Advanced UVM Register Modeling

There's More Than One Way to Skin A Reg

Mark Litterick, Marcus Harnisch
Verilab GmbH
Munich, Germany
mark.litterick@verilab.com, marcus.harnisch@verilab.com

***Abstract*— This paper provides an overview of register model operation in the UVM and then explains the key aspects of base class code that enable effective complex register modeling. Several possible solutions to common modeling problems are discussed in detail with a focus on supporting both active and passive operation. In addition the performance impact of large register models is analyzed and improved solutions provided.**

Keywords—UVM, Register-Model, Register-Performance

I. Introduction

Writing effective register models for most complex designs involves modeling any number of imaginative register field operations, side effects and interactions between different registers. The Universal Verification Methodology (UVM) provides a standard base class libraries that enable users to implement just about anything the designers can dream up, but the documented examples, recommendations and guidelines are misleading and can result in ineffective implementation and poor performance.

This paper provides a brief overview of register model operation in the UVM and then explains the key aspects of base class code that enable complex register modeling, including:

A. *Multiple Solutions*

By presenting and analyzing multiple possible implementations for several modeling problems, we are able to present the user with a clear understanding of what patterns are more appropriate to specific applications and why. From this analysis we derive guidelines on when it is appropriate to use callbacks, when custom fields are required, and illustrate the use of dynamic field access policies.

B. *Passive Operation*

The basic UVM register model application programming interface (API) is very stimulus-centric. In particular the user is presented with the opportunity to customize derived fields using pre/post read/write hooks in the base field class and associated callbacks. As a result many register models are built up using hooks that do not result in correctly modeled register behavior in a passive context, i.e. under any circumstances when the Device Under Test (DUT) register receives stimulus that was not a direct result of running a corresponding sequence on the register model; typically passive reuse in an environment with alternate stimulus, or where registers are updated from internal sources such as an embedded microcontroller. For passive modeling the standard access hooks suggested by UVM documentation are inappropriate and the implementer needs to make use of alternative callbacks.

C. *Performance*

In a typical complex System on Chip (SoC) device we can easily have tens of thousands of register fields in the model. If we follow the standard guidelines for factory generation as proposed by the UVM we suffer from a significant load and build-time performance penalty. The paper observes that since register models are generated to-order from the documentation source, it is not part of the standard use-case to perform factory overrides on registers or fields. We demonstrate what the factory overheads are and provide alternative guidelines on how to minimize register model overhead without compromising usability. In addition we also explore the possibility to build registers on-demand.

II. UVM Register Overview

In a verification context, a register model (or register abstraction layer) is a set of classes that model the memory-mapped behavior of registers and memories in the DUT in order to facilitate stimulus generation and functional checking (and optionally some aspects of functional coverage). The UVM provides a set of base classes that can be extended to implement comprehensive register modeling capabilities as detailed in [1] and [2]. The fundamental structure of the actual register model is illustrated in Figure 1.

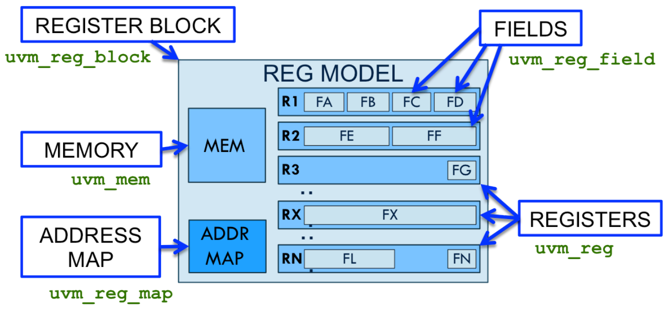


Fig. 1. Register Model Structure

The register model itself is a set of DUT-specific files that extend the UVM base classes. Due to the number of registers required and their regular structure, the model files are usually generated from a textual register description source (such as XML) in parallel with the generation of actual RTL for the DUT registers. The top-level register block is instantiated in the verification environment in parallel with other interface verification components, as shown in Figure 2.

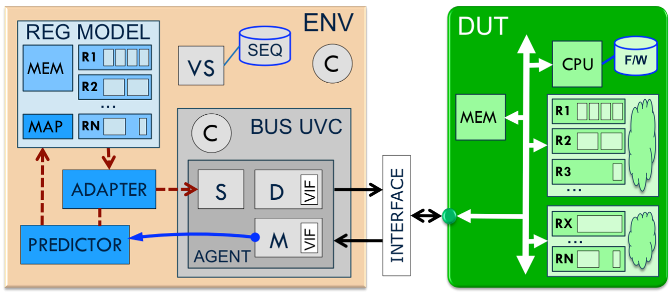


Fig. 2. Environment with register model, adapter and predictor

Note that the reusable interface components do not contain a reference to the register model (since this would compromise portability), but the enclosing environment implements any required interaction between the interface components and the model. The main interaction between the register model and the rest of the verification environment is via adapter and predictor components:

• *uvm_reg_adapter* converts between register model read and write methods and the interface-specific transactions

• *uvm_reg_predictor* updates the register model based on observed transactions published by a monitor

A good introduction to register model usage and how to set up the adapter and predictor is provided by [3].

A. Active and Passive Operation

The UVM register model supports both active and passive modes of operation and is often used in an environment where both are required as shown in Figure 3. In an active context, register operations are sourced via the register model read and write methods, which in turn get converted to sequence items for an interface verification component via the adapter (path 1 in Figure 3). Passive operation refers to any register operation where the stimulus did not explicitly use the register model access methods directly, for instance if an interface component sequence is executed directly by the virtual sequencer without calling the register model access methods (path 2 in Figure 3), or if the register content is affected by a bus transaction from an alternative stimulus source, for example an embedded CPU running firmware (as shown by path 3 in Figure 3).

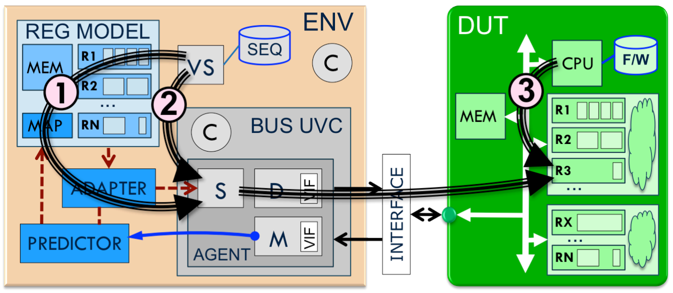


Fig. 3. Access Paths for Active and Passive Operations

Another form of passive operation is when the register model is used in an environment where the stimulus is provided by some other mechanism entirely (such as executable software, or a legacy non-UVM environment), but the model is still required to be accurate for checks and functional coverage. For actual bus traffic from an alternative source to be observed, a predictor must be present on the corresponding bus hierarchy. The UVM register model API is very stimulus-centric and the documentation is focused on the active usage of the class constructs, which leads to confusion in partial and fully passive contexts.

It is not just the field modeling aspect that needs to be aware of passive usage, but also any checks that make use of the register model state. For example implementing register checks using the *mirror* method in a sequence (which reads the register and checks the mirrored value against the read value) is not portable to a vertical reuse scenario; so monitors should be used to implement passive checks instead [4].

B. Predictable and Volatile Modeling

The accurate modeling of all register fields in the DUT goes beyond the capabilities provided by the register model constructs, since many register fields are modified by the DUT in response to alternative stimulus and not just bus operations. Such register fields are not predictable, based only on described behavior in relation to read and write operations, and are termed volatile from the model viewpoint. Typically these registers are not checked directly by the model, but rather the RTL value is probed using active monitoring via the specified HDL path (i.e. the actual path to the RTL register field in the DUT), in order to keep the field up to date [5]. This means the model adapts to DUT behavior and additional external checks should be used to validate that the DUT field, available in the register model, has the correct value under all circumstances dictated by the specification. In some cases it may also be possible to model status registers as read-only but non-volatile, and analyze external stimulus directly to predict new register values based on observed events.

Modeling and checking volatile register behavior in the larger context is non-trivial, but is part of the overall verification environment modeling responsibilities. For instance, how do we validate that the resultant value on a volatile register bit is correct if both the bus and the hardware try to update the bit at the same time? Which operation has priority and has that been implemented correctly in the DUT? More fundamentally, what external stimulus should cause the volatile register field to be updated to a particular value and when may this occur deep inside the design? A full analysis of all aspects of volatile register field modeling in the context of functional behavior of the application is outside the scope of this paper; the remaining sections focus on maintaining an accurate model of non-volatile register fields.

III. Register Model Operation

A. *Register Access API*

An overview of the register and field access API is provided by [1], with a more detailed explanation provided by [6]. Each field has a corresponding predicted and mirrored value, as well as reset state and hooks for additional operations such as randomization via a value field, as shown in Figure 4.

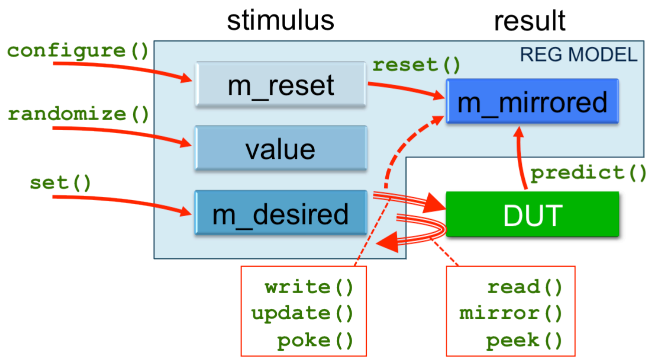


Fig. 4. Register Field Variables and Access Methods

For non-volatile register fields the mirrored value provides the current state of the register field based on all active and passive bus operations. For volatile fields additional modeling is required to maintain the state of the mirrored value based on other non-bus related application-specific behavior. Register operations can be summarized as:

• *write*, *poke*, *set-update* and *randomize-update* are all active operations which update both the mirrored and DUT register values

• *read*, *peek* and *mirror* are all active operations which update the mirrored value based on DUT register values

• *reset* and *predict* are passive operations which update the mirrored value independent of active model stimulus

Note that the model is normally reset during construction and for all subsequent observed reset events and is expected to match the DUT reset state. The *predict* method is called in response to observed bus transactions published via the associated predictor component, but it is also called as part of all active write and read operations, including backdoor operations which have no bus traffic at all.

B. *Hooks and Callbacks*

The term callback is often abused in software engineering in general, and the UVM documentation in particular, even though the *uvm_callback* mechanism is quite well defined. In the context of register modeling, we need to make a clear distinction between the two main mechanisms, hooks and callbacks, which are used to affect register model behavior.

Register and field base classes provide a set of empty virtual method hooks (*pre_write*, *post_write*, *pre_read* and *post_read*), which are called at specific points in the execution of the register operations. The register model developer can implement one or more of these methods in the derived class in order to affect the functionality. This is normal object-oriented behavior and nothing to do with the callback mechanism.

```
class my_reg_field extends uvm_reg_field;
  ...
  virtual task post_write(uvm_reg_item rw);
    // specific implementation
  endtask
  ...
endclass
```

The callback base classes also have empty virtual methods (*pre_write*, *post_write*, *pre_read*, *post_read*, *post_predict*, *encode* and *decode*), but they are only executed if a callback is registered with a specific field or register (which causes it to be added to a list). The developer can derive a user-defined callback class from the base class and implement one or more of the virtual methods to affect the functionality.

```
class my_field_cb extends uvm_reg_cbs;
  ...
  function new(string name="my_field_cb", ...);
  ...
  virtual task post_write(uvm_reg_item rw);
    // specific implementation
  endtask
  ...
endclass
```

The callback class must then be constructed and registered with the field or register for which the method will be called. Callbacks are allocated dynamically and we can register multiple callbacks for different orthogonal features with the same register field. In addition the constructor for the callback does not have a fixed signature, so the functionality of each callback is very flexible.

```
my_field_cb my_cb = new("my_cb", ...);
uvm_reg_field_cb::add(regX.fieldY, my_cb);
```

Hence the method implementation is done in a derived *uvm_reg_field* class for hooks, and in a derived *uvm_reg_cbs* class for callbacks. The virtual method hook is always called but the related method from the callback is only called if the callback is registered with the corresponding field, as shown in the following extract from the *uvm_reg_field* file:

```
task uvm_reg_field::do_write(uvm_reg_item rw);
  ...
  pre_write(rw); // virtual method hook
  for (uvm_reg_cbs cb=cbs.first();
       cb!=null; cb=cbs.next())
    cb.pre_write(rw); // callback method
  ...
  rw.local_map.do_write(rw); // actual write
  ...
  post_write(rw); // virtual method hook
  for (uvm_reg_cbs cb=cbs.first();
       cb!=null; cb=cbs.next())
    cb.post_write(rw); // callback method
  ...
endtask
```

For passive modeling of the register behavior the *post_predict* callback method is the most important since it is executed by the *uvm_reg_field::predict* method after any observed read or write operation, irrespective of the source of the bus traffic, including backdoor accesses [7]. Note that it is important to register the callbacks which implement *post_predict* with fields and not registers, since *post_predict* is only called from a *uvm_reg_field::predict()* operation; in other words if we associate the callback with the register object, then we effectively add a *post_predict* method to the derived class which will not be called by the super class!

C. Field Access Policies

The UVM provides a comprehensive set of pre-defined field access policies [1], which are summarized in Table I. The field access policy is normally setup during *build* by the *configure* method.

TABLE I. SUMMARY OF PRE-DEFINED FIELD ACCESS POLICIES

	NO WRITE	WRITE VALUE	WRITE TO CLEAR	WRITE TO SET	WRITE TO TOGGLE	WRITE ONCE
NO READ	-	WO	WOC	WOS	-	WO1
READ VALUE	RO	RW	WC W1C W0C	WS W1S W0S	W1T W0T	W1
READ TO CLEAR	RC	WRC	-	WSRC W1SRC W0SRC	-	-
READ TO SET	RS	WRS	WCRS W1CRS W0CRS	-	-	-

An important aspect of field access policies is that they are necessarily self-contained, in that the field's behavior can be explained in terms of register operations in isolation from other fields and registers as shown in Figure 5.

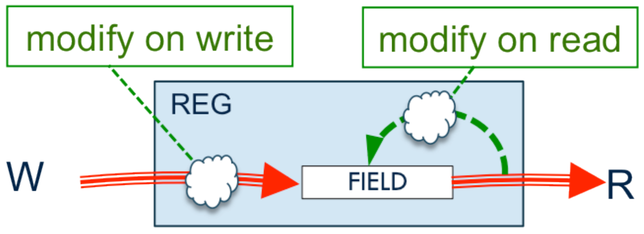


Fig. 5. Field Access Policy Behavior

UVM provides mechanisms to support the addition of user-defined access policies, which are defined using the static *uvm_reg_field::define_access* method. Furthermore, the access policy can be changed dynamically by calling *set_access*, even after the register model is locked. There is also a *get_access* method that returns the current access policy for the field.

It is important to be aware that defining, adding and using a user-defined access policy does not actually create any specific modeling behavior in itself - in fact the behavior of a user-defined access policy defaults to that of the RW policy. To get specific user-defined policy behavior the user must alter the behavior of actual accesses to the corresponding register field by implementing virtual methods in either the hooks or callbacks.

Field access policies should not be confused with the access rights declared when a register is added to a particular address map. Registers can be added to more than one map (corresponding to different interfaces in the DUT) with different access rights (RW, RO or WO are the only choices); whether a register field can be read or written depends on both the field's configured access policy and the register's rights in the map being used to access the field [2].

Although comprehensive, the set of pre-defined field access policies can never be exhaustive; for example we have dealt with designs requiring read-to-reset, write-to-reset, write-key-to-set/clear/reset, and write-sequence-to-set/clear/reset/write behavior - all of which can be solved with dedicated user-defined access policies.

D. Register Field Interaction

In practice, most special register operations tend to require interaction between different register fields, rather than self-contained field access policy behavior. These register field side effects are not modeled out of the box by the UVM register abstraction layer; however, mechanisms are provided to enable users to develop advanced modeling relationships using the underlying hooks and callbacks. Typically this involves adding a handle from one register field to another, and connecting it in a higher-level code (e.g. register block) that has access to the interacting fields. If the field names are unique throughout the model, we can also use *get_field_by_name* method to connect the cross references, otherwise it is safer to use hierarchical names (as shown in the examples). Figure 6 shows the interaction of two fields, from separate registers, in a generic manner; both register field operations (i.e. read and write) as well as field value can affect the operation of the affected field.

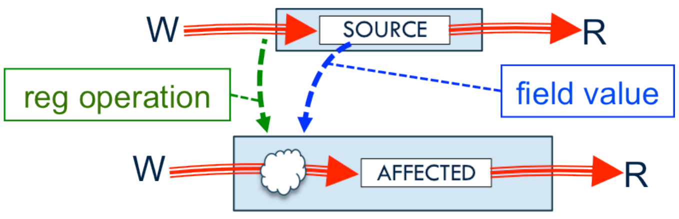


Fig. 6. Field Interaction & Side Effects

Typical examples of field interaction include locking or protecting a register field based on another fields content, or controlling the timing of a buffer update based on a write to separate trigger field which may be in the same register, a different register in the same scope, or a different register block.

It normally makes sense to identify a field as having a user-defined access policy even though the behavior is not entirely self-contained in accesses to this field (but rather is modeled by side-effects from other register fields) in order to identify the field as special. Care is required here due to the very loose binding between the affected field and the controlling field; this encapsulation problem does not exist in self-contained user-defined access policies.

E. Register Side Effects

Register side effects are not limited to interaction of fields. In fact register field operations can be extended to influence many other aspects of verification environment behavior, such as modifying configuration fields in an interface or legacy verification component when a related configuration register is updated in the model.

Implementation of these kinds of side effects is similar to that for register field interaction, except the relationships between the two structures must be handled at an even higher level, typically the environment which instantiates both the register model and the other verification components.

IV. WORKED EXAMPLES

This section provides some worked examples for modeling certain types of register field behavior and interaction. In each case we discuss several possible solutions and look at the relative benefits of each of these.

A. Example User-Defined Access Policy - Write-to-Reset

While write-to-clear (WC) and write-to-set (WS) are available as pre-defined field access policies, some devices have a requirement for write-to-reset behavior that can be implemented as a user-defined field access policy (which we will call WRES). In this case the register field is set to its current reset value (which was initialized using *configure*, or by *set_reset,* and is presumably not all ones or all zeros, otherwise the pre-defined policies could be used) on a write. Other variations of this access policy include read-to-reset, write-1-to-reset, write-0-to-reset, etc. The basic mechanism is illustrated in Figure 7.

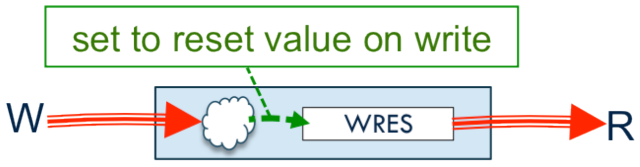


Fig. 7. Write-to-Reset User-Defined Access Policy

1) Write-to-Reset Using post_write Hook

One possible implementation for this access policy is to use a derived register field, which is extended from the *uvm_reg_field* base class, and implement the *post_write* virtual method hook to set the mirror to the reset value as shown in the following code.

```
class wres_reg_field extends uvm_reg_field;

  `uvm_object_utils(wres_reg_field)

  local static bit m_wres = define_access("WRES");

  function new(string name = "wres_reg_field"); ...

  virtual task post_write(uvm_reg_item rw);
    // set the mirror to reset value after a write
    if (!predict(rw.get_reset())) `uvm_error(...)
  endtask

endclass
```

The enclosing register definition needs to declare a field of the corresponding derived type, and can set the access policy in the *configure* method for the field as shown in the following code example:

```
class wres_reg extends uvm_reg;

  `uvm_object_utils(wres_reg)

  rand wres_reg_field wres_field; // special field

  function new(string name="wres_reg"); ...

  virtual function void build();
    wres_field = wres_reg_field::type_id::create(
      "wres_field");
    wres_field.configure(
      this,8,0,"WRES",0,8'hAB,1,1,1);
  endfunction

endclass
```

The corresponding register block only has to create the enclosing register and add it to the required address maps, as shown:

```
class my_reg_block extends uvm_reg_block;
  virtual function void build();
    ...
    wres_reg = wres_reg::type_id::create("wres_reg");
    wres_reg.configure(this, null, "wres_reg");
    wres_reg.build();
    default_map.add_reg(wres_reg, 'h24, "RW");
    ...
```

2) Write-to-Reset Using post_write Callback

This user-defined field access policy can also be implemented using callbacks. In order to make a clearer comparison between the hook and callback implementations, we will first illustrate overloading the *post_write* virtual method in a callback class and registering this callback with the corresponding field. For callbacks we need to define a derived class that extends from the *uvm_reg_cbs* class and add the user-defined behavior to this class.

```
class wres_field_cb extends uvm_reg_cbs;

  local static bit m_wres =
    uvm_reg_field::define_access("WRES");

  function new(string name = "wres_field_cb"); ...

  virtual task post_write(uvm_reg_item rw);
    // set the mirror to reset value after a write
    if (!predict(rw.get_reset())) `uvm_error(...)
  endtask

endclass
```

In this case the enclosing register definition just uses the base *uvm_reg_field* class for the field, and configures it as before:

```
class wres_reg extends uvm_reg;

  `uvm_object_utils(wres_reg)
  rand uvm_reg_field wres_field; // base field type

  function new(string name="wres_reg"); ...

  virtual function void build();
    wres_field = uvm_reg_field::type_id::create(
      "wres_field");
    wres_field.configure(
      this, 8, 0, "WRES", 0, 8'hAB, 1, 1, 1);
  endfunction

endclass
```

In order to use the callback functionality, we need to construct the callback and register it with the corresponding register field. This additional work is typically done in the generated code for the corresponding register block:

```
function void my_reg_block::build();
 wres_field_cb wres_cb = new("wres_cb");
 ...
 wres_reg = wres_reg::type_id::create("wres_reg");
 wres_reg.configure(this, null, "wres_reg");
 wres_reg.build();
 default_map.add_reg(wres_reg, 'h24, "RW");
 ...
 uvm_reg_field_cb::add(wres_reg.wres_field, wres_cb);
 ...
```

The key thing to be aware of at this stage is that both of the above implementations using *post_write* are inadequate since they do not correctly handle register updates in a passive context. Specifically the *post_write* methods from both the field and the callback classes are only executed if the *write* method is called on the corresponding field of the register (or enclosing register class). If another master, for example an embedded CPU, performs a write to the register without going through the register model bus adapter, then these methods are not called and the register operation is not correctly mirrored in the model.

3) *Write-to-Reset Using post_predict Callback*

The following code shows a better implementation of the write-to-reset user-defined field access policy that works for both active and passive register updates. In this case the *post_predict* method from the callback class is used to set the field value whenever any type of write is observed on the register field. The basic register field class does not have a *post_predict* virtual method, it is only available in the callback class; it is called by *field::predict* whenever an active front or backdoor operation occurs or passively when a predictor component observes a published bus transaction.

```
class wres_field_cb extends uvm_reg_cbs;
  ...
  virtual function void post_predict(
    input  uvm_reg_field   fld,
    input  uvm_reg_data_t  previous,
    inout  uvm_reg_data_t  value,
    input  uvm_predict_e   kind,
    input  uvm_path_e      path,
    input  uvm_reg_map     map
  );
    // set the mirror to reset value after a write
    if (kind == UVM_PREDICT_WRITE)
      value = fld.get_reset();
  endfunction

endclass
```

The enclosing register definition and callback registration are identical to that shown previously for the *post_write* callback implementation.

B. *Example Field Interaction - Locked/Protected Field*

A common requirement for registers is that a field can be locked, or protected, when another register field has a specific value. This locking can be relatively permanent (so we configure the registers and then lock them down for the duration of the application phase) or it can be temporary (for example some registers are frozen temporarily while we reconfigure another aspect of device operation). The actual locking value can be a single bit or a multi-bit key. The basic operation is illustrated in Figure 8.

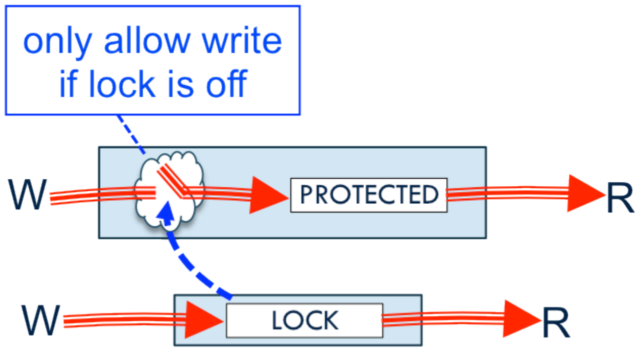


Fig. 8. Lock/Protect Field Interaction

1) *Lock/Protect Operation Using pre_write*

The UVM User Guide [1] provides examples of how to implement protected operation using *pre_write* hooks and callbacks; this is repeated here for the purposes of illustration. The first example implements *pre_write* in a derived register field with a handle to the locking field (called *protect_mode* in this example) as shown:

```
class protected_field extends uvm_reg_field;
  local uvm_reg_field protect_mode;

  virtual task pre_write(uvm_reg_item rw);
    // Prevent the write if protect mode is ON
    if (protect_mode.get()) begin
      rw.value = value;
  endtask
endclass
```

The second example implements *pre_write* in a callback class which uses a handle to a *protect_mode* field:

```
class protected_field_cb extends uvm_reg_cbs;
  local uvm_reg_field protect_mode;

  virtual task pre_write(uvm_reg_item rw);
    // Prevent the write if protect mode is ON
    if (protect_mode.get()) begin
       uvm_reg_field field;
       if ($cast(field,rw.element))
         rw.value = field.get();
    end
  endtask
endclass
```

```
protected_field_cb protect_cb = new(
  "protect_cb",protect_mode)
uvm_callbacks#(my_field_t,uvm_reg_cbs)::add(
  my_field, protect_cb);
```

Both of these implementations interfere with the actual write operation operation by changing *rw.value* in the *pre_write* method. This means the write method will continue with the wrong (protected) data value. For a locked field we must explicitly check that if we write a different value to the register when the register is locked or protected, then the new value should be ignored by the DUT register. Additionally, the *pre_write* method is only called during an active write operation and does not work in a passive context.

2) Lock Operation Using post_predict Callback

This section illustrates an improved implementation of the locked field operation using the *post_predict* callback that also works for passive modes of operation. The *post_predict* method has a parameter that provides the *previous* value of the register field prior to the operation, hence we can restore this value if a write is attempted and the register is locked.

```
class protect_field_cb extends uvm_reg_cbs;

  local uvm_reg_field lock_field;

  function new (string name, uvm_reg_field lock);
    super.new (name);
    this.lock_field = lock;
  endfunction

  virtual function void post_predict(...);
    if (kind == UVM_PREDICT_WRITE) begin
      if (lock_field.get()) begin
        // Revert to previous value if protected
        value = previous;
      end
    end
  endfunction

endclass
```

In this case the callback is registered with the protected field, and it is passed a handle to the lock field (which can be in another register or block), which is taken into account when a write operation is attempted on the protected field.

```
protect_field_cb prot_cb = new(
  "prot_cb",lock_reg.lock_field);
uvm_reg_field_cb::add(
  prot_reg.prot_field, prot_cb);
```

3) Lock Operation Using Dynamic Access Policy

Another alternative that works for both active and passive modes is to modify the protected field access policy dynamically in response to changes in the lock field. In this case a callback is defined to change the access policy of a field accessed by a handle. The callback is registered with the locking field (not with the protected field) and therefore when a write operation is performed on the lock field the access mode for the protected field is modified from RW (read/write for unlocked or unprotected) to RO (read-only for locked or protected state).

Modifications to access policies using *set_access* are allowed after the model is built and locked, but normally discouraged since this leads to confusion relative to the static register definition. In this case it is quite a good fit though because if any code does a *get_access* on the protected field, the model will return the truth about the current state of the field - either RW or RO.

```
class lock_field_cb extends uvm_reg_cbs;

  local uvm_reg_field protected_field;

  function new (string name, uvm_reg_field prot);
    super.new (name);
    this.protected_field = prot;
  endfunction

  virtual function void post_predict(...);
    if (kind == UVM_PREDICT_WRITE) begin
      if (value)
        void'(protected_field.set_access("RO"));
      else
        void'(protected_field.set_access("RW"));
    end
  endfunction

endclass
```

In this case the callback is registered with the locking field, and is passed a handle to the protected field, such that a write operation observed on this lock field may alter the access policy of the protected field.

```
lock_field_cb lock_cb = new(
  "lock_cb", prot_reg.prot_field);
uvm_reg_field_cb::add(
  lock_reg.lock_field, lock_cb);
```

C. Example Field Interaction - Triggered Buffered Writes

Another common requirement, especially in applications with a lot of pseudo-static configuration registers, is that the register writes are buffered and not applied to the functional part of the DUT until a trigger field or register is accessed (sometimes with a particular key value, or even a sequence of writes). When the trigger is written, a coherent set of buffered register fields is then copied to the active registers and operation continues. The basic mechanism is illustrated in Figure 9.

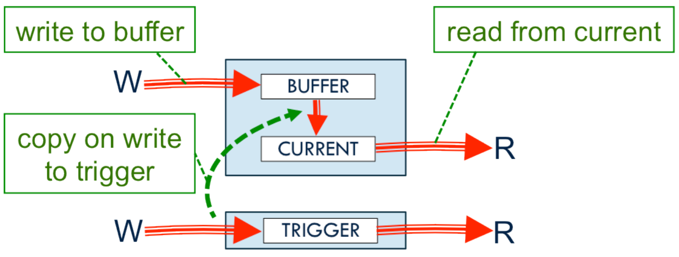


Fig. 9. Triggered Buffered Write Operation

1) Buffered Write Using Overlapped Registers

One possible solution here is to define two registers that are located at the same address but have different access rights. One register is WO and contains the buffer value - so the user can always write to the buffer. The other register, which is co-located at the same address, is RO and contains the actual operational value after the most recent trigger (this is really the current value in the application). A callback can be used to copy from the buffer to the current register whenever a trigger occurs. As discussed previously, for this to operate in passive mode we need to specialize the *post_predict* method rather than the *post_write* method in the callback that is registered with the trigger field, then copy from the buffer to the current register in the other field. An example of the trigger callback code is shown below:

```
class trig_field_cb extends uvm_reg_cbs;

  local uvm_reg_field current, buffer;

  function new (string name,
                uvm_reg_field current,
                uvm_reg_field buffer);
   super.new (name);
   this.current = current;
   this.buffer = buffer;
  endfunction

  virtual function void post_predict(...);
   if (kind == UVM_PREDICT_WRITE) begin
    uvm_reg_data_t val = buffer.get_mirrored_value();
    if (!current.predict(val)) `uvm_error(...)
   end
  endfunction

endclass
```

Two registers are required just to implement the basic functionality (remember we do need to store the buffered data somewhere!):

```
class cur_reg extends uvm_reg;
  `uvm_object_utils(cur_reg)
  rand uvm_reg_field cur_field;
  function new (...);
  virtual function void build();
    cur_field = uvm_reg_field::type_id::create(
      "cur_field");
    cur_field.configure(
      this, 32, 0, "RO", 0, 32'h00004444, 1, 1, 1);
  endfunction
endclass

class buf_reg extends uvm_reg;
  `uvm_object_utils(buf_reg)
  rand uvm_reg_field  buf_field;
  function new (...);
  virtual function void build();
    buf_field = uvm_reg_field::type_id::create(
      "buf_field");
    buf_field.configure(
      this, 32, 0, "WO", 0, 32'h00004444, 1, 1, 1);
  endfunction
endclass
```

In this case the callback is constructed with handles to the extra buffer register and the target destination register, but is registered with the trigger field (i.e. the one experiencing the dynamic operation). Notice that both the current and buffer registers are added to the register map:

```
class my_reg_block extends uvm_reg_block;
  ...
  virtual function void build();
   ...
   trig_field_cb trig_cb;
   ...
   default_map.add_reg(cur_reg, 'h10, "RO");
   default_map.add_reg(buf_reg, 'h10, "WO");
   ...
   // create callback and register with trigger
   trig_cb = new(
    "trig_cb", cur_reg.cur_field, buf_reg.buf_field);
   uvm_reg_field_cb::add(
     trig_reg.trig_field, trig_cb);
```

Even though the proposed implementation works in both active and passive contexts, there are three problems with this implementation:

- it is not possible to share the address with another register
- it is harder to generate two registers in place of one
- it is more confusing since we add a dummy register

The first problem arises because we have already shared two registers at the same address. Hence it is not possible to do a triggered buffered write-only register shared with another independent status register sharing the same address (which turned out to be a requirement in one project). Since we have to really treat the register as two separate registers, the generation is probably more complicated (depending on your generator tool) and the implementation is certainly more confusing (since the user sees multiple registers sharing an address when the source register description does not imply that is the case).

2) Buffered Write Using Derived Field and Callbacks

A better solution for this problem is to implement a derived field class that contains the buffer field. This buffer field is used to store a persistent value of the data for all writes to the register. Whenever an appropriate write occurs to the trigger register, the buffer field is copied to the register mirror. This approach also works if the triggered buffer register is WO and has to share an address with an actual RO status register.

There are two minor complications here; firstly the buffer typically needs to be reset to the same value as the nominal register; but this is easily achieved (without caring about actual values) using the *get_reset* method in the field *reset* function as shown below.

```
class buffered_reg_field extends uvm_reg_field;

  local uvm_reg_data_t buffer;

  `uvm_object_utils(buffered_reg_field)

  function new(string name); ...

  virtual function void reset(string kind = "HARD");
    super.reset(kind);
    buffer = get_reset(kind);
  endfunction

endclass
```

The second minor complication is that the extended field only has access to active methods such as *pre_write* and *post_write* which would result in an implementation that was not tolerant of passive operation if we did the setting of the buffer value there. So we need to use the *post_predict* method in an additional callback in order to set the buffer and restore the mirrored value for all observed write operations on the register. This callback is registered with the buffered register field.

```
class buffered_field_cb extends uvm_reg_cbs;
```

```
  local buffered_reg_field buf_field;

  function new(string name, buffered_reg_field buf);
    super.new (name);
    this.buf_field = buf;
  endfunction

  virtual function void post_predict(...);
    if (kind == UVM_PREDICT_WRITE) begin
      // save the write value to the buffer
      buf_field.buffer = value;
      // restore the previous value to the mirror
      value = previous;
    end
  endfunction

endclass
```

Another callback is required which is registered with the trigger field. This callback contains a handle to the buffer field, and implements the *post_predict* method in order to transfer the buffer to the mirror in the other register when the required value is written to the trigger.

```
class trig_field_cb extends uvm_reg_cbs;

  local buffered_reg_field buf_field;

  function new(string name, buffered_reg_field buf);
    super.new (name);
    this.buf_field = buf;
  endfunction

  virtual function void post_predict(...);
    // update the target mirror from the buffer
    if (kind == UVM_PREDICT_WRITE) begin
      if (value==1) // any write, boolean or key...
        if (!buf_field.predict(buf_field.buffer))
          `uvm_error(...)
    end
  endfunction

endclass
```

Note that since multiple callbacks can be registered with the same trigger field we can update many buffer registers with one trigger, but each requires a callback to be registered. Alternatively, we can add many handles to various buffered registers into the same callback and register this once with the trigger field (which might give better performance in cases where there are many buffered registers and frequent triggers). The corresponding register definitions look like the following:

```
class buffered_reg extends uvm_reg;

  `uvm_object_utils(buffered_reg)

  rand buffered_reg_field buf_field; // special field

  function new (...);

  virtual function void build();
    buf_field = buffered_reg_field::type_id::create(
      "buf_field");
    buf_field.configure(
      this, 32, 0, "RWB", 0, 32'hBBBBBBBB, 1, 1, 1);
  endfunction

endclass
```

For each buffered field we need both callbacks, one registered with the buffer field and the other registered with the trigger field, since both fields undergo observed operations (a write to the buffer needs to be stored, a write to the trigger causes the other register to update mirror from the buffer). Remember to register both of these callbacks with fields and not registers, since the *post_predict* method is only called from *uvm_reg_field::predict* operation.

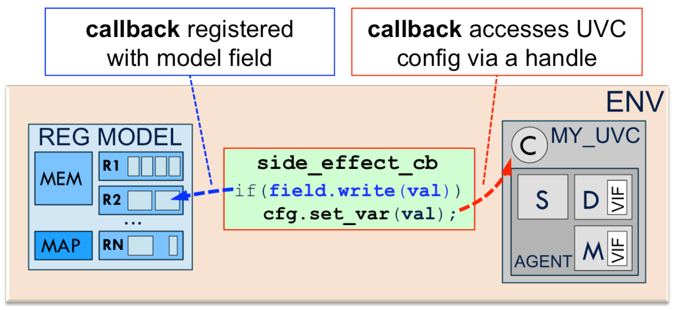
```
class my_reg_block extends uvm_reg_block;

 `uvm_object_utils(dut_vlog_reg_block)
 ...
 virtual function void build();
  trig_field_cb trig_cb;
  buffered_field_cb buf_cb;
  ...
  default_map.add_reg(buf_reg, 'h10, "RW");
  ...
  // create callback and register with trigger
  trig_cb = new(
    "trig_cb", buf_reg.buf_field);
  uvm_reg_field_cb::add(
    trig_reg.trig_field, trig_cb);

  // create callback and register with buffered
  buf_cb = new(
    "buf_cb", buf_reg.buf_field);
  uvm_reg_field_cb::add(
    buf_reg.buf_field, buf_cb);
```

D. Side Effects Outside of Register Model

Register side effects are not limited to interaction of fields and can be extended to influence many other aspects of verification environment behavior. For example, the UVM documentation describes randomization of the register model, and then applying the random configuration registers to the DUT using the update method; but this overlooks the fact that many interface verification components also need configuration object fields to be updated in response to DUT register operations (including passive updates from other sources). Implementing verification component configuration updates directly from sequences is not recommended since this will not work in a passive context. Furthermore implementing configuration updates by snooping on observed bus traffic inside the DUT using a passive monitor will not function correctly for backdoor writes to the registers. Figure 10 illustrates how we can spy on register operations using a callback in order to maintain the configuration variables for a generic or legacy verification component, which does not have a reference to the model.


callback registered with model field
callback accesses UVC config via a handle
ENV
REG MODEL
MEM
R1
R2
...
MAP
RN
side_effect_cb
C
MY_UVC
S
D
VIF
AGENT
M
VIF

```
if(field.write(val))
cfg.set_var(val);
```

Fig. 10. Interaction Between Register Field and VC Config

1) Configuration Update using Callback

The actual implementation in this case is similar to what was shown previously for field interaction using callbacks, except that software encapsulation demands that neither the register model code nor the reusable interface verification component can contain a direct reference to one another. Hence the callback is defined, constructed and registered by the enclosing scope - specifically the environment that instantiates both the register model and the hierarchy which included the interface verification component. All callback classes can be defined enclosed in a single file (in a similar manner to sequence library files) and imported. In order to support passive operation, the *post_predict* method will be used to call an access method from the required configuration object. If a translation between register value and protocol configuration parameter is required (for example from integer to enumerated type) then it can be encapsulated in the callback.

```
class reg_cfg_cb extends uvm_reg_cbs;

  local my_config cfg; // handle to config object

  function new (string name, my_config cfg);
    super.new (name);
    this.cfg = cfg;
  endfunction

  virtual function void post_predict(...);
    if (kind == UVM_PREDICT_WRITE)
      // call the config access function
      cfg.set_my_var(my_enum_t'(value));
  endfunction

endclass
```

This callback is constructed with a reference to the target configuration object, and registered with the required register field that undergoes the write operation as shown below.

```
class my_env extends uvm_env;
  ...
  reg_cfg_cb cfg_cb;
  ...
  virtual function void build_phase(...);
    super.build_phase(phase);
    uvc_inst = my_uvc::type_id::create(...);
    reg_model = my_reg_model::type_id::create(...);
    reg_model.configure(...);
    reg_model.build();
    reg_model.lock_model();
    ...
  endfunction

  virtual function void connect_phase(...);
    super.connect_phase(phase);
    cfg_cb = new("cfg_cb", uvc_inst.cfg);
    uvm_reg_field_cb::add(reg_model.field, cfg_cb);
  endfunction
  ...
endclass
```

V. REGISTER MODEL PERFORMANCE

When working with small stand-alone block-level verification environments, it is easy to overlook any potential register model performance concerns. However, when validating a complete SoC the register model code can become a significant performance burden. The main problem is due to the sheer volume of registers in a complex SoC, which can easily get to more than 10k register fields for a device with a lot of configuration and flexibility (this is typical for instance in applications with a lot of analog sub-components). Since the register model has each field, register and hierarchical block defined by classes, we often have considerably more SystemVerilog classes in the register model than the rest of the verification environment. This section looks at two possible aspects of register model performance improvement.

A. Life Without the Factory

In order to demonstrate the issues, consider an actual project where we observed a performance hit during compilation, load, build and execution phases of an SoC environment with a big register model (over 14k fields contained in about 7k registers), compared with running the environment without the model in place. Some of this performance is clearly due to the overhead of just having many more class objects declared and built in the verification environment, but we were also suspicious about the factory's role in dealing with the large number of classes.

It is important to note that the register model use-case is different to that for normal verification components and reusable environments (where the factory is, and should be, used as a matter of course). Specifically, the register model is generated on demand from a single source (XML in our case); it is already specialized for the intended purpose and regenerated for each derivative of the DUT. So the main questions are:

• can we avoid the factory for register model operation?

• how much of a performance benefit does that achieve?

The UVM user and reference guides state that we must use the *uvm_object_utils* to register the user specified field classes and registers with the factory and that we should use the factory *create* method instead of constructing directly using the *new* function. However, if our use-case does not involve using any features of the factory and we have a large number of register model classes, then using the factory seems like unnecessary overhead. In fact the UVM register model functions correctly without *uvm_object_utils*; specifically, if the normal factory enabled register code is represented by the following code:

```
class my_reg_type extends uvm_reg;
  rand uvm_reg_field fld_a;
  rand uvm_reg_field fld_b;

  `uvm_object_utils(my_reg_type)
  ...
  virtual function void build();
   fld_a = uvm_reg_field::type_id::create("fld_a");
   fld_a.configure(...);
   fld_b = uvm_reg_field::type_id::create("fld_b");
   fld_b.configure(...);
  endfunction
  ...
endclass

class my_reg_block extends uvm_reg_block;
  rand my_reg_type my_reg;
  ...
  virtual function void build();
    ...
    my_reg = my_reg_type::type_id::create(
      "my_reg",,get_full_name());
    my_reg.build();
    my_Reg.configure(...);
    default_map.add_reg(...);
    ...
  endfunction
  ...
endclass
```

Then, a functionally similar register definition can be achieved by omitting the *uvm_object_utils* macro and replacing the corresponding *type_id::create* methods with direct call to *new* for fields and registers as shown below:

```
class my_reg_type extends uvm_reg;
  rand uvm_reg_field fld_a;
  rand uvm_reg_field fld_b;
  ...
  virtual function void build();
   fld_a = new("fld_a");
   fld_a.configure(...);
   fld_b = new("fld_b");
   fld_b.configure(...);
  endfunction
  ...
endclass

class my_reg_block extends uvm_reg_block;
  rand my_reg_type my_reg;
  ...
  virtual function void build();
    ...
    my_reg = new("my_reg");
    my_reg.build();
    my_Reg.configure(...);
    default_map.add_reg(...);
    ...
  endfunction
  ...
endclass
```

To give some idea of the performance impact of deprecating the factory for this (and only this) aspect of verification environment operation we measured various parameters for several projects. Table II illustrates the number of classes registered with the factory, relative size of the compiled object, and measured times for the compile, load and build phases for a project with 6700 register classes containing 14400 fields. Nearly all of the fields are of the base *uvm_reg_field* type, so these have little impact on the factory overhead, but just represent an increase in content for the environment, whereas the register classes are all new derived types. The model also uses several register blocks in the hierarchy and a few small memories.

TABLE II. PERFORMANCE MEASUREMENTS

Mode	Factory Types	Compile Time	Load Time	Build Time	Disc Usage
no regs	598	23	9	1	280M
reg (with factory)	8563	141	95	13	702M
reg (no factory)	784	71	17	1	398M

The first row in Table II illustrates the measurements, without the register model, for number of types registered with the factory, compile, load and build times (in seconds) and disk usage for the compiled code. The second row illustrates the same measurements in the environment when the full factory-enabled register model is included for this project. Notice that by adding the register model the compile time is increased, the load time is significantly higher and the build time as well. Of course we have added significant capability compared to the plain environment without a register model.

However, if we generate the register model without using the factory, then we significantly impact each of the compile, load and build times without losing register model functionality as shown in row 3 of Table II.

To understand why there is a significant performance impact when using the factory for the register model we have to look behind the scenes in the UVM base code. The *uvm_object_utils* macro is summarized in the following simplified code snippet:

```
`define uvm_object_utils(T) \
  `m_uvm_object_registry_internal(T,T) \
  `m_uvm_object_create_func(T) \
  `m_uvm_get_type_name_func(T) \
  ...

`define m_uvm_object_registry_internal(T,S) \
  typedef uvm_object_registry#(T,`"S`") type_id; \
  static function type_id get_type(); ...\
  virtual function uvm_obj* get_object_type(); ...\

`define m_uvm_object_create_func(T) \
  function uvm_object create (string name=""); ...\

`define m_uvm_get_type_name_func(T) \
  const static string type_name = `"T`"; \
  virtual function string get_type_name (); ...\
```

So basically the *uvm_object_utils* macro just declares some utility methods for the name and type API to the factory, and more importantly declares a *typedef* specialization of the *uvm_object_registry* class called *type_id* (or more specifically *T::type_id* since it is declared inside the class that called the macro). The most significant performance impact is related to the *uvm_object_registry* operation, which constructs an instance of a proxy class (of type *type_id*) for each class that uses the *uvm_object_utils* macro (which is typically all register classes, but only a few specialized fields since most fields are of the *uvm_reg_field* base type). This part of the registry operation is shown in the following code snippet:

```
class uvm_object_registry #(type T, string Tname)
  extends uvm_object_wrapper;

  typedef uvm_object_registry #(T,Tname) this_type;

  local static this_type me = get();

  static function this_type get();
    if (me == null) begin
      uvm_factory f = uvm_factory::get();
      me = new;
      f.register(me);
    end
    return me;
  endfunction

  virtual function uvm_object create_object(...);
  static function T create (...);
  static function void set_type_override (...);
  static function void set_inst_override(...);

endclass
```

For each class that uses the *uvm_object_utils* macro, a corresponding proxy class if defined. This proxy class (*this_type*) is a lightweight replacement for the real class (*T*), and only really knows how to construct a class of type T (i.e. the *create* method) in addition to a few utility methods. The proxy for each object type is automatically constructed using *new* and added to a list in the factory by calling *f.register* when the static *get* function is called to initialize the proxy class variable (*me*). This results in an additional 7k classes being statically declared (which also affects the memory footprint) and takes time to construct during the static initialization phase, when the code is loaded into a simulator.

When we build the register model hierarchy using *create*, the factory class API is invoked for construction of each of the register class instances (even though we never register type overloads) which increases the build time, since the factory has to look-up data stored in associative arrays to determine if the class has been overridden by the user.

So if we avoid the static declaration and initialization overhead and construct registers without searching for non-existing overrides, then we get a significant performance improvement at the start of the simulations, but we also lose the *get_type_name* implementation from the *uvm_object_utils* macro. However, if we look into the UVM code base we can see that this method is only called by *do_print* and as part of some error messages; since we do not print our huge register model ever, and the error messages identify the actual register using *get_full_name* (which uses *get_name* to return the name parameter from the constructor call), then we do not actually need *get_type_name*.

Note that the time values from Tables II are measured in seconds, so although it is proportionally a big performance improvement, compared to using the factory for the model, the absolute numbers are not huge. The decision on whether several minutes additional overhead is tolerable for each and every simulation depends on the application and perhaps also phase of the project. Our recommendation is to include the capability into your register model generator and provide users with the choice of whether to use the factory or not.

B. Register Creation On Demand

Even in the presence of a large number of registers in the DUT and corresponding register model, individual simulations may only require to access some small subset of the registers to validate a particular set of functionality. For example, in the environment used to illustrate the factory performance issues in the previous section, most complex top-level simulation scenarios only accessed less than 500 of the 14k register fields.

So one possible performance improvement that we considered is whether or not we could just create the registers we needed in our model when they were required. This idea is not new, in fact the vr_ad register model for *e* was recently enhanced to allow on-demand static info creation, in order to minimize memory consumption and increase performance for just this reason [8]. When enabled, the static info for a register is only created when (and if) the register is accessed.

In fact there are two major barriers to this approach when using SystemVerilog and the UVM. Firstly we cannot affect the timing of the static initialization operation, so we would always have to conclude static preparation at the start of the simulation when the files are loaded. Secondly, no fields or registers may be added to the UVM register model after the *lock_model* method has been called and *lock_model* is required to initialize the address map prior to functional use of the model. We did analyze in some detail whether we could modify the functional behavior to achieve closure on the address map without building all of the registers, but it did not seem worth the trouble since the observed performance impact is related to the static initialization phase for the model. All attempts at performance improvements by this approach were unsuccessful.

VI. Conclusion/Results

All of the examples illustrated in the paper were used in real projects. In fact it was only during the evolution of these projects that we started to develop best practices that were aligned with the UVM operation and not the generic guidelines provided by the UVM documentation. A summary of the performance improvements is also provided as well as a discussion on the limitations that prevented on-demand register building from happening with the current UVM base-classes. All implementations were validated in UVM-1.1d and also in OVM-2.1.2 using a version of *uvm_reg_pkg* derived from the original OVM-2.1.2 version and updated with all the bug fixes and code improvements from UVM-1.1d, which Verilab has released back to the community [9].

References

[1] Accellera, “UVM User Guide, v1.1”, www.uvmworld.org

[2] Accellera, “UVM Reference Guide, v1.1d” , www.uvmworld.org

[3] V. Cooper, “Getting Started with UVM - a beginner’s guide”, ISBN-978-0615819976

[4] S. Rosenberg, “Register This! Experiences Applying UVM Registers”, DVCon 2012

[5] J. Bromley, “I Spy with My VPI - Monitoring signals by name, for the UVM register package and more”, SNUG 2012

[6] ClueLogic, “UVM Tutorial for Candy Lovers - 16. Register Access Methods”, www.cluelogic.com

[7] S. Holloway, “The UVM Register Layer - introduction, experiences and recipes”, DVClub 2012

[8] Cadence, “UVM *e* User Guide, v13.1”, www.cadence.com

[9] Verilab, “uvm_reg for OVM - uvm_reg_pkg-1.1d”, www.verilab.com