e/eRM to SystemVerilog/UVM

Mind the Gap, But Don't Miss the Train

Avidan Efody
Mentor Graphics, Corp.
10 Aba Eban Blvd.
Herzilya 46120, Israel
avidan_efody@mentor.com

Michael Horn
Mentor Graphics, Corp.
1811 Pike Rd.
Longmont, CO 80501, USA
mike_horn@mentor.com

***Abstract*— With industry trends showing a clear move to SystemVerilog and the Universal Verification Methodology (UVM)[1], verification teams using e and the e Reuse Methodology (eRM) are increasingly looking for a language and methodology migration path. Though migration does entail risk, it holds the long-term promise of multi-vendor support, wide choice of verification IPs and access to a growing market of complementary tools. Planning carefully and minimizing productivity setbacks means asking and answering several questions: Does switching involve redesign of verification architecture and flow or only a syntax change? Will it be possible to build random generators, bus functional models (BFMs), scoreboards, coverage models and tests the "old" way, or a "new" way will have to be adopted instead? And, in general, what part of the experience and knowledge acquired over years of using e/eRM can be reused with the new approach?**

Answers to these questions usually won't be found in training materials that teach SystemVerilog/UVM from scratch. These tend to take SystemVerilog/UVM solutions as a given, without considering the requirements at their base, or comparing them against alternative solutions such as those provided by e/eRM. While helpful, pre-UVM papers that compare specific language features[4] are often limited since they don't account for the many e/eRM-like features added on top of SystemVerilog through UVM. This paper draws on our hands-on experience to provide an updated map of the gaps between e/eRM and SystemVerilog/UVM, and of the ways in which those can be bridged.

Keywords-eRM, UVM, migration, AOP, when inheritance, random generation, reflection, DUT-testbench connection

I. INTRODUCTION

With SystemVerilog/UVM (Unified Verification Methdology) gradually coming to dominate the verification landscape, many e/eRM (e Reuse Methodology) teams are starting to experiment with it at varying levels of intensity. In some cases selected individuals might go through a SystemVerilog/UVM training or be asked to test language features deemed crucial. In others a pilot project will take place, to make sure that a list of requirements can be addressed with SystemVerilog/UVM. If the results are satisfactory, a management decision might state that, moving forward, all verification projects should use SystemVerilog/UVM. Whatever the concrete situation is, there will probably be a point when a few e/eRM verification engineers are grouped in front of a whiteboard trying to figure out what a SystemVerilog/UVM testbench for a specific block should look like. This paper aims to help them.

The paper is divided in two main parts. The first is a "quick migration reference" allowing e/eRM teams to detect the potential hotspots when moving to SystemVerilog/UVM and focus on those rather than on the more intuitive parts. It lists e/eRM features alongside their SystemVerilog/UVM counterparts and estimates how similar/different they are. To keep things in a practical context, features are ordered according to the typical testbench parts in which they are most often used. Hence *ports* and *static hierarchy* are discussed as part of a **testbench skeleton** section, *transaction modeling* and *sequences* as part of a **stimuli** section, and *packing/unpacking* as part of a **drivers and monitors** section.

Each testbench part has a dedicated table with the rows representing features commonly used in it. The rows are color coded: Green means that there is a high level of similarity between e/eRM and SystemVerilog/UVM. Yellow means there are significant technical differences but these don't tend to have a profound impact on testbench architecture. Red means the differences are conceptual and might influence testbench architecture.

The second part contains in-depth discussions of five selected areas: **AOP** (Aspect Oriented Programming), **when inheritance**, **reflection, memory allocation during randomization**, and **connecting testbench to DUT (Device Under Test) signals**. Because they are often perceived as migration blocking points we see *AOP* and *when inheritance* as entitled to a more in-depth discussion than the table would allow. *Reflection* influences many distinct testbench parts, and its full effects are therefore somewhat hidden when a testbench is considered block by block. Like *reflection* the differences in memory allocation during randomization have horizontal effects that can be better pointed out when discussed standalone. However, the stronger incentive for analyzing these differences in-depth is that they often take e/eRM users by complete surprise. Finally, connecting testbench to DUT signals is a relatively technical issue with limited impacts that usually poses an obstacle to migrating groups all the same.

When writing this paper we always aimed to stay at the conceptual/architectural level. Technical details are only discussed insomuch as we think they might influence testbench

structure. Otherwise, readers are simply referred to the detailed LRM vs. LRM comparisons already in place [2]. Constraints are an example in point: although there are numerous syntactical differences in the way they are coded, none of these really has any high level implications. Therefore, they are not discussed at any length in this paper. The same goes for *soft constraints*, a well-known difference between e and SystemVerilog, discussed at considerable detail in earlier publications [4]. We see it as yet another difference that might have some local affects, but nothing that e/eRM users should be aware of upfront before they migrate.

It is our firm belief that customers don't have to figure out solutions on their own. SystemVerilog, and even more so UVM, have already gone to considerable length to try and satisfy sensible requirements that e/eRM users had. There is absolutely no reason why this healthy process should not go on. As part of the work on this paper we have created a "migration kit" that aims to minimize some of the more prominent gaps between SystemVerilog/UVM and e/eRM. This migration kit" is comprised of UVM add-ons, coding guidelines and examples and is publicly available [5]. We refer the readers to various parts of this kit where relevant.

II. SIMILARITIES AND GAPS BY TESTBENCH PART

TABLE I. TESTBENCH SKELETON

Category	e/eRM	SystemVerilog/UVM
Static Hierarchy	A hierarchy of *unit*s exists from simulation time 0 to simulation end and is used to implement interface drivers, monitors, reference and coverage models, and in general everything that must be up and running whenever the DUT is alive.	A hierarchy of *uvm_component*s exists from simulation time 0 to simulation end and is used to model the same testbench parts as in e.
	The *unit* hierarchy is randomly generated.	The *uvm_component* hierarchy is sequentially created and not randomized by default. It is up to the user to randomize it when required. Special care has to be taken when the *uvm_component* topology is itself partially random (i.e. contains arrays of *uvm_component*s of random size). See **memory allocation during randomization** in the section 3.D, for an in depth discussion and suggested solutions.
	e_path() can be used to get the unique e path to a specific unit.	Every *uvm_component* has a unique SystemVerilog path, but there is no SystemVerilog API to get it (see **reflection** in section 3.C for more details). Therefore, *uvm_component*s also have a string field called *name* and the full name of a component serves as its unique ID. It is usually recommended to keep the UVM name of a component equal to the SystemVerilog handle name.
	get_enclosing_unit(),*get_all_units()*, *get_parent_unit()*, and other APIs can be used to find units in the hierarchy.	*find()*/*find_all()* can be used to find a *uvm_component* by its name. UVM versions of the e/eRM API can be easily implemented. Examples can be found as part of our migration kit [5].
Port connections	e/eRM *port*s of all kinds are used to connect *unit*s in a reusable way that is decoupled from actual implementation.	UVM ports of all kinds are used for the exact same purpose.
	e/eRM *method_port*s are usually used to pass *struct*s and other information from one *unit* to another; for example, to pass a transaction from monitor to scoreboard.	UVM *tlm_analysis_ports*/*exports* are usually used to pass transactions between *uvm_components* in the exact same way. The main difference is that a *method_port* could pass a few parameters in parallel, while a *tlm_analysis_port*/*export* can pass only one, usually a transaction. UVM can be seen as encouraging a stricter TLM approach than e/eRM. The channels UVM provides require users to package all information they want to pass in a single transaction object.
	e/eRM *event_ports* are used to broadcast events. They are also used to listen to signal level events.	*tlm_analysis_ports/exports* can be used to broadcast events. Connecting to DUT signals is done in a different way with UVM and discussed in detail under **connecting the testbench to DUT signals** (section 3.E)
	simple_ports are used to pass basic types and to connect to DUT signals.	*tlm_analysis_ports/exports* can be used for that as well. Connecting to DUT signals is discussed under **Connecting the testbench to DUT signals** (section 3.E)
	buffer_ports buffer data until retrieved by other side.	tlm_fifos can be used for the same purpose.

TABLE II. STIMULI

Category	e/eRM	SystemVerilog/UVM
Transaction modeling	A transaction is seen as the basic data unit and a building block for more complex sequences. Outlining transaction borders and the amount of control a user will have over its contents via API is a major part of designing random stimuli for an interface.	Transactions play the same role in UVM.
	when inheritance is often used to model transactions (i.e. multiple transaction types are often modeled using a single *struct* with multiple *when subtypes*)	SystemVerilog doesn't support "when inheritance". See **when inheritance** in section 3.B for in-depth discussion and alternative approaches.
	Random generation and constraints are often used with transactions.	SystemVerilog's randomization process is different than e's and migrating users should be aware of the differences. See **memory allocation during randomization** in section 3.D for an in depth discussion and suggested solutions. There are also many differences with regards to constraints at the technical/syntactical level, but not at the conceptual level. For a detailed technical comparison see [2].
Sequences	eRM sequences are a way to constrain transactions in a time/state dependent way. They are used for system bring up at the initial stages of verification, for closing coverage holes at the final stages or for areas that require less randomness such as initialization.	The use model for sequences in UVM is identical.
	Sequence hierarchies can be created to descend/ascend abstraction layers and implement protocol stacks.	Same in UVM.
	sequence_driver arbitrates between sequences according to configurable arbitration policy. *get_next_item()*/*try_next_item()* can be used to retrieve arbitration output from a BFM.	A *uvm_sequencer* plays the exact same role in UVM and has an almost identical API.
	Sequence *body()* executes sequence items or subsequences using *do…keeping*.	Sequence *body()* executes items using a similar looking macro: *`uvm_do_with(item, {constraints})* Or with a straightforward API (easier to debug): *start_item(item); // go through arbitration* *item.randomize() with {constraints};* *finish_item(item); // wait for item_done()*
Virtual sequences	eRM virtual sequences are used to synchronize stimuli on independent interfaces.	Same in UVM
	Virtual *sequence_driver* simply holds a list of sequence_drivers attached to real interfaces.	A virtual sequencer can be created in UVM. However, there is not a need since, the list of sequencers is usually stored in a generic sequence that starts sub-sequences on those sequencers.
	A sequence item must be tied to a specific sequencer prior to execution.	Same in UVM
Test termination	eRM allows for a test to be terminated when a specific stimuli has been injected, a number of elements checked by the scoreboard or certain coverage reached. This is done through the	UVM implement the exact same mechanism, with the exact same API.

	objection mechanism, allowing each element to "object" to test end.	

TABLE III. DRIVERS AND MONITORS

Category	e/eRM	SystemVerilog/UVM
Transaction to signal level conversion	High level verification requires work with transactions, while the DUT usually understands signal level activity. e/eRM has a built in way of supporting conversion between abstraction levels.	Same in UVM.
	pack()/unpack() used to automatically turn all physical fields into a list of integers of any size. *do_pack()*, *do_unpack()* can be used to customize default implementation.	The API is similar. Built in implementation supports only packing into 1/8/32 bit integers, and only big endian and small endian packing modes.
	Automatic implementation is based on e's reflection API	Automatic implementation is based on UVM field macros. See **reflection** in section 3.C of this paper for an in-depth discussion.
Connecting testbench to DUT signals	e/eRM implements an abstraction layer on top of HDL signals allowing users to refer to those as ordinary struct fields. This is mainly achieved through an automatic code generation stage that is not a native part of SystemVerilog/UVM, but can be implemented for them as well. See section 3.E for more details.	

TABLE IV. SCOREBOARDS AND COVERAGE

Category	e/eRM	SystemVerilog/UVM
Reference model	Coding a reference model is a very specific task. e/eRM infrastructure doesn't provide a lot of help in this area.	Same in UVM.
Storing/ searching transactions	A typical e/eRM scoreboard calculates expected DUT output transactions through a reference model, stores the transactions in a list and matches them against actual output transactions. e lists allow for efficient searching/matching according to any user criteria.	SystemVerilog arrays and queues have approximately the same capabilities as e lists. Unlike lists, multiple dimensions are allowed.
Comparing transactions	*compare()/deep_compare()* automatically compare all transaction fields. *do_compare()* allows customization. Implementation is based on reflection API.	API and capabilities are similar. Implementation is based on UVM field macros. See **reflection** in section 3.C for in-depth discussion.
Message/error control	To enable efficient debugging e/eRM allows errors/messages to be configured/disabled based on various criteria.	UVM gives users the same control of errors and messages
	Messages can be filtered based on origin (specific IP), tag (horizontal screening), or string match (pinpointed). They can be sent to files/screen and formatted. Control of messages is always done through the enclosing unit.	Same in UVM.
	Errors are mainly filtered using string match. Control and configuration of errors is done through an API that is separate and different from messages.	UVM has a unique API for errors and messages. Errors are just messages with different severity, but can be configured by the user.
Coverage model	Coding a coverage model is a very specific task, e/eRM infrastructure doesn't provide a lot of help in this area.	Same in UVM.
	Few differences at the technical/syntactical level, but no conceptual ones. For a detailed technical comparison see [2].	

TABLE V. TESTS

Category	e/eRM	SystemVerilog/UVM
Test/ Testbench separation	Test/testbench separation is required to prevent users from frequently changing the verification code base, while allowing them to test all the different features of the DUT. e gives users convenient means to achieve this separation.	Same in UVM.
	The API of an e/eRM testbench is loosely defined. Users can, in theory, change/override anything, including private fields and methods from[1] a test file. However, a good verification architect can make some parts more accessible than others. For example, by making fields public or providing hook methods.	The API of a SystemVerilog/UVM testbench has a more compulsive nature. Users will be able to access only things that fall within the API from a test file. For example, a field can be constrained only if it is public, and a function can be overridden only if it is virtual.
	AOP and *when inheritance* are used to extend structs and units from a test, to configure fields, create new subtypes and shape stimuli.	See **AOP** in section 3.A and **when Inheritance** in section 3.B for how to use UVM factory, UVM configuration and some SystemVerilog coding guidelines to achieve similar capabilities in UVM.

[1] Although not straightforward it is possible. For example, An e method defined as private can only be accessed from the same package and same or derived structs. However, a user can add any file to an existing package and override private members from this file.

III. IN DEPTH DISCUSSION OF SELECTED AREAS

In this section we discuss five selected areas in depth: **AOP (Aspect Oriented Programming)**, **when inheritance**, **reflection**, **memory allocation during randomization** and **connecting testbench to DUT signals**. For each of these we describe the requirements behind the e/eRM solution and the ways that are available within SystemVerilog/UVM to address some or all of them. For the missing parts, we provide UVM extensions and/or suggest some coding guidelines that would help users find an adequate alternative.

A. Aspect Oriented Programming (AOP)

Aspect Oriented Programming (AOP) allows e users to add fields, constraints, events and methods to *struct*s and *unit*s; to override existing implementations of methods or to append/prepend them; to override events; to override and extend coverage; and to extend enumerated types. Thus *AOP* gives a skilled e user the ability override just about anything from an external test, including fields and methods defined as private. However, **a well planned e/eRM testbench will have parts that are easy to extend and override, and others that will require significant amount of effort**.

The main requirement addressed by e's *AOP* is test/testbench separation. *AOP* allows users to configure, modify and extend testbench behavior without ever touching testbench code. As already mentioned, this is a necessity if a multitude of users are to check a multitude of different DUT features using the same code base.

1) UVM solutions

Although AOP is not supported by SystemVerilog, UVM does provide a couple of means for configuring, modifying and extending testbench behavior from a test. Configuration, unless required to be random, is usually done via the UVM configuration database. Modifications and extensions can be achieved through the UVM factory.

The UVM *configuration database* is a simple mechanism for passing configuration data from a test file to the testbench. Whereas an e user would add a "*keep has_coverage"* line to a test to configure a specific unit to collect coverage, in UVM that would be done via the configuration API. For configuration parameters that are not required to be random it is usually a sufficient replacement.

The UVM *factory* is a global object that exists in every UVM testbench and is used to create instances of UVM objects and components. Instead of constructing an object by directly calling *new()*, UVM users ask the factory to create an object of a specific type for them. Pushing the factory into the middle of the object creation process enables overrides of specific object types later on, for example, from a test file. When an override is defined, it will make the factory return a different type of object than the one originally requested. This allows for external extensions or customizations of specific testbench parts, but in a more limited way than AOP, since instances returned by the *factory* are still accessed by a reference to the original object type.

UVM's *configuration table* and *factory* require that the areas a user would like to access from a test are defined upfront, during testbench coding. Depending on one's point of view they might be perceived as more restrictive than *AOP* or as enforcing a more structured use model and an accurate definition of the API. **A well-planned UVM testbench will have parts that are easy to extend and override, and others that can't be since they are outside the API**. Table IV below, compares e's *AOP* against the capabilities of UVM's *configuration table* and *factory*.

Using UVM's *configuration table* and *factory* adds a small coding overhead. For the *configuration table,* any parameter that is required to be configurable from a test must be pulled from the *configuration table* prior to use. For the *factory,* every object must be registered with the *factory* in order to allow it to be overridden later. Also, object instantiations must not be done directly using *new()*, but rather using the factory API since the object might be overridden from a test.

TABLE VI. THE EXTENSION CAPABILITIES OF AOP VS. THOSE OF THE UVM FACTORY.

e/eRM AOP	UVM Factory/ Configuration
Set configuration fields to specific values.	Users can set non random configuration via the *configuration database*. Random configuration can be constrained using the *factory*.
Add fields, methods, events, coverage.	Users can add any of these through the factory
Add constraints.	Users can add constraints to any object through the factory. Since SystemVerilog's *randomize()* is a virtual function, the added constraints will be taken into account by the solver.
Override methods, append or prepend actions to method.	The factory lets users override only virtual functions that were defined upfront as candidates for override. For such functions, users will also be able to append or prepand functionality to existing implementation through the use of super.
Override events.	SystemVerilog events are different than e events since they are always emitted by tasks or functions. The ability to override the triggering of these events depends on the

	ability to override the emitting functions/tasks.
Override/extend coverage.	SystemVerilog/UVM covergroups can't be extended. To disable or re-implement them users should wrap covergroups within classes that are registered with the UVM factory. This allows replacement of a coverage class with another when required. Another option is to use SystemVerilog/UVM configuration space to control coverage.
Extend enumerated types.	SystemVerilog doesn't allow enumerated types extensions. A typical use case is discussed under **when inheritance** in section 3.B

2) Examples

a) Extending Specific Instances

AOP can be used together with *when inheritance* to extend only a specific instance of a *struct* in a hierarchy. This is usually done by extending only *struct*s with a specific name value, where name is an enumerated type.

```
// in testbench
type agent_name [AGENT0, AGENT1, AGENT2]

// in test
extend AGENT0  agent {
   keep delay < 20;
};
```

The UVM *factory* can be used in the same way through an instance rather than a type override:

```
// in test
class agent0 extends agent;
  // factory registeration
  `uvm_component_utils(agent0)

  //override the constraint with the same name
  constraint delay { delay < 20; };
endclass

set_inst_override_by_type(
  agent::get_type(),   // overridden type
  agent0::get_type(),  // overriding type
  "uvm_test_top.env.agent0"
  // path to overridden instance
);
```

b) Base Class Extension

One characteristic of *AOP* is that it allows users to add class members, such as fields and functions, to a base class after it has been derived from. This type of extension is usually referred to as "orthogonal extension", and can be handy in many cases. For example, we have used an orthogonal extension of all *uvm_component* derived classes to create a package that simplifies the connection of testbench to DUT signals, as described in section 3.E below.

As already noted by earlier papers on the subject [4], an infrastructure for creating orthogonal extensions can be set up in any object oriented language by following the, so called, façade design pattern. The trick is to add a generic placeholder object, or façade, to the base class. Users wishing to extend a base class and all the classes that derive from it would then extend the façade, instead of extending the original class. To allow multiple independent extensions of the same base class, an array of facades is normally used. Just like the factory and configuration table, this technique is more restrictive than AOP since it requires the testbench designer to lay down the infrastructure for extending a specific base class upfront.

UVM contains a class called *uvm_callback* which implements façade objects. Testbench designers can instantiate those in strategic classes that they think user might want to extend later. UVM also has some of these façade objects in a few of its own base classes such as *uvm_report_object*, but not in all. For example, the following code makes use of the *uvm_report_object* callback, to apply special formatting to all messages coming from a specific component[2].

```
// extend the uvm_report_object façade
// to apply formatting
class msg_formatter extends
                            uvm_report_catcher;
  local string format_string;

  function new(string name="msg_formatter",
               color_t font_color,
               color_t bg_color);
    super.new(name);
    format_string = create_format(font_color,
                                  bg_color);
  endfunction : new

  function string create_format(
                            color_t font_color,
                            color_t bg_color);
    //…
  endfunction : create_format

  // apply formatting to any message
  // from the component
  function action_e catch();
    string msg = get_message();
    $sformat(msg, format_string, msg);
    set_message(msg);
    return THROW;
  endfunction : catch
endclass : msg_formatter
```

[2] Full code for this example is available from http://verificationacademy.com/sites/default/files/uvm_colors.tar.gz

```
// This env delegates message formatting
// to the callback object
class colorful_env extends uvm_env;

  //…

  color_t font_color = BLACK;
  color_t bg_color = WHITE;

  local msg_formatter formatter;

  function void end_of_elaboration_phase(
                              uvm_phase phase);

    // get font_color and bg_color
    // from configuration table
    //…

    // create the callback object
    formatter = new("formatter",
                    font_color,
                    bg_color);
    // attach it to all components
    // under colorful_env
    uvm_report_cb::add_by_name("*",
                               formatter,
                               this);
  endfunction : end_of_elaboration_phase
endclass : colorful_env
```

One base class where it would be very useful to have facades integrated is *uvm_component*. There are many cases when users want to add some functionality to all the semi-static objects in their testbench, but to do so they have to extend all the classes that derive from *uvm_component*. Unfortunately, there are a multitude of those (*uvm_test*, *uvm_env*, *uvm_sequencer*, *uvm_driver*, and more). As part of this paper we have created a simple package that extends all these classes with an array of façades[3]. Users can then create a custom façade and add it to all uvm_component derived classes, to implement an additional functionality for all of them. The following code shows how the package can be used to add an hdl_path field, similar to the one that exists in e units, to all uvm_component derived classes[4].

```
// façade class implements some virtual hooks
// that could be extended later
class component_facade extends uvm_object;
  //…
  virtual function void build_hook();
  endfunction : build_hook
endclass : component_facade
```

```
// component add lists of facades and calls
// them at points where users might want
// to extend behavior
class aop_component extends uvm_component;
  component_facade m_facades[];

  // …

  function void build_phase(uvm_phase phase);
    super.build_phase(phase);
    foreach(m_facades[i])
      m_facades[i].build_hook();
  endfunction : build_phase
endclass : aop_component

// To create a custom façade users simply
// extend the component_facade base class

class smp_facade extends component_facade;

  //…

  string hdl_path;

  virtual function void build_hook();
    calculate_hdl_path();
  endfunction : build_hook

  function string calculate_hdl_path();
    //…
  endfunction : calculate_hdl_path
endclass : smp_facade
```

B. *When Inheritance*

When inheritance allows e/eRM users to add fields, constraints, methods, events and coverage based on the value of an enumerated field. It is often used together with AOP to extend or override functionality only for a specific subtype. *When inheritance* is widely used in e/eRM testbenches but has a significant added value over conventional Object Oriented (OO) techniques only in limited areas. In this section we look at the conventional OO techniques used to emulate *when inheritance* in SystemVerilog, and describe their respective pros and cons.

1) *SystemVerilog solutions*

There are two approaches that give *when inheritance*-like behavior in SystemVerilog: the object factory approach, which will be referred to here as "multi-class" to avoid confusion with the UVM factory, and the "single-class" approach. In the "multi-class" approach users simply define a class for every *when subtype* they have; in the "single class" approach, users sum up all *when subtypes* into one class.

The multi-class approach looks when-like since it allows different subtypes to have different fields, even with identical names. However, there are two points where it strongly diverts from *when*:

1) If some of the subtypes share groups of fields, the class hierarchy created might have duplicated parts, which can't be accessed through a base type. For example, if a user tries to model the packets shown in Fig. 1 below, two distinct types would have to be

[3] The full code for this package is part of our migration kit [5]

[4] The full code of the example is part of our migration kit [5]

created to represent AX and BX, each of them containing all X's fields.

2) In order to randomly generate a mix of different subtypes, perhaps with some user-defined distribution, the randomization process must be broken down in two steps: first, randomize the type of the object, then allocate the specific class required and randomize its contents.

If 1) is not true, then breaking down the randomization process in two steps usually won't have any major effects, since any constraints on the subtype fields actually imply the choice of a unique subtype.

Due to these differences, "multi-class" has been largely dismissed as a *when inheritance* replacement [4]. However, looking at the SystemVerilog parallels of some common e packages reveals that e's *when* is often replaced with "multi-class" on the SystemVerilog side. This is simply because in some cases the subtypes don't share any common groups of fields, That is, 1) is not true. The implementation of UVM sequences, and register package are two such examples. Both are implemented using *when inheritance* in e/eRM and the multi-class approach in UVM, but users will only rarely, if at all, notice any change. The main reason is that neither sequence subtypes nor register subtypes tend to have any group of fields in common, apart from what the base class provides.

As subtypes share more groups of fields, with more constraints tying fields from one group to fields from other groups, the multi-class approach gradually gets more and more cumbersome to implement. It becomes practically impossible when verification architecture requires that constraints on fields affect subtype choices. For example, if a user would like to specify the constraint X1 == 8 from a test, and get all packet types allowed by Fig. 1 for that value, then a multi-class approach simply can't be used.

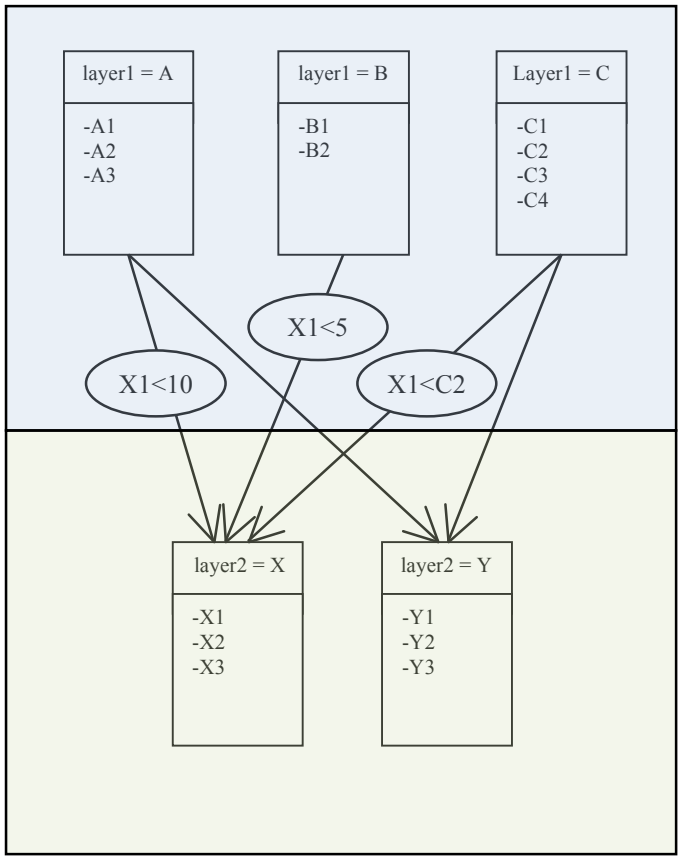

Figure 1. Multiple field groups bounded with constraints. Layer 1 can be either A, B or C. Layer 2 can be either X or Y. Some cross combinations such as AX, AY, BX, BY, CX are allowed, with some constraints, and some, such as CY are not.

In cases where the "multi-class" is not an option, a "single class" can always be used. "Single-class" simply means that all *when subtypes* are merged into one SystemVerilog class. When choosing this way, we recommend grouping fields that would have been defined under a *when* subtype into a class of their own (for example, "A fields", "Y fields", etc. for diagram 1). This will prevent name collisions, reduce congestion during debugging and help identify irrelevant field bundles if the user takes care to set those to *null* at *post_randomize()*. Of course, the more when subtypes one has, the larger the single class grows, and the harder it becomes to tell relevant from irrelevant information. However, in cases we have seen, this never seemed to be an issue.

In general the multi-class approach is easier on debug and performance but limited in its application areas. The opposite holds true for the single-class approach.

2) Examples

a) Creating new subtypes from a test

A common way of using AOP with *when inheritance* is to extend an enumerated type from a test, then use the additional enumerated value to create a complete subtype. In general, it is possible to do the same with SystemVerilog/UVM, whether "multi-class" or "single-class" is used to model the transaction. We will show how this is done with "single-class" since this case is slightly more challenging and requires some upfront preparation during testbench coding.

The main idea is to have an extra placeholder enumerated value for every enumerated type expected to be extended from a test. This placeholder value lets the user make a neutral choice from the test, which won't activate any constraints in the testbench. The user can then add fields and a set of extra constraints for the new subtype. The example below shows how this is done[5]:

```
// testbench code
typedef enum [TYPE_A, TYPE_B, TYPE_C,
              PLACEHOLDER] packet_type_t;

class a_fields extends uvm_object;
   rand int A1, A2, A3;
endclass : a_fields

class packet extends uvm_sequence_item;
  rand packet_type_t type;

  // disabled from test
  constraint not_placeholder
      { type != PLACEHOLDER; };

  rand a_fields A;
  rand x_fields X;

  constraint X1_lt_10_when_A
      { type == TYPE_A ->  X.X1 < 10; };

endclass : packet

// test code
typedef enum {D} extra_packet_types_t;

// the extended packet can generate all
// previous subtypes plus the new one
class extended_packet extends packet;
  rand extra_packet_types_t extra_type;

  // disable constraint
  constraint not_placeholder {};

  constraint placeholder_is_d
      {(type == PLACEHOLDER) ==
                        (extra_type == D);};

  rand d_field D;

  constraint some_constraint
      {extra_type == D -> X.X1 < D.D1; };
endclass : extended_packet
```

[5] The full code of this example is part of our migration kit [5]

```
set_type_override_by_type(
   packet::get_type(),
  extended_packet::get_type(),
  "*"
);
```

C. Reflection

Reflection (or introspection) is not designed to address any specific verification requirements, but rather to create an infrastructure on which solutions for specific requirements could be built. At the technical level it means that e keeps a database for each of its types. For a *struct* or *unit*, this database would include an entry per field, which holds information such as the field type, the field name, whether it is physical/ungenerated or not and so on. This database can then be used, for example, to reduce code maintenance effort by providing a default implementation for widely used functionality such as copy() or print(). Or it could be used to allow e *struct*s and *unit*s to access their absolute e instance name, which would help users understand where messages are coming from. There are of course other ways, including user defined ones in which it could be used.

Since SystemVerilog doesn't support reflection, replacements for specific reflection-based applications must be discussed on a per-application basis. We will do so now for the most widely used reflection based application: the implementation of print(), copy(), pack() and other predefined *struct* utilities.

1) print(), copy(), pack(), unpack(), compare(), deep_compare()

Without reflection it is not possible to have a default implementation that will actually do something valuable for any of the methods above. More often than not, only a limited subset of those methods will actually be useful. However, in the very few cases where they are all useful, implementing them by hand might result in a very verbose code, and therefore, in a code maintenance problem.

To solve this problem, *field automation macros* were introduced into UVM. These can be used to implement any or all of the above mentioned utilities for a subset or for all fields of an object. While they are convenient and can reduce the amount of code the user has to write and maintain, they have some downsides that users should be aware of. Additional debugging effort due to the use of long macro sequences, an overly complex implementation and a minor performance pain, are probably the most important ones. For a comprehensive list see "Are Macros Evil?"[3]. The code below shows an example of how these macros are used.

```
class transaction extend uvm_sequence_item;
  rand int data[];
  `uvm_object_utils_begin(transaction)
    `uvm_field_array_int(data, UVM_ALL_ON)
  `uvm_object_utils_end
    //…
endclass : transaction
```

We recommend using UVM field macros very carefully and only where absolutely required. In the case of *uvm_components*, most of the utility functions that are automatically created are not relevant in any use case. Like e *units*, *uvm_components* can't be copied or unpacked, and there's probably little need to compare or deep compare them. The only relevant method for those is probably *print()*, so it might just as well be implemented manually instead of using the macros. The same is often true for sequences. Transactions are where field macros contribute the most since all of the utilities can be relevant to them. However, we strongly recommend a case-by-case analysis.

As part of the work on this paper we have implemented a version of the UVM field macros[6] that uses template classes to reduce macro code to almost zero. This can improve significantly the debugging difficulties and a few of the other known disadvantages, but unfortunately not the complexity of the implementation, since it is an almost exact copy of the macros.

D. Memory allocation during randomization

To allow objects to be allocated according to the random values generated, e combines together memory allocation and random generation. SystemVerilog, on the other hand, keeps them separated, in order to simplify debugging and reduce random generation complexity. This seemingly technical difference has considerable implications on testbench structure and code, that migrating e/eRM users should be aware of.

1) SystemVerilog solution

The first time e/eRM users will notice the difference in generation, is when they try to randomize an object that contains a child random object. While in e the child object will be allocated in memory automatically after a *gen* action on its parent, in SystemVerilog it will remain *null* after a call to *randomize()* on its parent. To allow the child object to be randomized along with its parent SystemVerilog users usually extend the pre-defined *pre_randomize()* function of every class, to allocate the child object.

```
class parent_object;

  rand nested_object nested;

  function void pre_randomize();
    //Don't create a new object every time
    // randomize() is called
    if (nested == null)
      nested = new("nested");
  endfunction : pre_randomize

  //…
endclass : parent_object
```

a) Object arrays of random size

Things become slightly more complicated when the parent object contains an array of child objects. If the size of the array is not random, this is just a matter of introducing a loop that will allocate all required elements into the example above. When it is random users are faced with a choice that has similar aspects to the "single-class"/"multi-class" one presented in the **when inheritance** section (3.B). They can either go for a two-step generation and randomize the size before they randomize the array contents, as shown below:

```
class obj_array_container;
  rand int unsigned obj_array_size;

  constraint obj_array_size_max_c {
                        obj_array_size < 30; }

  obj obj_array[];

  // after the size is randomized allocate
  // the array accordingly, then randomize
  // each element
  function void post_randomize();
    obj_array = new[obj_array_size];

    foreach (obj_array[i])
      begin
        obj_array[i] = new();
        obj_array[i].randomize();
      end
  endfunction : post_randomize
endclass : obj_array_container
```

Or, they can allocate the maximal possible array size prior to randomization and randomize the size of the array together with everything else. Although SystemVerilog will not allocate new memory during generation, it will de-allocate it, so in the second case, it will automatically remove any array elements beyond the chosen size.

Obviously, each solution has its own advantage and disadvantage and users must choose which of these matters most. The first solution is cheap in memory, but breaks randomization in two steps, which means that object array contents can't affect the size of the array. The second solution is high in memory consumption, but allows the generated random array elements to influence the size.

In cases where the maximum possible array size is not known, users can try and break the nested object into individual arrays of basic types. SystemVerilog will allocate dynamic memory for those, so their sizes can be randomized in-line with the rest of the random variables. Such cases, however, are rare and when they are found, might imply that the transaction object has not been delineated correctly.

2) uvm_component arrays of random size

When the type of the objects in the random size array is derived from *uvm_component*, most UVM users automatically opt for a solution similar to the two-step one shown above. They do so because *uvm_component*s, once created, can't be destroyed, so allocating a maximal size array and then cutting it down to the right size, is not an option. Implementing the two-step solution in this case can be done in various ways, but the most common one is to generate the size of the array within some random configuration object, then make this object available to the parent of the array during the *build_phase*().

[6] The code for this version of the macros is available as part of our migration kit [5]

It often happens that when the components in the array require some random configuration themselves, this configuration will find its way into the random configuration object as well. In some cases, to avoid having some configuration as fields of the components, and some inside the random configuration object, users simply move all configuration to the configuration object, creating a sort of a shadow semi-static hierarchy before they build the actual one.

An alternative solution, created as part of the work on this paper, allows users to randomize their entire static hierarchy in a single step, in a very similar way to static generation of e. This is achieved by setting redundant *uvm_component*s in arrays to an inactive state rather than de-allocating them. An inactive component skips all the phases following the *build_phase()*. While this eliminates the need for an external configuration object, and allows all components and configuration to be randomized together, it does come at the price of higher memory consumption. Whether this disadvantage outweighs the advantage depends on a case-by-case analysis, and on user preferences. The code below shows what this solution looks like:

```
// on_off_pkg
class on_off_test extends uvm_test;

  function void end_of_elaboration_phase
                            (uvm_phase phase);
    this.randomize();
  endfunction : end_of_elaboration_phase
endclass : on_off_test

class on_off_component extends uvm_component;
  rand bit off;
  constraint off_if_parent_off
                     { m_parent.off -> off; };
  task run_phase(uvm_phase phase);
    if (off) return;
  endtask : run_phase

  function void check_phase(uvm_phase phase);
    if (off) return;
  endfunction : checkphase

  //more default implementations for phases…

endclass : on_off_component

// Testbench.

class my_component extends on_off_component;
  task run_phase(uvm_phase phase);
    // call the super-class to check for
    // the off bit. This should be done
    // in every overridden phase
    super.run_phase(phase);

    // do stuff
  endtask : run_phase
endclass : my_component

class base_test extends on_off_test;
  rand unsigned int component_array_size
  rand my_component component_array[];

  // array size is always the maximal size. We
  // "delete" component by setting them to off
  constraint component_array_size
      { component_array.size() == MAX_SIZE; };
  constraint component_on_size
      { component_array_size <= MAX_SIZE;

        foreach (component_array[i])
          if (i<component_array_size)
            component_array[i].off == 0;
        else
          component_array[i].off == 1;
      };

  function void build_phase(uvm_phase phase)
    component_array = new[MAX_SIZE];
    foreach(component_array[i])
      component_array[i] ==
       my_component::get_type::
          create({"component_array[", i, "]"},
                 this);
  endfunction : build_phase
endclass : base_test

// test
class test1 extends base_test;
   constraint component_array_bt_10
       { component_array_size > 10; };
endclass : test1
```

E. Connecting testbench to DUT signals

e adds a layer of abstraction on top of HDL signals. From the e user point of view, signals, represented by *simple_port*s are just additional *unit* members that can be read and written in an almost identical way to normal *unit/struct* fields. e users are typically fully unaware of the way in which the actual connection to the HDL signals is performed and of any nuances that exist between Verilog and VHDL, reg and wire, etc..

To support this abstraction e simply generates automatic connection code prior to actual running, during the *stub creation* phase. The automatic code generator takes care of the specific requirements of the HDL in question, so that the users don't have to deal with those. Since the automatic generation is performed at run time, it allows users to refer to signal names via strings, and use string manipulation methods to specify names.

The actual signal connection code generated is similar to the code that is required in order to connect a SystemVerilog testbench to a DUT. In both cases this code is relatively simple and repetitive, and is made of a few simple patterns that are applied in a limited number of cases. SystemVerilog and UVM don't support automation of connection code natively, but as part of the work on this paper, we have implemented such automation. Users can choose to learn the few simple patterns for connecting various types of signals to their testbench, or to use this package.

1) Examples

a) Connecting a Verilog reg to a tesbench

As mentioned above, the patterns that users should follow for connecting signals depend on the HDL and the signal type. The code below shows the typical way of connecting a Verilog reg to a testbench. (If the code seems verbose this is because connecting one signal and hundred signals requires the same overhead)

```
// signal we want to connect to
module dut();
  reg x;

  //…
endmodule

// use an interface to connect to the signal
interface dut_if();
  // used for monitoring x
  wire x_in;
  // used for driving it
  reg x_out;
endinterface

module tb();
  //…

  dut dut_i();
  dut_if dut_if_i();

  // connect the monitor pin
  assign dut_if_i.x_in = dut_i.x;
  // connect the driver pin
  always @(dut_if_i.x_out)
    dut_i.x = dut_if_i.x_out;

  initial begin
    //put interface in configuration database
    uvm_config_db#(virtual dut_if)::
              set(null, "*", "vif", dut_if_i);
    // …
  end
endmodule

package test_pkg;
  //…
  class test extends uvm_test;
    //…

    virtual dut_if vif;

    function void build_phase(
                              uvm_phase phase);
      //…
      // pull virtual interface from
      // configuration database
      if (!uvm_config_db#(virtual dut_if)::
                   get(this, "", "vif", vif))
        `uvm_error("NO_VIF",
             "Couldn't find vif in config db")
    endfunction : build_phase

    task run_phase(uvm_phase phase);
      //…
      // use it to drive/monitor
      vif.x_out = 0;
      #1;
      vif.x_out = 1;
      //…
    endtask : run_phase
  endclass : test
endpackage : test_pkg
```

Similar patterns for other cases can be found through the link in the footnote[7].

b) Using the automation package to connect the testbench to DUT

The DUT-testbench connectivity package we have created allows users to:

1. Define signals inside classes rather than inside virtual interfaces.
2. Attach *uvm_component*s to specific HDL blocks using an *hdl_path* field.
3. Define signal names that relative to a component's *hdl_path* field.
4. Define signal names using strings and string manipulation methods at run time.
5. Control the actual connection code generated using normal configuration parameters.

The example below shows how this package is used to define signals and connect them.

```
// verification IP code
package vip;
  //…
  import smp_pkg::*;

  class vip_component extends aop_component[8];
    //…

    // define signals
    smp_port#(1) req;
    smp_port#(1) ack;

    // create them
    function void build_phase(
                              uvm_phase phase);
      //…
      req = smp_port#(bit)::type_id::
                           create("req", this);
      ack = smp_port#(bit)::type_id::
                           create("ack", this);
    endfunction : build_phase
```

[7] The patterns for various signal types as well as the connection automation package are all part of our migration kit [5]

[8] This package extends each uvm_component derived class with an hdl_path field. As mentioned in the AOP section, we have created a package that allows this to be done easily, which is being used here.

```
  // use them
  task run_phase(uvm_phase phase);

    req.set(1);
    ack.set(0);
    #1;
    `uvm_info("vip_comp",
     $sformatf("%s value at time %0d is %d",
      req.get_hdl_path(), $time, req.get()),
              UVM_MEDIUM)
    `uvm_info("vip_comp",
     $sformatf("%s value at time %0d is %d",
      ack.get_hdl_path(), $time, ack.get()),
              UVM_MEDIUM)
  endtask : run_phase
endclass : vip_component

// verification IP integration
class test_base extends aop_test;
  //…
  vip_env my_vip1;

  function void build_phase(
                           uvm_phase phase);
    super.build_phase(phase);

    // configure the hdl_path of
    // the verification IP instance
    uvm_config_db#(string)::
              set(this, "my_vip1",
                  "hdl_path", "verilog_top");
    uvm_config_db#(string)::
         set(this, "my_vip1.comp",
              "hdl_path", "verilog_block_i");

    // configure the relative hdl_path
    // of the signal
    uvm_config_db#(string)::
               set(this, "my_vip1.comp.req",
                  "hdl_path", "req");

    // configure its HDL and type
    uvm_config_db#(signal_hdl)::
               set(this, "my_vip1.comp.req",
                   "sig_hdl", Verilog);
    uvm_config_db#(signal_net)::
               set(this, "my_vip1.comp.req",
                  "sig_net", Verilog_reg);
  endfunction : build_phase
endclass : test_base
```

IV. SUMMARY

Understandably, migration from one language/methodology to another is not something that users look forward to. It is a hard learning process that results in reduced productivity for a significant amount of time. From a verification engineer's perspective, it is nothing but another obstacle on the way to the real goal of getting a bug-free project out. Unfortunately, from a high level view that must take long term industry support into account, it might be a necessary evil.

Our aim in this paper was not to persuade e/eRM users that migration is going to be a pleasant walk in the park. Rather, we have tried to show that despite the hard work and productivity setbacks that are always a part of it, it can also be a golden opportunity to update old conventions. By focusing on verification requirements, either those associated with specific testbench blocks, or with specific e/eRM features, we were hoping to make readers ask themselves what is it that they really need, and how much are they willing to pay for it. If they actually do so, whether or not they decide to migrate to SystemVerilog/UVM at the end, we have done our job.

V. ACKNOWLEDGMENTS

The authors of this paper would like to thank Ionel Simionescu and Mentor Graphics Verification Methodology team for reviewing and commenting on this paper. We would also like to thank Geoff Koch for helping to edit this paper.

VI. REFERENCES

[1] 2010 Wilson Research Group Functional Verification Study - SystemVerilog usage is up from 24% of the testbenches written in 2007 to 60% of testbenches written in 2010.

[2] "SystemVerilog for the Specman Engineer" – Marriott, Paul 2005

[3] "Are OVM & UVM Macros Evil? A Cost Benefit Analysis" – Erickson, Adam DVCon 2011

[4] "SystemVerilog for e Experts – Understanding the Migration process" – Janick Bergeron, 2006

[5] http://verificationacademy.com/uvm-ovm/ERM2UVM/Overview