Replacing Hardcoded Register Values with Hardcore Abstraction

Alex Melikian
Hilmar Van Der Kooij

Verilab
Ottawa, Canada

www.verilab.com

ABSTRACT

Today's complex protocols typically involve built-in register functionality for configuration and operation purposes. This requires corresponding VIPs to mimic register functionality in order to enable exact behaviour and complex operations. VIP developers may resort to using a large number of hardcoded values representing register addresses, reset values or field locations. However, this approach is prone to errors and demands a high level of maintenance during the course of a project as register definitions often change. Our solution enables VIP developers to include register functionality in a manner that is not only scalable, but also flexible and robust to any changes. The solution involves leveraging the derived UVM register model classes, typically defined for mirroring and automating checks on DUT registers. These derived register classes can be used to create a layer of abstraction, allowing the VIP to be transparent to all pre-defined register related values in the protocol.

Table of Contents

Table of Figures

Table of Code Examples

1. Introduction

In addition to defining data formats, signal timing and operational procedures, some protocols and electronic standards define an addressable register set a device implementation must provide. These register sets serve to directly influence parameters affecting protocol operations or configurations. An example of such would be a low-level register which sets clock speeds on a serialized interface. This implies a Verification IP (VIP) created for the protocol must not only model the register access mechanism, but also model the consequential actions of these accesses on protocol operations.

In order to meet this objective, VIP developers may resort to using hardcoded values when generating the code related to handling and decoding the register accesses. These hardcoded values can represent anything from the register address, or a register field position. The following code would be an example where hardcoded values are used:

```
// if address and data is for register address, trigger field, do something
if ( virtual_if.addr == 16'h0100) && (virtual_if.data[5] == 1'b1 )
   do_something();
```

Code Example 1: Hardcoded Method to Decode Address & Field

Though the above code achieves its objectives, it has key shortfalls and vulnerabilities during the development cycle of the VIP or using it to verify a Design Under Test (DUT). Firstly, it is prone to buggy behaviour when register attributes change, such as a changes in address or field bit offset. Secondly, it may require a lot of code maintenance over the course of a project, especially in the early part of the design cycle when register definitions can frequently vary. Though the use of pre-defined macros could be employed to alleviate these issues, the approach would have its own set of pitfalls regarding code maintenance.

Fortunately, alternatives exist to make the above functionality in a VIP far more robust, self-maintainable and easy to read. The proposed solutions in this paper involve the use of register model classes in the VIP, derived from the UVM register model portion of the UVM.

2. Use of UVM Register Model Classes

Traditional Use Case of UVM Register Model for DUTs

First, let us review the traditional use case of the UVM register model. As elaborated in [1], the UVM register model is a set of classes created to facilitate the modeling of a DUT's memory-mapped register set and internal memories. Hence, the UVM library provides a set of classes a verification engineer can use to derive specialized classes to implement the specific functionality of each register in the DUT. The most commonly used classes from the reference document [2] are the following:

- `uvm_reg_field` : base class for an individual atomic field of a register

- `uvm_reg` : base class for an individual register, composed of a set of fields
- `uvm_mem` : base class for a collection of contiguous storage locations
- `uvm_reg_map` : utility class representing an address map, or collection of accessible registers and/or memories
- `uvm_reg_block` : base class representing a register and/or memory hierarchy which may possess one or multiple address maps.

The relationship between each class type is depicted in the figure below.

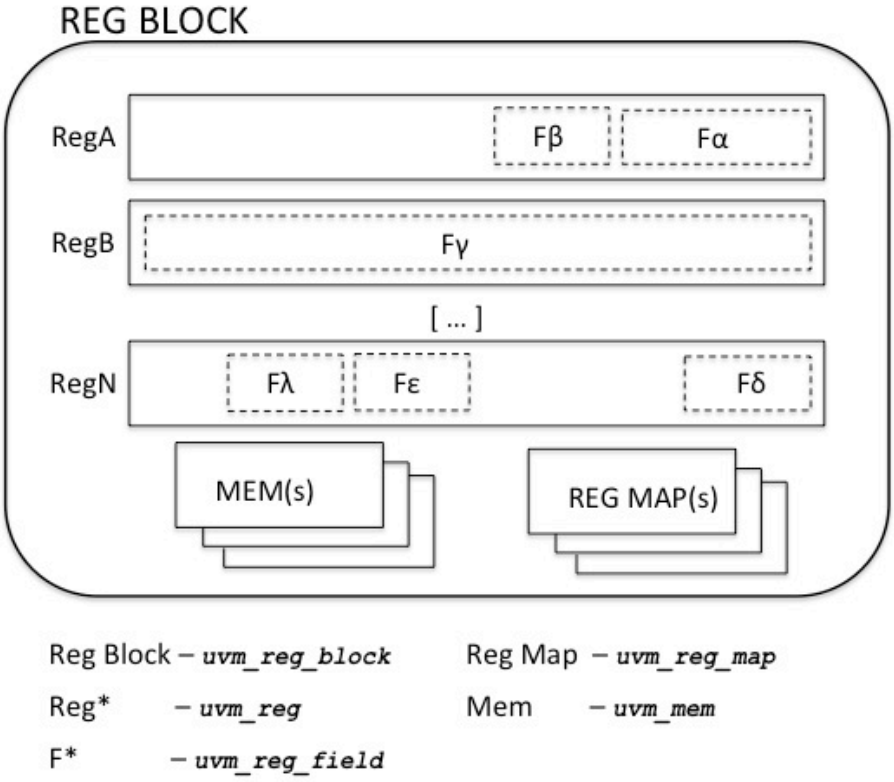

Figure 1: Standard UVM Register Model Structure

Along with other register related UVM provided classes [2], such as the `uvm_reg_adaptor` and `uvm_reg_predictor`, verification engineers can quickly integrate the register model they have assembled into the testbench for automated checks and stimulus for the DUT's register functionality.

Use Case of UVM Register Base Classes for VIPs

To clearly state the objectives of this paper, the recommended solutions proposed are not a replacement of this established UVM methodology for verifying DUT registers.

Rather, this paper outlines a set of implementation guidelines for a VIP, enabling it to emulate register access functionality of the protocol. These guidelines are based on requirements to create

a layer of abstraction between the actual functionality tied to the registers and its low-level details, such as specific addresses, offsets and field locations. In other words, these guidelines enable the VIP to avoid the pitfalls of using hardcoded values, and enable it to adapt to any low-level changes to register attributes.

The guidelines showcase not only the use of UVM register classes in VIPs, but also the rich set of built-in introspection and query methods provided by these classes.

3. Guidelines for Abstracting Registers from VIP Functions

Abstraction of the Register File

As mentioned previously, the UVM class library provides building blocks to quickly generate an address mapped register and memory model of a DUT. These building blocks can also be used to model the register set in a VIP, defined by the protocol specification it implements.

In other words, it is a model construct of derived `uvm_reg_field`, `uvm_reg`, `uvm_reg_block` classes that can be constructed for strict use by the VIP code. It serves not only to organize the protocol register set, but also encapsulate all low-level register details such as addresses, offsets and field indexes. The code for such a model register block would look just like one constructed for DUT verification, often autogenerated with a third party tool. Hence this structure would look like the code example below:

```
class my_vip_ref_regblock extends uvm_reg_block;
[…]
   protocol_reg1 protocol_reg1_reg;
   protocol_reg2 protocol_reg2_reg;
   protocol_reg3 protocol_reg3_reg;

   virtual function void build();
      protocol_reg1_reg = protocol_reg1::type_id::create("...");
      protocol_reg1_reg.configure(this);
      protocol_reg1_reg.build();

      protocol_reg2_reg = protocol_reg2::type_id::create("...");
      [...]
      protocol_reg3_reg = protocol_reg3::type_id::create("...");
      [...]

      default_map.add_reg(protocol_reg1_reg, 'h100, "RW");
      default_map.add_reg(protocol_reg2_reg, 'h104, "RW");
      default_map.add_reg(protocol_reg3_reg, 'h108, "RW");
   endfunction
endclass
```

Code Example 2: Typical UVM Register Block Declaration

The register block declaration in Code Example 2 serves as a reference for all register related functionality a VIP must model, and is critical for the abstraction of register details as described in this paper. A practical location where this reference register block can be located in the structure of a VIP would be at the agent level, as depicted in the following figure:

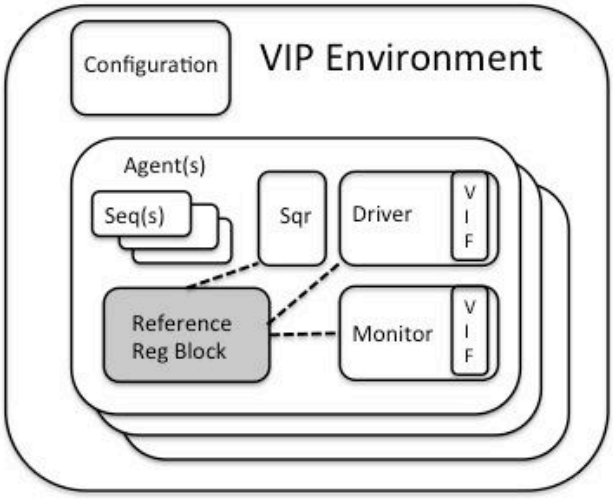

Figure 2: Proposed Location of Reference Register Block in VIP

This would allow the instance of the reference register block to be visible in all essential parts of the VIP through the use of simple pointers.

The reference register block example uses a flat hierarchy, but it is not the only way to structure a register model. In some situations, more complex hierarchical structures, such as an aggregation of `uvm_reg_block` derived instances defined with sub-maps, could be used as a register model for the VIP. Since a flat hierarchy satisfies most situations at a VIP level, for the sake of clarity, this paper initially presents code examples based on such a hierarchy. Further on, we will explore modelling more complex structures, and the impact they have on the VIP code.

Abstraction of Registers Accesses

Code Example 1 presented in the introduction is an implementation of a VIP capturing a write access attempt to a register containing a field named `trigger`. The context here is where the DUT, or other test-bench component, executes an access to the register holding the `trigger` field in order to activate some functionality in the VIP.

In Code Example 3, the same write access to the VIP is implemented with the hardcoded values replaced by UVM classes and their built-in methods. Before presenting the example, we can assume the VIP package contains or has access to the uvm class derived register reference block described in the previous section. In the example below, `protocol_reg1` is derived from the

uvm_reg class and included in the reference register block. It represents a protocol specific register containing the trigger field.

```
my_vip_ref_regblock  ref_vip_regblock;

ref_vip_regblock = my_vip_ref_regblock::type_id::create("...");
if ((virtual_if.write == 1) &&
    (virtual_if.addr == ref_vip_regblock.protocol_reg1.get_offset()) &&
    (virtual_if.data [ref_vip_regblock.protocol_reg1.trigger.get_lsb_pos()] == 1))
  do_something()
```

Code Example 3: Abstract Method to Decode Address & Field

There are a few points worth exploring and clarifying with the example:

- We declare and create using the UVM factory instance of the my_vip_ref_regblock. Once created, all registers included in the block have low level register details such as addresses, field indexes and offsets initialized.

- The second expression in the boolean statement eliminates the use of the hardcoded value for comparing register address by using the standardized uvm_reg::get_offset() of the protocol_reg1 instance of the reference register block.

- The third expression of the boolean statement eliminates the use of the hardcoded value for the register field index by using the uvm_reg_field::get_lsb_pos() of the trigger field instance in the same protocol_reg1 instance of the reference register block.

Though Code Example 3 is more involved than Code Example 1, there are a number of clear advantages. Firstly, it removes the use of hardcoded values which are vulnerable to creating hard to find test bench bugs during simulation. Secondly, the code is self-maintainable, meaning it adapts to any changes in address or register field locations in the class definition. These are benefits of having abstracted low-level register details into the reference register block. Furthermore, class definitions of these registers would be typically located in a centralized file and be common to all components of the VIP (driver, monitor, agent, etc). Hence the use of this abstract method would allow the entire VIP to automatically adapt to any change in the centralized definitions file.

It should be conceded that in extreme cases where registers radically change structure or locations, the self-maintainability of the code will likely break down. However, in these situations, the use of this abstracted code structure such as in Code Example 3 results in compilation errors which are trivial to fix. In the case where hardcoded values are used, not only are vendor tools unable to indicate any sign of a problem, but the use of hardcoded values can result in hard to find test bench bugs that occur during simulation run time, which will require more effort to debug.

Abstraction of Registers Accesses with Transaction Items

The previous guideline used dedicated 'address' and 'data' signals of a generic virtual interface that does not reflect any specific protocol. However, certain protocols, particularly serial-based ones, do not have dedicated interface signals involved with register access operations. Rather, an encoded data or signaling pattern is used. In such a case, the VIP in question, more specifically the monitor component of the agent, already decodes the data or signal patterns for its usual collection and reporting of protocol transactions. The same abstraction technique can be applied once the VIP monitor has interpreted and collected a transaction indicating protocol level register accesses.

The following code would be a variation of Code Example 3, where properties of a collected transaction item holding data and address values are used for register related operations.

```
my_transaction = my_vip_transaction::type_id::create("");
  [ … ]
  if ( vip_detected_write_operation()) begin
     my_transaction.register_operation = WRITE;
     my_transaction.addr = vip_decoded_write_address( );
  end
  [ … ]
  if (my_transaction.register_operation == WRITE) &&
     (my_transaction.addr == ref_vip_regblock.protocol_reg1.get_offset()) &&
     (my_transaction.data[ref_vip_regblock.protocol_reg1.trigger.get_lsb_pos()]==1)
    do_something()
```

Code Example 4: Abstract Method Using VIP Transaction Items

Typically, an ideal location in the VIP's monitor to tap-off a collected transaction for processing register functionality would be just before it transfers the transaction into its analysis port. At this point, the monitor has completed the collection and interpretation of the transaction and thus all contents of the transaction would be ready for use of processing potential register operations.

Abstraction of Registers when Storing VIP Register Values

The following example shows how some of the built-in UVM register class introspection methods can be used to interpret data and address values from the bus interface into storage structures. As demonstrated, no hardcoded values are needed to determine which slices of the bus data port map into which register fields serving as configuration parameters.

```
my_vip_reg2_class  storage_reg2_class_reg; // derived from uvm_reg

// Capture Data Values destined for reg2_class register
storage_reg2_class_reg = my_vip_reg2_class::type_id::create("...");
if ( (my_transaction.register_operation == WRITE) &&
     (my_transaction.addr == ref_vip_regblock.protocol_reg1.get_offset())
   storage_reg2_class_reg.set( virtual_if.data )
```

```
[ .. ]
// Extract config portions from corresponding register field values
config_param_a_value = storage_reg2_class_reg.config_field_a.get();
config_param_b_value = storage_reg2_class_reg.config_field_b.get();
```

Code Example 5: Abstract Method to Storing Register Values

Abstraction of Register Value Retrieval

The following is an example where an active VIP slave retrieves data to drive out on a bus for a read access. One manner to do so with hardcoded values may be the following:

```
if (virtual_if.addr == 16'h0100)
   virtual_if.data[ 7:5] = my_config_fieldA_value;
```

Code Example 6: Hardcoded Method to Returning Values on Interface

Applying the same approach of using standardized UVM register API functions, we can achieve the same functionality:

```
if (virtual_if.addr == ref_vip_regblock.protocol_reg2.get_offset()) begin
  lsb_position = ref_vip_regblock.protocol_reg2.fieldA.get_lsb_pos();
  for ( idx=0; idx < ref_vip_regblock.protocol_reg2.fieldA.get_n_bits(); idx++)
    data_val[ lsb_position + idx] = my_config_fieldA_value[idx];
  virtual_if.data = data_val;
```

Code Example 7 : Abstract Method to Returning Values on Interface

In Code Example 7, we use the introspection methods of a register instance (`protocol_reg2`) in the reference register block (`ref_vip_regblock`) to align bit values of an independent data structure (`my_config_fieldA_value`) to a data bus. As we will discuss in the following section, the independent data structure can be replaced.

Further Leverage of Reference Register Block

The reference register block is derived from the `uvm_reg_block` class, with properties that are typically an aggregation of `uvm_reg` instances. To verify correct DUT register operation, these `uvm_reg` derived instances are critical to retain, update and compare values reported by the testbench via `uvm_reg_adapter` and `uvm_reg_predictor` derived classes linked to the register model in the environment. By following the UVM guideline, the UVM reg model will automatically execute these functions by calling `uvm_reg::set()` and `uvm_reg::get()` access methods of each uvm_reg derived instance in the register model.

When modeling functionality of a VIP, the goal is not to verify the register functionality of a DUT, but rather mimic register operations in order to fully model a protocol's functionality.

Hence instead of using derived classes of `uvm_reg_adapter` or `uvm_reg_predictor`, the VIP code can directly call `uvm_reg::set()` and `uvm_reg::get()` access methods for retaining and retrieving register values.

With this approach, it can be easily argued that parallel data structures holding configuration values are not necessary. A VIP developer can use the reference register model as a "one stop shop" for not only adding a layer of abstraction over register attributes, but for register value storage and retrieval as well.

Complex Calculation Using Protocol Register Abstraction

Many protocols have configuration parameters that are based on multiple register values, often spread out across multiple registers. These are traditionally stored in a configuration object in the VIP. Building on the guidelines above, we can further leverage the abstractions the UVM register model provides to enhance the robustness of the configuration mechanism of the VIP. By making configuration parameters in their storage class accessible through access functions as opposed to directly exposing the fields, we enhance robustness of the configuration modeling by keeping configuration fields in a single location:

```
function field_a_type get_config_param_a();
      return ref_vip_regblock.protocol_reg2.config_field_a.get();
endfunction
```

Code Example 8: Abstract Method to Retrieve Register Values

Besides reducing the spread of configuration parameters, this approach also shows the relationship between the configuration parameter and its associated register location. Complex parameters can then be computed and made available as follows:

```
function param_x_type get_computed_param_x();
      return compute_param_x(.param_a(get_config_param_a()),…);
endfunction
```

Code Example 9: Operational Parameter Calculation Using Abstract Method

This strengthens the tie to the register model by directly reusing the data it contains, rather than keeping multiple copies of the same data around.

Abstraction with Complex Register Map Hierarchy

For most cases, a simple flat hierarchy is acceptable for the reference register block created for use in the VIP. This means the default map of the register block holds a set a of registers with distinct addresses. Should it be necessary for a VIP to have a more complex register map scheme, such as multiple maps, or the use of sub-maps, only a small adjustment would be necessary with all the code examples presented so far.

In such a scenario where a more complex mapping scheme is needed, the `uvm_register::get_offset()` is replaced with `uvm_register::get_address()` with the appropriate register map as argument in the cases presented so far. The latter method call accounts for factors affecting register addressing such as shifts in address due to different maps, or sub-maps. In other words, the `uvm_register::get_address()` provides the fully resolved address a register would be associated with, like a raw address value on a physical interface.

Abstraction to Determine Register in Operation

Often involved with register operations, specific accesses to some registers can trigger operations related to the protocol. For example, an access to a specific register can atomically select and activate a bank of registers to use as operational configuration. Therefore the code in the VIP must determine the register being accessed by using the extracted address from the protocol transaction in order to determine if there is a consequential action to be applied. The following example would be an implementation in the VIP to determine which register is being accessed:

```
case (virtual_if.addr)
   16'h0100 : [ ... ];
   16'h0101 : [ ... ];
   16'h0102 : [ ... ];
endcase
```

Code Example 10: Hardcoded Method to Locate Register in Operation

For reasons previously mentioned, this method is undesirable and can be improved upon by once again using built-in introspection methods available in the `uvm_reg_block` and `uvm_reg` derived classes. One solution would be to combine the `uvm_reg_block::get_registers()` and `uvm_reg::get_offset()` as such:

```
my_vip_ref_reg_block m_ref_regblock;  // VIP reference register block
uvm_reg              m_reg, m_regs[$];
string               m_reg_str;

m_vip_reg_block.get_registers(m_regs);
for ( regidx=0;  regidx<m_regs.size(); regidx++  ) begin
   if ( my_transaction.addr == m_regs[regidx].get_offset() ) begin
    m_reg_str = m_regs[regidx].get_name( );
    break;
   end
end
```

Code Example 11: Abstract Method to Locate Register in Operation

Code Example 11 can also be extrapolated to handle the case where multiple maps would exist in the reference register block. In such a case, instead of calling the `uvm_reg::get_offset()`

function, the code would have to iterate through the registers calling its `uvm_reg::get_address()` method with each valid register map property as argument.

The above code is effective and satisfies the property of being self-maintainable should register locations change. Unfortunately, it does have the potential of poor performance. This is due not only for the cost of the creating of the list of registers for every access attempt, but also due to the cost of recursively travelling the list in order to find a match in register offset (or address). Hence this code would degrade in performance if the register protocol set would grow.

Fortunately, other introspection methods are available allowing this operation to be more performance effective, regardless of the size of the register set. This would involve using the address map of the reference register block, which typically means the `uvm_reg_block::default_map` property and its available built-in methods. The following example demonstrates this approach:

```
m_reg = m_ref_regblock.default_map.get_reg_by_offset(my_transaction.addr)
if ( m_reg != null ) begin
  m_reg_str = m_reg.get_name( );
end
else begin
  [ … handle address not mapped to recognized address … ]
end

case (m_reg_str)
    "reg1_reg" : [ ... ];
    "reg2_reg" : [ ... ];
  ...
    "regN_reg" : [ ... ];
endcase
```

Code Example 12: Performance Optimized Abstract Method to Locate Register

In the case above, the code uses the `uvm_reg_map::get_reg_by_offset()` built-in method to locate the register instance in the reference register block mapped to the provided offset. Under the hood of this `uvm_reg_map` class method, an automatically pre-assembled associative array is used to execute a quick lookup of the register mapped to the offset. This ensures faster and more efficient performance than the iterative lookup of Code Example 11.

Once again, Code Example 12 can also be extrapolated for the case of multiple register maps, where the `uvm_reg_map::get_reg_by_offset()` would be called for every valid map property in the reference register block, instead of just the `default_map`.

Finally, a keen observer would notice a vulnerable point with the Code Example 12, specifically the use of the string literals (ie `"reg1_reg"`) used in the case statement. If registers would change their name, some of the items in the case statement would no longer match the entries in the reference register block. Furthermore there would be no explicit warning or error, but rather a

hard to find bug in the test bench. Fortunately, the use of another introspection method along with some organization with the string literals can help make the previous example code more robust as depicted in the next example:

```
const string c_reg1_str = "regA_reg";
const string c_reg2_str = "regB_reg";
const string c_regN_str = "regC_reg";
string c_register_table[] = { c_reg1_str , c_reg2_str, c_regN_str };


virtual function build_phase(uvm_phase phase);
   foreach(c_register_table[r];
     if (null == m_ref_regblock.get_reg_by_name(c_register_table[r]) )
       `uvm_fatal( …, $sformatf("Register string %s is no longer
                                 in the reg block", c_register_table[r]))
endfunction

[…]
case (m_reg_str)
  c_reg1_str : consequential_actions_regA(…);
  c_reg2_str : consequential_actions_regB(…);
  […]
  c_regN_str : consequential_actions_regN(…);
endcase
```

Code Example 13 : Adding Robustness for Locating Register in Operation

With the above code, should a string literal representing a register name no longer be present in the register block, this would be caught by the uvm_fatal statement since the `uvm_reg_block::get_reg_by_name()` would return a null value. Hence, if any disruptive changes were to occur, the use of the above code would make notification of a potential problem in the VIP code more explicit, thus trivial to fix.

4. Conclusions

UVM register classes offer a rich set of methods allowing a VIP developer to create a layer of abstraction and efficiently model register operations of the protocol it implements. The examples and techniques presented in this paper were used for creating a slave device VIP implementing the MIPI SoundWire [3] serial audio protocol.

The three main advantages to using this approach of abstracting register attribute details from the core VIP functionality are the following: (a) automatic synchronized modeling of the register with a centralized register definition of the VIP's protocol, (b) increase in code stability and robustness during the course of a project even as register attributes change, and (c) explicit error notifications such as compilation or fatal errors in the event of radical changes in register attributes instead of hard to find simulation time testbench bugs.

There are a few minor disadvantages to consider. Firstly, the code in the solutions proposed may appear to be more difficult to understand as it replaces all hardcoded explicit addresses. However, the learning curve to overcome this should be small as the appearance of register names should provide clarity of intent when read by users of the VIP code. Secondly, as discussed in [4], there can be non-negligible compile time costs, as well as run time costs when a register class derived object's `build()` method is called. However, as disclosed in the same paper [4], workarounds do exist to mitigate this extra overhead.

5. References

[1] Accelera, "UVM User Guide, v1.1" www.accellera.org/downloads/standards/uvm

[2] Accelera, "Standard UVM Class Reference, v1.2" www.accellera.org/downloads/standards/uvm

[3] MIPI Alliance, "Specification for Soundwire^{SM}", Version 1.0

[4] "Advanced UVM Register Modeling – There's More Than One Way to Skin a Reg" – M.Litterick, M. Harnisch, DVCon 2014