Random Stimuli Models for UVM Registers

Jacob Sander Andersen, CTO, SyoSil ApS
jacob@syosil.com
Lars Viklund, Expert Engineer, Axis Communications AB
lars.viklund@axis.com
Laura Montero, Verification Engineer, SyoSil ApS
laura@syosil.com

accellera SYSTEMS INITIATIVE

2019 DESIGN AND VERIFICATION DVCON CONFERENCE AND EXHIBITION EUROPE

Motivation

- Most companies (that do) functional verification of RTL designs are using UVM
  - Thus, most companies have adopted the usage of the UVM register layer as well

- The UVM register layer provides
  - A uniform way of modelling registers across testbenches
  - Easier maintenance of tests/ UVM sequences as they use real register names and field names
  - The models can be nested so block/module level models can be merged into subsystem/chip level models

Typical UVM Register Model Usage

- UVM provides the base classes for registers, fields etc.
- An instance can then either be
  - Handwritten
  - Generated from a high-level register specification (SystemRDL)
    - May be vendor specific

- Single instance in TB
  - Instantiated typically in the ENV
  - Connected to the UVC handling the register accesses to the DUT

Role of the Register Model

- Verification engineers apply randomized stimuli to the DUT
  - Need to randomize stimuli models
- What roles do the register model play
  - Sometimes the model is queried and used for control flow of the tests/sequences
  - Can also be a direct part of the stimuli randomization/application
- So randomization of registers play a fundamental role in applying stimuli

How To Randomize Registers

- Invoke .randomize() on register field
  - Default is to get the register first, then get the register field and do: `<reg field>.randomize()`
  - Root and register randomization success depends on the generated register model
- Most generated register models allows direct access to registers and fields
  - `<reg root>.randomize()`
  - `<reg root>.<reg>.randomize()`
- Parallel access
  - Only `write`, `read`, `mirror`, `update` etc. is guarded by the `semaphore`
  - If multiple processes, try to randomize the register model in parallel then it might yield erroneous results

Register Model Constraints

- Add the constraints directly in the generated code
  - Constraints can be reused through derivation but not during runtime
  - Problematic to support multiple constraints sets in the same simulation

- Use with clause
  - Better/worse reuse model
    - Better: Sequences can be executed multiple times
    - Worse: Derived sequences for fine tuning constraints will repeat constraint code leading to redundant implementations
  - Multiple constraints sets can be supported

Attempts From StackOverflow

- Constrain register related to each other using with clause

```
reg_model.randomize(register_a,register_b) with
                    {register_a.get() > register_b.get();}
```

  - Assumes reg model has rand handles of registers

- Constraint all registers to 0

```
regm.get_registers(regs, UVM_HIER);
for (int unsigned r=0; r<regs.size(); r++) begin
    assert(regs[r].randomize with { regs[r].get() == 0; });
end
```

  - Assumes reg model has rand handle to register fields

Problem Solved - Well Almost 😐

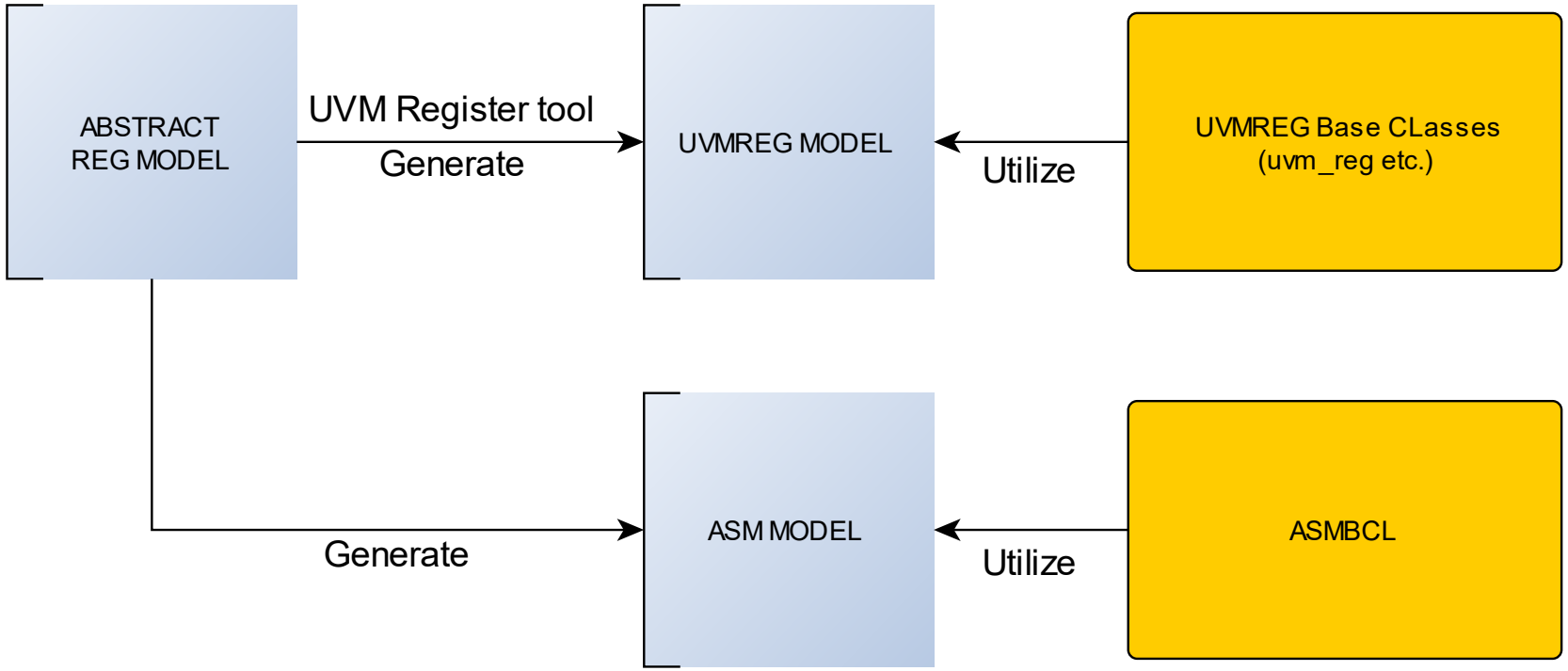
- Problems can be solved by implement a small alternative stimuli model (ASM)
  - Captures relevant register values and constraints
  - Can be randomized and applied to the register model
  - 1:1 mapping to the hierarchy of the register model
- Benefits
  - Independent from the register model
  - Reuse model intact. Constraint set can be reused and fine tuned through derivation
  - Parallel access problems fixed as the ASM is independent
- New Problem(s)
  - When the current register values must be used for randomization then the ASM has to be updated with these values
  - More classes to be compiled

Problem Solved ☺

- ASM can be generated from the same meta data as the register model itself
- Idea
  - Add method for preloading the ASM with the current register values in the register model prior to randomization
  - Add method for applying the randomized values to the register model

Framework

- Generation Tool for generating the ASM model
- ASM Base Class Library used for ASM model API definition



ASM Base Class for `uvm_reg`

```
package asmbcl;
  class reg_rnd_base extends uvm_object;
    rand bit apply_default;

    constraint co_default_apply_default {
      soft this.apply_default == 1'b0;
    }

    extern virtual function void set_apply_default(bit apply_default);
    extern virtual function bit  get_apply_default();
    extern virtual function void transfer_from_model();
    extern virtual function void transfer_to_model();
    extern virtual function void set_reg_root(uvm_reg register);
    extern virtual function void set_rnd_mode(bit mode);
  endclass
endpackage: asmbcl
```

Test Scenarios

- SC0 [Legal]
  - Randomize the register model with legal values and apply them to the DUT
- SC1 [Illegal]
  - Randomize the register model with illegal values and apply them to the DUT
- SC2 [Partly re-randomization]
  - Randomize once the model with legal values and apply them to the DUT
  - Randomize it again keeping some fields with the previous value and apply the new values to the DUT
- SC3 [Reset to default]
  - Randomize some fields and others reset to default and apply the new values to the DUT
- SC4 [Mix of reset and re-randomization]
  - Randomize once the model with legal values and apply them to the DUT
  - Randomize it again keeping some fields with the previous value and others reset to default and apply the new values to the DUT.

Example SC0

- `reg`: 32 bit register with
  - `value0`, `value1` and `value2`: Each 8 bit fields
  - `threshold`: 8 bit field
  - Legal constraint: `value0+value1+value2<=threshold`
- Generated ASM model

```
class reg extends asmbcl::reg_rnd_base;
  typedef bit [7:0] t_val;
  typedef bit [7:0] t_threshold;

  rand t_val       value0, value1, value2;
  rand t_threshold threshold;

  constraint co_apply_default {
    this.apply_default == 1'b1 -> (this.value0 == 0 && this.value1 == 0 && this.value2 == 0);
    this.apply_default == 1'b1 -> this.threshold == 'h4;
  }
endclass
```

Example SC0

- Legal reg class

```
class reg_legal extends reg;
  constraint co_legal {
    this.threshold >= this.value0+this.value1+this.value2);
  }
endclass
```

Example SC0

- Create, Randomize and Apply

```
class reg_legal_seq extends uvm_sequence;
  …
  task body();
    reg_legal rl = reg_legal::type_id::create("rl");

    rl.set_reg_root(this.model.reg);

    if(!rl.randomize()) begin
      `uvm_fatal("SC0", "Unable to randomize")
    end

    rl.transfer_to_model();

    this.model.update(status);
  endtask: body
endclass: reg_legal_seq
```

Results

- Writing Constraints
  - Constraining is also a lot easier since constraints can just be expressed directly without the `.value` etc.
  - Natively support randomization of all hierarchies
- Applying Constraints
  - Application and reuse has also been improved since constraints now can be naturally reused via derivation.
- Parallel Access
  - ASM model created locally and dynamically in e.g. a `uvm_sequence`
  - ASM model populated in zero time to preload it with the current values prior to randomization
  - Once randomized then it can return the randomized register values in zero time

Conclusions and Future Work

- It has been demonstrated that the ASM model framework provides
  - A better reuse model for reuse of constraints
  - A more natural way of expressing constraints on register values
  - Removes problems with parallel access
  - Extra overhead from additional files to compile proved to be insignificant
- Nested ASM models are currently not supported
- Further development of our in-house register tool in Python
  - Uses the SystemRDL parser: https://github.com/SystemRDL/systemrdl-compiler

Questions