Simpler Register Model Package for UVM Testbenches.

Sanjeev Singh
Juniper Networks
2530 Meridian Parkway, Durham
North Carolina 27713
sanjeevs@juniper.net

***Abstract*-A register model (or register abstraction layer) is a model of the software visible registers and memories in the design. Currently the "UVM" distribution (UVM package) [1] ships with the UVM_register package (UVM registers) that can be used to model these registers and memories in the design. However, this package adds a significant load and build time performance penalty for large system-level test benches where we have thousands of registers and embedded memories. Also the current UVM register access API is confusing to use. It mixes multiple independent concerns of randomization and access policies in the same API. Hence the itch, for developing an open source, clean slate, simpler register model (SRM) framework that is lightweight and has a cleaner API for the test writer. The source code is released under MIT License and available on GitHub.[2]**

I. INTRODUCTION

Currently the UVM distribution ships with a register package (UVM registers) that can be used to model registers and memories in the design. It integrates well with the UVM verification flow since it clearly separates register access from the actual bus transactions. Using the Adapter design pattern, it allows the generic bus transactions generated by the register model to be converted to the actual bus operation. This makes test sequences easier to write and reusable across multiple levels of abstraction.

However, there were multiple issues when migrating to the UVM register model. These are detailed below.

A. Large Memory Footprint

UVM register package causes memory to be statically allocated for all the locations in the address map, independent of the actual locations accessed by the test. In fact, each register needs to be as large as the largest register in the design. It is well known that most tests, especially at the system level, access only a small fraction of the address space. Hence it is wasteful to pay for all the locations in each test. [3]

UVM register does provide a "vreg" lightweight api. This is just an access API with no storage inside the model intended for memories with their own custom model (like external memories). Hence this is not a valid solution for on chip structures.

B. Randomization Fails For Large Tables

In current chips there are extensive tables implemented as embedded memories inside the design. UVM register has no equivalent classes to model these tables and instead implements them as static array of registers. Time taken for randomization on these static arrays increases exponentially with the number of entries. Practically only tables of less than 32K entries can be randomized.

C. Confusing Access API

The UVM register package exposes 3 values associated with each register ("mirrored value", "desired value" and "random value") to the test writer. To manipulate these, the test writer has to choose a multitude of access functions like "mirror", "update", "predict" etc. This strategy of creating multiple access methods to implement independent concerns of randomization and access type, confusing and hard to remember for the test writer. It was interesting to find that others on the internet had reached the same conclusion. [4] [5]

D. Hard to Understand Code

In UVM 1.1d, the 26 files are containing over 22,000 lines of source code for the register package. In the User Guide, the description of the UVM Register layer, is alone 26% of the number of pages.[5] Hence it is hard to understand and modify it.

II. SIMPLER REGISTER MODEL

Simpler Register Model (SRM) is an open source, system verilog package, released under MIT license. It can be used in UVM test benches instead of UVM register package. It is easy to integrate as it follows the same Adapter design pattern of having the register model generate generic bus transactions, and the user providing adapter classes, that translate these to bus specific transactions. However, it fixes the problems outlined above and provides some extra benefits over UVM registers.

This paper gives key code excerpts for certain concepts, but for full understanding, the complete working code is available at the URL below.

https://github.com/Juniper/simple_reg_model

A. Efficient Memory Footprint

SRM models tables by the "srm_table" which uses an associative array for storage. Hence entries are allocated only when written. The registers are modeled as template classes "srm_reg" and so the storage is limited to the actual size of the register.

The graph below in **Figure 1** shows the memory footprint of a UVM testbench with a different number of entries in an 8bit table. This testbench is a modified version of the popular example on the web Cluelogic[6]. The source code is available on GitHub.[7]

In this benchmark, the test accesses a single location, but we increase the depth of the table in the register model. Running on 64-bit Incisive simulator 15.10, we find that the memory footprint of the testbench starts increasing exponentially after 10K entries. By 400K entries, it has risen by over 1Gbyte of memory. This clearly does not scale even for current chips. With SRM, the memory is used only for locations written to.

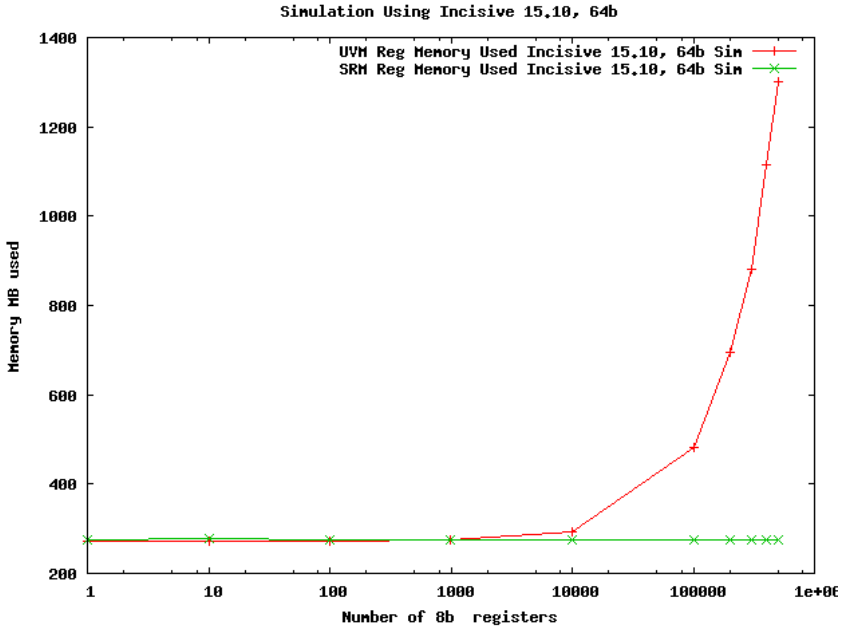

Figure 1 Memory Footprint in Benchmark Sim

B. Randomization Works For Large Tables

Randomization in SRM is implemented separately as a separate hierarchy of system verilog constraint classes. The test writer can apply the constraints on each entry of the table individually in a loop instead of applying it to the entire table in one shot.

As shown below in **Figure 2**, the time taken by the randomization call on the table is linear instead of being exponential with the size of the table as in UVM register package.

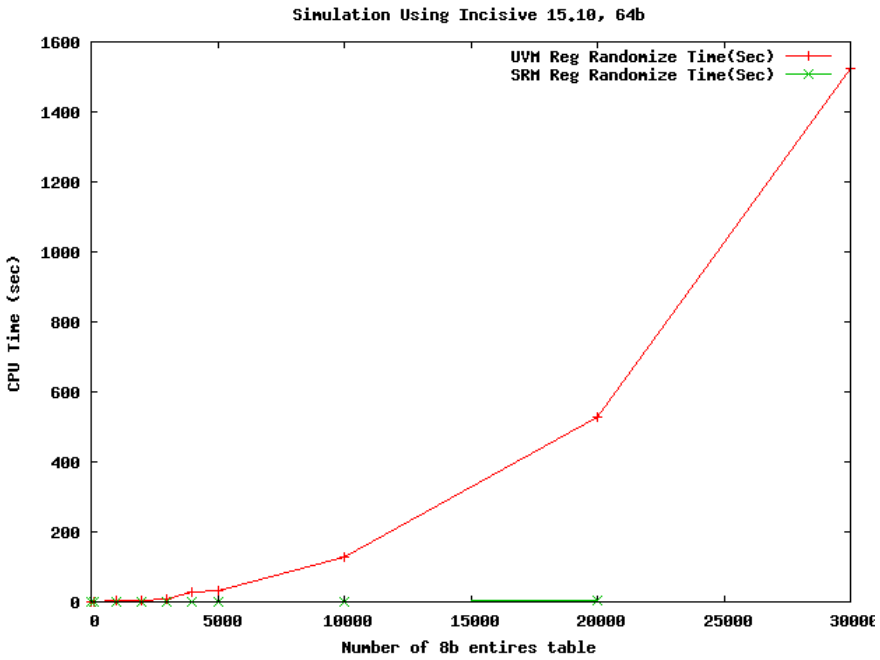

Figure 2 Randomizing Tables in Benchmark Sim

C. Clean Access API

The SRM access API exposes only the last value written by the test writer. Writes to the register always update this value and read, by default, checks all the nonvolatile fields in this value. Field access are allowed and the generic bus transactions have byte enables to identify the bytes being written and read to. It is the responsibility of the bus adapter to translate these partial transactions to actual writes and reads.

D. Modular Code

The core SRM consists of < 4K lines of code. It cleanly separates randomization, access type (frontdoor/sidedoor/backdoor) into a separate class hierarchy, so each class has only a single responsibility. Each class has unit tests to enable high coverage.

III. SRM TESTBENCH INTEGRATION

The steps to integrate SRM into test benches are similar to UVM registers. The first step is to generate the register model by deriving from the SRM base classes. This process is tedious and boilerplate, so it is expected that the user will use a register generation tool to generate it. SRM does have its own python utility "srm_rgen.py" that can convert the user specification in python to system verilog code.

The test bench can be setup in either active or passive operation.

A. Active Operation

In active operation, the register access is sourced from the register model via the access API. These read and write operations on the register model are converted to generic bus transactions by the model. These are then forwarded to the selected bus adapter as per the configured adapter policy. The bus adapter chosen implements the actual conversion from the generic transaction to the specific bus transaction. Unlike UVM registers, even backdoor is implemented as a separate adapter rather than being a part of the register model.

This is shown below in Figure 3. Here the model has a handle to 3 adapters. The backdoor adapter may just use dpi to directly set/get the design flops. A frontdoor adapter instead would run the actual bus transaction to implement the transfer. A sidedoor adapter may skip the DUT host block and issue the access directly to the block's local address decoder (Referred as "Pio Decode" in the figure).

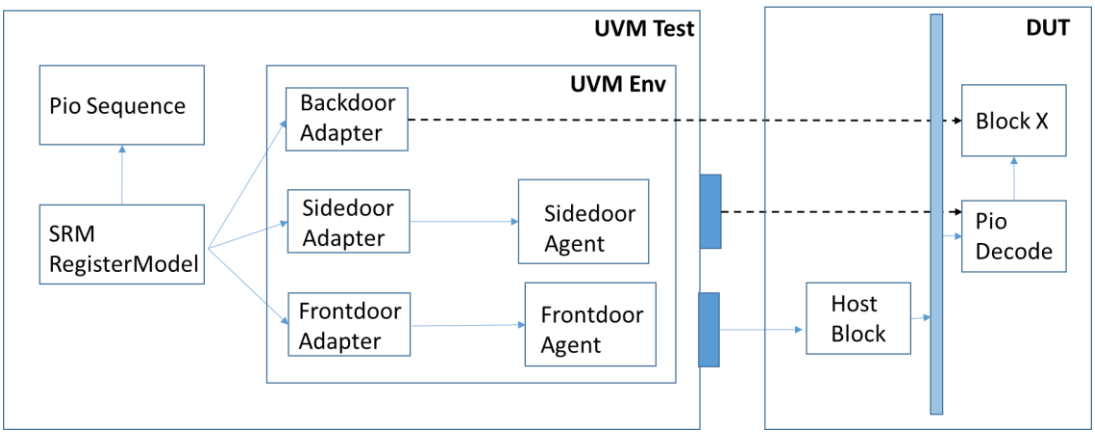

Figure 3 Testbench Active Operation

B. Passive Operation

For passive operation, the register access is not generated by the register model. This could be created by some other mechanism like embedded software, or a legacy non UVM environment but the model is still required to be accurate for the checks and functional coverage.

In this case, a predictor is used to observe the bus transactions and then forward them to the SRM register model. An example of Instruction set simulator based test bench is shown below in **Figure 4.**

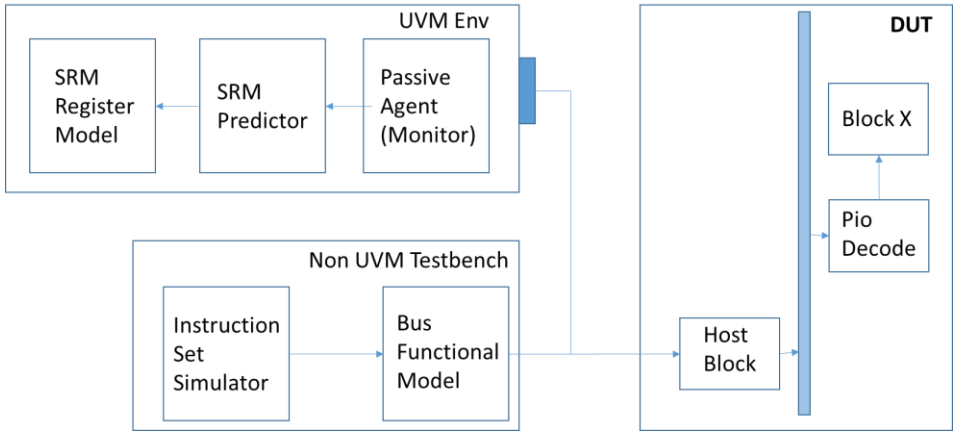

Figure 4 Testbench Passive Operation

III. SRM OVERVIEW

SRM models two types of elements, a register, and a table. A register is a collection of fields which is a group of an arbitrary number of bits treated as one unit. A table is an array of registers. It is modeled using the Composite design pattern. The function "entry_at(i)" returns the table entry at index 'i' which is an instance of the register class "srm_reg". Hence both register and a table entry are equivalent, and I use the generic term "SRM register" to refer to either of them.

The handles to the registers and table can be stored in "srm_node" class which can also hold instances to other nodes. Using this we can construct a tree with the "srm_reg" and "srm_table" being the leaf nodes. Such a tree represents an address map. There can be multiple instances of a tree, corresponding to different address maps, inside a register model. Each address map defines all the control information like the unique offset of all the nodes in the tree, the policies of each field, etc. For leaves shared between address maps, aliases can be setup, to ensure that the model only keeps a single storage across multiple address maps.

IV. SRM DATA MODEL

SRM uses a packed structure to describe the structure of the register. Each field of the register has a single storage element holding the last data written to the field or the reset value, if any, at startup. For illustration let us assume that we have an entry "r1" that has single 32 bit read, write field. The sample code in **Figure 5** describes a srm register.

```
// Struct for describing fields of register
typedef struct packed {
  bit [31:0] field;
} r1_struct_t;

// 32b Register Model
class r1_reg extends srm_reg#(r1_struct_t); // Template base class
    srm_field#(bit[31:0]) field;

    function new(string name, srm_component parent);
        super.new(name, parent);
        field = new(.name(field), .parent(this), .n_bits(32),
                    .lsb_pos(0), .volatile(0));
        add_field(field);
    endfunction

endclass
```

Figure 5 SRM Register Declaration

Similarly, we can also create a 100 entry table having the same fields as shown in **Figure 6**. The class "srm_table", holds a handle to the prototype register entry. On the first write to the index, it clones the prototype to create the entry and stores it in an associative array.

```
// SRM Table model with 100 entries.
class r1_table extends srm_table#(r1_struct_t);
    class r1_entry extends srm_table_entry#(r1_struct_t);
       srm_field#(bit[31:0]) field;
       // boiler plate code ….
    endclass

    function new(string name, srm_component parent);
        r1_entry entry;
        super.new(name, parent, .num_entries(100));
        entry = new(.name("r1_entry"), .parent(this));
        _prototype = entry;
    endfunction

endclass
```

Figure 6 SRM Table Declaration

V. SRM ACCESS METHODS

The SRM register has a set of 4 primitive access methods described in **Table 1** and 3 composite methods in **Table 2**.

The primitive methods consist of 2 tasks "read" and "write" that access the DUT and 2 functions that "get" and "set" the value of the model only with no side effects. These primitive methods apply to only leaf nodes and can be executed either on the entire entry or any field of the entry. The first argument to the read/write task is an instance of "srm_pio_handle" which holds all the configuration information required for the access.

Name	Example	DUT	SRM Model
Read	`model.r1.read(handle, rd_data);` `model.r1.field.read(handle, rd_data)`	Reads the value from DUT.	Checks nonvolatile field against the read value and updates volatile fields.
Write	`model.r1.write(handle, 0x1);` `model.r1.field.write(handle, 0x1)`	Writes the value to DUT.	Writes the value to the model.
Get	`rd_data=model.r1.get()`	Ignored	Reads the value from model in zero time.
Set	`model.r1.set(wr_data)`	Ignored	Updates the value in the model in zero time.

Table 1 Basic Access API

The 3 composite methods are tasks that apply to any node. For non-leaf nodes, SRM will call the methods on all the leaf nodes below the node.

Name	Example	DUT	SRM Model
Load	`model.node.load(handle)`	Reads all the DUT leaf nodes values.	Updates all the leaf nodes with the read value.
Store	`model.node.store(handle)`	Writes all the DUT leaf nodes with data from model.	No change
Store_Update	`model.node.store_update(handle, new_model);`	Write all the values to DUT from new_model that are different from current model.	Update all the values that are different in the new_model.

Table 2 Composite Access API

Typical use of composite methods are:

1) At the end of the test, the test writer wants to read say all the counters inside the STAT block. By specifying the load command at the STAT node, causes all the registers inside it to be read and the register model populated.
2) Test writer can create test configurations using the register model alone. Once complete it can be downloaded to the DUT by applying "store" at the root node.
3) When multiple configurations need to be applied, the test writer can save simulation time, by using "store_update". The method will cause writes to the leaf nodes which have the same value to be skipped.

VI. SRM RANDOMIZATION

Support for randomization is supported by a separate constraint class that is registered with the UVM factory. The test writer can override this type to generate any custom constraints. The constraint class has the method "get_data" that can return the final random value to be passed to the access methods. An example of a basic constraint class for a 32b read/write register is shown in **Figure 7**

```
// Struct for describing fields of register
typedef struct packed {
  bit [31:0] field;
} r1_struct_t;

// Register Base Constraint class
class r1_constr extends uvm_object;
    // Register with factory
    `uvm_object_utils(r1_constr)

    rand bit [31:0] field;           // Field values are made rand.

    function new(string name="r1_constr");
        super.new(name);
    endfunction

    function r1_struct_t get_data(); // Retrieve the data struct
       r1_struct_t d;
       d.field = field;              // Fill the data struct with fields
       return d;
    endfunction

endclass
```

Figure 7 SRM Constraint Class Example

A. *Register Randomization*

Constraints can be applied to the register by extending the base constraint class. These can be added dynamically using the Decorator design pattern as outlined in [8]. The test writer can simply override the constraint class in the factory before creating it. It is then randomized and data extracted using the "get_data" method. This is shown below in **Figure 8**.

```
// Register Randomization
r1_struct_t wr_data;
r1_constr c1 = r1_constr::type_id::create("r1_constr");   //Factory Create

assert(c1.randomize());              // Randomize the contents of register
wr_data = c1.get_data();             // Extract the data

model.r1.write(handle, wr_data);     // Write the data
```

Figure 8 SRM Register Randomization

B. *Table Randomization*

SRM tables can be randomized one entry at a time. Using the method "entry_at(i)" the test writer can access the register at each location and randomize it. This is shown below in **Figure 9** for a table of 32K entries.

```
// Table (32K entries) Randomization

r1_struct_t wr_data;
r1_constr c1 = r1_constr::type_id::create("r1_constr");// Factory Create

for(int i = 0; i < 32*1024; i++) begin                  // Loop over table
    assert(c1.randomize());                             // Randomize
    wr_data = c1.get_data();                            // Extract data
    model.r1_table.entry_at(i).write(handle, wr_data); // Write entry i
end
```

Figure 9 SRM Table Randomization Example

VII. SRM ACCESS MECHANISM

One of the significant differences between the UVM register model and SRM model is in the selection of the access mechanism. It is common for a single register to have multiple access mechanism like "Frontdoor", "Backdoor" or other proprietary mechanisms in a simulation.

For reusability purposes, it is crucial for the sequences to be agnostic of the access method. Depending on the context, the same sequence could use different access mechanism. For example, during configuration, the sequence would like to use backdoor access for saving simulation time. When spawned as part of pio background noise thread, the same sequence would like to use "frontdoor" even though backdoor may be available for a register. Similarly, for gate level simulations, the hierarchal path for the backdoor access to the register may no longer be available. In that case, it may be desired to run the same sequence unchanged using frontdoor mechanism only.

SRM handles this dynamic selection of access mechanism by creating an adapter class for each of the access mechanisms. Unlike UVM register, the SRM model generates only generic transactions and so even backdoor access are treated as just another adapter.

The user then initializes the list of all available adapter handles at the appropriate node in the address map hierarchy. At runtime, the user can specify the search algorithm to use in the srm_pio_handle (which is the first argument to the write and read tasks).

The read/write task then will search the adapters associated with the node for the correct one. If no match then it will walk up the hierarchy matching the list of adapters at its parent. Once found, it will dispatch the transaction to that adapter. This way the user can cause the sequence to use different adapters by passing the configuring the srm_pio_handle appropriately.

An example is shown in **Figure 10**. Here the dut consists of a single address map having 2 registers "c" and "d". The user has configured the register model address map to have the root node point to the frontdoor adapter and the leaf node "c" point to the backdoor adapter.

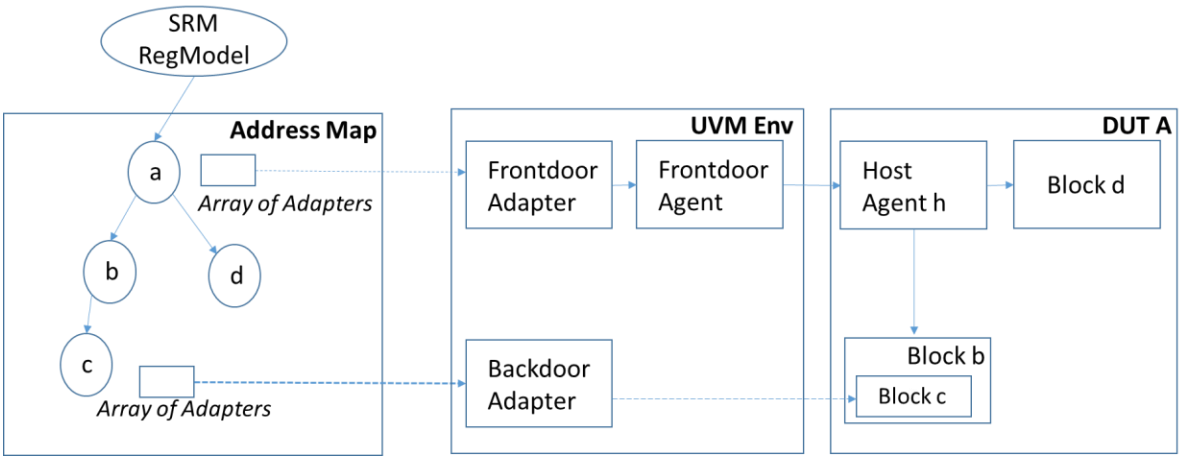

Figure 10 Access Mechanism Example

Now the user can initialize the srm_pio_handle to choose the first available adapter as shown in **Figure 12**.

```
// Custom selector class to choose the first adapter.
class first_adapter_selector extends srm_adapter_selector;
    function bit is_correct_adapter(srm_bus_adapter adapter);
        return 1;
    endfunction
endclass

// SETUP:Initialize the srm pio handle with the adapter selector
first_adapter_selector = new();                              // Create it
cpu_pio_handle.adapter_selector = first_adapter_selector;    // Install it

// Sample PIO Write in a sequence
a.b.c.write(h, 32'hdeadbeef); // Write to a.b.c uses backdoor
a.d.write(h, 32'h0);          // Write to a.d uses frontdoor
```

Figure 11 First Adapter Selector Example

Now if the test writer wants to use just the frontdoor adapter (say for a gate level netlist where the backdoor adapter is not available), then the code can be modified as shown in **Figure 12**. When the access is done for leaf node "a.b.c", then SRM searches for the correct adapter to use. Since the backdoor adapter does not match, SRM then searches the parents for the correct adapter. In this case the match will happen at the root node, and the write will be dispatched to the frontdoor adapter for all accesses.

```
// Custom selector class to choose only frontdoor adapter
class frontdoor_adapter_selector extends srm_adapter_selector;
    function bit is_correct_adapter(srm_bus_adapter adapter);
        if(adapter.name == "frontdoor") return 1;
        else return 0;
    endfunction
endclass

// SETUP: Initialize the srm pio handle with the adapter selector.
first_adapter_selector = new();
cpu_pio_handle.adapter_selector = frontdoor_adapter_selector;

// Sample PIO Writes in a sequence UNCHANGED.
a.b.c.write(h, 32'hdeadbeef); // Write to a.b.c uses frontdoor
a.d.write(h, 32'h0);          // Write to a.d uses frontdoor
```

Figure 12 Frontdoor Adapter Selector

VIII. SPECIAL REGISTER MODELLING

SRM supports 27 built-in field policies ranging from default read, write to esoteric strategies. Each address map can support different policies for the same field. In case, these do not fit; the user can implement any arbitrary policy. The user can add callback functions to the leaf register. These are detailed in **Figure 13**. On detecting a read or write to these callbacks get invoked. The user is then responsible for implementing all the state changes in the model.

```
// A return value of 1 means that processing is done else execute the next
one from the callback list. If list is empty then do the default behavior.

// Write callback to override
int function srm_write_callback(srm_reg_model model,  // Register Model
                                srm_base_reg node,    // Leaf node
                                int index,            // -1 for reg
                                ref srm_data_t data );// Data to be
                                                      // written
// Read callback to override
int function srm_read_callback(srm_reg_model model,   // Register Model
                               srm_base_reg  node,    // Leaf node
                               int index,
                               const ref srm_data_t dut_data); // Data read
                                                               //  from DUT
```

Figure 13 Callback API

IX. MISC FEATURES

SRM supports all the introspection methods of UVM registers. This can be used to write generic sequences. Functional coverage model can be created by the user by registering functional coverage callbacks with the node for read and writes.

X. UVM TESTBENCH EXAMPLE

A full uvm test bench using SRM package is available at the following URL.

https://github.com/sanjeevs/srm_sap1

In this example, a simple DUT is created having 2 blocks, "host" and "blockX". The blockX has single register and a single table consisting of 1024 entries that are 32b wide. This is modelled using SRM register package. Adapters are hooked up so that the registers in blockX can be accessed either through the host agent, or through the block's local pio decoder or through backdoor.

XI. SUMMARY

The UVM Register package is popular, and powerful, but complex. [5] However it is expensive to use and has an intricate API for the test writer. This paper proposes a new open source register model package "SRM" that is lightweight and has a more elegant API. The major weakness with SRM is that the new API is not backward compatible with UVM register and so can be used only for new test benches.

SRM register package is open source under MIT license. I hope that others in the community find the work interesting and can actively join in the development work.

ACKNOWLEDGMENT

Thanks to my employer Juniper Network for supporting and allowing me to release the code under open source license.

REFERENCES

[1] SystemVerilog UVM http://accellera.org/downloads/standards/uvm
[2] Simpler Register Model https://github.com/Juniper/simple_reg_model
[3] Litterick http://www.verilab.com/files/litterick_register_final_1.pdf
[4] AgileSOC, http://agilesoc.com/2014/03/09/youre-either-with-me-or-youre-with-the-uvm-register-package/
[5] Rich Edelman, Bhushan Safi, "Beyond UVM Registers – Better, Faster, Smarter," in DVCon India 2015
[6] cluelogic.com http://cluelogic.com/2012/10/uvm-tutorial-for-candy-lovers-register-abstraction/
[7] https://github.com/sanjeevs/uvm_reg_gotchas
[8] Design Patterns by Example for System Verilog Verification Env, Eldon Nelson, https://dvcon.org/sites/dvcon.org/files/files/2016/08_2.pdf