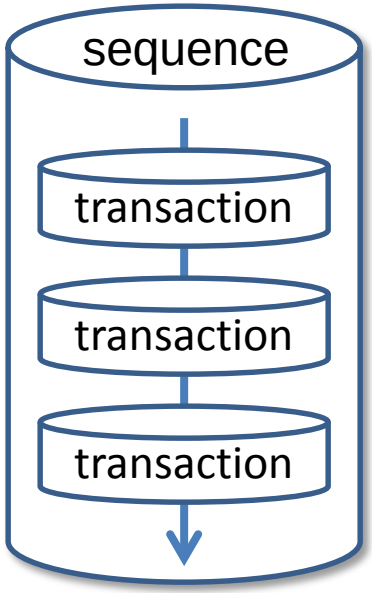
Call to the factory which creates and instantiates this component dynamically



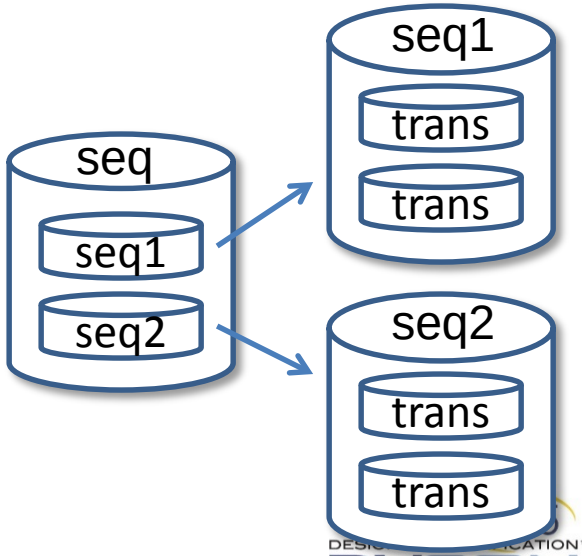

- In this example, the UVM verification component (UVC) contains only one agent. In practice there will be more agents instantiated

# UVM sequences and sequencers

# 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 testbench 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

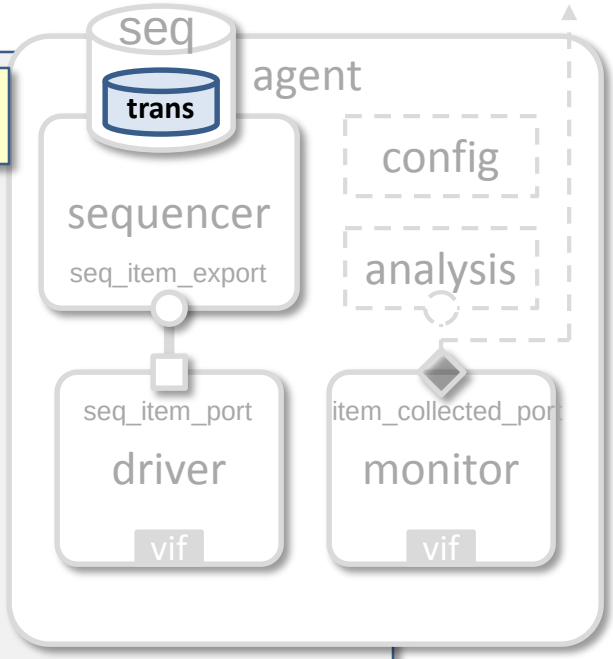


# UVM sequence item – example

```
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 { ... }

}; // class vip_trans
```

Inherits from uvm_transaction

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 and LRM under review – subject to change

# UVM sequence – example

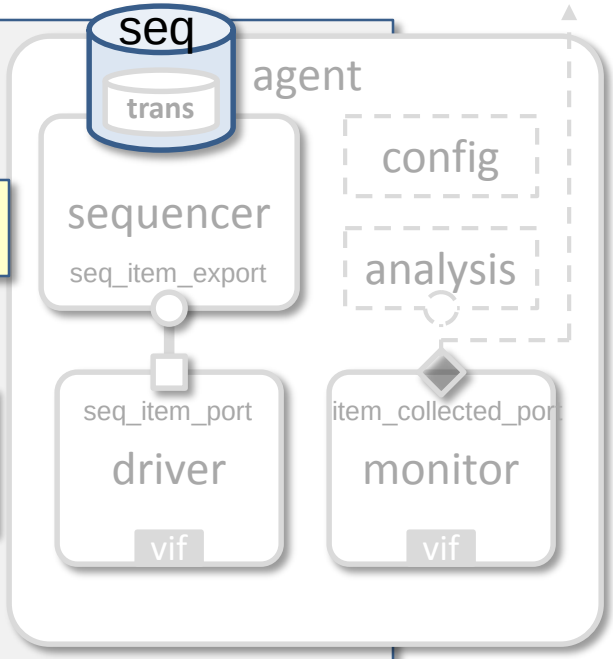
```
template <typename REQ = uvm_sequence_item, typename RSP = REQ>
class sequence : public uvm_sequence<REQ,RSP>
{
 public:
  sequence( const std::string& name )
    : uvm_sequence<REQ,RSP>( name ) {}

  UVM_OBJECT_PARAM_UTILS(sequence<REQ,RSP>);

  void body()
  {
    REQ* req;
    RSP* rsp;

    for(int i = 0; i < NUM_TRANS; i++)
    {
      req = new REQ();
      rsp = new RSP();
      ...
      start_item(req);
      // req->randomize();
      finish_item(req);
      get_response(rsp);
    }
  }
}; // class sequence
```

Factory registration supports template classes

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

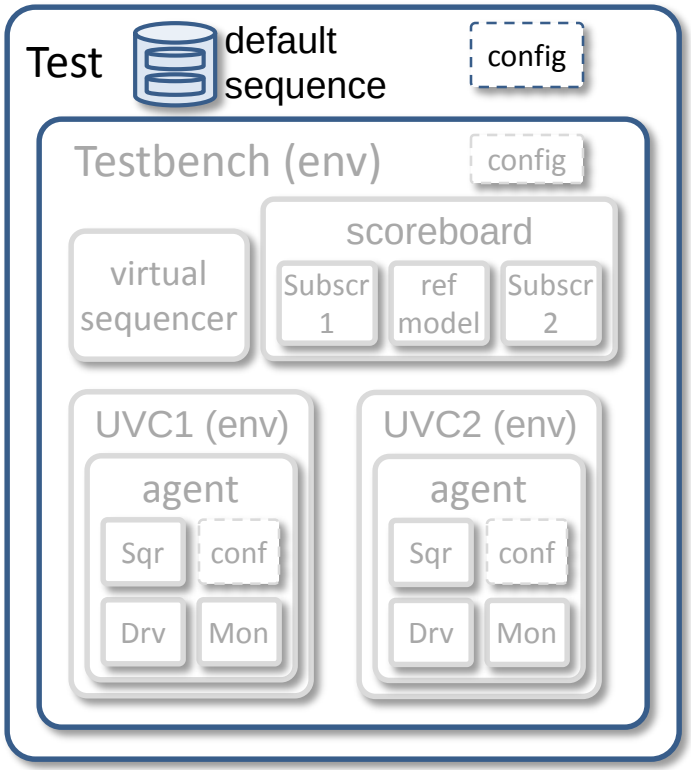
Compatibility layer to SCV or CRAVE not yet implemented

Optional: get response



# UVM virtual sequence

- A virtual sequence encapsulates one or more sequences, which are executed on the sub-sequencers in each UVC agent, which are all connected to the parent virtual sequencer
- A virtual sequence can be configured as *default sequence* in a test, to facilitate automatic execution on a virtual sequencer or a sequencer which belongs to a UVC agent
- Base class: `uvm_sequence` (same as 'normal' sequences)







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

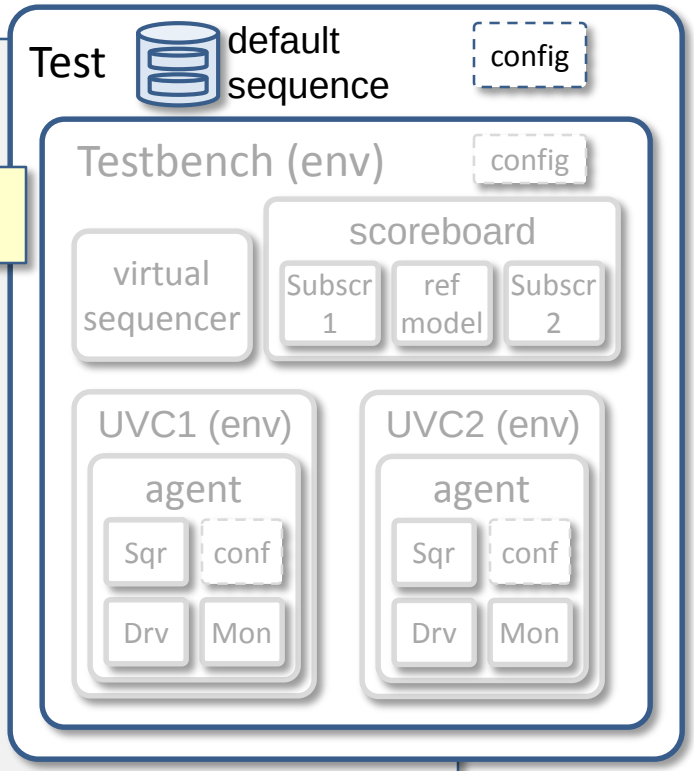
# UVM virtual sequence – example

```
class virt_sequence : public uvm_sequence<>
{
 public:
  vip_sequence<vip_trans>* vip_seq;

  virt_sequence( const std::string& name = "virt_sequence" )
  : uvm_sequence<>( name ) {}

  UVM_OBJECT_UTILS(virt_sequence)

  UVM_DECLARE_P_SEQUENCER(virt_sequencer)

  void body()
  {
    UVM_INFO( get_name(),
              "Virtual sequence starts here...", UVM_NONE)

    UVM_DO_ON(vip_seq, p_sequencer->vip_seqr)

    UVM_INFO( get_name(),
              "Virtual sequence finished.", UVM_NONE);
  }
}; // class virt_sequence
```

Use of same base class, without parameters

Declaration of the parent sequencer

Macro to start sequence on a specific sequencer using its member function **start**

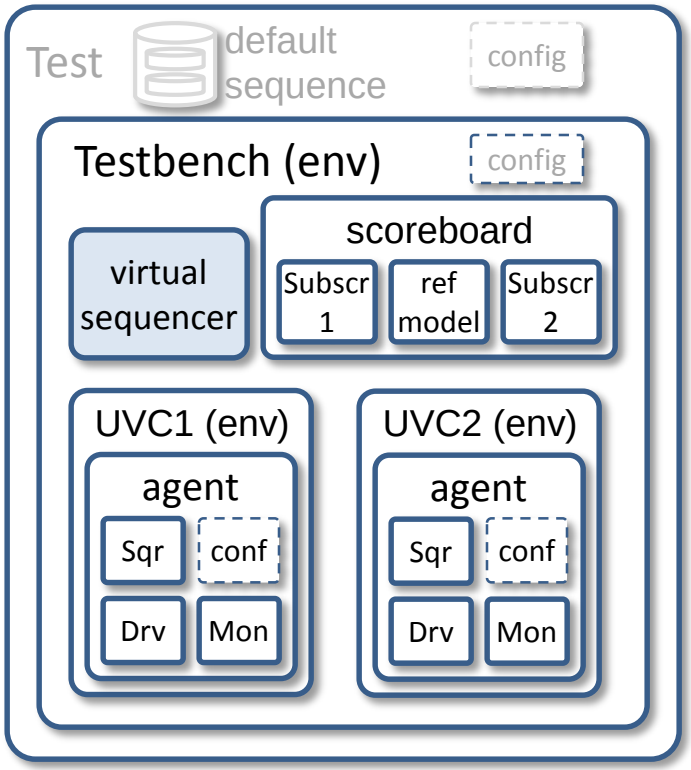




NOTE: UVM-SystemC API and LRM under review – subject to change

# UVM virtual sequencer

- A virtual sequencer contains references to its subsequencers such as UVC sequencers or other virtual sequencers
- Virtual sequencers process virtual sequences which encapsulate sequences for multiple verification components
- Virtual sequencers do not execute transactions on themselves but 'offload' this to its subsequencers
- Base class: `uvm_sequencer` (same as 'normal' sequencers)



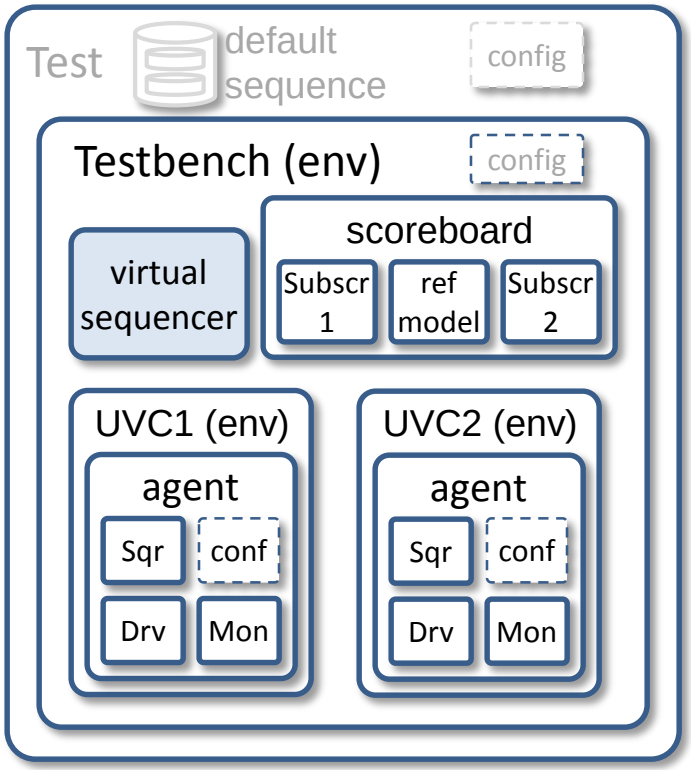




NOTE: UVM-SystemC API and LRM under review – subject to change

# UVM virtual sequencer

- Similar as with the sequencer in an agent, the communication between the virtual sequence and virtual sequencer is implemented in the base classes and therefore the application does not have to deal with this







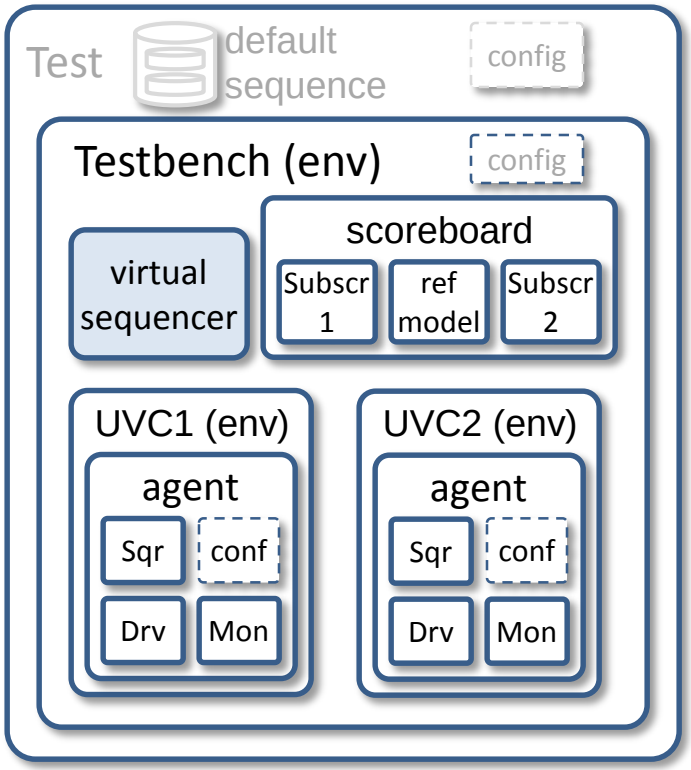
NOTE: UVM-SystemC API and LRM under review – subject to change

# UVM virtual sequencer – example

As the virtual sequencer does not process transactions itself, we do not specify a template parameter

```
class virt_sequencer : public uvm_sequencer<>
{
 public:

  vip_sequencer<vip_trans>* vip_seqr;

  UVM_COMPONENT_UTILS(virt_sequencer)

  virt_sequencer( uvm_component_name name )
  : uvm_sequencer<>( name ) {}

}; // class virt_sequer
```

Placeholder to associate one subsequencer to this virtual sequencer



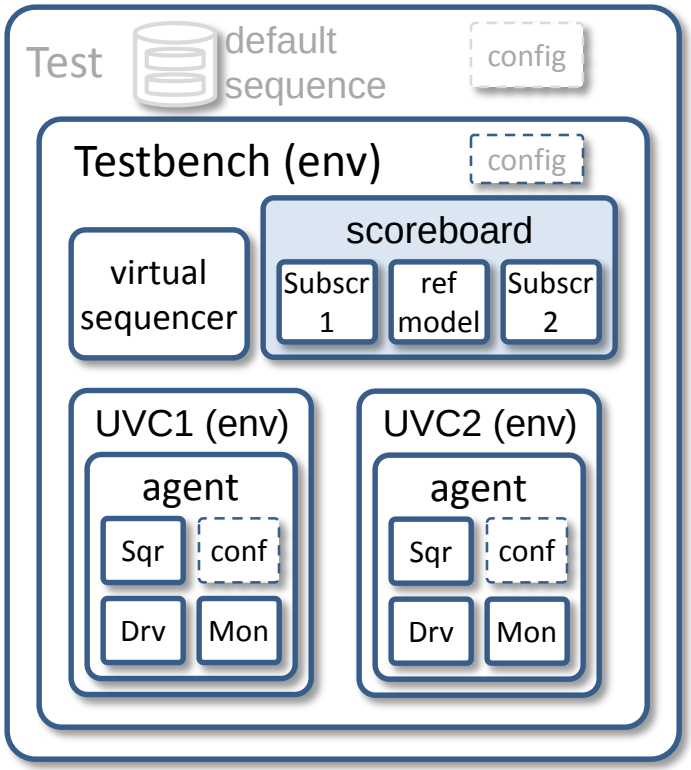


NOTE: UVM-SystemC API and LRM under review – subject to change

# UVM scoreboard and subscribers

# UVM scoreboard

- The scoreboard performs *end-to-end* checking by comparing expected and processed transactions
- These transactions are retrieved by dedicated *subscribers* or *listeners*, which implement the **write** method of the analysis ports of each monitor, to which these subscribers are connected
- A scoreboard may contain a predictor, which acts as reference or golden model. Alternatively, the scoreboard may contain an algorithm to calculate the expected transaction
- Base class: `uvm_scoreboard`





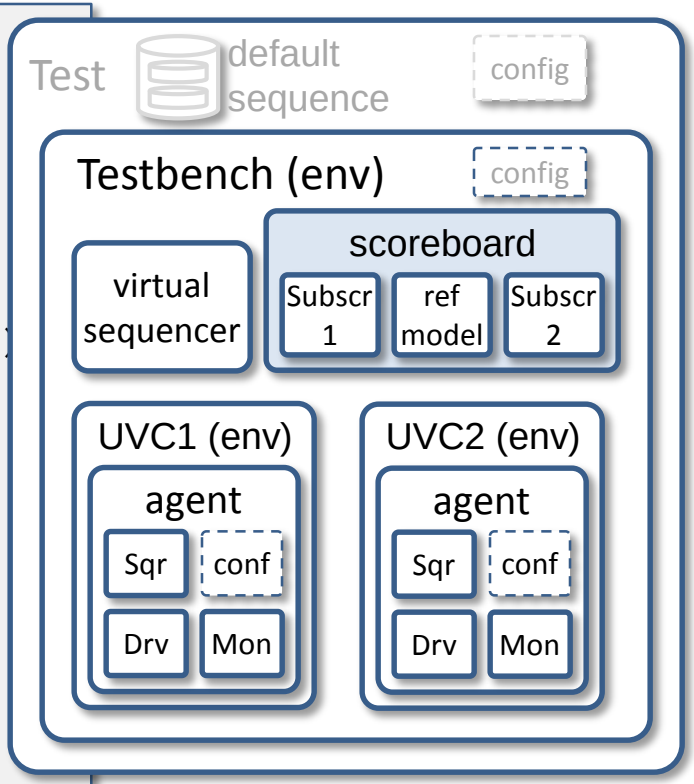


NOTE: UVM-SystemC API and LRM under review – subject to change

# UVM scoreboard – example

```
class scoreboard : public uvm_scoreboard
{
 public:
  uvm_analysis_export<vip_trans> listener1_imp;
  uvm_analysis_export<vip_trans> listener2_imp;
  subscriber1* subscr1;
  subscriber2* subscr2;

  scoreboard( uvm_component_name name ): uvm_scoreboard(name)

  UVM_COMPONENT_UTILS(scoreboard)

  void build_phase( uvm_phase& phase )
  {
    uvm_scoreboard::build_phase(phase);
    subscr1 = subscriber1::type_id::create("subscr1", this);
    subscr2 = subscriber2::type_id::create("subscr2", this);
    ...
  }

  void connect_phase( uvm_phase& phase )
  {
    listener1_imp(subscr1->analysis_export);
    listener2_imp(subscr2->analysis_export);
  }
  void write_listener1(const vip_trans& t) { ... }
  void write_listener2(const vip_trans& t) { ... }
};
```

Exports used to connect to the subscribers




Via the subscribers the expected and the processed transactions become available which are used for the actual checking




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

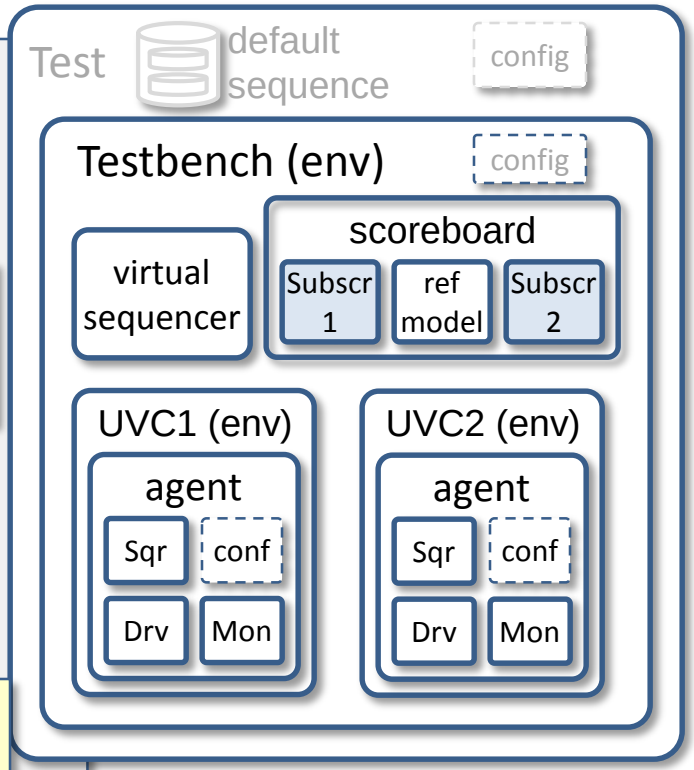
# UVM subscriber – example

```
class subscriber1: public uvm_subscriber
{
 public:

  subscriber1(uvm_component_name name): uvm_subscriber(name)

  UVM_COMPONENT_UTILS(subscriber1)

  void write(const vip_trans& t)
  {
    uvm_object* obj;
    scoreboard* sb;

    uvm_config_db<scoreboard*>::get(0, "", “sb", sb);
    assert(sb);


    sb->write_listener1(t);
  }
};
```

Implementation of the write method for the export in the monitor

The scoreboard in which the subscriber is used is retrieved via the configuration mechanism

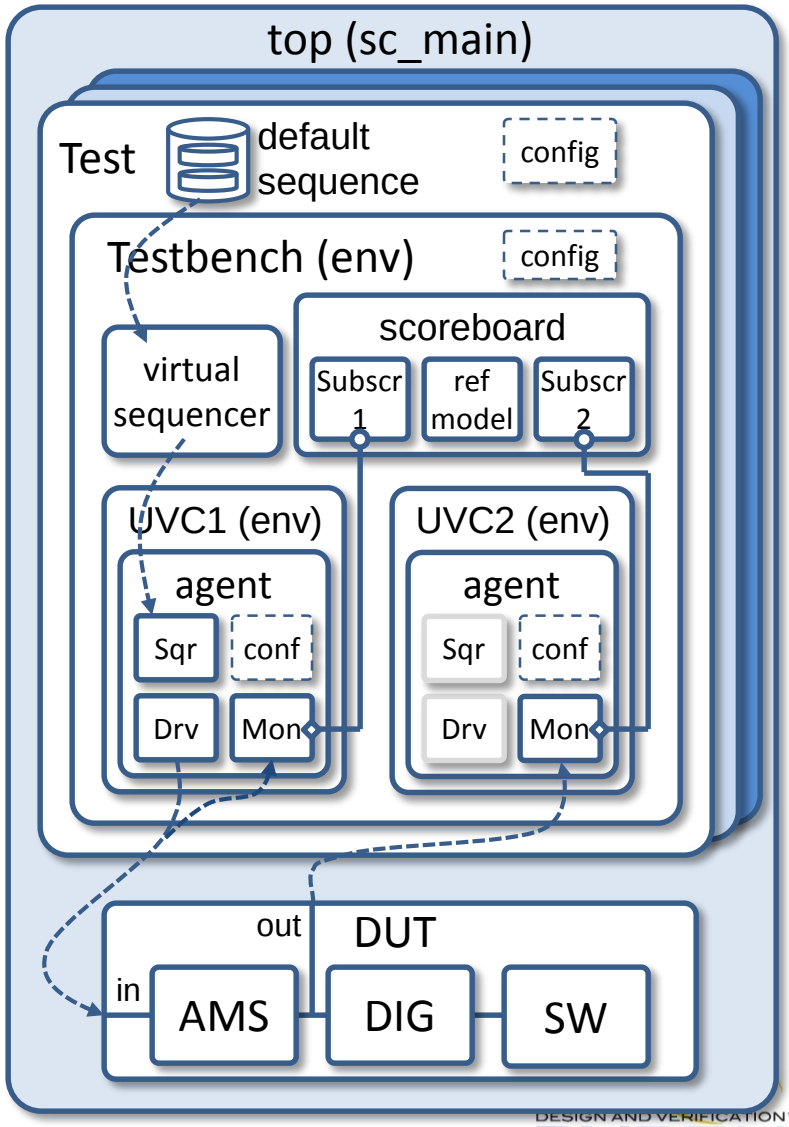
Transactions are passed to the scoreboard



# UVM top, test and testbenches

# Top, Tests and Testbench

- 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`







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

# Top – example

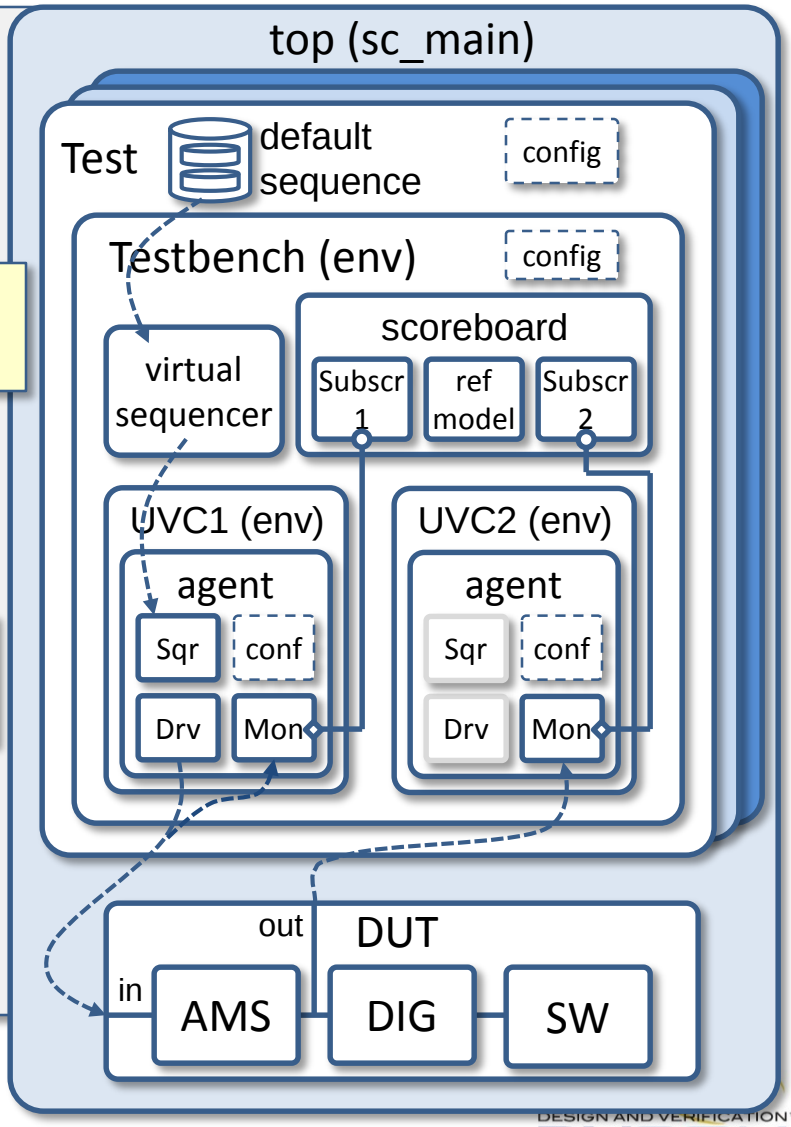
```
int sc_main(int, char*[])
{

  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;

}
```

Instantiate the DUT and interfaces

register interface using the configuration database

Connect DUT to the interface

Dynamically instantiates the test if given as argument and start it

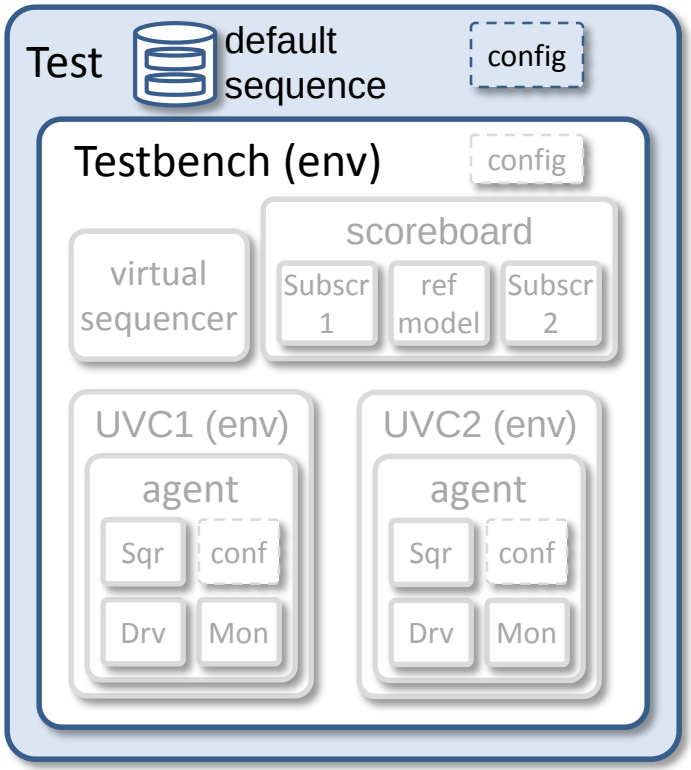




NOTE: UVM-SystemC API and LRM under review – subject to change

# UVM test

- Each UVM test is defined as a dedicated test class, which instantiates the testbench and defines the test sequence(s)
- Reuse of tests and topologies is possible by deriving tests from a test base class
- The configuration and factory concept can be used to configure or override UVM components, sequences or sequence items
- Tests can be selected (passed) as command line option*
- Base class: `uvm_test`




* Not yet available in UVM-SystemC

# UVM test – example

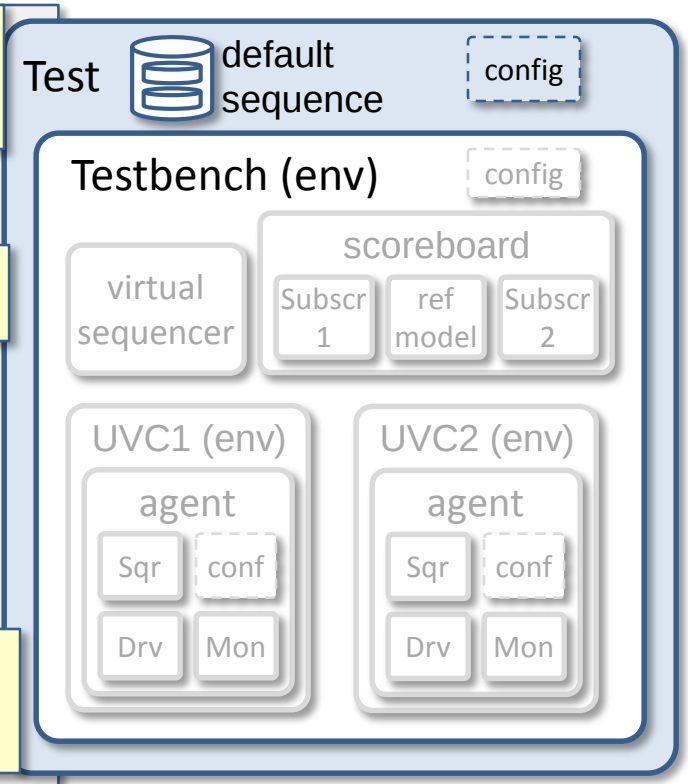
```
class test : public uvm_test
{
 public:
  testbench* tb;
  test( uvm_component_name name ) : uvm_test( name ) {}

  UVM_COMPONENT_UTILS(test)

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

    uvm_config_db<uvm_object_wrapper*>::set( this,
      "tb.virtual_sequencer.run_phase",
      "default_sequence“, virt_sequence::type_id::get() );
  }

  void run_phase( uvm_phase& phase )
  {
    UVM_INFO( get_name(), "** UVM TEST STARTED **", UVM_NONE )
  }

  void report_phase( uvm_phase& phase )
  {
    UVM_INFO( get_name(), "** UVM TEST ENDED **", UVM_NONE )
  }
};
```

Specific class to identify the test objects for execution using the method **run_test**

The test instantiates the required testbench

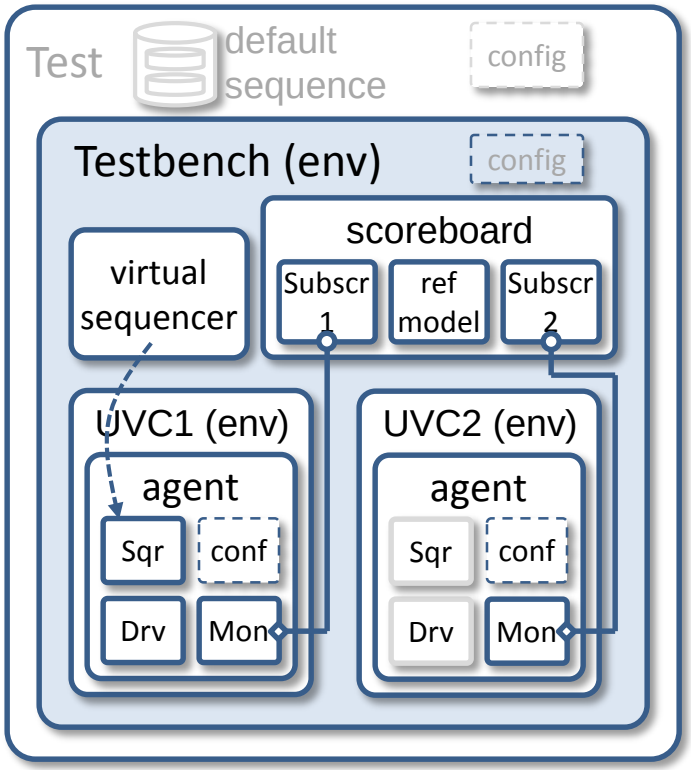
Configuration of virtual sequence for virtual sequencer

Reporting macros for information, warnings, errors and fatal errors are available



# UVM testbench

- A testbench is defined as the complete environment which *instantiates* and *configures* the UVCs, scoreboard, and virtual sequencer if available
- The UVCs are sub-environments in a testbench
- The testbench only makes the connections between the scoreboard and virtual sequencer to each UVC; the connection between UVCs and the DUT is arranged within the UVCs





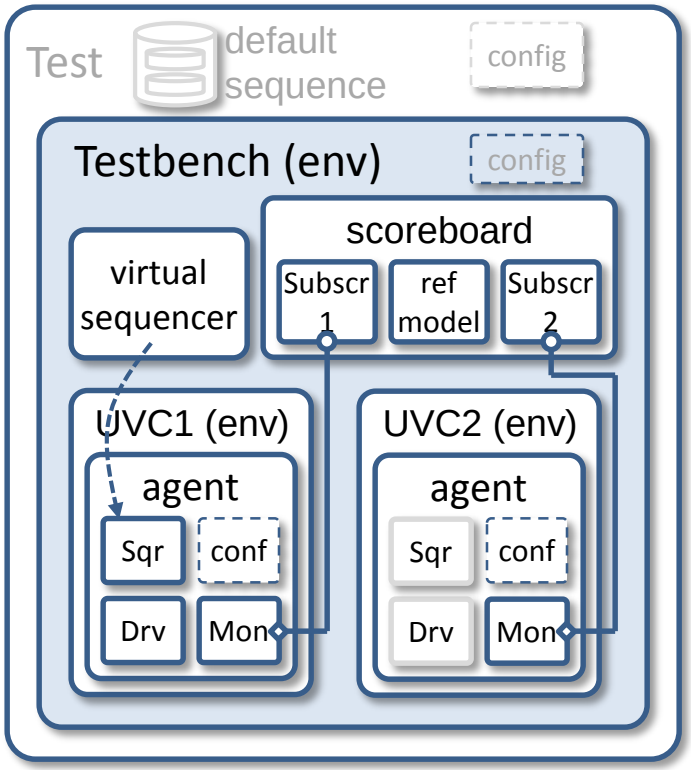


NOTE: UVM-SystemC API and LRM under review – subject to change

# UVM test bench – example -1

```
class testbench : public uvm_env
{
 public:
  vip_uvc*         uvc1;
  vip_uvc*         uvc2;
  virt_sequencer*  virtual_sequencer;
  scoreboard*      scoreboard1;

  UVM_COMPONENT_UTILS(testbench);

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

  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 overridden by the test if necessary



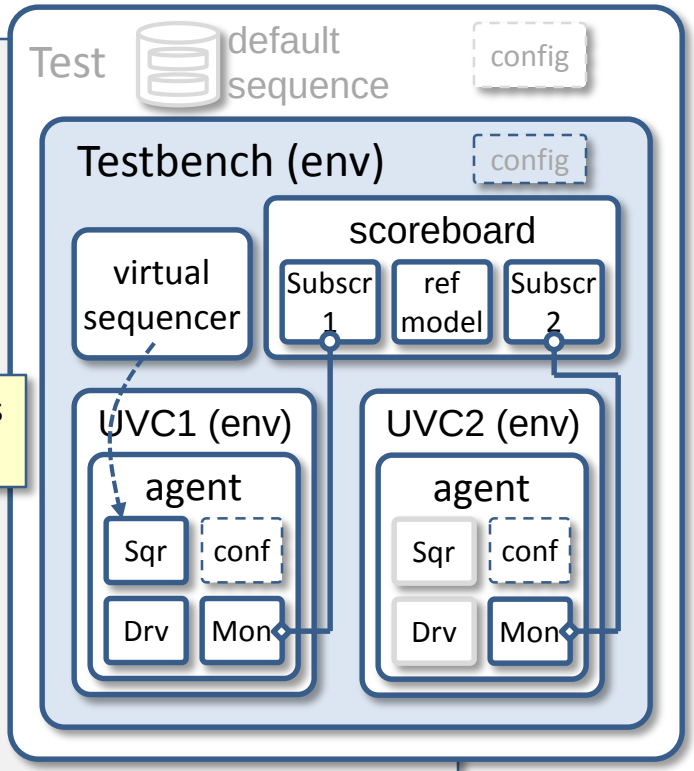

Definition of active or passive UVCs

# UVM test bench – example -2

```
    ...
    virtual_sequencer = virt_sequencer::type_id::create(
                          "virtual_sequencer", this);
    assert(virtual_sequencer);

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


  void connect_phase( uvm_phase& phase )
  {
    virtual_sequencer->vip_seqr = uvc1->agent->sequencer;

    uvc1->agent->monitor->item_collected_port.connect(
      scoreboard1->listener1_imp);

    uvc2->agent->monitor->item_collected_port.connect(
      scoreboard1->listener2_imp);
  }

}; // class testbench
```

Virtual sequencer points to UVC sequencer

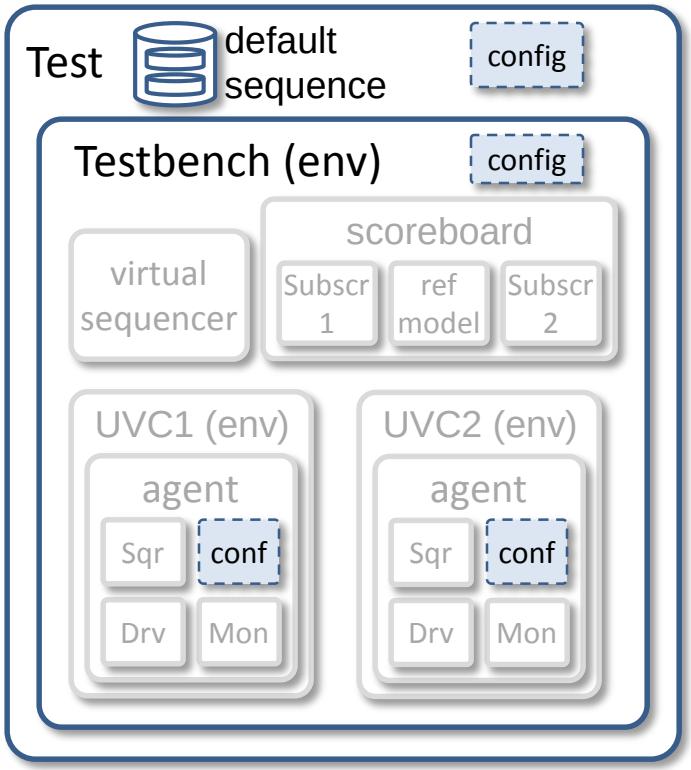
Analysis ports of the monitors are connected to the scoreboard subscribers (listeners)



# UVM configuration customization

# UVM configuration mechanism

- Central resource database to store and retrieve any type specific information of UVM and non-UVM objects at *any place* in the verification environment
- Configuration is facilitated during the build process and/or run time
- Information can be accessed by name (string) or arbitrary type
- Scope (context) of accessibility of information can be defined by the application
- Easy access to resource database via the configuration mechanism `uvm_config_db`
- Base class: `uvm_resource`



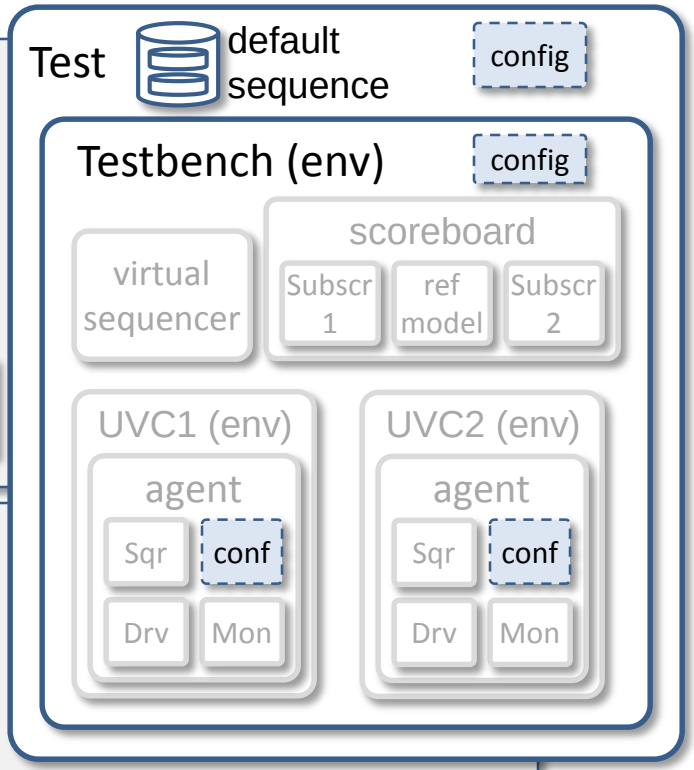


NOTE: UVM-SystemC API and LRM under review – subject to change

# UVM configuration – example -1

```
int sc_main(int, char*[])
{
  my_env* topenv;
  topenv = new my_env("topenv");

  uvm_set_config_int("topenv.*", "debug", 1);

  run_test();

  return 0;
}
```

Integer 'debug' declared for all components below topenv




```
class my_env : public uvm_env
{
 public:
  int debug;
  ...
  UVM_COMPONENT_UTILS(my_env);

  my_env( uvm_component_name name ) : uvm_env(name), debug(0) {}

  void build_phase( uvm_phase& phase )
  {
    uvm_env::build_phase(phase);
    uvm_get_config_int("debug", debug);
    std::cout << get_full_name() << ": In Build: debug = " << debug << std::endl;
  }
  ...
```

Retrieve value for integer variable 'debug'

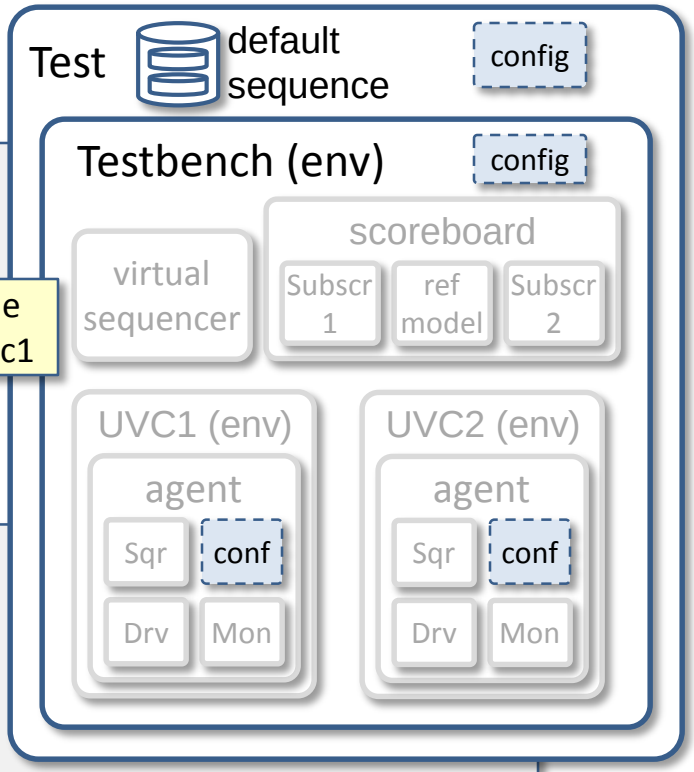
# UVM configuration – example -2

- Set UVC interface in top `sc_main`

```
...
vip_if* dut_if_in = new vip_if();
uvm_config_db<vip_if*>::set(0, "tb.uvc1.*", "vif",
                            dut_if_in);
...
```

Store handle of the interface

Only valid in the context of tb.uvc1




- Get interface in the build phase of the driver of UVC1

```
template <class REQ>
class vip_driver : public uvm_driver<REQ>
{
 public:
  vip_if* vif;
  ...
  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(), "Virtual interface not defined! Simulation aborted!")
  }
  ...
```

Get handle of the interface




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

# UVM factory

- Follows the classical C++ factory design pattern to create objects without specifying the exact class of these objects that will be created
- In UVM, the factory will be used to create and/or override objects for individual test scenario's
- Only objects which are registered to the factory can be instantiated or overridden
- Factory objects are dynamically instantiated using a dedicated static member function create

```
class vip_uvc : public uvm_env
{
  vip_agent* agent;
  ...
  void build_phase( uvm_phase& phase ) {
    ...
    agent = vip_agent::type_id::create("agent", this);
  }
};
```

Object dynamically instantiated

```
class vip_agent : public uvm_agent
{
  UVM_COMPONENT_UTILS(vip_agent)
  ...
};
```

Object is registered to the factory using this macro

# UVM factory overrides

- The factory can be used to substitute a predefined component type with another specialized type, without having to derive from its base class
- Various override functions are available
  - Type overrides: replaces all objects of the specified type with the new specified type

```
set_type_override_by_type( consumer<packet>::get_type(),
fifo_consumer<packet>::get_type() );
```

  - Instance overrides: replaces objects which match the instance path with the new specified type

```
set_inst_override_by_type( "parent_component.consumer",
                           consumer<packet>::get_type(),
                           fifo_consumer<packet>::get_type() );
```

  - In addition to type overrides, similar member functions exist to override by name





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

# *Work-in-Progress*: Register Abstraction Layer

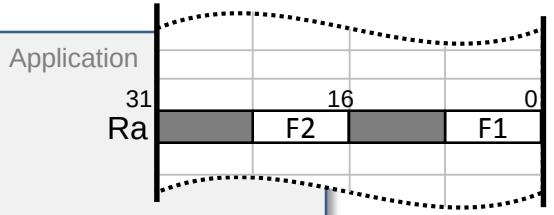
| Register Abstraction Layer | Status |
|---|---|
| Register model containing registers, fields, blocks, etc. | testing |
| Register callbacks | testing |
| Register adapter, predictor, sequences and transaction items | testing |
| Register front-door access | testing |
| Build-in register test sequencers | 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_component_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)

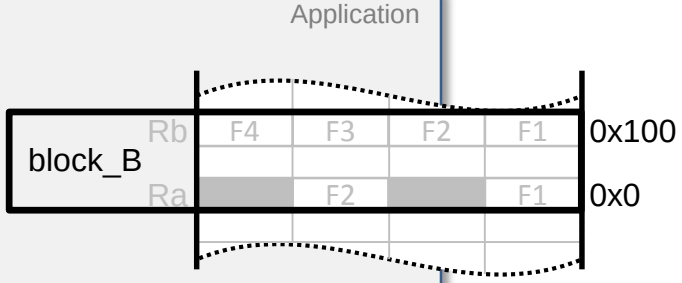
- Although the user can create a register model manually, the recommended use model is to generate this register model from an IP-XACT register description

# 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_component_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

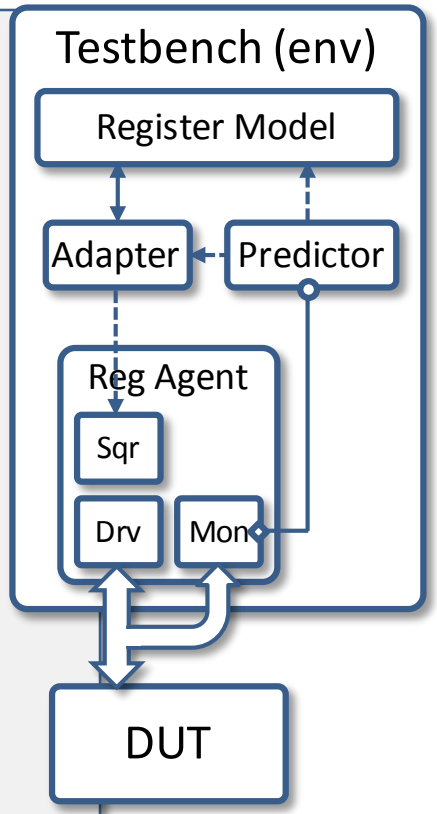
# Testbench including Register Model (1)

```
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

Application







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

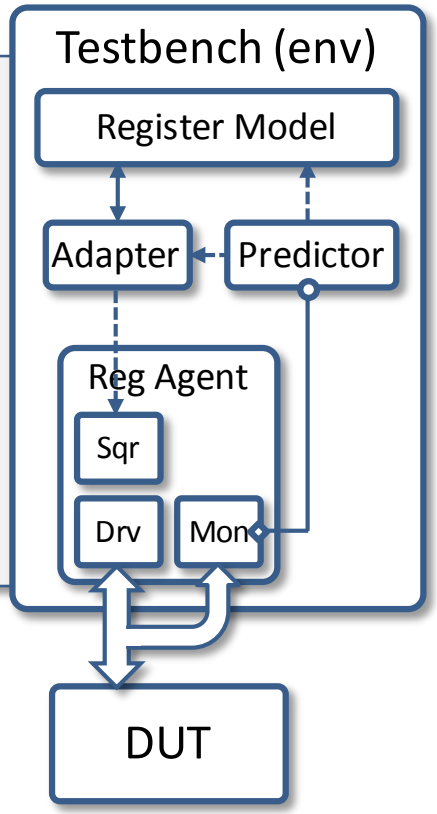
# Testbench including Register Model (2)

```
  ...
  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 and LRM under review – subject to change

# Execute Built-in Register Test (1)

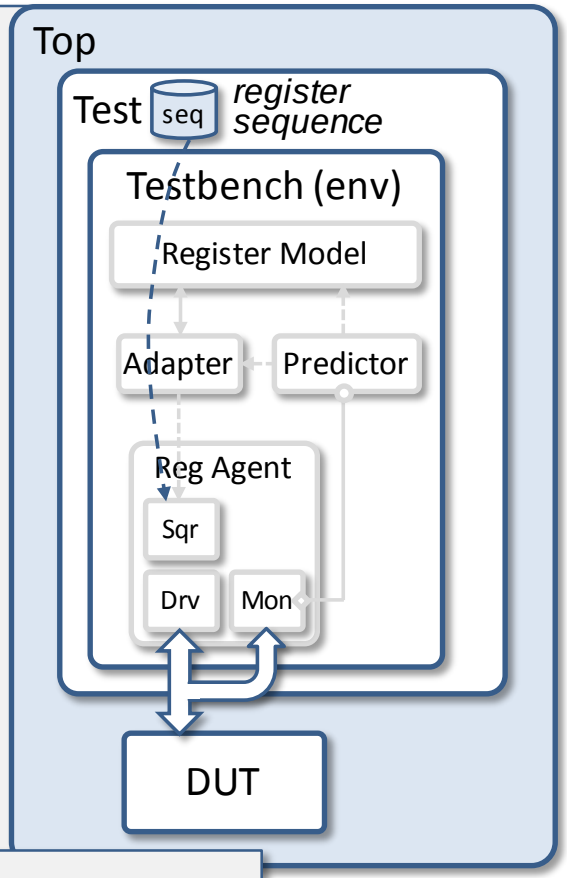
```
class test : public uvm_test
{
 public:
  tb_env* env;
  uvm_reg_sequence<>* seq;

  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 built-in "bit bashing" sequence

Start the register sequence




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




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

# Execute Built-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
```

# Application Examples

# UVM-SystemC Generator

- Generator is based on *easier uvm code generator for SystemVerilog* from Doulos (http://www.doulos.com/knowhow/sysverilog/uvm/easier_uvm_generator/)
- Generator uses template files as input, which are similiar 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

```
#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
```

- General Config File

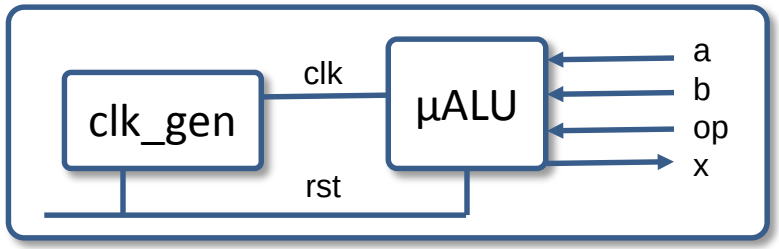
```
#DUT directory
dut_source_path = mydut
#Additional includes
inc_path = include
#DUT toplevel name
dut_top = mydut
#Pin connection file
dut_pfile = pinlist
```

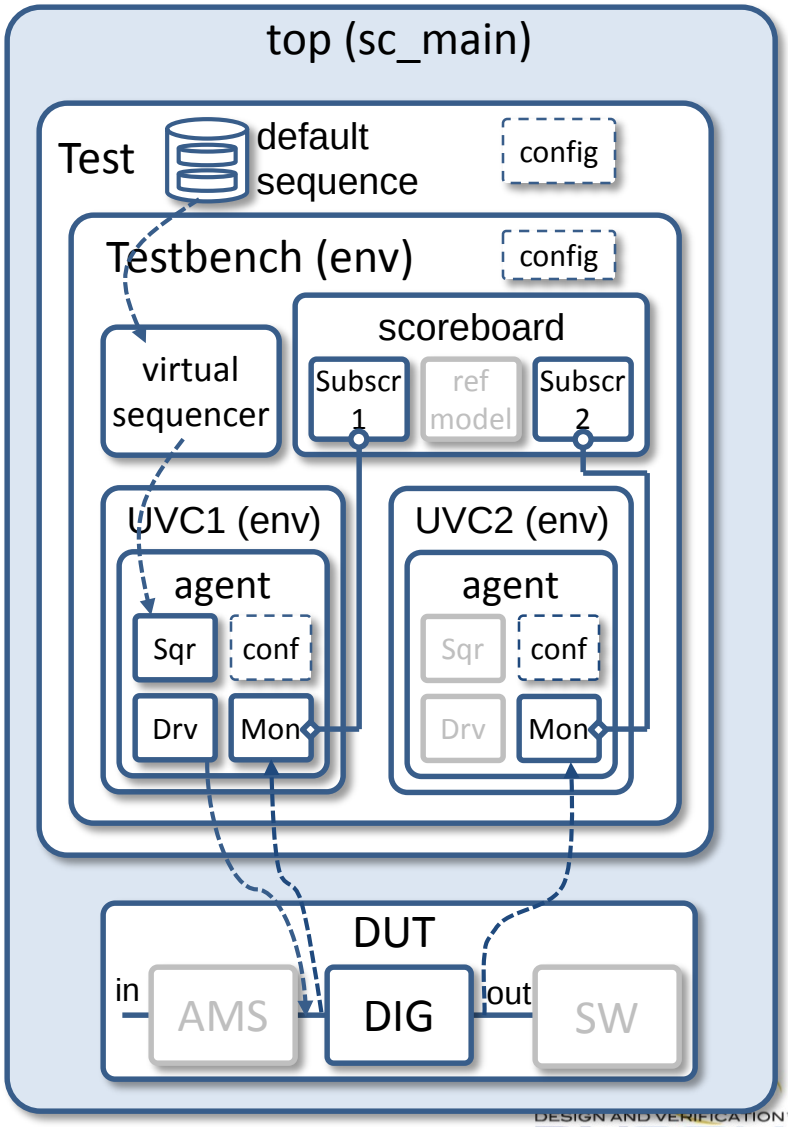
- DUT connection to agent interfaces (DUT port <-> agent port))

```
!clkndata_if
clk clk
reset_n reset_n
rw_master1 rw_master
scl1 scl
sda1 sda

!agent2_if
...
```

# Hands-on example (Generator)

- DUT is a minimalistic ALU
- Tests checks basic arithmetic with static operands
- Plain SystemC Testbench as reference
- Re-implementation with UVM-SystemC

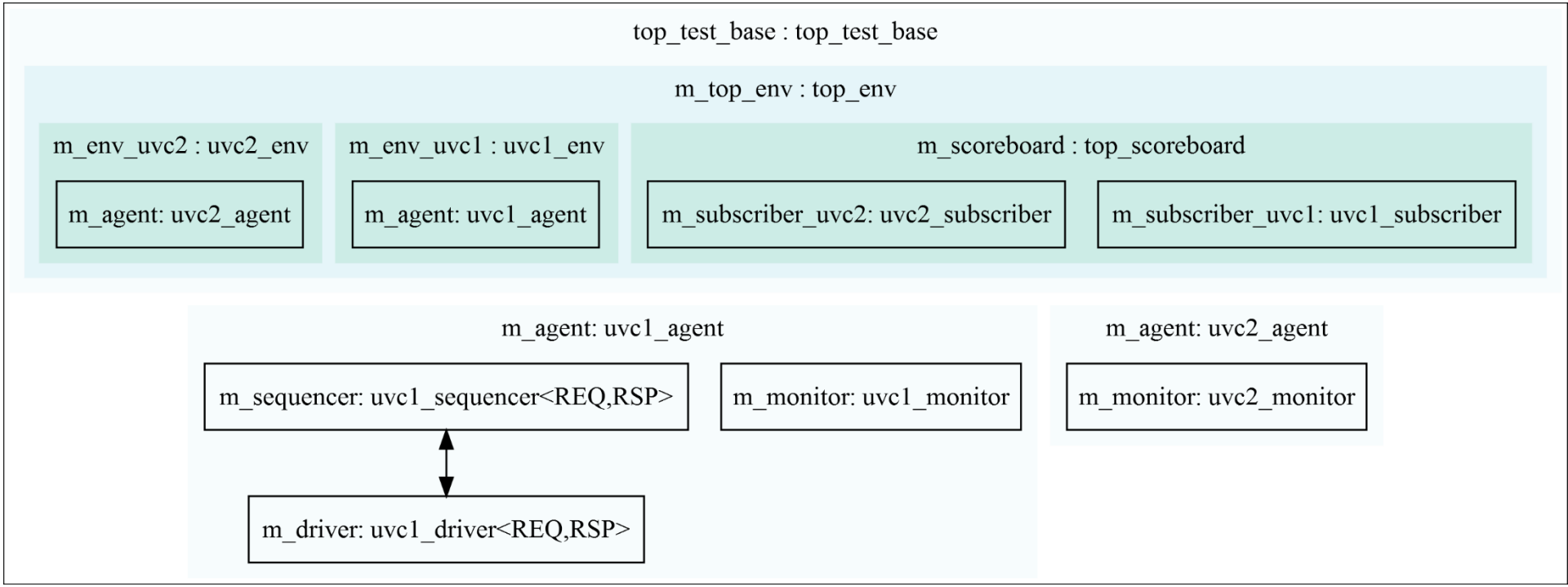


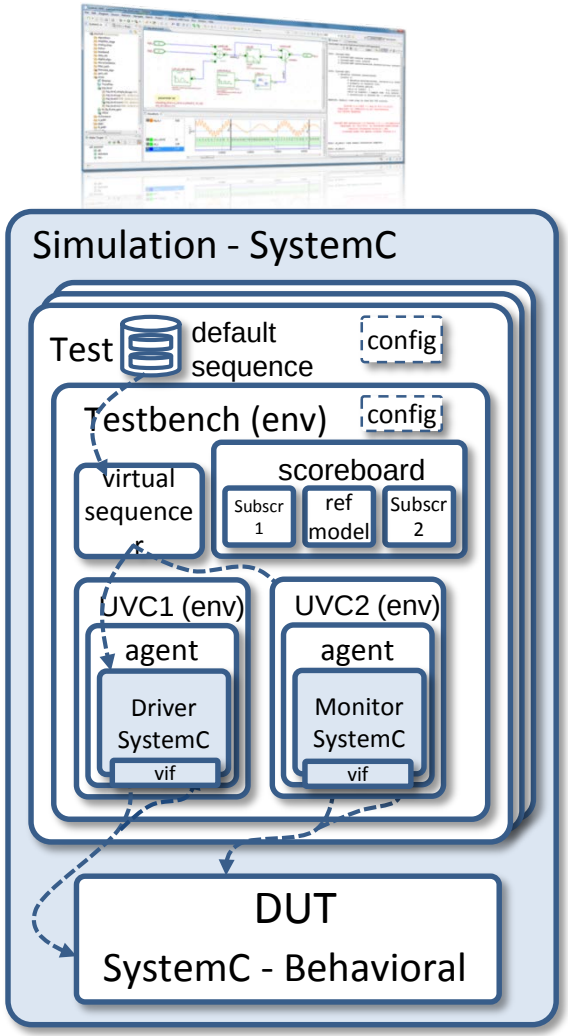
# Benefits

- Avoidance of boilerplate code copy & paste disasters
- Manual input amount as in hand-crafted testbench
  - DUT setup
  - Test sequence
  - Driver implementation for DUT driving
  - Monitor implementation for DUT interpreting
- UVM conformity
- Re-Usage because of modularity more likely

# Hands-on example (Visualizer)

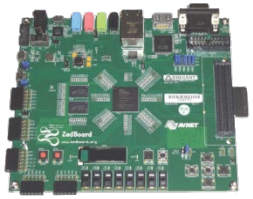


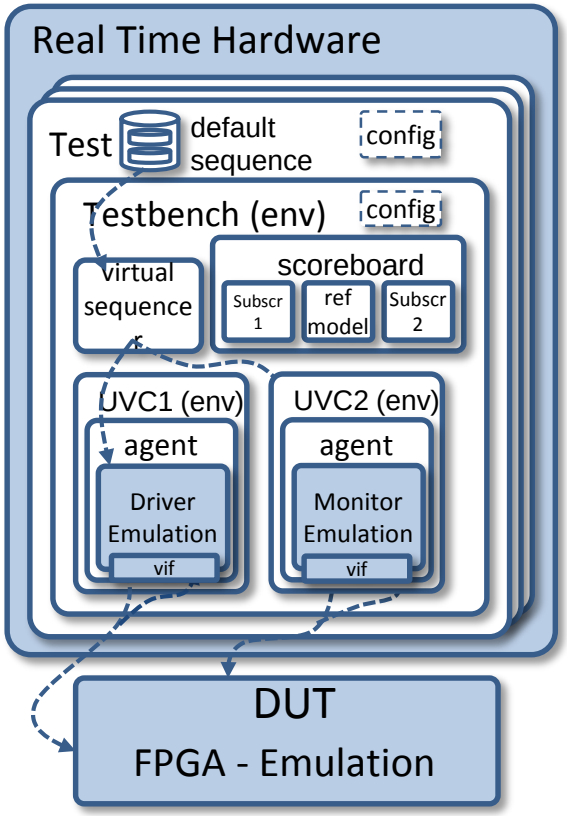
# Re-use across abstraction levels (1)




- 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

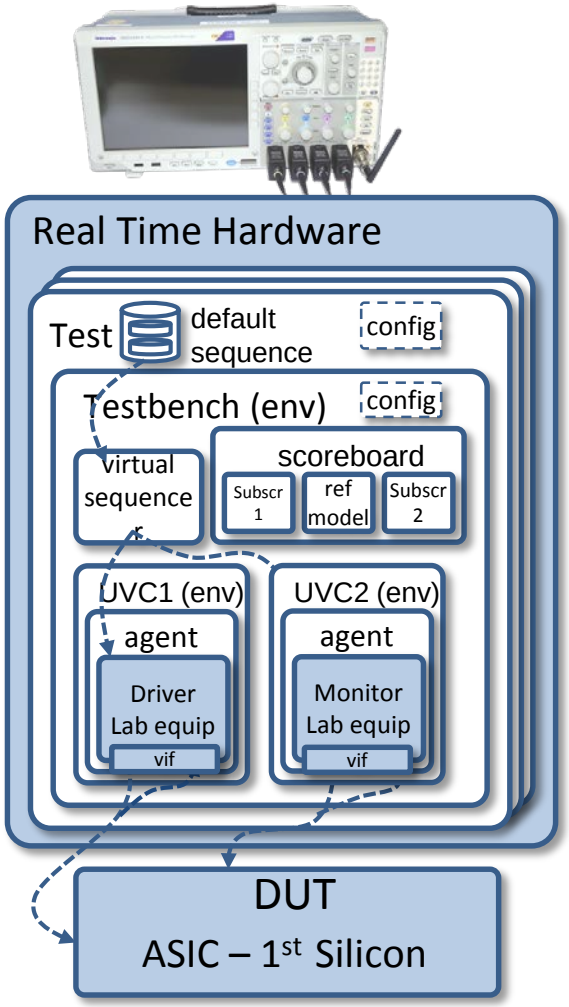
# Re-use across abstraction levels (2)






- 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)

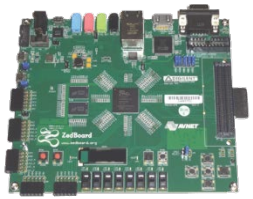



- Continued use of previous verification setup by running the verification environment as a real-time model on lab-test equipment
  - Exchange of UVM driver verification components necessary
  - Re-use of all tests possible

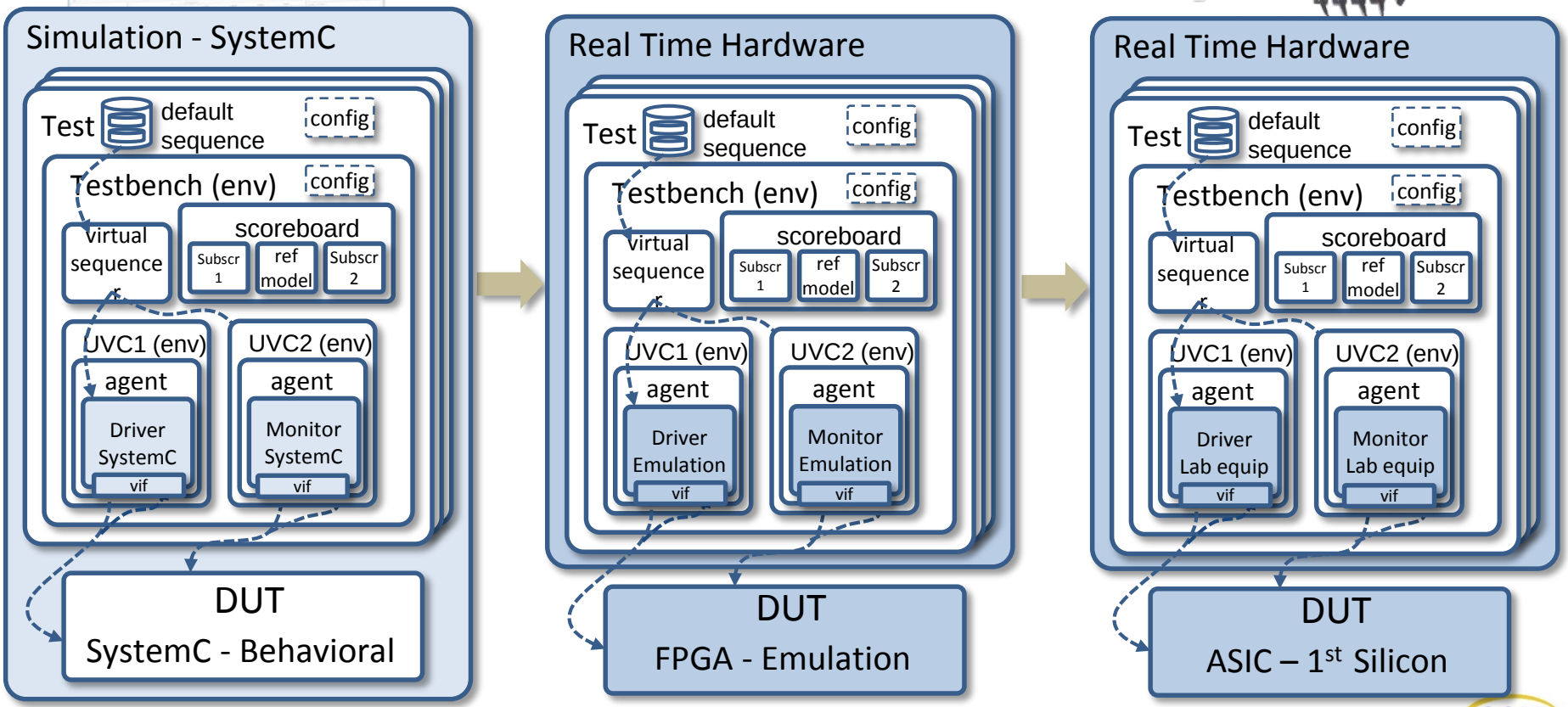
# Re-use across abstraction levels (4)

download

monitor



integrate

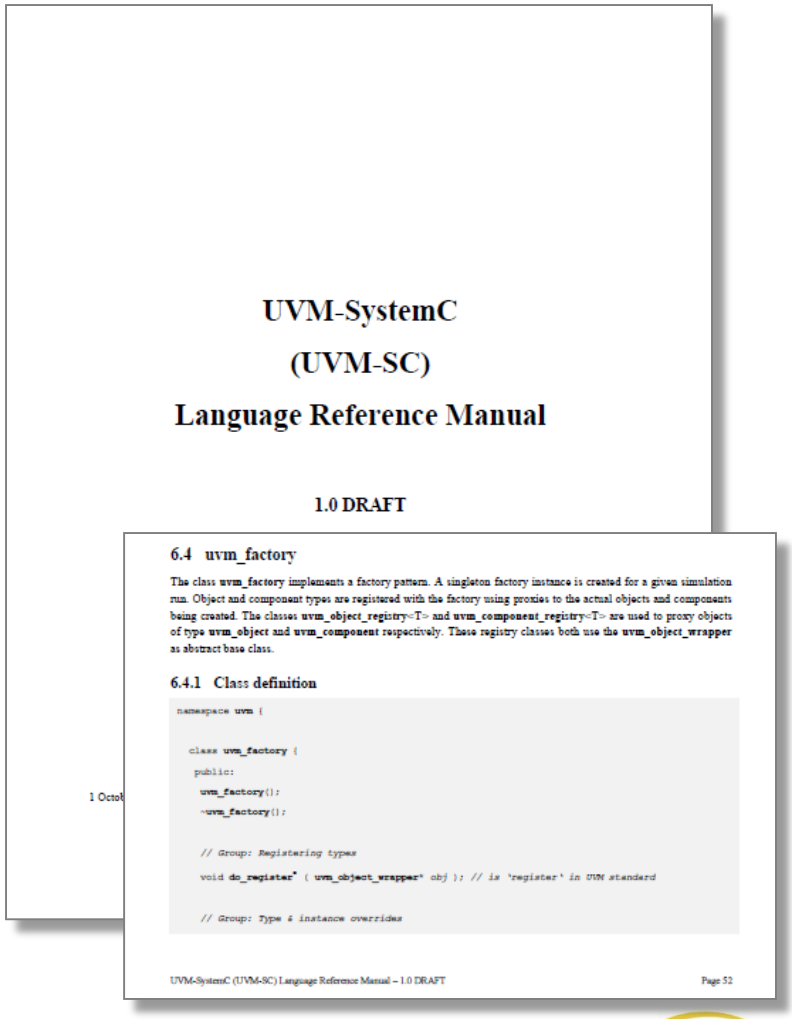


# Outline

- Part C – Further steps & Outlook
  - Standardization in Accellera
  - Next steps
  - Summary and outlook

# 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 CW48/49 2015
  - Both LRM and PoC available under the Apache 2.0 license





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

# 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 finished and Proof-of-concept implementation ongoing
  - Draft release of UVM-SystemC planned for CW48/49 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