Leveraging IEEE 1800.2-2017 UVM for improved RAL modelling

Vikas Sharma - (Mentor, A Siemens Business)
Manoj Manu - (Mentor, A Siemens Business)
Ankit Garg - (Mentor, A Siemens Business)

accellera SYSTEMS INITIATIVE™
Mentor® A Siemens Business
2019 DESIGN AND VERIFICATION™ DVCON CONFERENCE AND EXHIBITION INDIA

Agenda

- UVM RAL
  - Requirement
  - Challenges
- IEEE 1800.2 UVM RAL
  - Offerings
  - Solution
- Summary
- QA

accellera SYSTEMS INITIATIVE

2019 DESIGN AND VERIFICATION™ DVCON CONFERENCE AND EXHIBITION INDIA

Universal Verification Methodology (UVM)

- UVM, Stands always, Just like a true friend
- Guides & Supports everybody, and it could be
  - At times when executing, a verification test
  - At times when debugging, what went wrong
  - At times when pursuing, an unlawful timeline
- Offers many rational things & thoughts to be relied upon
- Register Abstraction Layer (RAL) is one of them, and stands out for
  - Being the most important one (automatic mirroring & predictions)
  - Being the most customizable one (implementation specific)

© Accellera Systems Initiative

Register Abstraction Layer (RAL)

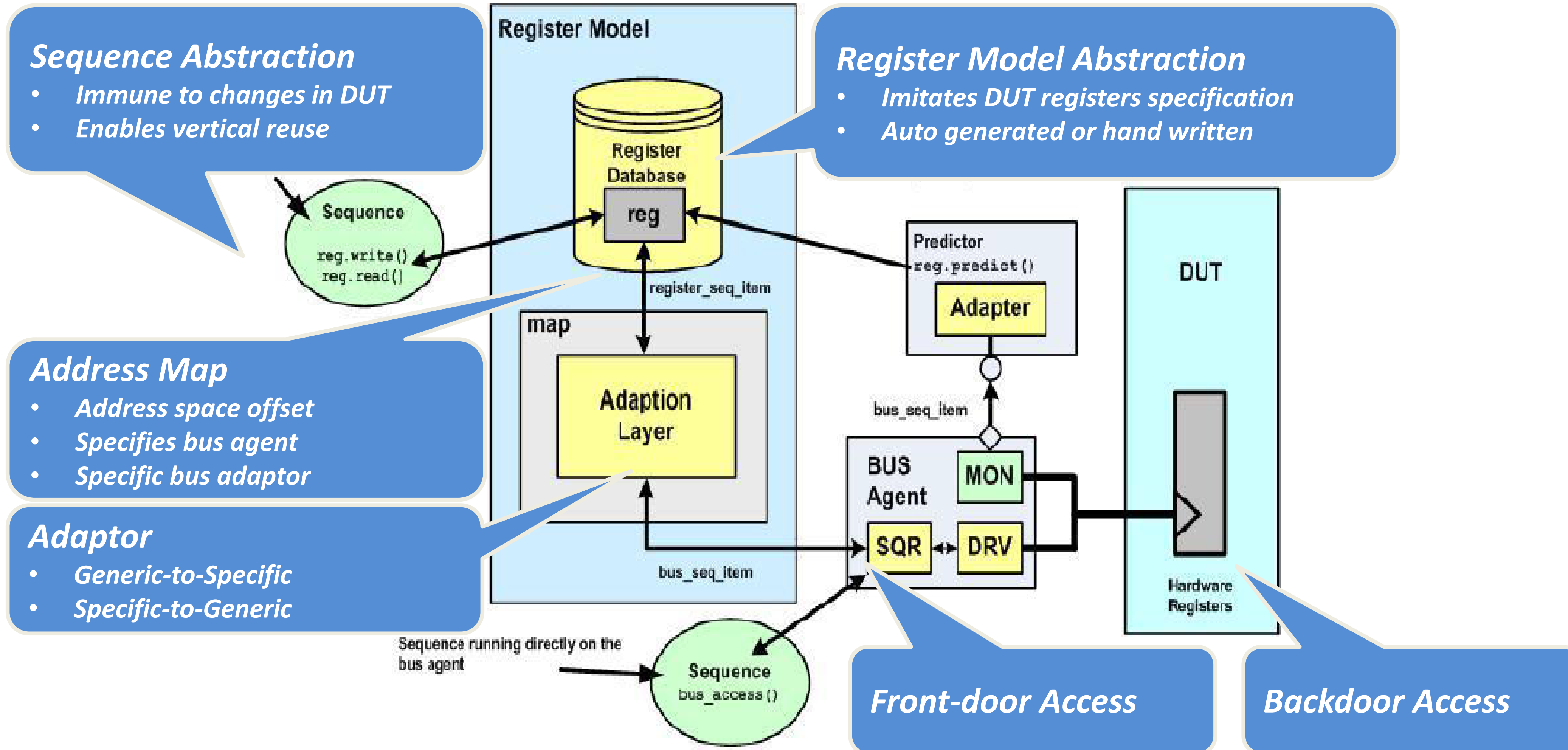


Image Courtesy : Verification Academy and UVM Cookbook

Requirements : UVM-RAL

- Dynamic address map
  - When address maps are required to be changed during the simulation
    - Addition/Deletion and Reconfiguration of registers and memories
  - When any new interrupt or hot-join occurs during simulation
- Multiple address map
  - When there are multiple masters in a system
    - Separate address map and adaptor for every bus interface
  - When there are multiple view (set of address offset) to the same set of registers
    - Separate address map for separate set of address offsets

Challenges : UVM-RAL

- Static address map
  - Allows only static locking with ***lock_model()*** at build time
  - It checks ***is_locked()***, and if found locked then no further structural changes can be made
  - If really wanted to do, then extend base and rewrite many functionalities
  - No standard mechanism is available to unlock the register maps
- Single address map
  - Doesn't allow a register to be added to multiple address map
  - Doesn't allow a memory to be added to multiple address map
  - Doesn't allow a sub-map to be added to multiple address map
  - Provides restriction of the same parent block

```
// Once locked, no further structural
// changes, such as adding registers
// or memories, can be made.
function void uvm_reg_block::lock_model();

   if (is_locked())
     return;

   locked = 1;

   foreach (regs[rg_]) begin
     uvm_reg rg = rg_;
     rg.Xlock_modelX();
   end

   foreach (mems[mem_]) begin
     uvm_mem mem = mem_;
     mem.Xlock_modelX();
   end

   foreach (blks[blk_]) begin
     uvm_reg_block blk=blk_;
     blk.lock_model();
   end
```

IEEE 1800.2 UVM-RAL

- Accellera in conjunction with IEEE launched 800.2 UVM standard
- IEEE 1800.2 standard comes with
  - Bug fixes,
  - Modified API's (backward compatible)
  - New API's and features
  - Deprecated APIs (activated again by using +UVM_ENABLE_DEPRECATED_API)
- Proposes the solution for the problems in traditional UVM-RAL
  - Dynamic and multiple address maps
- Allows removal or addition of registers and memories
- Allows an association to different address maps

Offerings : IEEE 1800.2 UVM-RAL

- New API ***unlock_model()***
  - Unlocks the register model, bringing the register model to the state before ***lock_model()***
  - Recursively unlocks the entire register model and sub-blocks
  - Sets the locked bit to 0
- New API ***wait_for_lock()***
  - Blocks untill ***lock_model()*** completes

```
// Brings back the register mode to a
// state before lock_model() so that a
// subsequent lock_model() can be issued
//
virtual function void unlock_model();

  bit s[uvm_reg_block]=m_roots;
  m_roots.delete();

   foreach (blks[blk_])
       blk_.unlock_model();

   m_roots=s;
   foreach(m_roots[b])
     m_roots[b]=0;

   locked=0;
endfunction


// Waits for the event to be triggered
// when locked using lock_model()
virtual task wait_for_lock();
  @m_uvm_lock_model_complete;
endtask
```


© Accellera Systems Initiative

Offerings : IEEE 1800.2 UVM-RAL

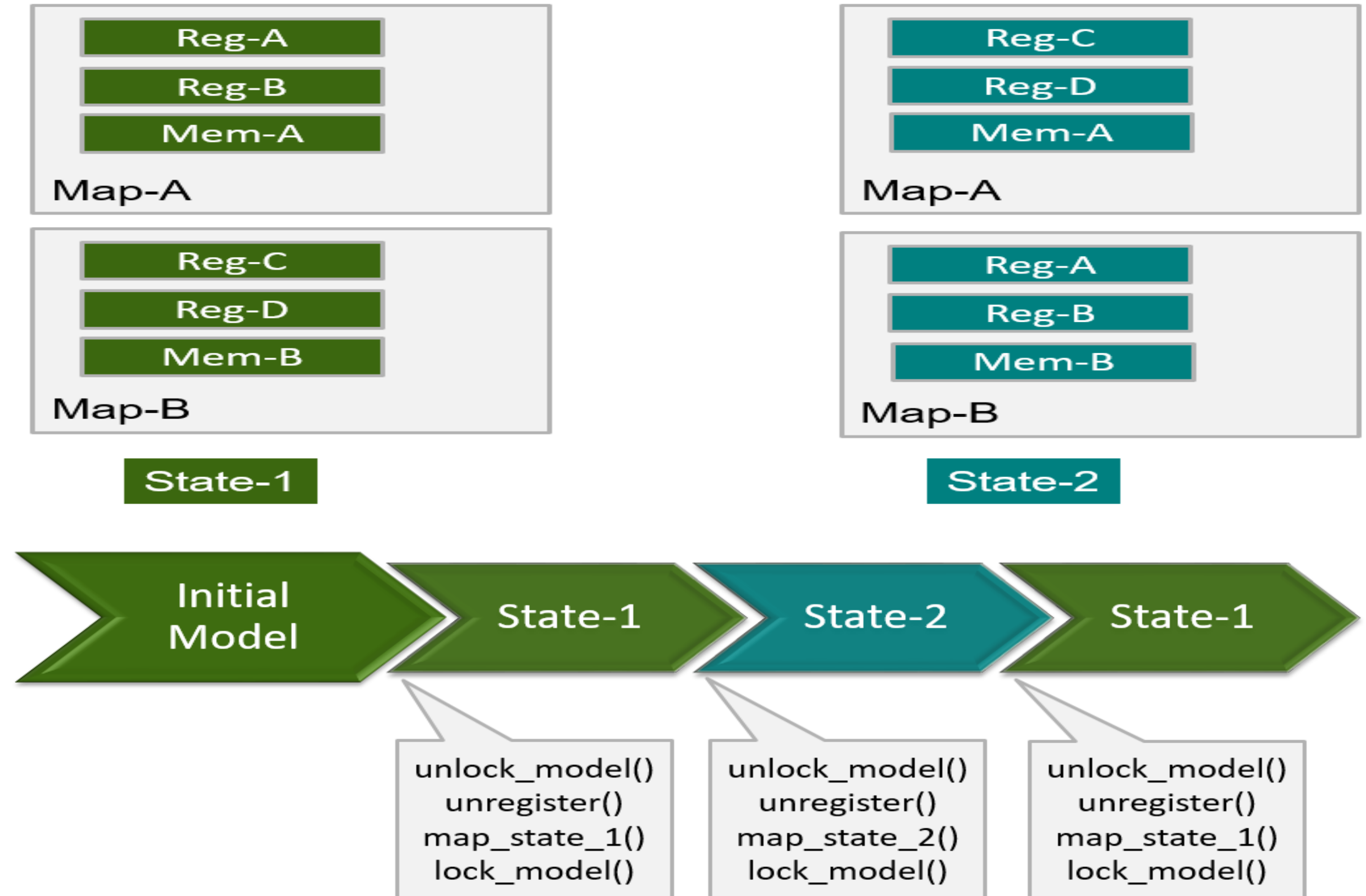
- New API ***set_lock()***
  - Sets the lock mode for the current register block and all its sub blocks
- New API ***unregister()***
  - Unregisters recursively all content from map
  - Removes all knowledge of map from current block, registers, memories, and virtual registers
  - Enables the reuse of the map content and objects with a fresh map instance by using the add_** APIs
  - If used ***this.unregister(map)*** then full register map will get unregistered
  - If used ***map.unregister(reg/mem)*** then only specific register or memory will get unregistered

```
virtual function void set_lock(bit v);
  locked=v;
  foreach(blks[idx])
    idx.set_lock(v);
endfunction


// Brings back the register mode to a
// state before lock_model() so that a
// subsequent lock_model() can be issued
//
virtual function void unlock_model();

  bit s[uvm_reg_block]=m_roots;
  m_roots.delete();

   foreach (blks[blk_])
       blk_.unlock_model();

   m_roots=s;
   foreach(m_roots[b])
     m_roots[b]=0;

   locked=0;
endfunction
```

© Accellera Systems Initiative

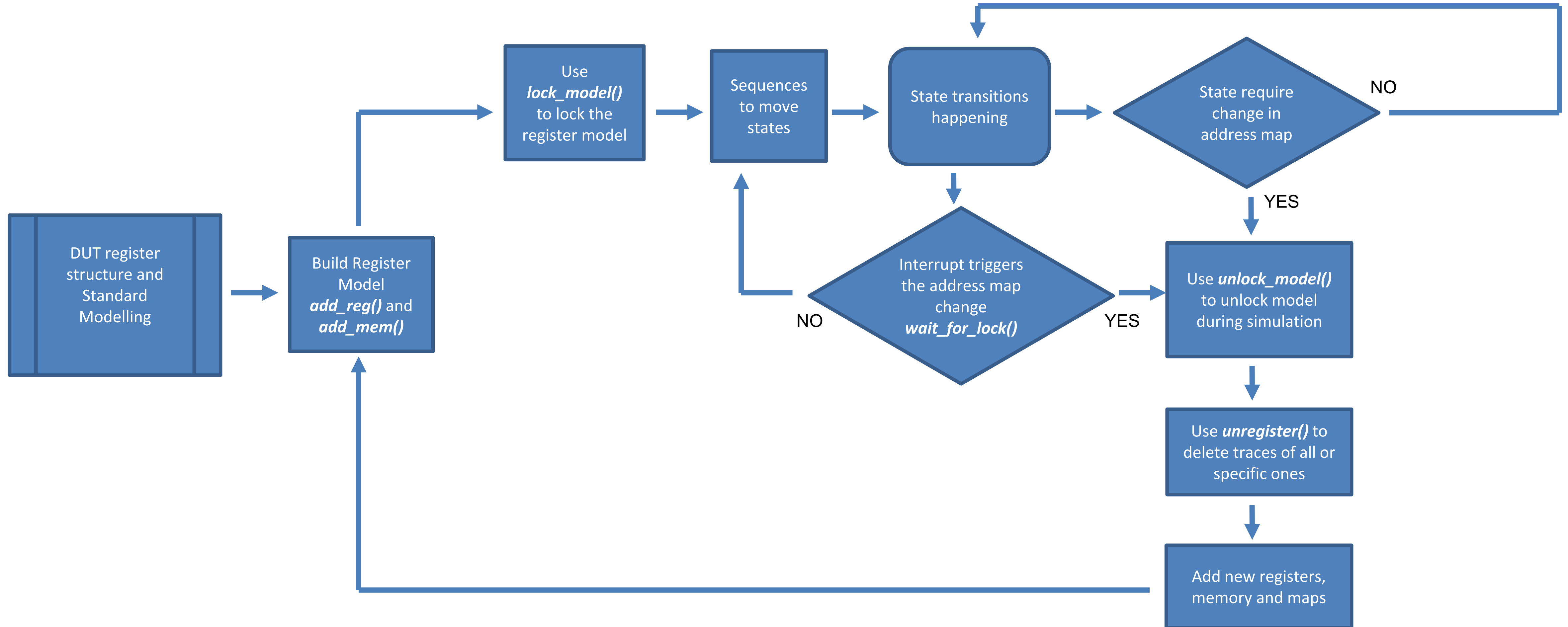
Solution : IEEE 1800.2 UVM-RAL

- Let's suppose that there is a test requirement where :
  - State-1 and State-2 specify different register structures and address maps as explained in the above figure.
  - The simulation starts from the initial model and moves the system from State-1 to State-2, and after some time the system returns to State-1 as explained in the figure below.
  - At every stage, user needs to reconfigure the model, but it cannot be done because the system is already locked during initial model setup.



accellera SYSTEMS INITIATIVE
2019 DESIGN AND VERIFICATION DVCON CONFERENCE AND EXHIBITION INDIA

Solution : IEEE 1800.2 UVM-RAL



© Accellera Systems Initiative

Solution : IEEE 1800.2 UVM-RAL

- ***map_as_per_states()*** function
  - Handles the register mapping based on the states of the system
  - Called every time the system state changes
  - Unlocks the register model and unregisters the register maps
  - Calls function ***map_state_1()*** and ***map_state_2()*** when in ***STATE_1*** and ***STATE_2*** respectively

```
// Remapping the register model as per states
function void map_as_per_states(state_e state);

  unlock_model();     // Unlocks the model

  unregister(map_A); // Unregisters the address map_A
  unregister(map_B); // Unregisters the address map_B
  map_A = null;      // Delete the address map_A
  map_B = null;      // Delete the address map_B

  case(state)
    STATE_1 : map_state_1();
    STATE_2 : map_state_2();
  endcase

  // Locks the model
  lock_model();

endfunction: map_as_per_states
```

Solution : IEEE 1800.2 UVM-RAL

```
function void map_state_2();
reg_A = ctrl_status_reg::type_id::create("reg_A");
reg_A.configure(this, null, "");
reg_A.build();
reg_B = ctrl_status_reg::type_id::create("reg_B");
reg_B.configure(this, null, "");
reg_B.build();
reg_C = ctrl_status_reg::type_id::create("reg_C");
reg_C.configure(this, null, "");
reg_C.build();
reg_D = ctrl_status_reg::type_id::create("reg_D");
reg_D.configure(this, null, "");
reg_D.build();

mem_A = mem_model::type_id::create("mem_A");
mem_A.configure(this, "");
mem_B = mem_model::type_id::create("mem_B");
mem_B.configure(this, "");

// Add memories and registers to the map_A
map_A = create_map("map_A", 'h0, 4, UVM_LITTLE_ENDIAN, 1);
map_A.add_reg(reg_C, 32'h0000_000C, "RW");
map_A.add_reg(reg_D, 32'h0000_000D, "RW");
map_A.add_mem(mem_A, 32'h000A_0000, "RW");

// Add memories and registers to the map_B
map_B = create_map("map_B", 'h0, 4, UVM_LITTLE_ENDIAN, 1);
map_B.add_reg(reg_A, 32'h0000_000A, "RW");
map_B.add_reg(reg_B, 32'h0000_000B, "RW");
map_B.add_mem(mem_B, 32'h000B_0000, "RW");
endfunction: map_state_2
```

```
function void map_state_1();
reg_A = ctrl_status_reg::type_id::create("reg_A");
reg_A.configure(this, null, "");
reg_A.build();
reg_B = ctrl_status_reg::type_id::create("reg_B");
reg_B.configure(this, null, "");
reg_B.build();
reg_C = ctrl_status_reg::type_id::create("reg_C");
reg_C.configure(this, null, "");
reg_C.build();
reg_D = ctrl_status_reg::type_id::create("reg_D");
reg_D.configure(this, null, "");
reg_D.build();

mem_A = mem_model::type_id::create("mem_A");
mem_A.configure(this, "");
mem_B = mem_model::type_id::create("mem_B");
mem_B.configure(this, "");

// Add memories and registers to the map_A
map_A = create_map("map_A", 'h0, 4, UVM_LITTLE_ENDIAN, 1);
map_A.add_reg(reg_A, 32'h0000_000A, "RW");
map_A.add_reg(reg_B, 32'h0000_000B, "RW");
map_A.add_mem(mem_A, 32'h000A_0000, "RW");

// Add memories and registers to the map_B
map_B = create_map("map_B", 'h0, 4, UVM_LITTLE_ENDIAN, 1);
map_B.add_reg(reg_C, 32'h0000_000C, "RW");
map_B.add_reg(reg_D, 32'h0000_000D, "RW");
map_B.add_mem(mem_B, 32'h000B_0000, "RW");
endfunction: map_state_1
```

Changing the address map of map_A from State-1 to State-2

Changing the address map of map_B from State-1 to State-2

accellera SYSTEMS INITIATIVE

2019 DESIGN AND VERIFICATION DVCON CONFERENCE AND EXHIBITION INDIA

Summary

- Identified the problem of static nature of the RAL model in traditional UVM
- Introduced new IEEE 1800.2 UVM RAL model
- Proposed solutions targeting dynamic address mapping
- Illustrated an example changing the register model dynamically using system states
- Demonstrated the usage of new APIs provided in the IEEE 1800.2 UVM RAL model

© Accellera Systems Initiative

Questions