Integrating e Verification IP in a VMM Testbench

JL Gray
Verilab
jl.gray@verilab.com

Adiel Khan
Synopsys
adiel@synopsys.com

30 March 2010

ABSTRACT

Modern testbenches often consist of components drawn from multiple languages. In many of these cases, multi-language and multi-methodology interaction is not well defined. In this paper, we will demonstrate the use of e verification components (eVCs) in a SystemVerilog/VMM testbench. Several complex issues arise when using SystemVerilog as the "primary" language. Initial simulator engine synchronization, random generation ordering, timing problems caused by program blocks, and methodology synchronization between the VMM and eRM will all be discussed.

Contents

1 Overview

Modern testbenches often consist of components drawn from multiple languages. SystemVerilog, e, OpenVera, SystemC, C, C++, Perl, and Python can be found in verification environments across the industry. Additionally, many languages have their own unique methodologies (whether open source, home grown, or vendor specific). What happens when users want to take advantage of verification IP, but the available choices were not written in the primary testbench language used for a particular project?

In many of these cases, multi-language and multi-methodology interaction is not well defined. This is certainly the case where e verification components (eVCs) are used in a SystemVerilog/VMM-based testbench. e has been around for over 14 years. Companies throughout the industry have libraries of eVCs they would like to reuse in new verification environments written in SystemVerilog and the VMM.

In this paper, we will demonstrate the use of e verification components (eVCs) in a SystemVerilog/VMM testbench. After defining a terminology mapping between the VMM and eRM, the paper will be divided into four main topics.

First, challenges to interoperability between the SV/VMM testbench and e will be discussed. These challenges include compatibility problems caused by use of program blocks in the VMM, and issues with method ports in different versions of Specman.

Next, we will review data communications strategies using method ports and user-defined adapters. Part of this discussion will center on identifying how to minimize the amount of adapter code (including SystemVerilog modules and interfaces) required to instantiate e code in a SystemVerilog testbench while maximizing configurability. We will cover a novel mechanism for customizing the **`specman.svh`** auto-generated header file to add custom VMM extensions to generated wrapper classes.

After describing the framework upon which our interoperability solution will be built, we will describe the basic proposed data flow for communicating between e and SV. We will discuss how to control component instantiation in e via configuration generated in SystemVerilog. We will then review the concept of a component registry, object wrappers, and synchronization.

Finally, we will discuss several possible communication paths between e and VMM/SV. The following paths will be addressed with an eye towards future research:

- **`vmm_channel`** to e sequencer
- **`vmm_ms_scenario`** to e sequencer
- Straight API calls into eVCs

We will also discuss synchronizing activities in e with VMM phases (both explicit and implicit).

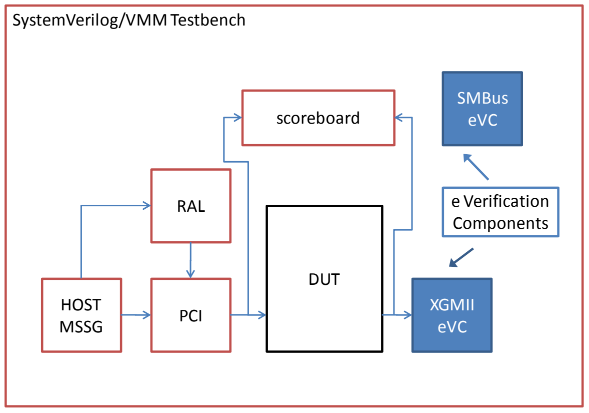


FIGURE 1: EXAMPLE TESTBENCH

1.1 Background

It is expected that readers of this document will be familiar with both e and SystemVerilog, and will have knowledge regarding general integration issues that can arise when communicating across the e-SV language boundary. Experienced e-language users will be familiar with techniques used to integrate SystemVerilog and Verilog components in testbenches where Specman is the master. When using SystemVerilog as the master, there are some key differences that must be taken into account in areas such as:

- Simulation initialization
- Environment randomization
- Testbench setup (Program block vs. module-based testbenches)
- The need to access methods in e that are part of dynamic SVTB components (as opposed to program blocks or modules)

In addition to basic language issues there are methodology issues to deal with, such as how to deal with:

- Push vs. pull-mode stimulus
- Testbench phase synchronization
- Coverage collection
- Data randomization strategies

One of the driving goals of this paper is to describe a process that can be used to integrate the SystemVerilog and e testbenches from a basic language perspective in the absence of significant tool automation support. Specifically, an attempt has been made to minimize the amount of glue logic that must be written and maintained. In some cases, that may reduce the functionality available across the interface. Another goal is that, to simplify the manual maintenance involved, extensions to testbench components will be written in the most convenient (usually the native) language. Methodology issues involving passing configuration between languages and using appropriate base classes for data items will be discussed. Stimulus, testbench phase synchronization, coverage collection, and randomization strategies will be reviewed with an eye towards future development.

1.2 Terminology

The e Reuse Methodology (eRM) and VMM use different terminology when referring to similar testbench components. The following map should help users familiar with one of these methodologies.

TABLE 1: TERMINOLOGY MAPPING

e/eRM	VMM	Notes
Agent	Subenv, Group	
Bus Functional Model (BFM)	Transactor, Driver	
Monitor	Transactor, Monitor	
Sequencer, Sequence Driver	Scenario Generator, Multi-Stream Scenario Generator	
Env	Group, Subenv	(1)
Sequence	Scenario, Multi-Stream Scenario	
e Verification Component (eVC)	VMM Group	(2)
Method	Function	
Time consuming method (TCM)	Task	

(1) The **vmm_env** class was intentionally omitted from the table because an **env** in an e testbench does not really map well to a **vmm_env**. **vmm_env** is a top-level component and cannot be reused. The e **env** most closely matches with **vmm_group** since the group can be reused at any level of the hierarchy and is implicitly phased. Implicit phasing aligns well with the "infinity-minus generation" in e.

(2) While there is no direct correlation between the eRM and VMM, the closest match is that an eVC is most like the VMM group (i.e. both are self-contained collections of verification components)

1.3 Other Useful Terminology

Some documentation refers to the primary language as the "master", and the foreign methodology as the "slave". For the purpose of this document we will use the same terminology as was used in the Accellera VIP Recommended Practices document. [1]

2 Challenges

Documentation on the interface between Specman and SystemVerilog, including configuring Specman to run with VCS, can be found by running the command **`cdnshelp`**. However, there are some issues that should be addressed before getting started.

2.1 Method Ports

e-language code naturally runs through its "generate" phase before SystemVerilog testbench code has a chance to activate. Additionally, method ports are not bound until after generation in all but very recent versions of Specman, preventing calls to method ports from being used during random generation.

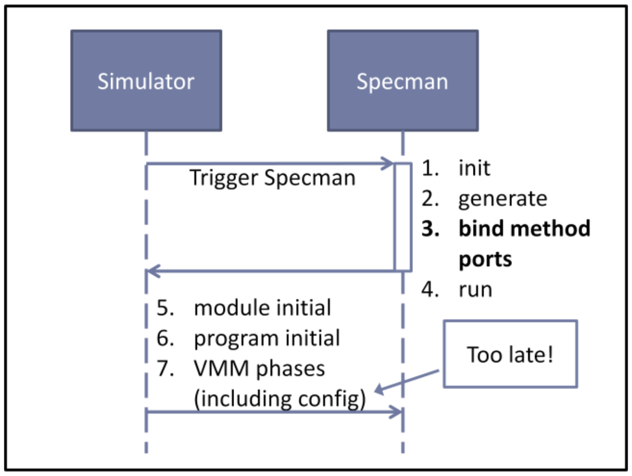


FIGURE 2: SPECMAN PHASES BEFORE VMM GEN_CFG()

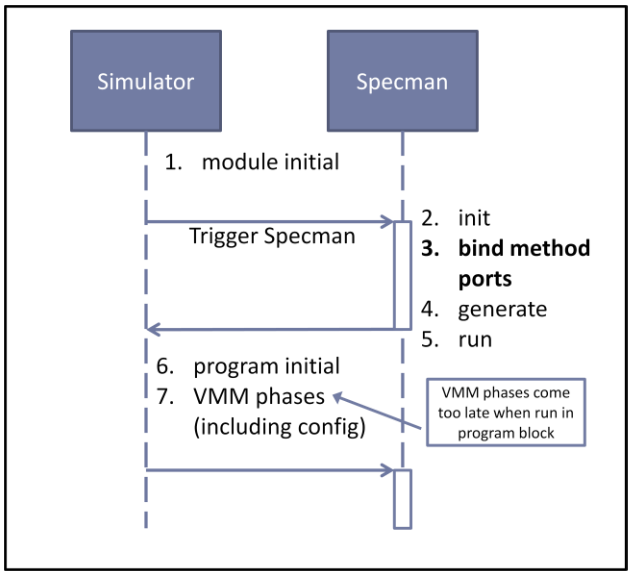


FIGURE 3: UPDATED SPECMAN, PROGRAM BLOCK-BASED VMM

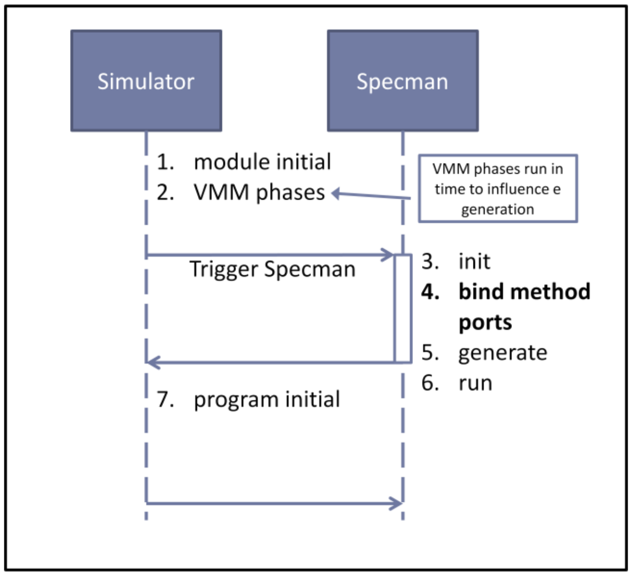


FIGURE 4: UPDATED SPECMAN, MODULE-BASED VMM

There are a couple of implications to this. In a testbench where SystemVerilog is the master, it would be ideal to control such factors as number of instance of eVCs, address ranges, etc, from random variables created in SystemVerilog. However, that information is not available during the generation phase for the reasons described above. Users must either assume all random configuration activity for eVCs will be controlled on the Specman side, or they must dump relevant configuration generated in SystemVerilog to an e-language file that can be loaded in when Specman is restarted during the build phase.

In recent versions of Specman, method ports are bound *before* the generation phase and calls to such methods can be used during randomization on the e-language side. For the rest of this document, we will assume access to a version of Specman with the relevant method-port support.

2.2 PROGRAM BLOCKS

Another potential pitfall for VMM users is related to use of program blocks to build testbenches. The VMM recommends that all testbenches be instantiated within a program block. However, Specman is optimized to work with testbenches written without the use of program blocks. Because of this mismatch, there could be unexpected timing differences between eVCs wrapped with SystemVerilog instantiated within a program block and e code effectively instantiated within a module. The module instance would behave in the expected fashion; the program block instance could have an extra clock cycle of delay as events are passed across the e-SV boundary.

Program blocks also cause another complication. In versions of Specman where it is possible to delay test generation, the call to **`test`** must occur *before* the Reactive region. That means the **`test`** command must be executed twice: once at the beginning of the simulation, and once during the VMM build phase executed from a program block. Testbenches written without the use of program blocks will not suffer this limitation.

Additionally, as of March 2010 the VCS “separate compile” flow relies on testbench components being encapsulated by a program block. Writing a testbench without program blocks will result in a loss of ability to use the “separate compile” feature.

3 COMMUNICATION

The key to successfully integrating e and SystemVerilog verification components is developing a mechanism for easily passing data back and forth between the two languages. The recommended approach is to use method ports in combination with Specman-generated and user-generated adapters.

3.1 METHOD PORTS

The most efficient way to pass data between e and SystemVerilog is to use method ports. A method port is effectively a tunnel that allows users on either side of the tunnel to call TCMs and non-TCMs. One limitation of method ports is that they must be bound to a specific module instance in SystemVerilog. That is, any SystemVerilog function or task a user would like to call from e must exist in a module instance corresponding to a SystemVerilog dynamic testbench component that exists elsewhere, as shown in Figure 5. The reverse is also true. If there are multiple instances of an e unit that contain a function to be called from SystemVerilog, each unit instance must have a unique **`hdl_path()`** as shown in Figure 8. Alternately, it is possible to reference methods in units based on a unique ID. Figure 6 demonstrates how a user can access multiple eVC instances from a single SVTB object by using the eVCs ID. Figure 7 shows a more complex scenario where there are multiple SVTB and e objects that must communicate with each other. The ID provides us with the ability to create a registry that can be used to more easily map between the e and SVTB domains.

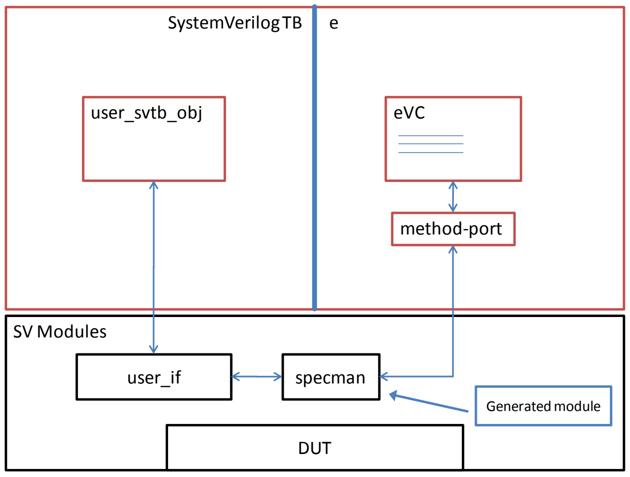


FIGURE 5: SINGLE METHOD PORT COMMUNICATION

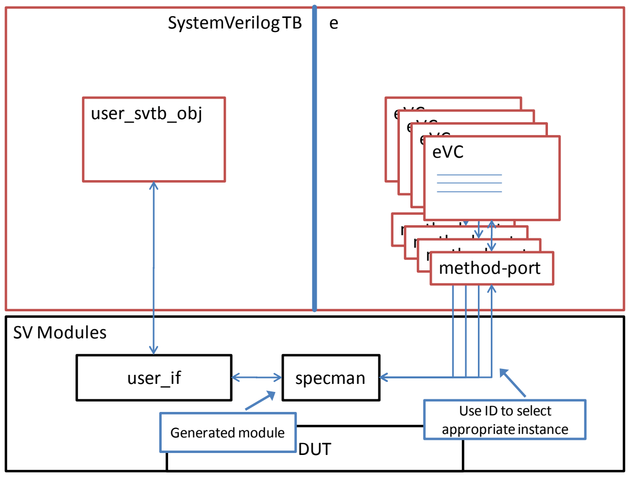


FIGURE 6: MULTIPLE METHOD PORT COMMUNICATION

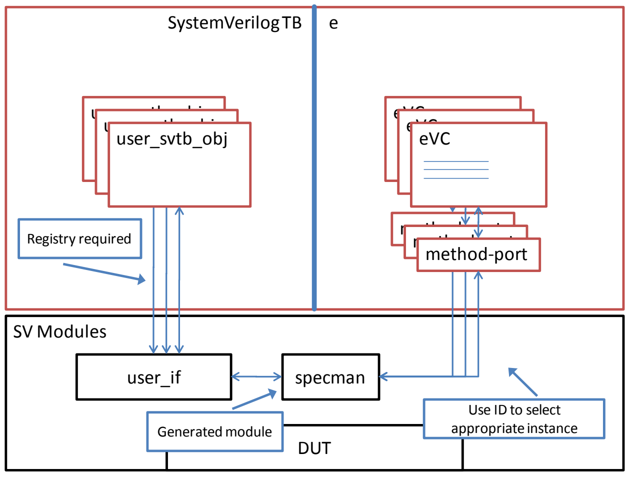


FIGURE 7: REGISTRY REQUIRED TO ACCESS MULTIPLE SVTB OBJECTS

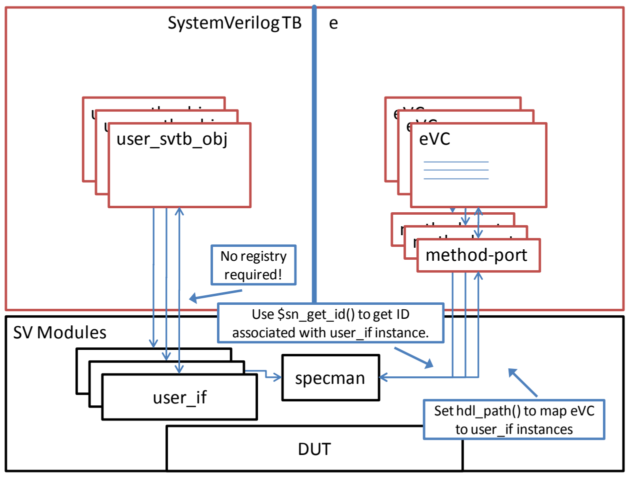

SystemVerilog TB
e
user_svtb_obj
eVC
method-port
No registry required!
Use $sn_get_id() to get ID associated with user_if instance.
SV Modules
user_if
specman
Set hdl_path() to map eVC to user_if instances
DUT

FIGURE 8: NO REGISTRY REQUIRED: 1:1 EVC TO USER_IF MAPPING

Table 2 summarizes the different types of connections possible between e and SV, and whether a registry is required.

TABLE 2: REGISTRY AND ID REQUIREMENTS BASED ON E, SV INTERFACE, AND SVTB INSTANCE COUNT

SVTB	Interfaces	e	Registry	ID
1	1	1	NO	NO
1	1	Many	NO	YES
Many	1	1	YES	NO
Many	1	Many	YES	YES
Many	Many	Many	NO	NO

The recommended solution for dealing with the situation shown in Figure 7 is as follows. For each desired connection between e and SV, create one SV interface to be used to connect the e **env** and with its corresponding SVTB **group** or **subenv**. Instantiate each interface once for each object it will correspond with. If the number of objects is unknown at compile time, create one interface that communicates with all, depending on a unique identifier. For example, a user may want to call the **write()** method of an eVC from SystemVerilog. If there are multiple instances of that eVC in the verification environment, and they are contained in a list in e:

```
// p_wr = packet write
method_type p_wr(data: enet_packet);

unit enet_env {
  p_wr: in method_port of p_wr is instance;
  keep bind(p_wr, external);
  keep p_wr.hdl_path() == “write”;

    p_wr(data: enet_packet) @clk$ is {};
};

unit my_toplevel_env {
    enet_env_list: list of enet_env is instance;
};
```

FIGURE 9: E CODE - ACCESSING A LIST OF EVCS

The associated SystemVerilog interface would look something like:

```
interface enet_env_sv2e_if ();

  // If only one instance of eVC exists id
  // is determined by call to $sn_get_id().
  task write_unique(sn_enet_packet packet);
    specman.write ($sn_get_id(), packet);
  endtask: write_unique

  // If multiple instances exist, select
  // appropriate instance with id field.
  task write(int id, sn_enet_packet packet);
    specman.write(id, packet);
  endtask: write

endinterface: enet_env_if
```

FIGURE 10: SV INTERFACE FOR E CODE ACCESS

Now, when calling the write method from instance 3 (for example) of the **enet_env**, users can say

```
my_enet_env_sv2e_if.write(3, my_pkt);
```

FIGURE 11: CALLING AN E METHOD FROM SV

3.2 Data Conversion Adapters

By default, Specman can generate adapter classes (written to the **specman.svh** file) that make it possible to transform e structs and units to SystemVerilog classes. These pre-defined classes have some limitations:

- They are not derived from any base class[1]
- Random constraints and coverage are not included
- Methods are not implemented

If the converted classes are not derived from VMM base classes, they are significantly less useful in a VMM-derived environment. When classes are derived from (for example) **vmm_data**, they can be used by other components in the SystemVerilog testbench, such as a scenario feeding a **vmm_channel**.

There are two ways to work around the limitations described above. The first mechanism is to create a user-defined adapter to convert objects from SystemVerilog to their e-based equivalent. This approach is the *only* way to convert a class defined in SystemVerilog to e. Instructions on creating an adapter can be found in the e-language documentation [2]. If a data structure exists in e and needs to be used in “SystemVerilog, an alternate approach can be used. This approach involves post-processing the **specman.svh** file dumped by Specman when generating a stubs file to force the class to be extended from the desired base object (and to use any desired shorthand macros).

3.2.1 Modifying the Stubs File

Maintaining class structures and adapters across multiple languages can be a time consuming and error-prone process. An automated mapping process significantly reduces maintenance and debug effort. Automating the process of converting e data structures to SystemVerilog can be accomplished using a combination of two techniques. First, the **sv_adapter_unit** can be modified to change the way data structures are named and dumped to the **specman.svh** file. Second, the **specman.svh** file itself can be modified via a post-processing script to add desired characteristics such as:

- User-defined base class
- Support for VMM shorthand macros
- Randomization of data members

Once these modifications have taken place, users can either take advantage of the generated class directly or create a derived object that includes additional capabilities such as customized **vmm_data** functionality and random constraints.

[1] Unless Specman is used in conjunction with the Cadence OIG tool, in which case, classes are derived from OVM base classes.

As an example, here is an e struct that would be useful to work with in SystemVerilog.

```
// A simple data structure modeling a read or
// write command, address, and data.
struct vlab_simple_data_s like
                        any_sequence_item {
    data_type: simple_data_t;

    %addr: uint(bits:32);
    %data: list of byte;

    keep soft data.size() in [1..5];

    short_print() : string is {
        // ...
    };

    do_print() is only {
        // ...
    };

};
```

FIGURE 12: VLAB_SIMPLE_DATA_S

When this data structure is used as a parameter to a method port, Specman will automatically create an equivalent SystemVerilog class in the stubs file. Struct data members will be part of the converted class definition. Methods, constraints, and functional coverage statements will not be converted.

Before modification, the specman.svh file will look similar to the following. Code that is not relevant to the conversion has not been shown. Also note the accessor methods created during the stubs generation process.

```
1   /*************************
2   package specman_types
3   *************************/
4   package specman_types;
5
6   ///////////////////////////////////////
7   //  Forward Declarations             //
8   ///////////////////////////////////////
9
10  typedef class sn_vlab_simple_data_s;
11
12  ///////////////////////////////////////
13  //    Type Definitions               //
14  ///////////////////////////////////////
15
16  // class definition for specman struct
17  // vlab_simple_data_s for use in method port
18  // with dynamic parameters' size
19  class sn_vlab_simple_data_s;
20
21     ///////////////////////////////////////
22     // Public fields
23     ///////////////////////////////////////
24     int unsigned addr;
25     byte unsigned data[];
26
27     ///////////////////////////////////////
28     // Field access methods
29     ///////////////////////////////////////
30     extern function int get_data_size();
31     extern function void set_data_size(int
          new_size, bit keep_old_values);
32     extern function byte unsigned
          get_data_elem(int index);
33     extern function void set_data_elem(int
          index, input byte unsigned val);
34     // ...
35
36  endclass : sn_vlab_simple_data_s
```

FIGURE 13: SPECMAN.SVH BEFORE PROCESSING

Look at line 19 above. The e struct **vlab_simple_data_s** has been converted to a new SystemVerilog class **sn_vlab_simple_data_s**. Unlike the original e struct, the data members of this new class are not randomized.

After post processing, the class **sn_vlab_simple_data_s** has some additional capabilities.

```
1   /*************************
2    * package specman_types
3    *************************/
4   package specman_types;
5
6   import vmm_std_lib::*;
7
8   //////////////////////////////////////
9   // Forward Declarations             //
10  //////////////////////////////////////
11
12  typedef class sn_vlab_simple_data_s;
13
14  //////////////////////////////////////
15  // Type Definitions                 //
16  //////////////////////////////////////
17
18  // class definition for specman struct
19  // vlab_simple_data_s for use in method port
20  // with dynamic parameters' size
21  class sn_vlab_simple_data_s extends vmm_data;
22     `vmm_typename(sn_vlab_simple_data_s)
23
24     //////////////////////////////////////
25     // Public fields
26     //////////////////////////////////////
27     int unsigned addr;
28     byte unsigned data[];
29
30     //////////////////////////////////////
31     // Field access methods
32     //////////////////////////////////////
33     extern function int get_data_size();
34     extern function void set_data_size(int
35        new_size,bit keep_old_values);
36     extern function byte unsigned
37        get_data_elem(int index);
38     extern function void set_data_elem(int
39        index,input byte unsigned val);
40
41  // BEGIN POST PROCESSING BY svh2vmm.py
42  // SHORTHAND MACROS
    `vmm_data_member_begin(sn_vlab_simple_data_s)
43     `vmm_data_member_scalar(addr, DO_ALL);
44     `vmm_data_member_scalar_array(data, DO_ALL);
45  `vmm_data_member_end(sn_vlab_simple_data_s)
46  // END POST PROCESSING BY svh2vmm.py
47  // SHORTHAND MACROS
48
49  endclass : sn_vlab_simple_data_s
```

FIGURE 14: POST-PROCESSED STUBS FILE

A closer look reveals that four important modifications have been made to the stubs file. First, on line 6, the VMM standard library package has been imported into the **specman_types** package. This allows access to all relevant VMM functionality from within the stubs file.

Second, on line 21 **sn_vlab_simple_data_s** now extends from **vmm_data**. This will allow the class to be used as a data item passed via channel, and provides access to copy(), compare(), pack(), unpack(), and other relevant VMM data functions using their default implementations (subject to the third and fourth additions described next).

Third, the call to **`vmm_typename(...)** has been included on line 22. And finally, the shorthand macros wrapping each of the member variables have been added on lines 41 through 47.

If users need to add random constraints, custom functions, or customized implementation of built-in VMM data functions they will need to extend from this data item and create a new item to be passed throughout the SystemVerilog testbench.

Certain characteristics of the dumped SystemVerilog class can be controlled from Specman itself using the reflection API and a customized version of the **sv_adapter_unit**. This topic is beyond the scope of this paper. However, as an introduction, the **addr** field of **vlab_simple_data_s** can be forced to be random in the stubs file via the following code.

```
unit vlab_simple_data_adapter_unit like
   sv_adapter_unit {

   // Output vlab_simple_data_s as a class
   // instead of a struct (the default).
    convert_struct_to_class(cur_struct:
       rf_struct) : bool is {
        return(TRUE);
    }; // convert_struct_...

    // Randomize the "addr" field of
    // "vlab_simple_data_s"
    randomize_field (cur_field : rf_field) :
       bool is {
       var cur_struct : rf_struct =
          cur_field.get_declaring_struct();
       if (cur_struct.get_name() ==
          "vlab_simple_data_s" and
           cur_field.get_name() == "addr" ) {
              result = TRUE;
        };
    };

};
```

FIGURE 15: CONFIGURE 'RAND' KEYWORD USING SV_ADAPTER_UNIT

The **vlab_simple_data_adapter_unit** should be instantiated somewhere within the environment as shown.

```
extend sys {
  data_adapter: vlab_simple_data_adapter_unit
     is instance;

};
```

FIGURE 16: INSTANTIATE ADAPTER UNIT UNDER SYS

The stubs file will now contain the "rand" keyword before the **addr** field in the **sn_vlab_simple_data_s** class definition.

There are a few drawbacks to this approach. First, modifying the stubs file is not recommended by Cadence as its contents are not guaranteed to remain consistent between versions of Specman. However, it is believed the changes suggested are to areas of the file unlikely to change significantly, if at all, between Specman versions.

Second, modifying the stubs file requires an additional step in the compilation flow. After the stubs file is generated, modifications must be made before the compile continues. At most, a few additional lines of code were added to the example **`Makefile`** to support post-processing. If this level of modification to the build process is not possible, modifying the stubs file may not be a practical approach.

Third, if more than one base class is needed the scripting effort involved could become a gating factor. The examples in this section assume that each class that must be instrumented should be based on **`vmm_data`**. If that is not going to be the case, additional logic would need to be scripted to determine which classes should be based on **`vmm_data`**, and which should be based on others (such as **`vmm_e_xactor`**, which will be discussed in section 4.1).

Finally, depending on the complexity of the data structures involved, it could be difficult to make the necessary additions to the stubs file without a more advanced SystemVerilog parsing capability. This could be especially true in the case where a data structure is represented by a hierarchical set of objects.

3.2.2 Creating the User-Defined Adapter

Another approach possible when using eVCs in a VMM environment is to hand-code adapter classes. Classes that represent data objects in e (for example, derived from **`any_sequence_item`**) should be derived from **`vmm_data`** on the SystemVerilog side. When creating the adapter class, users must determine whether or not they want to randomize the resulting data object in SystemVerilog or only randomize the object in e.

4 Basic Solution

It is possible to define a robust, fully-featured interconnect between e and SystemVerilog/VMM[2] with a significant amount of manual coding effort. Since one of the goals of this paper is to avoid as much manual effort as possible, a scaled down version of the interface will be defined. The framework presented here will allow users to:

- Start and configure eVCs from the primary SystemVerilog/VMM testbench
- Start and stop eVCs using a **`vmm_xactor`**-based interface
- Pass arbitrary configuration from SV to e via **`vmm_opts`**
- Call arbitrary e methods (with no parameters or return values)[3] from SystemVerilog
- Easily pass data between e and SV, as long as the data structure was originally defined in e
- Pass data between e and SV with manual effort assuming data structure was originally defined in SV

The steps to accomplish these goals are described below.

4.1 The General Flow

Figure 17 demonstrates the high level architecture of the infrastructure required to communicate between e and SystemVerilog.

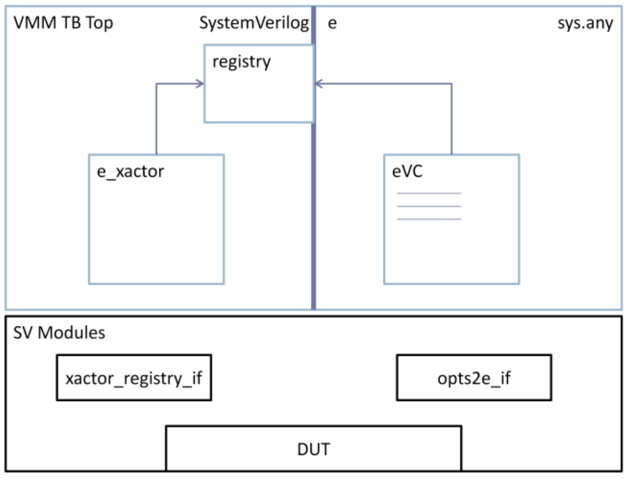


FIGURE 17: MULTI-LANGUAGE INFRASTRUCTURE

The basic execution flow for our e-SV framework is as follows:

1. VMM executes all phases up to and including **`build`**.
2. All randomized configuration relevant to the e component hierarchy will be generated in SystemVerilog.
3. During the build phase, a VMM-based wrapper for each expected e object will be built. Objects will be derived from a newly proposed class: **`vmm_e_xactor`**. Each of these objects will register itself with a central registry maintained in SystemVerilog

[2] Cadence's OIG testbench generator can facilitate e to SV/OVM when run on the IUS simulator.

[3] For this to work, parameters must be passed via the Drop Box described in section 4.5.

(**vmm_e_xactor_registry**). During registration, the following information will be passed:

   a. Pointer to the object
   b. Logical name (string) of the object
   c. Type of object (driver, monitor, generator, group, etc). Currently, the type will be used for debugging purposes only, but could be used in the future for other purposes.

4. At the end of the build phase, call the "**test**" command in e.
5. e components are instantiated and generated using constraints from SystemVerilog and the **vmm_opts** integration.
6. During the **post_generate()** step, e components add themselves to the registry of e components on the SystemVerilog side. Two important pieces of information are registered:
   a. Pointer to the e object (as a string e object name, such as **my_inst-@3**)
   b. Logical name of the component as a string. For example, **sys.my_vlab_evc.bfm**. Though the logical name could be passed back relative to SystemVerilog if that can be determined either during generation or by the user who creates the e wrapper to begin with.
7. When each e component is added to the registry, the registry looks for a matching SystemVerilog component. If none is found, an error is issued. Additional checking could be added at the end of this step to flag an error if a SystemVerilog component has no matching e component.

Wrappers between VMM/SV and e can be created to deal with the following types of interfaces:

- **vmm_xactor::start_xactor** and **vmm_xactor::stop_xactor()**
- sequence/scenario-related methods
- TLM methods
- user-defined methods[4]

On the SystemVerilog side, users can create wrappers deriving from the class **vmm_e_xactor**. On the e side, users can instrument their eVCs by extension via a set of macros:

- **enable_vmm_xactor_integration**
- **enable_vmm_scenario_integration**[5]
- **enable_vmm_tlm_integration**[5]

[4] Only **start_xactor()** and **stop_xactor()** have been implemented in example code.

For example:

```
extend vlab_simple_seq_driver_u {
    enable_vmm_xactor_integration;
};
```

Given an eVC and a VMM testbench, the following files will need to be created by the user:

- **myevc_config.e**
  - Contains extensions to enable interaction with the VMM base classes, plus hooks to pull information on unit configuration from SystemVerilog during randomization.
- **myevc_wrapper.sv**
  - Parameterized with SV-e adapter based on **vmm_data**
  - User wrapper based on **vmm_e_xactor** base class. Users must implement custom methods as needed to deal with SystemVerilog-e integration issues.
- **myevc_data_type.sv**
  - Derived from **vmm_data**
  - Data items that users want to pass between e and SV should be defined. Add fields to control which random variables should be used in e and which should be ignored.

4.2 Configuration with VMM_OPTS

vmm_opts is a VMM utility class that allows users to pass configuration values from the simulation command line or testbench source code to other portions of the environment [3]. By wrapping this class appropriately, it can be used to easily pass basic types (i.e. all types except user-defined types) of configuration data from SystemVerilog to e.

There are three ways to save configuration values to the **vmm_opts** database: call to the **set_*** API, from an external options file, or from the simulation command line. [3] Each value stored in the configuration database can be referenced via a unique string id. For example, the number of agents to be instantiated within an eVC could be randomized and stored in the VMM configuration database.

[5] Scenario and TLM integration are proposed.

```
class vlab_tb_top extends vmm_group;

  rand logic [3:0] num_agents;

  constraint e_agent_count {
      num_agents < 5;
      num_agents > 1;
  };

  // ...

  virtual function void build_ph();
      this.randomize();
      vmm_opts::set_int(
        "num_agents", num_agents);
  endfunction: build_ph

  //...
endclass: vlab_tb_top
```

FIGURE 18: POPULATING VMM_OPTS DATABASE FROM SV

With the proper integration between e and SystemVerilog, the **num_agents** parameter can be read out during the generation phase of an e component as shown below.

```
extend vlab_simple_env_u {
  // ...

  // vmm_opts option processing
  opts2e: vmm_opts2e_u;

  // Instantiate a simple agent.
  keep simple_agent_list.size() ==
            opts2e.vmm_get_int$("num_agents");

};
```

FIGURE 19: RETRIEVING VMM_OPTS VALUES FROM E

Built-in types can easily be passed across the language boundary using this approach. User-defined types do not work well without user customization as the **vmm_opts** API stores these objects as **vmm_object** types.

The **vmm_opts** integration is effectively a special case of the drop box approach described in section 4.5.

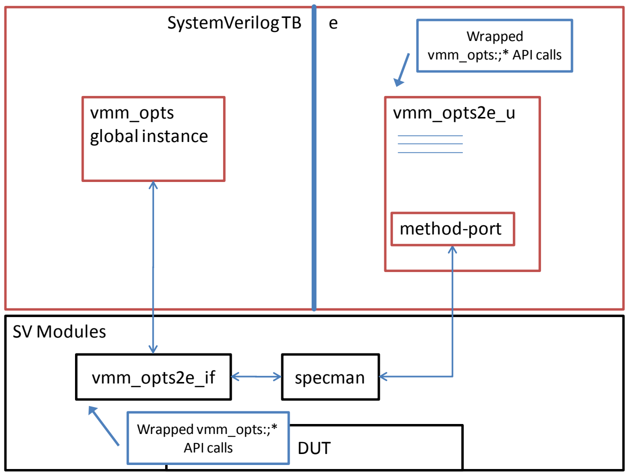


FIGURE 20: VMM_OPTS -> E HOOKUP

As shown in Figure 20, a SystemVerilog interface, vmm_opts2e_if, is created to facilitate calls from e components. A call to each of the functions of **vmm_opts** is made from an equivalently named function in the interface. The interface is instantiated once, and is paired with a single instance of an e unit called **vmm_opts2e_u**. **vmm_opts2e_u** contains method ports corresponding to each of the wrapper functions in **vmm_opts2e_if**.

Any e component with a pointer to the vmm_opts2e_u instance can query the configuration registry in SystemVerilog. In order to use the configuration registry during generation, any required values must be written to the registry before or during the build phase **build_ph** in the VMM.

4.3 REGISTRY

e method calls must be associated with a specific **hdl_path()**. If there are multiple eVC instances, a custom ID must be used to call the methods in each one. Maintaining a mapping of IDs to instances can be tricky as the numbers and types of eVCs and instances scale. Automated code generation of the mapping could help alleviate the issue, as could adding the ability to map e method calls to SVTB dynamic calls to Specman. This type of support does not currently exist between Specman and VCS. Therefore, an alternate strategy is proposed.

A registry can be used to facilitate communication between e and SystemVerilog. To simplify coding and maintenance, our registry relies on two unique aspects of the e language to simplify mapping. Usually, pointers to objects in another language cannot be stored. However, in e it is possible to save a string that serves as a pointer to a unique instance of an object. Pointers are strings in the form of **my_vlab_class-@3** which means the **@3** instance of **my_vlab_class**. Also, it is possible in

Specman to call commands using the **specman()** method. For example:

```
// object_from_sv has been converted from SV
// to e by the Specman adapter and
// placed in the relevant drop box.
specman(
  “my_class-@3.send_data(object_from_sv)”);
```

FIGURE 21: USING 'SPECMAN()' TO EXECUTE ARBITRARY E CODE

As described in section 4.1 above, e objects that must be able to communicate with SystemVerilog will register with the central SystemVerilog registry during generation. During the registration process, the string reference to the e object will be stored in the registry. The **enable_vmm_xactor_integration** macro defined below automatically registers instrumented eVCs.

```
define
  <enable_vmm_xactor_integration'struct_member>
  "enable_vmm_xactor_integration" as {

  opts2e: vmm_opts2e_u;
  keep opts2e == sys.vmm_opts2e;

  xactor_registry: vmm_e_xactor_registry_u;
    keep xactor_registry ==
         sys.vmm_e_xactor_registry;

  // Functionality to start and stop this
  // eVC. Must extend and implement
  // this method.
  start_e_xactor() is empty;
  stop_e_xactor() is empty;

  // ...

  post_generate() is also {
    var return_val: bit;
    // Add this instance to the SystemVerilog
    // object registry.
     vmm_e_xactor_registry.\
       register_vmm_e_xactor$(
       short_name_path(), appendf("%s", me));
  };
};
```

FIGURE 22: ENABLE_VMM_XACTOR_INTEGRATION MACRO DEFINITION

The registry itself has two key functions.

```
function bit register_sv_xactor(
          string name, vmm_e_xactor x);
function void register_vmm_e_xactor(
          string name, string var_name);
```

FIGURE 23: VMM_XACTOR INTEGRATION FUNCTIONS

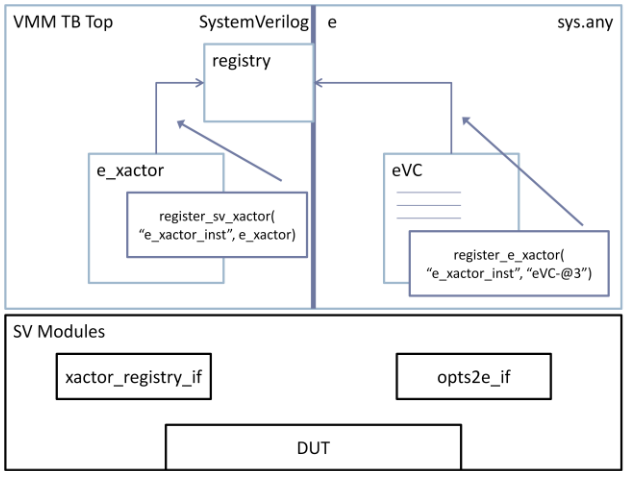

VMM TB Top
SystemVerilog
e
sys.any
registry
e_xactor
register_sv_xactor(“e_xactor_inst”, e_xactor)
eVC
register_e_xactor(“e_xactor_inst”, “eVC-@3”)
SV Modules
xactor_registry_if
opts2e_if
DUT

FIGURE 24: REGISTRY API

When registering the e component and its wrapper, the current mechanism to ensure they are matched in the registry is to use the same name for each transactor instance. One way to deal with names is to use the **short_name_path()** of the e unit being registered, and to use the same name for the VMM transactor wrapper object as well.

4.4 SYNCHRONIZATION

vmm_e_xactor objects will be stopped and started via the regular VMM **start_xactor()** and **stop_xactor()** methods. This will provide a first level of synchronization between the two language domains. eVCs will be “stopped” by default, and must be activated initially via the **start_xactor()** command. There are a variety of ways users could modify e components to cause them to start and stop as desired. One example involves starting and stopping a sequence driver.

Sequence drivers in e are responsible for sending traffic generated by sequences to the BFM. Each sequence driver has a default sequence that runs (or not) based on some user-defined scheme. Usually this scheme involves setting the count in the default sequence to 0 telling it not to generate any new sequences. To accomplish something more complex, users must come up with a strategy on their own. VMM transactors based on **vmm_xactor**, on the other hand, have a convenient mechanism for controlling when traffic generators should start or stop – the **vmm_xactor::start_xactor()** and **vmm_xactor::stop_xactor()** functions. When using the VMM together with e, the VMM scheme can be used to our advantage to easily control when sequence drivers are activated to send traffic to the driver, or when the BFM itself should even be requesting new transactions from the driver in the first place. This mechanism could be used with just about any e environment component, enabling the component to be turned off or on as needed. One important point to mention is that sequence drivers should not be disabled if they are expected to receive requests from virtual sequences in other parts of the testbench.

Using the registry described in section 4.3 and the **vmm_e_xactor** base class, any call to **vmm_e_xactor::start_xactor()** or **stop_xactor()** will call the **start_e_xactor()** and **stop_e_xactor()** methods in the correspondingly instrumented e units. Users must implement **start_e_xactor()** and **stop_e_xactor()** in any units that will take advantage of this feature. In this example, the approach used for starting and stopping the sequence driver is first to set a flag letting the driver know whether it should plan to start or stop at the next opportunity. Instead of stopping the sequence driver itself, we will stop the BFM. A more complex scheme could be used instead depending on the requirements of the interface.

```
extend vlab_simple_seq_driver_u {

  // Keep a pointer to the BFM so we
  // can turn it on and off. It will no longer
  // pull transactions from the driver.
  !bfm: vlab_simple_bfm_u;

  start_e_xactor() is also {
    bfm.is_enabled = TRUE;
  };

  stop_e_xactor() is also {
    bfm.is_enabled = FALSE;
  };

};
```

FIGURE 25: START/STOP E XACTOR IMPLEMENTATION

The BFM has been coded to respond on per-transaction boundaries to the state of the **is_enabled** flag.

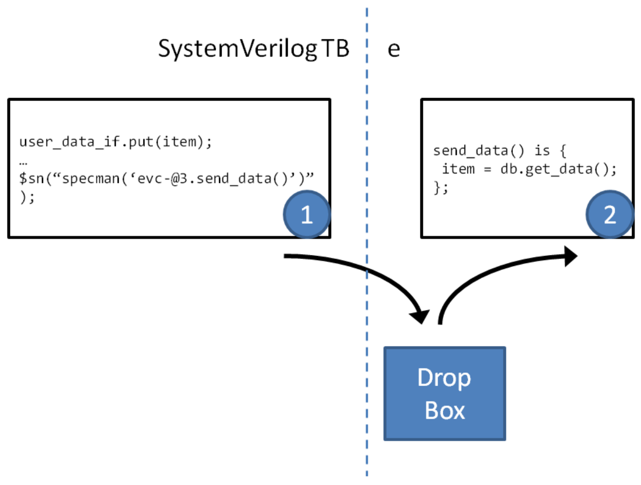
```
1   // Pull item from driver, process it, then
2   // inform using item_done
3   extend vlab_simple_bfm_u {
4     driver: vlab_simple_seq_driver_u;
5
6     // Flag to be set by vmm_e_xactor start/stop
7     // is_enabled: bool;
8     keep soft is_enabled == FALSE;
9
10    execute_items() @clk$ is {
11      var seq_item: vlab_simple_data_s;
12      while TRUE {
13
14        // Don't start executing items unless
15        // this driver is enabled.
16        wait true(is_enabled);
17
18        seq_item = driver.get_next_item();
19        drive_simple_data(seq_item);
20
21        emit driver.item_done;
22      };
23    };
24
25    run() is also {
26      start execute_items();
27    };
28
29  };
```

FIGURE 26: ADDING XACTOR CAPABILITIES TO E DRIVER

As can be seen on line 16, the driver will only attempt to get another item if **is_enabled** is TRUE. It will not check **is_enabled** again until after it has completed the current transaction.

4.5 DROP BOX

The registry described in section 4.3 provides a mechanism for dealing with many instances of an object containing methods users would like to call. However, each time a method is added to the object it must also be added to a SystemVerilog interface and a method port must be created in e. The drop box provides a means to allow new e methods to be created and called without the need for a new interface or method port to be created.



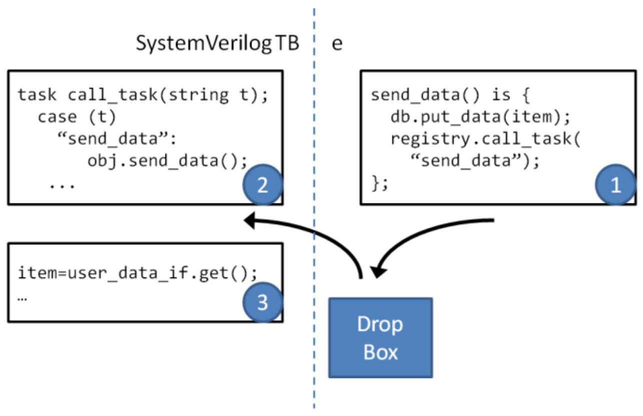
```
user_data_if.put(item);
…
$sn("specman('evc-@3.send_data()')"
);
```

```
send_data() is {
 item = db.get_data();
};
```

FIGURE 27: SV TO E DROP BOX

As shown in Figure 27, a SystemVerilog method places data in the drop box by means of an intermediary interface instance. It then calls the appropriate method in e via the **$sn/specman()** commands. The e method (in this case, **send_data()**) immediately gets the relevant data item from the drop box. The call to **get_data()** must take place before any time is consumed in the method to prevent other methods using the drop box from overwriting the data item. A new drop box must be created for each data type to be passed between SystemVerilog and e.

It is also possible to use the drop box to send data from e to SystemVerilog.



```
task call_task(string t);
  case (t)
    "send_data":
       obj.send_data();
   ...
```

```
send_data() is {
  db.put_data(item);
  registry.call_task(
    "send_data");
};
```

```
item=user_data_if.get();
…
```

FIGURE 28: E TO SV DROP BOX

Figure 28 shows the steps required to call a generic method in SystemVerilog from e using the drop box to pass a data item. In order for this approach to work users must implement a lookup table in SystemVerilog mapping strings to actual method calls. One way to accomplish this task would be to take advantage of the VMM Callback mechanism. Users could add callbacks that followed the following procedure:

1. Check to see if current string matches
2. If yes, call method

If the details of the interface between e and SystemVerilog are known in advance and little change is expected, the more straightforward approach is to create relevant method ports directly to pass required data via method parameters.

5 Future Work

Once the basics of the SystemVerilog interface have been established, users will immediately hit a number of issues critical to building a successful testbench. The most pressing of these will be in the areas of randomization, coverage collection, and stimulus generation.

5.1 Randomization and Coverage

When using eVCs in a SystemVerilog testbench, it is natural to want to control randomization and collection of functional coverage as much as possible in the testbench's native language (SystemVerilog). However, since it is not possible to pass actual data structures across the language interface, but instead only copies, several limitations quickly become evident. How many constraints from the original e struct need to be rewritten to make randomization effective in SystemVerilog? The same question applies to Functional Coverage as well. What strategy should be used to communicate which fields have already been randomized in SystemVerilog, and which should still be randomized in e?

If randomization in SystemVerilog is desired, classes could be constructed with control fields letting Specman know which fields were randomized by e, and which were randomized already in SystemVerilog.

```
class packet extends vmm_data;
   rand reg[47:0] dst_addr;
   // If 1, randomize dst_addr in e. If 0, it
   // has already been set in SystemVerilog
   bit randomize_dst_addr;

   constraint dst_addr {
      ...
   };
endclass:packet
```

FIGURE 29: RANDOMIZATION BETWEEN E AND SV

5.2 Stimulus

One of the key capabilities in both the VMM and eRM is the ability to generate stimulus using sequences (eRM) and scenarios (VMM). At a high level, the two methodologies behave similarly. At a lower level, there are differences that come into play that affect the way the libraries interact. The main differences come in the usage model – push (VMM) vs. pull (eRM). See [4] for more detailed description of the issue.

There are a number of possible use models that will need to be examined in order to identify a complete solution for integrating eVCs into a VMM environment, such as:

- Multi-Stream Scenarios (MSS) calling Sequences
- MSS executing sequence items
- Scenario generators passing data through a channel to a sequencer
- Adding new “sequences” that are actually written as SystemVerilog scenarios

Each of the above use cases is potentially valuable. However, several of them require a significant amount of manual coding and maintenance to enable them to scale properly in a large verification environment without some sort of automated solution. A tradeoff would likely be made to sacrifice some flexibility and the ability to write all future stimuli in SystemVerilog in order to make the final solution easier to maintain.

Often engineers need to take advantage of verification IP written in languages and methodologies other than the primary ones used in their testbench. Users with eVC libraries can take advantage of these existing components within a SystemVerilog/VMM framework by following the suggestions outlined in this paper. Specifically, techniques used to deal with method ports, a description of the process required to bring up the e and SystemVerilog simulations in the correct order, and the methodology to be used to deal with passing data across several permutations of standard VMM interfaces have been addressed.

6 Works Cited

[1] Accellera. (2009, August) Verification Intellectual Property (VIP) Recommended Practices v1.0. [Online]. http://www.accellera.org/activities/vip/VIP_1.0.pdf

[2] Cadence Design Systems. (2009) Specman Integrators Guide.

[3] Synopsys, Inc. (2009) VMM 1.2 User Guide. [Online]. http://vmmcentral.org/onlinedoc/wwhelp/wwhimpl/js/html/wwhelp.htm

[4] JL Gray and Scott Roland. (2010, February) Stimulating Scenarios in the OVM and VMM.

[5] Cadence Design Systems. (2009) Specman e Language Reference.

[6] Cadence Design Systems. (2009) e Reuse Methodology (eRM) Developer Manual.