UVM-SystemC Randomization - Updates from the SystemC Verification Working Group

Thilo Voertler, COSEDA Technologies GmbH
Dragos Dospinescu, AMIQ
Martin Barnasconi, NXP Semiconductors
Stephan Gerth, Bosch Sensortec GmbH

Overview

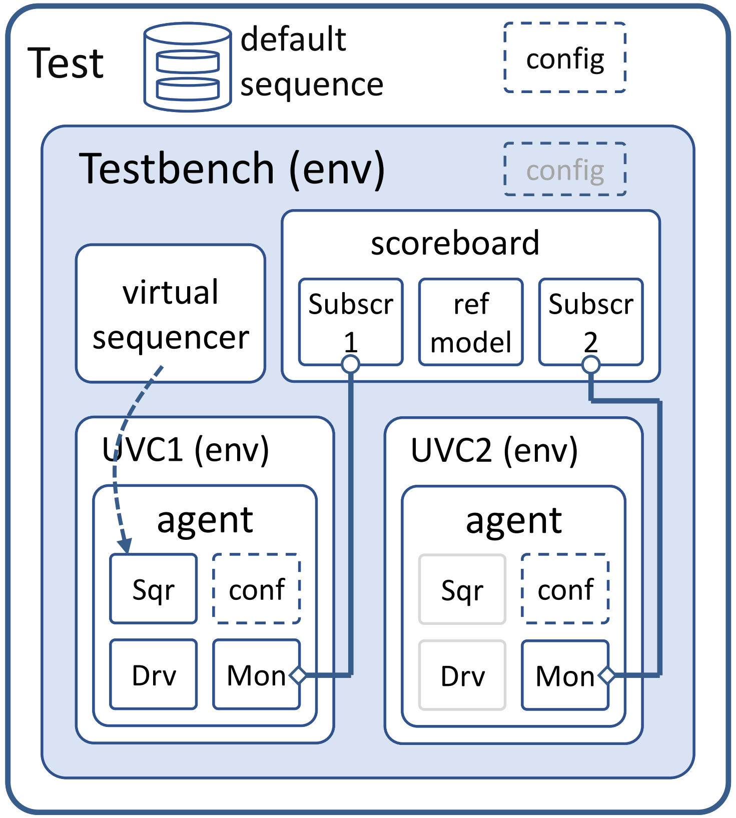
- Introduction and current VWG activities
- Randomization for UVM-SystemC
- Functional Coverage
- Summary and Outlook

Introduction and current VWG activities

Martin Barnasconi / Stephan Gerth

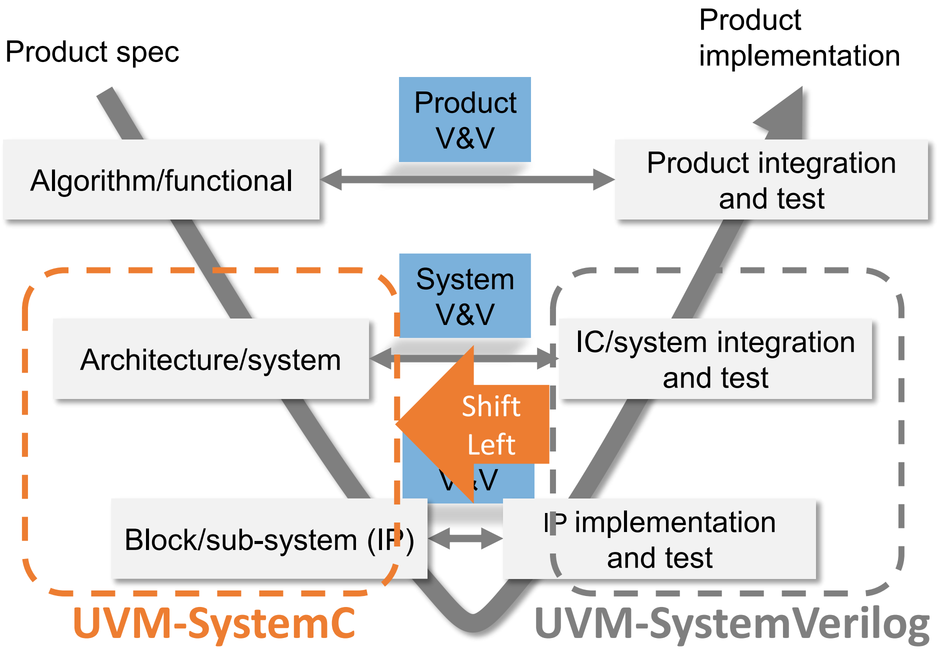
For SystemC Folks: What is UVM ?

- Methodology to create **modular, scalable, configurable and reusable** testbenches
  - Reuse Verification IP with standardized interfaces
- **Standard defined as class library** providing a set of built-in features dedicated to verification
  - E.g., phasing, component overriding (factory), configuration, score-boarding, reporting, etc.
- Verification environment supporting **Coverage Driven Verification**
  - Using constrained random stimulus generation, independent result checking and coverage collection



Why UVM in SystemC/C++ ?

- Growing need for a standardized **system-level** verification methodology
  - Support inclusion of embedded software
  - Early V&V in a **standardized way** ("Shift Left")
- Reuse of tests and test bench IP in the verification and validation phases
  - V&V intent exchangeable when using **SystemC/C++ as base language**
  - Support methodologies such as Hybrid prototyping, Hardware-in-the-Loop (HiL) simulation and Rapid Control Prototyping (RCP)

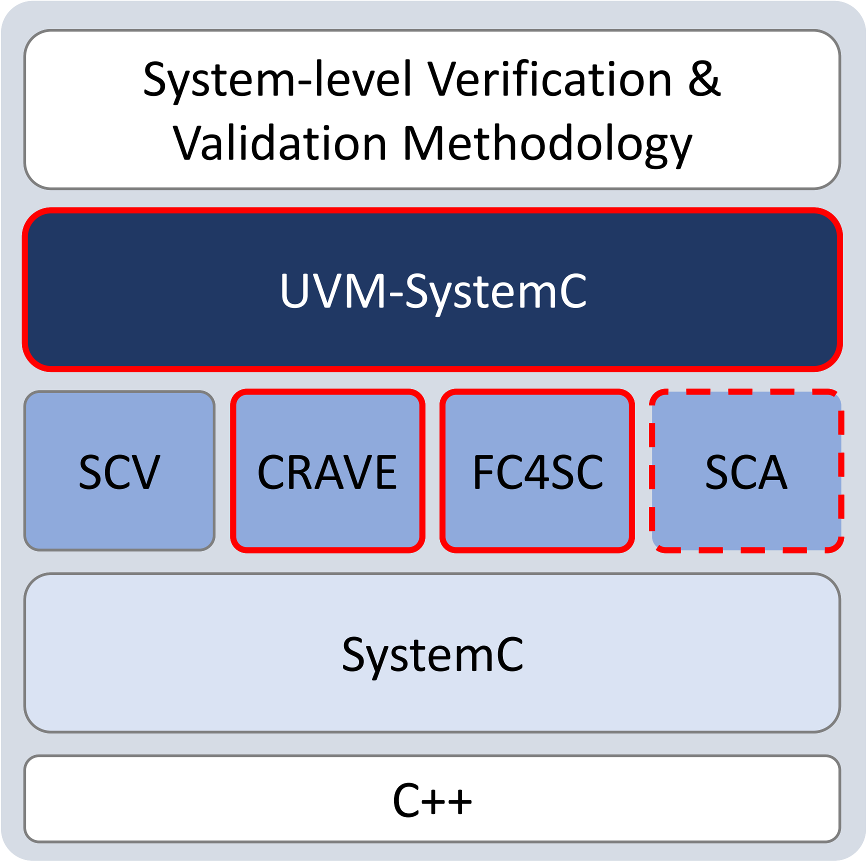


UVM-SystemC Principles

- Comply with UVM-SystemVerilog (IEEE 1800.2) Standardized API
  - Identical class definitions, methods and other definitions in the LRM
  - Very limited API changes to address SystemC/C++ reserved keywords
- Comply with SystemC standard and execution semantics
  - Follow SystemC-defined TLM1 and TLM2 communication mechanism
  - SystemC modules capture testbench hierarchy, test sequences as transient objects
- UVM-SystemC Reference implementation based on C++11
  - Compatible with most EDA vendor solutions and flows
  - Limited use of add-on libraries to keep dependencies low

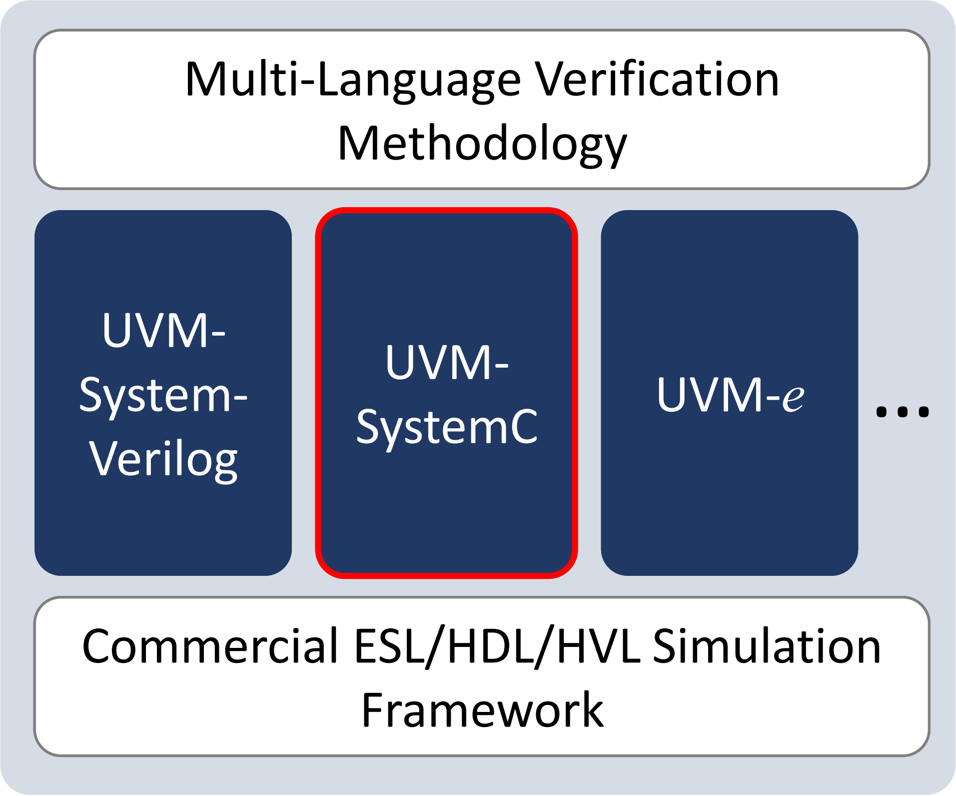
Evolving System-Level V&V Ecosystem

- SystemC-centric V&V ecosystem is evolving
  - Constrained Randomization (CRAVE)
  - Functional Coverage (FC4SC)
  - SystemC Assertions (SCA) – early development
- UVM-SystemC as "unification layer" supporting various V&V methodologies
  - Consolidated and consistent methodology
  - Supported by a standardized API and reference implementations maintained by Accellera and its members



Multi-Language Verification Support

- UVM-SystemC is an integral part of the Accellera Multi Language Verification (MLV) Standard under development
  - Enabling creation of "best of all worlds" verification environments
  - Standard not restricted to UVM in SystemVerilog, SystemC or *e*, integration of other languages such as Matlab or Python is considered
- More details on Multi-Language Verification in DVCon US 2021 Short Workshop (March 1st 11:30 PST)



UVM-SystemC Standardization Developments

- First version released in 2016
  - Standardized the foundational elements such as verification components, factory, configuration database, sequences
- Three beta versions released between 2017-2020
  - Incremental additions such as register classes, data access policy classes, core-services, class defaults, etc.
  - Improved code scalability to address more complex verification scenario's
- Outlook 2021 and beyond
  - Finalize register classes and backdoor access concepts
  - Seamless inclusion of constraints and functional coverage capabilities
    - CRAVE and FC4SC remain available as separate library

Randomization for UVM-SystemC

Thilo Vörtler, Daniel Große, Muhammad Hassan

Randomization for UVM-SystemC

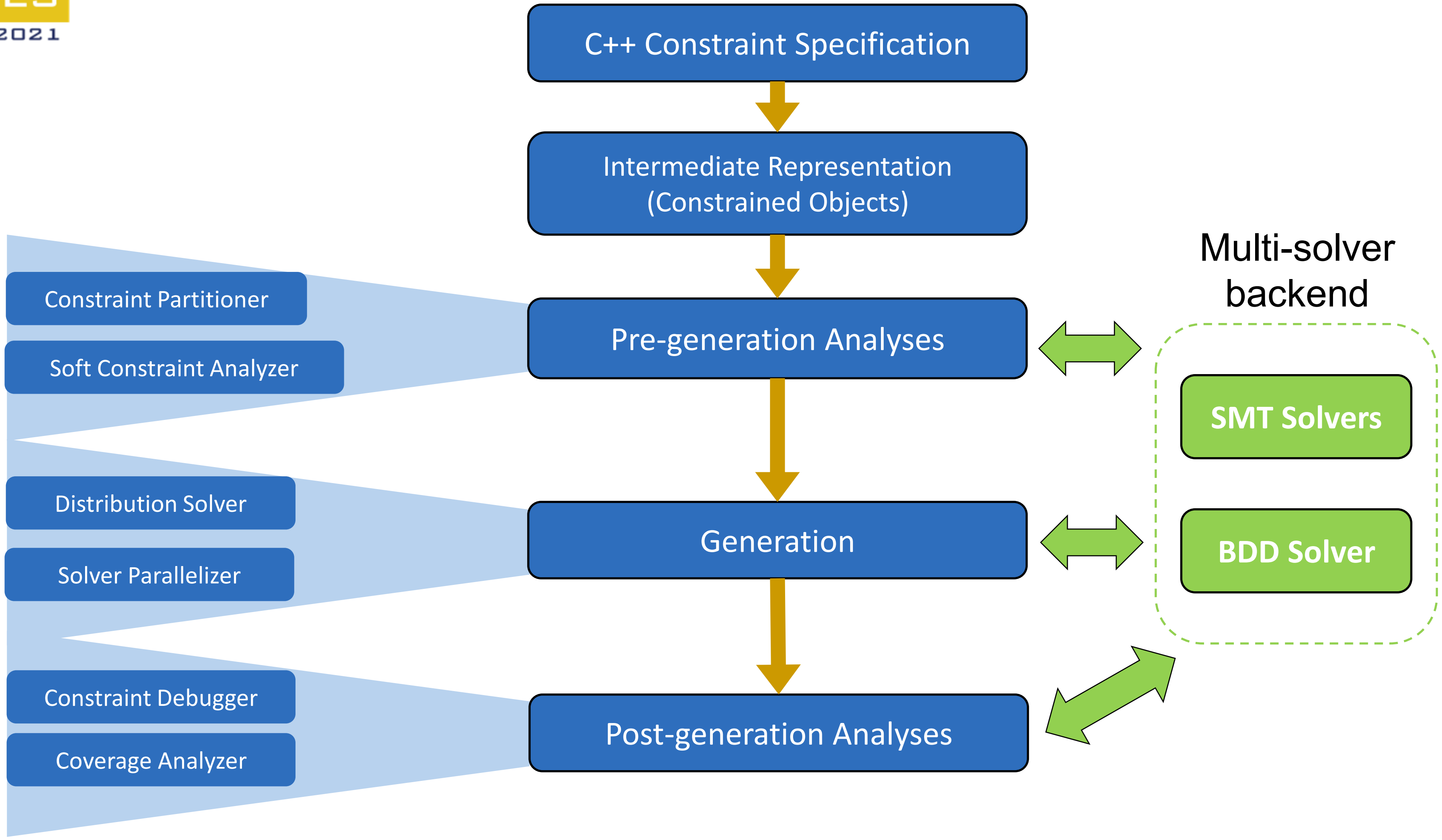
- UVM-SystemC library has no randomization features as randomization is not part of the UVM standard → Randomzation part of SystemVerilog
- SystemC currently has no support for randomization within the core language → Additional libraries requires
- CRAVE library has been donated to Accellera Uni Bremen, DFKI GmbH, Johannes Kepler University Linz
- Last year randomization adaption layer for UVM-SystemC was released

Randomization for UVM-SystemC

- **<u>C</u>**onstrained **<u>Ra</u>**ndom **<u>V</u>**erification **<u>E</u>**nvironment
- Syntax and semantics closely followed SystemVerilog IEEE 1800 std
- Random objects
- Random variables
- Support for C++ and bitlevel SystemC datatypes
- Hard/soft constraints
- Efficient constraint solvers
- MIT license

Crave is available at: http://www.systemc-verification.org/

Randomization for UVM-SystemC



Randomization for UVM-SystemC

- CRAVE Syntax is similar to SystemVerilog,
- Can be used without UVM

SystemVerilog syntax

```
class item;
  rand int v;
  constraint c { v < 10; }
endclass
```

CRAVE syntax

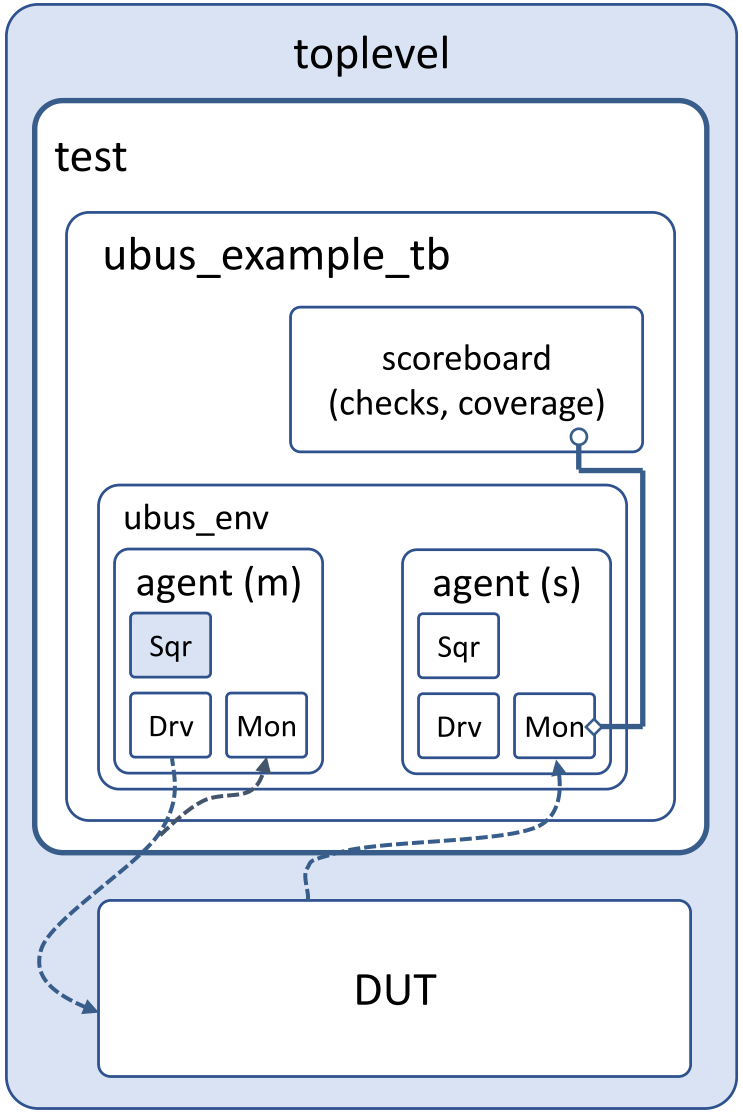
```
class item : public crv_sequence_item {
  crv_variable<int> v; //Random Variable
  crv_constraint c { v() < 10 }; //Constraint
  item(crv_object_name) {}
};
```

CRAVE syntax might change in Accellera standard

UBUS Example

- Ubus example in current UVM-SystemC Beta
  - simple non-multiplexed
  - Synchronous
  - no pipelining
  - Address bus: 16 bit wide
  - Data bus: 8 bit wide

- UVM example provided in the UVM Users Guide (http://accellera.org/downloads/standards/uvm)
- N number of Masters & Slaves supported
- Version supporting randomization using CRAVE available



UBUS Example (sequence_item)

```
class ubus_transfer extends uvm_sequence_item;

  rand bit [15:0]           addr;
  rand ubus_rw_enum         read_write;
  rand int unsigned         size;
  rand bit [7:0]            data[];
  rand bit [3:0]            wait_state[];
  rand int unsigned         error_pos;
  rand int unsigned         transmit_delay = 0;
  ...

  constraint c_read_write {
    read_write inside { READ, WRITE };
  }
  constraint c_size {
    size inside {1,2,4,8};
  }
  constraint c_data_wait_size {
    data.size() == size;
    wait_state.size() == size;
  }
  constraint c_transmit_delay {
    transmit_delay <= 10 ;
  }
```

SystemVerilog

```
class ubus_transfer : public
uvm_randomized_sequence_item {
public:
  crv_variable<ubus_rw_enum> read_write;
  crv_variable<sc_bv<16>> addr;
  crv_variable<unsigned> size;
  crv_vector<sc_bv<8>> data;
  crv_vector<sc_bv<4>> wait_state;
  crv_variable<unsigned> error_pos;
  crv_variable<unsigned> transmit_delay;
  ...
  crv_constraint c_read_write {inside(read_write(),
    std::set<ubus_rw_enum> {
      ubus_rw_enum::READ, ubus_rw_enum::WRITE
    })};
  crv_constraint c_size {inside(size(),
    std::set<int> { 1, 2, 4, 8 }
  )};
  crv_constraint c_data_wait_size {
    data().size() == size(),
    wait_state().size() == size()
  };
  crv_constraint c_transmit_delay {
    transmit_delay()<=10;
  };
```

SystemC

CRAVE syntax might change in Accellera standard

UBUS Example (uvm_sequence)

```
class write_double_word_seq extends ubus_base_sequence;
  ...
  rand bit [15:0] start_addr;
  rand bit [7:0] data0; rand bit [7:0] data1; rand bit [7:0] data2;
  rand bit [7:0] data3; rand bit [7:0] data4; rand bit [7:0] data5;
  rand bit [7:0] data6; rand bit [7:0] data7;
  rand int unsigned transmit_del = 0;
  constraint transmit_del_ct { (transmit_del <= 10); }

  virtual task body();
  ...
```

SystemVerilog

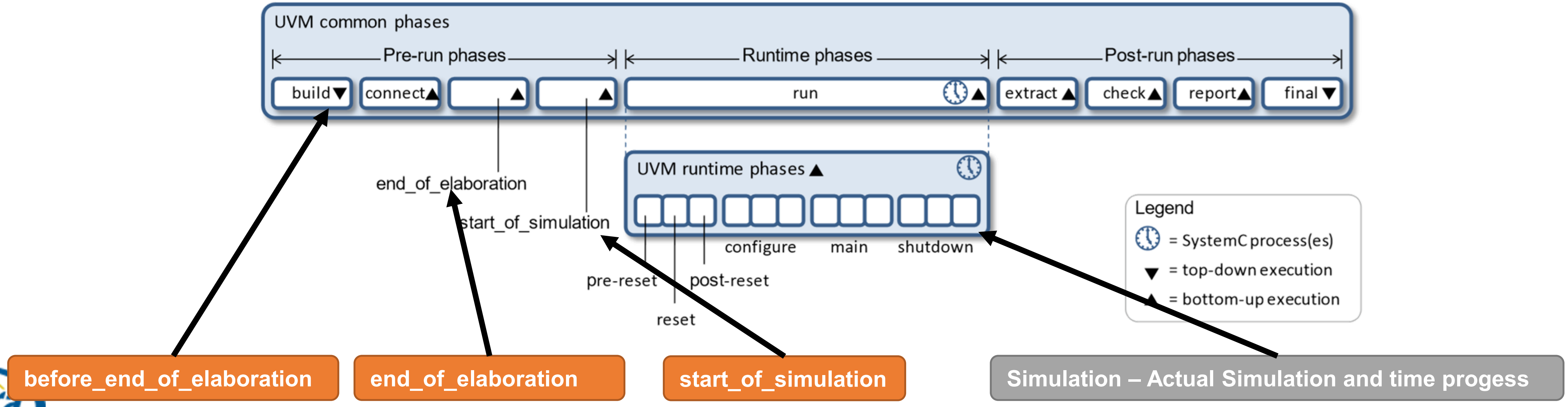
```
class write_double_word_seq : public ubus_base_sequence<REQ, RSP> {
public:
    crv_variable<sc_bv<16>> start_addr;
    crv_variable<sc_bv<8>> data0, data1, ... data7;
    crv_variable<unsigned int> transmit_del;
    crv_constraint transmit_del_ct { transmit_del() <= 10 };

    void body() {
    ...
```

SystemC

CRAVE syntax might change in Accellera standard

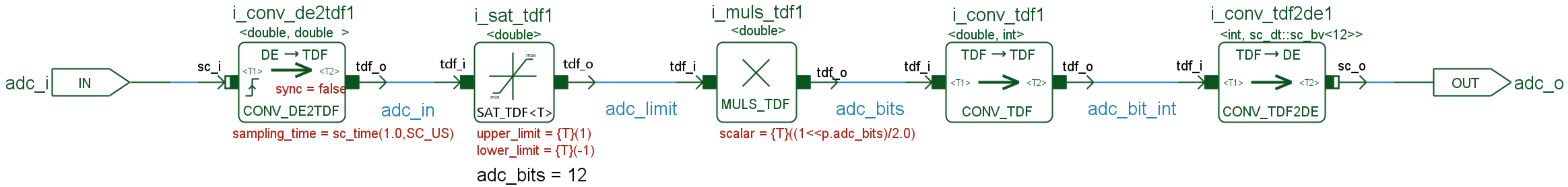
Randomzation of DUTs

- Randomization can be used also during set up of the DUT
- Possible to modify DUT before instantiation using constraints
- UVM-SystemC allows access to SystemC phases from a UVM test



Randomizing SystemC AMS DUTs

- Simple example: Setting stuck faults in ADC behavioral model

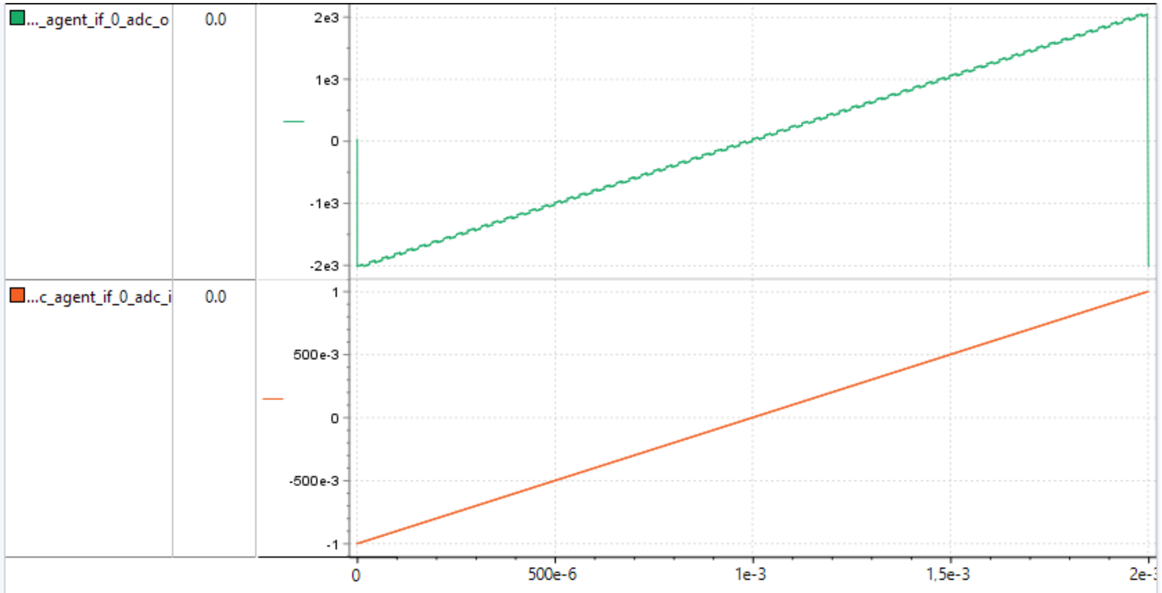


- Set one output bit of adc output to „1“ → Stuck at fault
- Several bits can be also set to „1“
- Modification of DUT just from UVM test possible

Randomizing SystemC AMS DUTs

```
void top_test_base::build_phase(uvm::uvm_phase& phase) {
// connect to SystemC DUT error location probe
dut_connector<int> stuck_bit_int;
stuck_bit_int.bind("*i_conv_tdf2de1.tdf_i");
// define bit error functor to model fault
auto stuck_at_1_error =
[&](int error_mask, double time, const int& oldval) {
return int{error_mask | oldval};
};
rand_one_hot rand_bits;
rand_bits.randomize();
auto error_class = std::bind(stuck_at_1_error, rand_bits.bits,
std::placeholders::_1, std::placeholders::_2);
stuck_bit_int.change_dynamic(error_class);
                                …
m_top_env = top_env::type_id::create("m_top_env", this);
}
```

```
struct rand_one_hot : public crave::crv_sequence_item {
  crave::crv_variable<unsigned int > bits;
  crave::crv_constraint c_one_hot{ "c_one_hot" };
  rand_one_hot( crave::crv_object_name = "rand_one_hot") {
  c_one_hot = {crave::onehot(bits()) && (bits() < (1<<12)) };
  }
};
```



Random stuck at 1 error (bit 5)

Randomization for UVM-SystemC

- Standardization efforts:
  - Write standard document
  - Adapt implementation to use existing SystemC concepts (sc_object, sc_any_value)
  - Check with Portable Stimulus Group constraint syntax definition

- Randomization - Proof of concept Implementation
  - License cleanup (done)
  - Adapt syntax according to standard
  - Update build System

Functional Coverage

Dragos Dospinescu

Agenda

- The FC4SC library & its features
- Coverage model
- Runtime data introspection
- Output database generation
- Next steps & work in progress

The FC4SC library & its features

- Pure C++11 based, header-only, no dependency on other libraries
  - FC4SC core library (AMIQ) + improvements and new features (NVIDIA)
- Complete coverage model (based on SystemVerilog)
  - Define coverage model, collect coverage data
  - Hierarchy & scoping support
  - Separation between the coverage model and data
  - Dynamic model creation (built at runtime)
- On-the-fly coverage data introspection via visitor pattern
- UCIS-DB output – interoperable with commercial tools

Coverage model: basic structure

The model is based on SV:

- covergroups
- coverpoints
- crosses
- bins
- Type & instance
- Options
- Sample function for collecting data
- Conditional expression for sampling

```
class cvg_ex: public fc4sc::covergroup {
public:
  int data, dir;
  CG_CONS(cvg_ex){/*constructor*/}

  COVERPOINT(int, data_cp, data) {
    bin<int>("zero", 0),
    bin<int>("positive", 1, 2, 3)
  };
  COVERPOINT(int, dir_cp, dir) {
    bin<int>("write", 1),
    bin<int>("read", 0)
  };
  auto dir_x_data = cross<int,int>(
  "dir_x_data", &dir_cp, &data_cp);
};
cvg_ex cg;
cg.dir = 0; cg.data = 3;
cg.sample();
```

Coverage model: scoping

- Hierarchical coverage model representation
  - Type & instance
  - Similar to scopes in SV (packages, modules etc.)
  - Parent -> child hierarchy
- In code:
  - Extend from *fc4sc::scope*
  - Declare the scope via the macro call ***SCOPE_DECL***
  - Use ***CG_SCOPED_CONS*** in the covergroup

```
class cov_model : public fc4sc::scope {
  SCOPE_DECL(cov_model)

  class cvg_ex : public fc4sc::covergroup {
    // declaration scope: cov_model::cvg_ex
    CG_SCOPED_CONS(cvg_ex, cov_model) { }
    // ...
  };
};

class main : public fc4sc::scope {
  SCOPE_DECL(main)
  // creation scope: main.cg
  cvg_ex cg;
  // scope assigned later, at instantiation
};
```

Coverage model: context

- Each coverage model sits in a context
- Separates coverage models instantiated in the same simulation
- FC4SC creates a default context – no need to explicitly define one

```
class cov_model : public fc4sc::scope { // …
};

int main(char* argv[], int argc) {
  auto ctx1, ctx2;
  ctx1 = fc4sc::global::create_new_context();
  ctx2 = fc4sc::global::create_new_context();

  cov_model m1("model1",__FILE__,__LINE__,ctx1);
  cov_model m2("model2",__FILE__,__LINE__,ctx2);

  // run simulation and sample covergroups ...

  fc4sc::global::delete_context(cntxt1);
  fc4sc::global::delete_context(cntxt2);
}
```

Coverage model: dynamic models (1)

- Coverage model defined during runtime
- Create a ***dynamic_covergroup_factory*** object
- Insert coverpoints (or crosses) & bins dynamically

```
fc4sc::dynamic_covergroup_factory dynamic_cvg_fac("dynamic_cvg_fac");
auto cvp_sum = cvg.create_coverpoint<int(int,int),bool(int)>(
  "cvg_sum",                              /* name of the coverpoint */
  [](int x, int y) {return x+y;},        /* lamba sample func     */
  [](int x){return x == 1;});             /* lamba condition func */
cvp_sum.create_bin("ZERO", 0);
cvp_sum.create_bin("ONE", 1);
```

Coverage model: dynamic models (2)

- Instantiate a dynamic covergroup
- Bind sample function and condition

```
// Dynamic Instantiation
int v1, v2;
fc4sc::dynamic_covergroup inst(dynamic_cvg_fac,"inst",__FILE__,__LINE__);
cvp_sum.bind_sample(inst, v1, v2);
cvp_sum.bind_condition(inst, v2);
// sample
v1 = 0; v2 = 1;
inst.sample();
```

Runtime data introspection

- Coverage data != coverage model
  - fc4sc::bin -> model -> User defined lifetime
  - fc4sc::bin_data_model -> data -> context defined lifetime
- Visitors visit the internal data representation

```
class visitor : public fc4sc_visitor {
  // virtual functions inherited from fc4sc_visitor
  void visit(fc4sc::cvg_base_data_model& base);
  void visit(fc4sc::coverpoint_base_data_model& base);
  void visit(fc4sc::cross_base_data_model& base);
  void visit(fc4sc::bin_base_data_model& base);
};
```

Output database generation

- Output = UCIS DB (XML)
- xml_printer::coverage_save
  - Visitor
  - Name argument
  - (Optional) Context argument
- Context isolated
- Can be called at any point during runtime

```
int main(char* argv[], int argc) {
  auto ctx1, ctx2;
  ctx1 = fc4sc::global::create_new_context();
  ctx2 = fc4sc::global::create_new_context();

  // create covergroups ...
  // run simulation and sample covergroups ...
  // ...

  xml_printer::coverage_save("ctx1.xml", ctx1);
  xml_printer::coverage_save("ctx2.xml", ctx2);

  fc4sc::global::delete_context(cntxt1);
  fc4sc::global::delete_context(cntxt2);
}
```

Next steps & work in progress

- Standardize the API
- Create better documentation
- Implement missing cross functionality
- Decouple coverage implementation from declaration (user)
- Work on an official release

Summary and Outlook

Stephan Gerth

Summary and Outlook

- UVM-SystemC 1.0-beta4 release
  - Few register related blocking items
- CRAVE integration layer for UVM-SystemC released
  - Standardization documentation efforts in progress
  - Donation process of CRAVE to Accellera kicked-off
- Functional Coverage w/ FC4SC
  - AMIQ‘s functional coverage implementation (FC4SC) part of Accellera
  - API standardization for functional coverage major upcoming topic

- Input and support from interested parties welcome!

Summary and Outlook

- References
  - SystemC Verification Working Group
    - https://www.accellera.org/activities/working-groups/systemc-verification
  - UVM-SystemC
    - https://accellera.org/images/downloads/drafts-review/uvm-systemc-1.0-beta3.tar.gz
  - FC4SC
    - https://github.com/amiq-consulting/fc4sc
  - CRAVE
    - http://www.systemc-verification.org/crave