Automated Generation of RAL-based UVM Sequences

Vijayakrishnan Rousseau
Validation Lead, Intel Corporation
1900 Prairie City Road, Folsom, CA 95630
vijaykrishnan.rousseau@intel.com

Satyajit Sinari
Validation Engineer, Intel Corporation
1900 Prairie City Road, Folsom, CA 95630
satyajit.j.sinari@intel.com

Benjamin Applequist
Validation Engineer, Intel Corporation
1900 Prairie City Road, Folsom, CA 95630
benjamin.t.applequist@intel.com

Timothy McLean
Validation Engineer, Intel Corporation
1900 Prairie City Road, Folsom, CA 95630
timothy.a.mclean@intel.com

Geddy Lallathin
Validation Engineer, Intel Corporation
1900 Prairie City Road, Folsom, CA 95630
geddy.lallathin@intel.com

***Abstract*-This paper addresses the need for automated generation of UVM sequences for validating extensive and complex designs. We derive a method of parsing an architect level master spec and auto-generating UVM sequences along with the Register Automation Layer (RAL). We discuss a method to improve on simulation time by compartmentalizing the testbench randomizations using some links between the RAL and DUT-based config objects within the config database facilitating complex constraint definitions.**

I. Introduction

In the hardware industry, the Universal Verification Methodology (UVM) is used widely for verifying designs. Hardware behavior is typically controlled by a set of registers, and the Register Abstraction Layer (RAL) in UVM is used for modelling the registers within a Design Under Test (DUT) on the testbench side. [1] UVM sequences are used to provide stimulus which consists of register reads, writes and polls. The architecture specification is used to dictate a specific order of programming the registers, in order to enable different features within a DUT. The need for automation arises when there are multiple registers being programmed for a feature within a DUT and the hardware supports multiple such features. Manually defining these many UVM sequences turns out to be cumbersome and difficult to maintain, as any minor changes in a feature specification, a register definition or a programming restriction needs to be reflected manually in the updated sequences. To solve this, we propose a way to automatically define UVM sequences straight from the design specification. To achieve this, we define a machine readable hierarchical 'state machine' format to define feature programming sequences within the DUT. UVM sequences will be autogenerated using parser scripts directly from these state machines. This paper also proposes a framework for complex constraint definitions and partial RAL randomization through configuration objects.

II. Overview

Registers need to be programmed in a specific order to enable a feature within a DUT. The RAL in UVM is used for modelling the registers on the testbench side, and UVM sequences are used to provide stimulus comprising of these register reads and writes to the DUT using a UVM driver through a test. [2] This is a typical usage of the RAL and the sequences in a UVM environment. We build on this behavior to accommodate autogenerating the UVM sequences as well as the RAL with the appropriate links to the configuration database as explained below. This allows us to handle validating extensive designs and constraint complexities without extensive overhead.

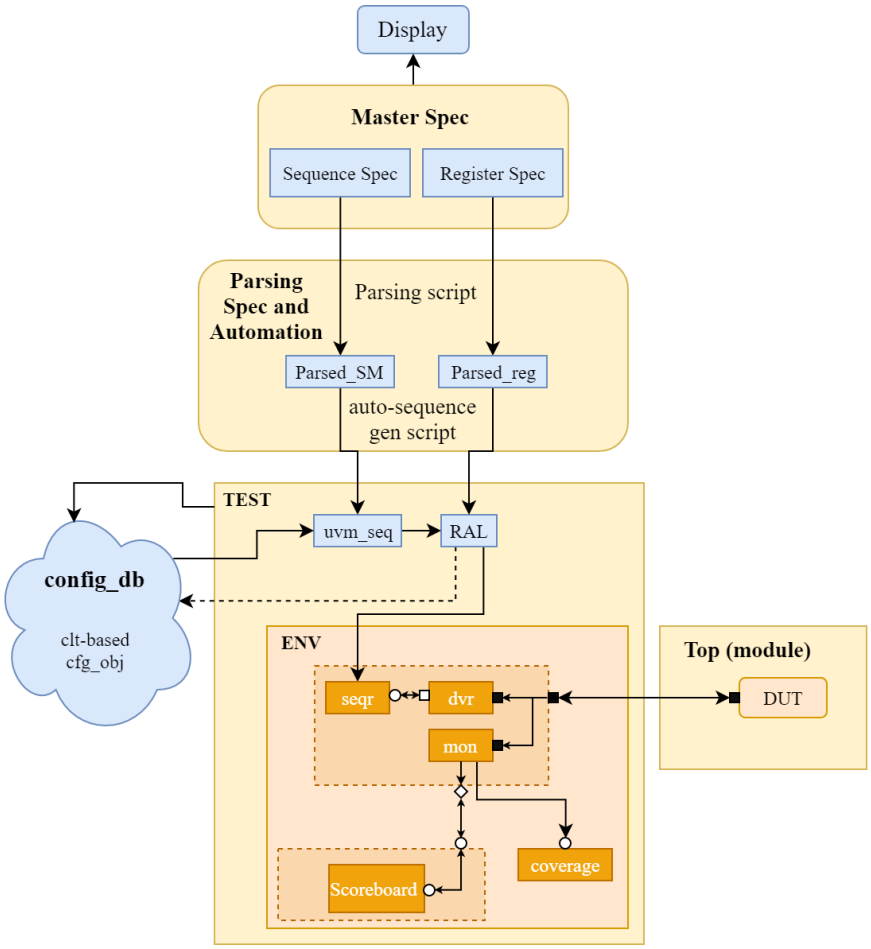

Figure 1. Modified UVM test framework integrated with the RAL and UVM sequence auto-generation scripts.

A. Modified UVM Testbench

A Master Spec dictates the registers and programming flow of the hardware. We use parsing scripts to autogenerate the RAL from the Register Spec, and all the basic constraints including legitimate register field values, bit field length, etc. are auto populated in the RAL.. In parallel we define the Sequence Spec within the Master Spec. Typically, sequences are defined in the specification using steps which are in plain English. However, in order to auto-generate the register programming sequences of a DUT, it is required that the specification is available in a machine-readable format. We introduce using State-Machines to define sequences within the Sequence Spec. This is achieved by representing each functional block of the smallest programmable hardware entity as a state which contains the associated register programming sequence. These State-Machines, implemented as Directed Acyclic Graphs (DAGs), have states connected in a hierarchical fashion to represent the programming model of the DUT at different levels of verification. We chose DAGs as they ease up the process of parsing and auto-generation with minimal branches. A library of UVM sequences (uvm_seq) will be generated and the test-writers can choose to start the sequences from any level of abstraction varying from unit level, IP (Intellectual Property) level, and block level to sub-system level etc. within the test depending on the verification environment.

The sequences dictate the order of register reads and writes, thus we need to have a link between the uvm_seq and the RAL for the UVM sequences to fully access and manipulate the registers. For a fully randomized testbench, we randomize the values of the registers in the RAL at runtime. Randomizing the RAL introduces additional problems for large and complex designs, due to constraints becoming complicated with dependencies across multiple registers and bitfields. This is difficult to contain within the RAL itself and asking test writers to manually define a lot of in-line constraints is equally as tedious. We also want to avoid randomizing the entire RAL and only target the blocks we are validating under the DUT, as this will save considerable simulation time. For these challenges, we use a hierarchical configuration object in the configuration database (config_db) with direct links to the RAL. Every DUT will have a corresponding configuration object which includes both the registers owned by that unit and the configuration objects of any subunits. Any complicated constraints that are difficult to be auto-generated within the RAL are included in these configuration objects. These are constraints that apply to specific unit-based registers or which cannot be defined at a lower level. For example, constraints which have dependencies on multiple subunits' registers would fall in this category. The configuration object also allows for a higher-level view of the DUTs functions, providing test writers with an abstracted constraint layer not tied to the underlying registers.

B. Overall Implementation

To link the RAL with the configuration object, we initialize the registers in every configuration object as handles to the RAL's version of those registers. Then we randomize the configuration object of our DUT. This picks up any simple constraints that were able to be auto-generated along with the RAL (valid values for each of a register type's bitfields), any manually defined multi-bitfield dependent constraints in the configuration object class, as well as any in-line constraints in the randomization call (typically used by test writers for specifying test scenarios in our strategy). This will not randomize registers outside of the current DUT, and ignores constraints defined at higher level DUTs, which will improve our simulation time.

To give better controls to the functional blocks, we define linked signals that we call 'Intent Triggers' within the config_db. Intent Triggers are bits that have a one-to-one relationship with the functional blocks. These are exposed to all the sequences and derived tests in the UVM environment (ENV) and are primarily used by the UVM sequences to bypass unnecessary register commands based on the specific DUT.

III. IMPLEMENTATION

As an example, we define the generic registers, SYS_CTL and FEATURE_CTL to demonstrate the strategy. The first essential areas are the master specs. It is crucial that they are well defined and machine-readable.

```
//Example Register Definition schema:
Register SYS_CTL
Bitfields:
    0:0     enable
    1:1     low power mode
    2:31    unused
end

Register FEATURE_CTL
Bitfields:
    0:3     mode
                Valid Values:
                    disabled        0x0
                    minimum         0x1
                    average         0xA
                    fancy           0xF
                end
    4:31    unused
end
```

The basic example above can be parsed by a script and reformatted into register classes to be used in the RAL. This is a made-up one-off approach, but it would be wise to consider using more standardized formats such as JSON or XML as they are well supported and offer various libraries for easier development.

```
//Example Register class output:
class system_ctl_reg extends uvm_reg;
    `uvm_object_utils(system_ctl_reg)
    rand uvm_reg_field enable;
    rand uvm_reg_field low_power_mode;
    uvm_reg_field      unused;
endclass : system_ctl_reg

class feature_ctl_reg extends uvm_reg;
    `uvm_object_utils(feature_ctl_reg)
    rand uvm_reg_field mode;
    uvm_reg_field      unused;

    // Simple constraint easy enough to auto-generate:
    constraint feature_ctl_possible_values {
        mode.value inside { 4'h0, 4'h3, 4'hA, 4'hF };
    }
endclass : feature_ctl_reg
```

Now, we define the sequence spec. This has similar requirements to the register spec: consistency and unambiguity to allow the parsing scripts to do their job.

```
//Example register programming sequence specification:
<Sequence Name="FEATURE_ENABLE">
    <Task Name="Enable_System">
        <Write Register="SYS_CTL"/>
    </Task>
    <Task Name="Enable_Feature">
        <Write Register="FEATURE_CTL"/>
    </Task>
</Sequence>
```

Hiding in the spec above, there's an actual UVM sequence that can be generated:

```
//Example auto-generated UVM sequence:
class feature_enable_seq extends uvm_sequence;
     `uvm_object_param_utils(feature_enable_seq)
     ral_class registers; //RAL instance, retrieved from config db

     task body();
            super.body();
            ral_class.feature_ctl.update(status);
            if(status != UVM_IS_OK)
                  //Report error
            ral_class.sys_ctl.update(status);
            if(status != UVM_IS_OK)
                  //Report error
     endtask
endclass
```

Note that we only need the call to ‘update’. The register values are set in the test either explicitly through constraints or during the randomization of the configuration object before the sequence starts.

```
//Example test:
class feature_test extends uvm_test;

    sys_cfg cfg;                //DUT config object
    feature_enable_seq seq;     //Autogenerated sequence

    task run()
        uvm_test_done.raise_objection();

        //Randomize call randomizes the RAL register values
        cfg.randomize() with {
            cfg.feature_ctl.mode.value != 1'b0; //Ensures the feature is enabled
            cfg.sys_ctl.enable.value == 1'b1;   //Ensures the system is enabled
        };

        seq.start(environment.agent.sequencer); //Updates the RAL registers

        //Registers are configured.  Do other test stuff here.

        uvm_test_done.drop_objection();
    endtask
endclass
```

The remaining bit is the configuration object itself. The important thing here is to link the relevant registers to the RAL. Multi-bitfield programming restrictions should also be added here in the form of constraints. If the DUT has any subunits, its configuration object will also contain lower level configuration objects for them.

```
//Sample configuration object
class sys_cfg extends uvm_object;
    rand feature_ctl_reg feature_ctl;
    rand sys_ctl_reg sys_ctl;
    `uvm_object_utils(sys_config)

    //Difficult constraint to autogenerate, so it’s manually defined here
    constraint low_power_mode_nothing_fancy {
        if(sys_ctl.low_power.value == 1'b1)
            feature_ctl.mode.value != 0xF;
    }
    task connect();
        ral_class registers;

        if(!uvm_config_db #(ral_class)::get(null, "*", "registers", registers)) begin
            `uvm_error("dup_config.connect()", "Could not find RAL!  Where did it go?")
        end

        //Link to the RAL
        this.sys_ctl        = registers.sys_ctl;
        this.feature_ctl    = registers.feature_ctl;
    endtask
```

```
endclass
```

If this RAL was configured to output a log, the output of this test may look something like this:

Time	Action	Register	Value
2ps	Write	sys_ctl	0x3
806ps	Write	feature_ctl	0xA

Note that the values in the "Time" and "Value" columns can change per execution of this test depending on the randomization, but the "Register" and "Action" columns will be consistent.

IV. Future Improvement

One shortcoming of this strategy is that setting up the register value programming is still manual. It's feasible to construct a robust enough spec language to formalize complicated register configuration restrictions in a machine parseable way. In this case, even the configuration objects could be partially or completely auto-generated. This would save a considerable amount of time and effort.

V. Conclusion

These state machines will give architects the ability to directly define and maintain the spec, and generation of the UVM sequences keeps the test-benches at all levels of verification in sync with the spec without any manual intervention. This will remove the extra effort needed from the test-writers to manually check the spec periodically to generate valid stimuli to their DUT. Such a sequence generation script can even be integrated into testbench compilation process to guarantee up-to-date sequences any time a verification environment is built. This not only eliminates the man-hours required for spec to code translation, but also allows a direct architect to RTL control point, thus reducing the chance for human error. There is a significant onetime startup cost with the development of the spec languages and their corresponding parser scripts, though it will eventually be overshadowed by the time saved from automating the recurring task of manual sequence generation. Though this strategy falls short in supplying actual register values, the proposal addresses this by referencing the RAL directly which provides an intuitive way of defining complex constraints and adds DUT level randomization to the RAL.

References

[1] IEEE Standard for Universal Verification Methodology Language Reference Manual," in IEEE Std 1800.2-2017, 26 May 2017

[2] IEEE Standard for SystemVerilog--Unified Hardware Design, Specification, and Verification Language," in IEEE Std 1800-2017 (Revision of IEEE Std 1800-2012), 22 Feb. 2018