Creating Stimulus and Stimulating Creativity:
Using the VMM Scenario Generator

Jonathan Bromley

Doulos Ltd, Ringwood, England

jonathan.bromley@doulos.com

ABSTRACT

The Verification Methodology Manual for SystemVerilog (VMM) standard library provides a scenario generator that can be built automatically to match any user-defined stimulus data class. This ready-to-run generator can then be customized to provide structured streams of randomized stimulus for a verification environment.

Scenarios provide a natural and convenient way to impose appropriate and realistic structure on a stream of data items that would otherwise be independent of one another. Communications protocols, instruction streams for CPUs, and the generation of interrupts by peripheral devices all represent examples of stimulus in which truly random generation would be very unlikely to exercise important aspects of system functionality. Scenarios can impose relationships between data items (or increase the likelihood of certain relationships arising) while preserving the well-known benefits of stimulus randomization.

This paper presents an introduction to simple customization of the VMM scenario generator, and then reports experience with the management of more complex stimulus including hierarchical scenarios and the co-ordination of multiple parallel scenario streams. It will also compare the tradeoffs between procedurally-generated scenarios and purely declarative, constraint-driven scenarios, and examine the potential of well-designed scenario generation to enhance effective re-use of verification components.

Table of Contents

Table of Figures

1. Introduction

1.1 Organization of this paper

This introduction briefly reviews the motivation for structured random stimulus generation in published verification methodologies, and outlines the solutions offered by the Verification Methodology Manual for SystemVerilog (VMM).

Section 2 introduces a rather straightforward verification problem that nevertheless requires structured stimulus. This example will be used throughout the rest of the paper.

Section 3 describes the specific mechanisms provided in VMM for creating structured stimulus (scenarios). This section is written in a tutorial style, as it is the author's experience that many VMM users have chosen not to adopt scenarios in their own verification environments.

Section 4 describes an alternative approach to using the existing VMM scenario generator, creating stimulus procedurally rather than by randomization. Although this approach is insufficiently flexible for most realistic situations, it offers some hints for how to generate more elaborate hierarchical scenarios.

Sections 5 and 6 describe in detail the author's prototype extensions of the VMM scenario generator to create hierarchically structured scenarios. Section 7 sketches how the modified generator could facilitate the coordination of stimulus across multiple data streams.

Finally, section 8 summarises the results and indicates some further work that is required to make the extensions easily useful in mainstream verification problems.

1.2 Structured verification methodologies

Functional verification of any large digital design is, notoriously, a challenging problem that can consume unacceptably large amounts of effort. Clearly, the ability to use field-proven, reliable, well-understood blocks of verification software across multiple projects is an essential part of good verification practice. Such re-use is impracticable, however, unless a consistent framework is applied not only to the various components of a given testbench but also across multiple projects. Such frameworks are commonly known as *verification methodologies* and a number of such approaches have been widely publicized and are in common use [2,3,4,5,6].

Inevitably, the coding and architecture guidelines that go to make up a verification methodology are strongly influenced by the implementation vehicle – the programming language, and to a lesser extent the simulator or other verification tools – that are used to create and execute a verification environment. Of published methodologies making use of the SystemVerilog language [1], that defined in [2] is among the most popular and is the longest established. In the remainder of this paper we will refer to it simply as *VMM*.

1.3 VMM outline

A verification environment conforming to VMM is built using object-oriented programming techniques in SystemVerilog, and makes use of six key kinds of object:

- *transactions*, objects each representing a block of related data that will be passed around the testbench and ultimately used as stimulus or response data for the device under test (DUT);
- *transactors*, which generate, manipulate and transform transactions;
- *channels*, FIFO queues (with a few additional features) providing a means for a transactor to pass transactions to another transactor without being aware of its details;
- *callback classes*, which offer a means to extend freely the functionality of a transactor by having it invoke user-defined functions at well-specified points in its execution;
- *services*, extensible pre-defined environment-wide functionality that deals with a range of utility operations – notably the generation of messages to output files, and signaling from one part of the environment to another;
- *environment*, a top-level container within which the structure of the testbench will be created and will operate.

Among these kinds of object, transactions have a special position. They represent data rather than environment structure, and are likely to be created and used in very large numbers during the life of any given test. By contrast, the other kinds of object form parts of the test environment, and are likely to be constructed at the outset; they will rarely, if ever, be created part-way through the life of the test.

The VMM Standard Library, supplied with the VCS^{TM} simulator, provides a library of base classes that implement much of the core functionality of these objects. It also offers macros and code-generator tools that simplify the developer's task of creating VMM-compliant components.

1.4 Atomic stimulus generation

VMM transactions are represented by objects of class type. In SystemVerilog, such objects can have data members that are declared with the modifier `rand` so that, when the object's built-in `randomize()` method is invoked, all its `rand` data members are given new, random values. Furthermore, relationships among these random values (and relationships with the values of other, non-randomized data items) can be controlled using *constraints.*

As a consequence of this randomization behavior, it is rather straightforward to build a stimulus generator that creates a stream of transactions with randomized values. It is merely necessary to provide a transactor that has an instance of the transaction class, known as a *factory* instance. The transactor invokes the `randomize()` method of this factory instance, and then creates a clone of the newly-randomized factory object by calling the latter's `copy()` method (which all VMM-compliant transaction classes are required to implement). The clone object may then be sent through a channel to another transactor for use as stimulus, leaving the factory instance ready to be randomized all over again to provide the next stimulus data item. The library macro `` `vmm_atomic_gen `` offers a painless way to create a stimulus generation transactor that has this behavior (and a few additional features to make it easier to integrate into an environment).

1.5 The need for structured random stimulus generation

The randomized atomic transaction generator outlined in the previous section is straightforward, and its creation is fully automatic from any VMM-compliant transaction class. Because such a generator can be created so quickly and easily, it is an ideal tool to create a "bring-up" test giving confidence that the test environment has been correctly constructed and is successfully pushing transaction data through the DUT. However, in most cases it cannot create the stimulus that is needed to exercise specific DUT functionality as required by a verification plan. Randomization constraints can express and enforce structure within a single transaction, but the factory pattern described above cannot impose relationships between successive transactions in the stimulus stream.

2. An example design that needs scenarios for effective verification

2.1 Example protocol and DUT

The APB bus structure promoted and standardized by ARM Ltd [7] is a very simple example of a synchronous, parallel bus protocol suitable for connecting simple peripheral devices to a bus master. It is intentionally very straightforward; its functionality is restricted to data read and write cycles performed by the master and responded-to by a peripheral device. Early versions of APB had completely fixed bus cycle timing, with each read or write cycle occupying exactly two cycles of the bus clock. However, the more recent APB3 specification adds a *READY* signal to the bus, allowing peripherals to pause the bus cycle.

This facility to introduce wait states in a bus cycle makes it possible to design an APB bus arbiter, allowing more than one master to compete for access to one or more peripherals. The author is fully aware that this is an abuse of the APB specification's intent, since it was always designed as a single-master protocol. However, it allows for the creation of an example DUT that has a rather simple specification, but nevertheless exhibits non-trivial behavior and needs structured stimulus to verify it effectively.

Figure 1 shows the example DUT in context. It is a bus-matrix module for APB, and supports two masters and two peripherals. Each master can generate cycles on its own local APB bus. The DUT arbitrates between these two masters, inserting wait states as necessary to hold off a master's access until the other master has finished. In all other respects, bus cycles from each master are forwarded unaltered to the appropriate peripheral, chosen according to the value of the most significant bit of the 16-bit address. If the peripheral introduces wait states by holding its `PREADY` output low, the active master will see these additional wait states.

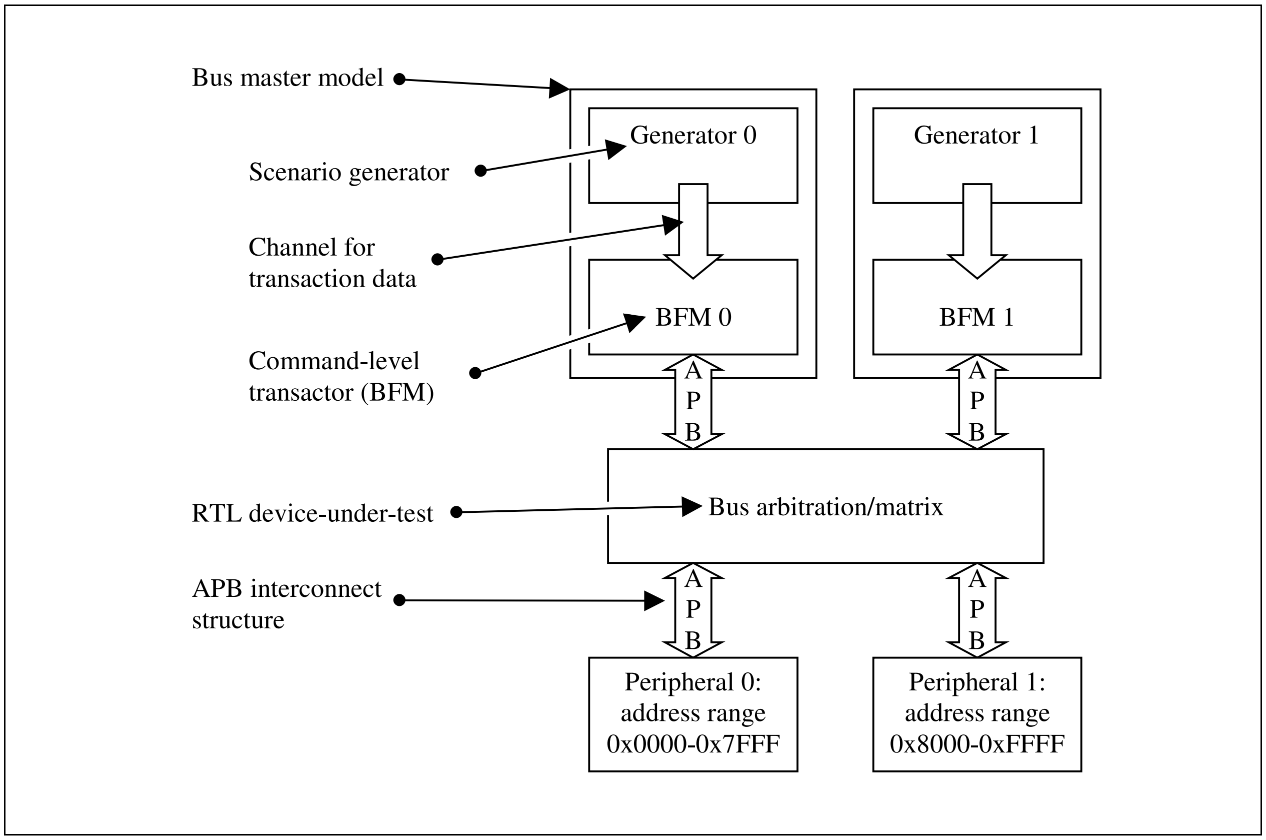
Bus master model
Scenario generator
Channel for transaction data
Command-level transactor (BFM)
RTL device-under-test
APB interconnect structure
Generator 0
Generator 1
BFM 0
BFM 1
APB
APB
Bus arbitration/matrix
APB
APB
Peripheral 0: address range 0x0000-0x7FFF
Peripheral 1: address range 0x8000-0xFFFF

Figure 1: DUT and its verification environment

2.2 Verification of the DUT

To verify the DUT in its simple form as described above, it is merely necessary to simulate random traffic from the two masters using the setup illustrated in Figure 1.

A user-defined transaction data class `apb_data`, derived from the VMM base class `vmm_data`, is created to model an APB bus cycle. Note that we model three different kinds of cycle, identified by the enumeration values `READ`, `WRITE` and `IDLE`.

Each master is modeled as a protocol stack consisting of two VMM transactors.

At the lowest level there is a bus-functional model (known as a *command level transactor* in VMM) that accepts `apb_data` objects on its input channel and performs appropriate manipulations on the physical APB bus signals. The design of this transactor is straightforward and is not discussed further here.

At the higher level we create a *generator* object whose purpose is to create a stream of randomized `apb_data` objects and deliver them to its output channel. Because the two generators are distinct instances of objects of class type, constructed sequentially in the environment class's `build` method, SystemVerilog creates a different starting value for the random number generator seed in each instance. Consequently, the two identical instances will deliver different randomized transaction streams without any special programming effort.

2.3 Additional DUT features requiring scenario stimulus for verification

The simple bus arbiter described above can be verified quite effectively with only random transactions as stimulus. However, the DUT as implemented has further functionality that effectively demands the use of structured scenarios for generation.

The arbiter DUT recognizes two special kinds of activity by a master: *read-modify-write* and *burst*. A read-modify-write is defined as two bus cycles by the same master, with no intervening idle states, to the same address, with the first cycle being a read and the second being a write. A burst is defined as an arbitrary number of bus cycles by the same master, with no intervening idle states, all of which have the same direction (read or write) and whose addresses are strictly incrementing. The arbiter treats both these special activities as non-interruptible: once such a cycle is in progress, the arbiter will not allow the other master to gain control until the activity has completed. Completion of the activity is indicated either by an idle state after its last cycle, or by the start of a bus cycle that does not conform to the specified pattern.

It is very unlikely for cycles of these new kinds to be generated by purely random streams of `apb_data` transactions. We must direct the stimulus generators to create such transactions with a significant probability, but at the same time we wish to retain all the usual benefits of randomized stimulus generation. The scenario generator is designed to meet such goals.

3. Scenarios in VMM

This section presents a brief overview of the facilities and structure of the VMM scenario generator, and then shows an example of its use applied to the bus arbitration device outlined in section 2. A general description of scenarios, with some useful strategies for their design and a thorough primer on how to create VMM scenarios, can be found in [8].

3.1 The VMM Scenario Generator

Consistent with `` `vmm_atomic_gen ``, VMM provides a somewhat more complicated macro `` `vmm_scenario_gen `` to automate the creation of a scenario generator specialized for any user-defined transaction class. This macro is invoked with the user-defined transaction class (invariably derived from `vmm_data`) specified as its first argument. In the discussion below, we assume that this user-defined transaction class is named *UserData*. The macro invocation then writes five new class definitions:

- a *generator* class *UserData*`_scenario_gen`
- a *scenario* class *UserData*`_scenario`
- a *scenario chooser* class *UserData*`_scenario_election`
- an *atomic scenario* class *UserData*`_atomic_scenario`, which is derived from *UserData*`_scenario`
- a *callback façade* class *UserData*`_scenario_gen_callbacks`

The callback façade will not be considered further in this paper; it plays the same role as the callback façade class for any generator class, discussed in detail in reference [2].

The new classes created by this macro are intended to work together to create stimulus with user-imposed relationships among the various data items that form the stimulus stream. The user is expected to create an instance (or perhaps several instances) of the generator class in their verification environment. This generator will then automatically create a stream of scenario objects, each of which is composed of an array of transaction objects. As usual with generator objects in VMM, the resultant stream of transaction objects is fed to a channel. This channel conveys the objects in FIFO fashion to another transactor, which may then deliver them to the device-under-test (DUT) or perhaps process them in some other way. Such channel-based communication between transactors is a standard feature of any VMM-based testbench and we do not discuss it further here.

It is noteworthy that, in this arrangement, scenario objects are instanced, created and managed entirely within the scenario generator object. The scenario generator's output takes the form of a stream of transaction objects whose number and contents have been controlled by the scenarios that created them. Scenarios themselves do not normally appear outside the confines of the generator object.

This encapsulation of scenarios within the generator object has some important benefits. It greatly simplifies the substitution of one generator for another, since all generators have as their output a simple stream of transaction objects. It also allows the rest of the testbench to remain ignorant of the scenarios, handling only transaction objects.

However, encapsulation of scenarios within the generator inevitably causes loss of visibility of scenario-related information elsewhere in the testbench. For diagnostics and coverage analysis, it may be helpful to know what high-level sequence of activity – what scenario – gave rise to any specific transaction object. VMM provides at least two ways to keep track of scenario-related meta-data. First, each transaction object in VMM has a data member `scenario_id` that indicates which scenario instance contained it. Secondly, the scenario generator's callbacks can be used to publish scenario-related information to subscribers elsewhere in the testbench, typically at the time of initiation or completion of each scenario.

3.2 Simple scenarios in VMM

Figure 2 shows how the read-modify-write and burst scenarios could be implemented using VMM's scenario generator.

```
`vmm_scenario_gen(apb_data,"APB scenarios")

class apb_RMW_burst_scenario extends apb_data_scenario;

// _____________________________________________________ APB_RMW ___

  int unsigned APB_RMW;

  constraint RMW {
    if (scenario_kind == APB_RMW) {
      length == 2;
      items[0].kind == apb_data::READ;
      items[1].kind == apb_data::WRITE;
      items[0].address == items[1].address;
      repeated == 0;
  }

// ___________________________________________________ APB_BURST ___

  int unsigned APB_BURST;

  constraint burst {
    if (scenario_kind == APB_BURST) {
      repeated == 0;
      length inside { 2, 4, 8, 16 };
      foreach (items[i]) {
        if (i > 0) {
          // Successive items have incrementing addresses
          items[i].address == items[i-1].address + 1;
          // All items have the same kind (read or write)
          items[i].kind    == items[i-1].kind;
        } else {
          // Special case for element [0]:
          // Only the least significant address bits can change
          (items[i].address % length) == 0;
          // Burst must be all-READ or all-WRITE
          items[i].kind inside { apb_data::READ, apb_data::WRITE } ;
        }
      }
    }
  }

// ______________________________________________________ common ___

  function new();
    super.new();
    APB_RMW = define_scenario("APB read-modify-write", 2);
    APB_BURST = define_scenario("APB burst", 16);
  endfunction : new

endclass : apb_RMW_burst_scenario
```

Figure 2: Read-modify-write and burst scenarios in VMM

The first line of code invokes the VMM scenario generator macro, automatically constructing the classes listed in Figure 3 – as discussed in section 3.1. Note that, in practice, users will not need to write the macro invocation at this point in the code, because it is generally included as part of the source file that defines the original transaction class (`apb_data` in this case).

```
apb_data_scenario
apb_data_scenario_election
apb_data_scenario_gen
apb_data_scenario_gen_callbacks
apb_data_atomic_scenario
```

Figure 3: Classes created by macro invocation `` `vmm_scenario_gen(apb_data) ``

Both generators in the testbench (see Figure 1) will be instances of `apb_data_scenario_gen`; it is the environment designer's responsibility to construct, connect and activate those instances appropriately. The remainder of the code shown in Figure 2 defines a new class `apb_RMW_burst_scenario`, inherited from class `apb_data_scenario` which was automatically generated by the macro. The extensions required to create the new scenarios are rather straightforward.

Considering first the read-modify-write scenario, a set of constraints is written to control randomization of the two transactions that together make up the scenario. The two transactions, both of type `apb_data`, are held in the scenario's predefined `items[]` array (which is implemented as a SystemVerilog queue). The number of transactions in the scenario is specified by the predefined `length` property, which is randomized along with the `items[]` array. Straightforward constraints on elements of `items[]` enforce the structure required for a read-modify-write. The predefined property `repeated` is constrained to zero; if it were not, the VMM scenario generator would run the scenario more than once each time it is created.

The set of constraints that give the read-modify-write scenario its structure is made conditional on the scenario's `scenario_kind` property being equal to the value of the user-defined variable `APB_RMW`. As can be seen in the constructor function `new`, the scenario's constructor registers the scenario kind by placing a call to `define_scenario`, a method of the scenario base class. For each new kind of scenario defined in such a derived class, a suitably named variable must be created and given its value by calling `define_scenario` in the constructor. These variables then act as symbolic names for the various different scenarios implemented by the class. When the scenario is randomized by predefined code within the generator object, the `kind` property will be given a random value selected from among the defined scenario kinds' identifying integers. This value can then be used in conditional constraints, so that the only constraints applied are those pertinent to the chosen scenario kind. This technique is used in our example to add a second scenario kind, `APB_BURST`. The two arguments to `define_scenario` are a human-readable scenario name string for use in debug messages, and the maximum possible number of transactions in the scenario.

The `APB_BURST` scenario is a little more difficult to constrain, because its length is not fixed. We constrain the burst length to be a small power of 2, using an `inside` (set-membership) constraint. The remainder of the constraint is written in an inductive style using a `foreach` construct, with a special case for the first index 0. An inductive constraint is generally to be preferred, in situations such as this where the constraint also controls the array length, because it is less likely to lead to insoluble constraints caused by cyclic relationships between the `items[]` array size and constraints on individual elements of that array.

3.3 Making use of the new scenarios

Although we have now created two new kinds of scenario, we have not yet deployed them into our verification environment. This is easily achieved by adding an instance of the derived scenario class `apb_RMW_burst_scenario` to the generator object's `scenario_set[]` array. Predefined code in the generator, executed each time the generator creates a scenario, will:

- choose one element of the generator's `scenario_set[]` array;
- randomize that element;
- deliver each transaction in the newly randomized scenario's `items[]` to its output channel.

We have already seen how randomization of the scenario (second step) can be customized by creating an extension of the scenario class. The other two steps can also be customized:

- The generator chooses a scenario from the `scenario_set[]` array by randomizing its `select_scenario` object. The resultant value of that object's `select` property is then used as an index into the `scenario_set[]`. Users are free to replace the predefined `select_scenario` instance with a customized derived-class instance. However, it is often sufficient to use the built-in default instance. In most cases, however, it is appropriate to disable its `round_robin` constraint to give random rather than cyclic choice of scenario.
- The generator delivers the scenario's `items[]` array to its output channel by invoking the scenario's `apply` method. This virtual method, predefined in the scenario base class, by default simply sends a copy of each element in turn to the generator's output channel. In the example of Figure 2, where control of the scenario's contents is achieved entirely through constraints, the default `apply` method is sufficient. However, as discussed in section 4, it can be overridden to create scenarios that are partly or entirely defined by procedural code rather than by constraints.

By default, the generator's `scenario_set[]` array is populated with a single instance of the automatically-created atomic scenario – in the present example, an instance of `apb_data_atomic_scenario`, which simply creates a single randomized transaction. The default behavior has the useful side-effect that a scenario generator can be installed in the testbench at the outset, before any user-defined scenarios have been created. This unmodified default scenario generator will, with no additional coding effort, produce a stream of random transactions. It is then very straightforward to upgrade the testbench to use scenarios as they become available, since the environment already contains a scenario generator; there is no need to replace an atomic generator with a scenario generator.

User code in the environment's `build` method, or elsewhere, can easily add a user-defined scenario object instance to a generator's `scenario_set[]` array by applying standard SystemVerilog queue operations, as outlined in Figure 4. Alternatively, users could create a derived generator class that automatically adds the newly defined scenarios to its own `scenario_set[]`, and then instance that derived generator class in the environment.

Once the new scenarios have been added to the generator, they will become part of the mix of scenarios that is randomly generated. The existing atomic scenario that already existed at `scenario_set[0]` can remain *in situ*, where it will provide random background traffic randomly interleaved with the custom scenarios. Alternatively it may be removed using one of the queue delete methods, or its likelihood of selection may be adjusted by adding constraints in an extension of the `scenario_election` class and using that to replace the generator's default `select_scenario` instance.

```
class apb_arbiter_env extends vmm_env;

  // One scenario generator for each master port
  apb_data_scenario_gen G1, G2;

  // Channels for the scenario generator outputs
  apb_data_channel GC1, GC2;

  ... declare other verification components

  virtual function void build();
    super.build();

    // Construct the channels
    GC1 = new(…);

    ... construct other components of the environment

    // Construct the generators attached to those channels
    G1 = new("channel 1 generator", 1, GC1);
    begin : add_custom_scenarios
      // create an instance of our custom scenario
      apb_RMW_burst_scenario s = new;
      // add that instance to the generator's scenario_set[]
      G1.scenario_set.push_back(s);
      ... add custom scenarios to other generators
    end

    // enable random scenario selection in the generators
    G1.select_scenario.round_robin.constraint_mode(0);

    ... add custom behaviors to other components

  endfunction : build
```

Figure 4: Adding scenarios to generators in the verification environment

3.4 Structuring a scenario by populating its `items` array

From the foregoing discussion it is clear that the generator's `scenario_set[]` is effectively an array of factory objects. The scenario election mechanism chooses an element of this factory array, and the generator then randomizes the chosen factory object to yield its output. However, the generator does not take a copy of the factory object in the usual manner. Instead it calls the factory scenario's `apply` method which, in the default case, passes copies of each element of the scenario's `items[]` array in turn to the generator's output channel. It is thus reasonable to think of this as a two-level factory mechanism, with the freshly-randomized members of a scenario's `items[]` array acting one by one as factories for the resulting stream of output.

This view of the scenario's operation leads to an alternative possibility for defining scenarios. If the user has already created a set of classes derived from the transaction data object, each derived class having a different set of constraints, then the scenario's `items[]` array can be populated with a variety of these derived-class objects. This specific set of derived-class objects will then impose structure on the resulting scenario when it is randomized.

To use this approach it is necessary to build the `items[]` array after the scenario's kind has been chosen, since different scenario kinds may require different sets of objects. The scenario base class has predefined methods `allocate_scenario` and `fill_scenario` to simplify this task, which is typically undertaken in the scenario's `pre_randomize` method. Further details can be found in [2].

4. Procedurally constructed scenarios

Section 3 described the conventional method of customizing a scenario generator by adding specialized constraints in one or more derived scenario class. While this mechanism is straightforward, flexible and easy to extend, randomization does not always provide the most appropriate way to construct scenario data items. As an alternative approach it is possible to construct scenarios using procedural code.

4.1 Limitations of preconfigured scenarios

All the techniques described thus far depend on constructing the entire scenario and then delivering its items one by one. There may be situations in which that is inappropriate or inconvenient. The user may want constraints on individual items to be affected by other activity at the time of the item's application, which – in a large scenario – may be some considerable time after the scenario was generated. It is inappropriate to re-randomize the item at that time, because scenario-wide constraints would not work correctly and so the re-randomization might destroy the integrity of the scenario. Reference [2] hints that the scenario's `apply` method can be adapted to meet this requirement, but does not give concrete examples. The next section discusses how this technique can be used in practice.

4.2 Using the scenario's `apply` method

As already noted, the scenario generator automatically calls each scenario's `apply` method after having randomized the scenario's contents. The `apply` method's default implementation simply streams each member of the scenario's `items[]` array to the generator's output channel. However, it is clear that the `apply` method could do much more than this. In particular, it can take full responsibility for constructing the scenario data items.

4.3 Simple example of a procedurally generated scenario

```
constraint RMW {
  if (scenario_kind == APB_RMW) {
    length == 0;
    repeated == 0;
  }
}

virtual task apply(apb_data_channel channel, ref int unsigned n_insts);
  if (scenario_kind == APB_RMW) begin
    apb_data Tr = new;
    int unsigned A;
    Tr.randomize() with { kind == apb_data::READ; };
    A = Tr.addr;
    channel.put(Tr.copy());
    Tr.randomize() with { kind == apb_data::WRITE; addr == A; };
    channel.put(Tr);
    n_insts = 2;
  end else
   super.apply();
endtask
```

Figure 5: Procedural implementation of read-modify-write scenario

Figure 5 shows an alternative implementation of the read-modify-write scenario already discussed. In this implementation the scenario length is constrained to 0 so that no transaction items are constructed when the scenario is first randomized. Instead, the `apply` method creates a transaction, randomizes it with READ direction, records the transaction's address in a local variable, sends a copy of the transaction to the output channel, then re-randomizes the transaction with the same address but with WRITE direction and similarly sends it to the output channel. Note that the `apply` method must take responsibility for reporting (through the `n_insts` argument) the number of transactions that it posted to the channel. If any other kind of scenario was applied, the custom `apply` method abdicates its responsibility by calling the parent class's `apply` method.

Even in simple cases such as that of Figure 5, procedural scenarios are typically more verbose and less clear than those driven by constraints. However, provided the output channel is a single-place FIFO, procedural scenarios have the interesting property that transactions are not randomized until they are needed. Consequently, their randomization can be directly influenced by the current state of other parts of the simulation.

5. Hierarchical scenario generation

This section outlines the objectives, challenges and design considerations for hierarchical scenario generation, *i.e.* scenarios that can be described in terms of other scenarios. Section 6 then describes the author's prototype implementation of hierarchical scenarios.

5.1 Motivation

As a verification project progresses, increasingly sophisticated stimulus generation is required to maximize functional coverage metrics and explore hard-to-reach corner cases in the DUT. An obvious way to manage such increased complexity of stimulus is to define new scenarios in terms of existing ones. A compelling example of this approach may be found in the generation of instruction streams for programmable processors, where the nature of an "instruction stream" is likely to be recursively defined. For example, an instruction stream may contain (amongst other things) arithmetic instructions, conditional branch instructions, and loop constructs. But a loop construct is merely an instruction stream that has as its last instruction a conditional branch back to its start. The instruction stream contained within a loop construct may, of course, contain other loop constructs; the stimulus can be thought of as a hierarchy.

Suppose we imagine that we have already defined an "instruction stream" scenario that does not incorporate the notion of a loop construct. If we could define scenarios in terms of other scenarios, it would be easy to add a "loop construct" scenario defined as an "instruction stream" followed by a "branch back to start of instruction stream". (In passing, we note that the first part of this scenario is essentially unconstrained, but the latter part is rather tightly constrained.) If the "loop construct" scenario could then be added to the repertoire of available components of an "instruction stream", then the nesting of loops would follow as a natural consequence. Adding a further scenario to represent "subroutine call" would immediately admit the possibility of loops that contained subroutine calls, and vice versa, to any nesting depth.

Followed through to its conclusion, this suggests that every scenario should be defined in terms of other (possibly randomly chosen) scenarios. The entire stimulus stream is then defined by a top-level scenario, which in its turn creates a stream of other scenarios. Ultimately the lowest-level scenarios would be atomic, and thus defined in terms of a single transaction data item. The creation of an atomic scenario entails delivery of its transaction data item to a specified output stream. The stimulus stream could then be seen as a tree with the top-level scenario at its root, and an atomic scenario at each leaf. The resulting stimulus is simply the sequence of transaction data in the leaf scenarios obtained by depth-first traversal of the tree.

5.2 Procedural generation of hierarchical scenarios

The `randsequence` statement of SystemVerilog provides an obvious means to implement such recursively-defined scenarios. However, it is not ideal for verification re-use because it is not readily extensible. As an example, the reader is invited to imagine a `randsequence` statement that yields instruction streams including loops, and then consider how to extend that statement so that it also yields subroutine calls. Although both are readily implemented, it is necessary to replace the whole `randsequence` statement when adding a new feature to it.

It would also be possible to implement hierarchical scenarios procedurally by writing each scenario as a subprogram (a SystemVerilog `task`) and arranging that each task could randomly choose which other scenario tasks to call. This solution, too, is straightforward to implement but very troublesome to extend. Each subprogram must contain some kind of randomly selected multi-way branch; when a new scenario is added, all such branch constructs must be rewritten to deal with the new possibility.

5.3 Object-oriented hierarchical scenarios

The inherently polymorphic nature of class instances (objects) naturally suggests another approach. Suppose we could define a base class to represent a scenario, and every new scenario is derived from this base class. Consequently, a scenario variable of base class type is capable of carrying a reference to any of the available classes of scenario object. Executing or generating a scenario is then merely a matter of calling its appropriate virtual method. When new scenario classes are created, they fit into this scheme without requiring any existing code to be modified.

Unfortunately, this attractively simple view is not easy to realize in practice. There are several areas of difficulty.

- Derived scenario classes typically have additional properties (data members) that are absent in the base class. Constraints over these additional properties can exist in the derived class without any other code being aware of their existence. However, if generator code wishes to apply useful constraints to a derived class object, it probably needs to have access to some of these additional properties and constraints. This cannot be done via a variable (reference) of the base class type, and so it is necessary to down-cast the object reference. This is at best tiresome, and is generally regarded as poor object-oriented programming practice.
- Randomization in SystemVerilog requires that the object to be randomized should already exist, *i.e.* it should already have been constructed. Consequently, it is not possible in a single step to make random choice of the specific derived class of a scenario and to randomize its contents. This is unfortunate, because it means that the definition of a scenario must have at least two distinct parts: a means to construct objects of appropriate derived class type, and a means to randomize their contents. A test-specific constraint as simple as "this sub-scenario should be a loop construct, and its first instruction should be an ADD opcode" must be unnaturally split into two isolated components, affecting different parts of the code.
- In the specific case of VMM scenarios, the randomization of scenarios is intimately linked to the operation of a scenario generator. For hierarchical scenarios (and, as we will see later, for coordination of multiple scenario streams) it would be preferable for the entire generation and randomization process to be managed by a scenario object itself. The scenario generator should be responsible only for accepting the ultimate output of the hierarchy of scenarios, a stream of transaction data items – and, perhaps, for launching just one scenario that is the root of a tree of hierarchical scenarios. This issue is discussed in more detail below.

5.4 Responsibilities of the VMM library scenario and generator classes

To create hierarchically defined scenarios that nevertheless fit into the VMM scenario framework, we must first analyze carefully the responsibilities of the various scenario classes created by the VMM Standard Library scenario generator macro.

The **scenario** class provides three core features that concern us: storage of the `items[]` array of transaction data, constraints for randomization of those items, and the `apply` method.

The **scenario generator** class contains storage for the `scenario_set[]`, an array of references to instances of every scenario class available to this generator. It also possesses an output channel, to which the items of all scenarios are sent. Finally it creates, randomizes and applies a certain number of scenarios. Each scenario in turn is created by using one member of the `scenario_set` as a factory, chosen using the generator's instance of a scenario chooser.

The **scenario chooser** (known as the *scenario election* class) provides the scenario generator with a policy for choosing which member of its scenario set is to be used as a factory for each new scenario. The generator has an instance of this class. Whenever it needs to choose a scenario factory, the generator randomizes this instance. The chooser has random constraints, and also maintains some history of previous choices so that it could (for example) implement a cyclic or non-repeating choice policy. Once randomized, the chooser has a new value of its `select_scenario` data member, which is simply an index into the generator's `scenario_set[]` indicating which element to use as a factory.

5.4.1 Responsibility for construction of scenarios

If a scenario itself were to take responsibility for the hierarchical manufacture of its child scenario objects, it would be necessary for that scenario to contain its own `scenario_set` and a means to choose randomly from it. Whilst this would be straightforward, it would represent unnecessary duplication of work already undertaken by the predefined scenario generator. Consequently, the approach described in section 6 makes use of existing factory capabilities of the scenario generator object. Despite this, there is still some duplication of effort. With hindsight it might have been preferable had VMM's scenario scheme provided a completely separate factory class to perform the task of choosing and constructing scenario objects – a task that, in the standard implementation, is undertaken entirely within the generator. Such a factory class could then have been used by any other part of the code needing to generate a scenario.

5.4.2 Management of constraints over a scenario

In the current VMM scenario implementation, scenarios are both chosen and randomized by code hidden in the generator. Consequently, to modify the constraints on scenario randomization it is necessary to create a new derived class from the existing scenario. This is, at best, somewhat cumbersome. It would be good to have a lightweight means to add new constraints to an existing scenario, rather as SystemVerilog's `randomize … with {…}` construct works today. Unfortunately, the `with` option on randomization cannot be imposed by any means other than its appearance in the `randomize` syntax. Section 6 suggests a useful compromise whereby additional constraints can be encapsulated in a very simple additional class that we call a *constrainer*, making it rather straightforward to write and impose new constraints on the randomization of existing derived-class scenario objects.

5.4.3 Managing scenario hierarchy using the `apply` method

As described above in sections 4.2 and 4.3, a scenario's `apply` method is called automatically by the generating transactor once the scenario has been randomized. This provides an opportunity for a scenario to describe itself hierarchically in terms of other scenarios. However, this implies that the child scenarios are created procedurally. The technique described in section 6 provides a novel compromise between the two extremes: fully randomized generation of a single-level scenario, and fully procedural creation of a scenario in its `apply` method.

6. A prototype extensible hierarchical scenario generation scheme

This section outlines a technique for implementing hierarchical scenarios in a flexible and extensible manner. As mentioned above, it duplicates some activity that is already part of normal operation of the scenario generator class created by VMM standard macros. The author is continuing work to reduce this duplication, and to improve its integration with other features of VMM such as notifications and callbacks. However, it has yielded promising results in a prototype application.

In this scheme, the role of the scenario generator transactor in creating transactions is restricted to creating and randomizing a small number (perhaps only one) of top-level hierarchical scenarios, each forming the root of a scenario hierarchy. The `apply` method of each of these top-level scenarios then creates, randomizes and applies other scenarios. Ultimately, this scenario creation process yields scenarios that have been defined in terms of transaction items rather than other scenarios. These items, forming the leaves of the scenario hierarchy tree, are then delivered to the scenario generator's output channel.

6.1 The hierarchical scenario class

As shown later in this section, a hierarchical scenario needs access to its parent scenario generator object. Unfortunately the VMM scenario does not have a reference to its parent generator. Consequently we must create an extension of the scenario class that adds this property, and requires it to be set in a constructor. Note that the names of any classes we create to implement hierarchical scenarios are of the form `apb_data_HS_???`, where `HS` is used as an abbreviation for *Hierarchical Scenario*. This derived class can easily be created using a macro. Further specialization of the scenarios will be achieved in user-written classes derived from that shown in Figure 6 below.

```
class apb_data_HS extends apb_data_scenario;
  apb_data_HS_gen parent_generator;
  function new(apb_data_HS_gen pg);
    super.new();
    parent_generator = pg;
  endfunction : new
  virtual task perform ... see
endclass
```

Figure 6: Hierarchical scenario class with reference to parent generator

6.2 Using the scenario generator's `apply` method to generate other scenarios

When the standard VMM scenario generator creates a scenario it also randomizes the scenario object, thereby creating values for all the transaction data in the scenario's `items[]` array. Clearly this is the wrong behavior for a hierarchical generator. Instead, the generated scenario's `apply` method must, in its turn, create further scenarios. As already observed, in SystemVerilog this must be a two-step process: first a scenario instance must be constructed, and then it should be given its value – typically by randomization.

In the standard VMM scenario generator, the "value" of a scenario is simply the set of transactions in its `items[]` array. If the scenario is hierarchical, though, this is inappropriate; instead, the scenario's value is a collection of data members of the scenario class that, collectively, control the hierarchical scenario's behavior. It is, of course, entirely possible that some parts of that behavior might be represented by transactions in the `items[]` array. Other parts of the scenario's behavior, though, must be controlled by data members that in their turn will influence the action of the scenario's `apply` method.

Within a hierarchical scenario, then, the creation of child scenarios must be essentially procedural, in the `apply` method. But we wish to retain the well-understood benefits of randomization in both steps of this creation process. The first step must offer randomization of *choice of scenario sub-class* from the `scenario_set[]`. Once the chosen scenario has been constructed, the second step must offer flexibility in adding *scenario-specific constraints* to the existing constraints in the scenario sub-class.

These new behaviors depend in part on functionality and data structures in the VMM scenario generator class. We therefore create a derived generator class with just one additional method `choose`, which returns an appropriate chosen scenario from the generator's `scenario_set[]`. This new method can be called from the `apply` method of a scenario that wishes to create a child scenario. Figure 7 shows the complete code required to create this derived class in the case of our APB scenario generator example.

```
class apb_data_HS_gen extends apb_data_scenario_gen;
  virtual function apb_data_scenario choose(apb_data_HS_chooser c = null);
    if (c == null)
      this.select_scenario.randomize();
    else
      c.choose(this.select_scenario);
    return this.scenario_set[this.select_scenario.select];
  endfunction
endclass
```

Figure 7: Adding a scenario chooser method to the scenario generator

To allow straightforward customization of the choice of scenario, the `choose` method takes an optional argument of type `apb_data_HS_chooser`. This base class is effectively a policy class that permits the choice of scenario to be decoupled from the generator, by adding policy-specific constraints to the randomization of the generator's own `select_scenario` object. Note that the new classes `apb_data_HS_gen` and `apb_data_HS_chooser` could easily be written by an extension of the macro that already creates the other scenario classes.

6.3 Choice of scenario sub-class

In the standard VMM scenario generator, the choice of scenario sub-class is controlled by randomization of the scenario generator's internal `scenario_election` object. To provide scenario-by-scenario control over this behavior, we create a very simple base class, derivatives of which will implement the desired scenario-choice policy. The base class is defined thus:

```
class apb_data_HS_chooser;
  virtual function void choose(apb_data_scenario_election it);
    it.randomize();
  endfunction
endclass
```

Figure 8: Scenario chooser base class

As can be seen, the base class contains only a single virtual method and is thus rather easy to extend. This method's default (un-extended) behavior is to randomize its `scenario_election` argument, just as the scenario generator itself does. Consequently, if the user does not extend this class, it benignly follows the scenario choice strategy of the generator. However, a user wishing to impose specific choice strategy can do so in one of two ways:

- Explicitly access one element of the `scenario_set[]` array.
- Provide a customized chooser subclass, and supply an instance of that subclass as an argument to the generator's `choose` method.

Customization of the chooser subclass is so simple that it can reasonably be done locally within the same source file that creates a new scenario subclass, as in the following example:

```
class special_chooser extends apb_data_HS_chooser;
  virtual function void choose(apb_data_scenario_election it);
    bit saved_constraint_mode = it.round_robin.constraint_mode();
    // avoid conflict between custom constraints and original constraint
    it.round_robin.constraint_mode(0);
    // custom randomization...
    it.randomize() with {select != 0};
    // restore original constraint
    it.round_robin.constraint_mode(saved_constraint_mode);
  endfunction
endclass

class DOUBLE_scenario extends apb_data_HS;
  ... other customizations ...
  virtual task apply(apb_data_channel channel, ref int unsigned n_insts);
    // Make an instance of the chooser class
    special_chooser c = new;
    apb_data_scenario scenario;
    repeat (2) begin
      // Make a child scenario that is not index 0 in the scenario_set[]
      scenario = parent_generator.choose(c);
      ... Randomize and apply the chosen scenario ...
    end
  endtask
endclass
```

Figure 9: New scenario class implementing a specialized chooser

6.4 Randomization of the chosen scenario; the *constrainer* class

The purpose of a scenario, or a scenario of scenarios, is to create a stream of data items that respect some specified structure but are otherwise randomized. Having chosen a child scenario as shown in Figure 9 above, it is now necessary to randomize its contents. The constraints specified in the chosen child scenario are unlikely to be sufficient to maintain the desired structure of its parent scenario. The simplest way to add new constraints is inline randomization using SystemVerilog's `randomize … with {…}` syntax, but this is impractical when writing generic code. The exact subclass of the scenario to be randomized is unknown: the scenario is held in a base-class reference and, therefore, constraints on any of its members other than those of the base class are inaccessible to this approach. To handle this problem we create a policy class whose sole purpose is to manage additional constraints. We refer to such a class as a *constrainer*. A hierarchical scenario appeals to the `do_randomize` virtual method of a suitable constrainer object to implement specialized randomization of its child scenarios' contents.

```
class apb_data_HS_constrainer;
  virtual function void do_randomize(apb_data_scenario it);
    it.randomize();  // default implementation is benign
  endfunction
endclass
```

Figure 10: Constrainer base class

Now consider how we might write a new, specialized constraint on our existing custom scenario `apb_RMW_burst_scenario` (see Figure 2) to constrain it to make a burst cycle with exactly 8 writes. The constraint is straightforward. Using the constrainer class makes it very simple to add such a constraint without the need to modify, or derive from, the existing scenario class. The constrainer object can use run-time dynamic casting `$cast` to check that it is being correctly applied, and to create a local object to which the desired constraints are applicable.

```
class apb_burst_W8_constrainer extends apb_data_HS_constrainer;
  virtual function void do_randomize(apb_data_scenario it);
    apb_RMW_burst_scenario s;
    assert ($cast(s, it)) else $error("Inapplicable Constrainer");
    s.randomize() with {
      scenario_kind == APB_BURST;
      length == 8;
      foreach (items[i]) {
        items[i].kind == apb_data::WRITE;
      };
    };
  endfunction
endclass
```

Figure 11: Specialization of a constrainer

As described in the next section, this `do_randomize` method is called by the parent scenario as part of its `apply` method.

6.5 Implementing a hierarchical scenario

We can now complete the apb_data_HS class whose skeleton was shown in Figure 6, and extend it to create new hierarchical scenarios. The only new feature of the base class is a method perform, which will be called by the apply method of hierarchical scenarios. Note that perform is essentially a combination of randomize and apply, and therefore almost exactly matches the behavior of the scenario *generator* when it has chosen a member of its scenario_set[]. The only difference is the constrainer argument, providing fine-grained control over randomization of each new scenario constructed by its hierarchical parent.

```
class apb_data_HS extends apb_data_scenario;

  apb_data_HS_gen parent_generator;

  function new(apb_data_HS_gen pg);
    super.new();
    parent_generator = pg;
  endfunction : new

  virtual task perform
    ( apb_data_channel channel
    , ref int unsigned n_insts
    , apb_data_HS_constrainer c = null
    );
    if (c == null)
      this.randomize();
    else
      c.do_randomize(this);
    this.apply(channel, n_insts)
  endtask

endclass
```

Figure 12: Completed implementation of hierarchical scenario base class

A scenario that wishes to implement itself in terms of one or more child scenarios can now do so by first appealing to its originating generator to choose a scenario, and then calling that scenario's perform method to randomize and apply the chosen scenario. Each of these steps can be adjusted locally – the first by using a chooser, the second with a constrainer. The chooser and constrainer objects are so simple to write that it is entirely reasonable to write a new one for any new scenario that you wish to create. Figure 13 shows an implementation of the apply method for a hierarchical scenario that implements an eight-cycle burst writes as described earlier, followed by a read-modify-write to address 0 based on a second constrainer class. This new constrainer class is not shown, but is very similar to the example in Figure 11.

Note that *all* scenarios must be derived from a hierarchical scenario base class for this to work correctly, necessitating a minor rewrite of already-written scenarios to derive them from the proper base class. This requirement arises because the parent scenario calls the child's perform method, which is defined in the hierarchical scenario's base class. It would be possible to work around this in the parent's apply method by using $cast to check whether the child is a hierarchical scenario, but it seems preferable to have the perform method implemented in all scenarios. A leaf scenario is then one whose `apply` method only streams data items to its output channel, and does not choose and perform any child scenarios.

```
virtual task apply
  ( apb_data_channel channel
  , ref int unsigned n_insts
  );
  // Establish appropriate chooser and constrainers
  special_chooser ch = new;
  apb_burst_W8_constrainer c_w8 = new;
  apb_RMW_addr0_constrainer c_rmw0 = new;
  // Select and perform the child scenario
  apb_data_HS child = parent_generator.choose(ch);
  child.perform(channel, n_insts, c_w8);
  child.perform(channel, n_insts, c_rmw0);
endtask
```

Figure 13: `apply` method of a simple hierarchical scenario

7. Coordination of scenarios - interrupts, multiple streams

The mechanism described in section 6 was originally conceived solely as a means to construct hierarchically specified scenarios. As part of that design it became necessary to encapsulate independently the generation and randomization of scenarios, outside the scenario generator object that takes responsibility for a given scenario stream. Serendipitously, this separation also facilitates the coordinated application of stimulus on multiple input streams. A single scenario can be composed of many other scenarios, and the `perform` method of each can be directed to a different channel – it is not necessary for a generator to drive scenarios into its *own* output channel. Scenarios and components of scenarios can now be constructed on-the-fly in response to coordination events such as notifications, and multiple scenario streams can be applied concurrently using `fork…join` and related process-control constructs. In such a setup, a single generator could take responsibility for the creation of a number of coordinated streams on several channels.

The VMM generators and transactors already make provision for interrupt-like stimulus, since they have an `inject` method that allows a scenario to be inserted into an existing stream. Alternatively, the `vmm_scheduler` class can coordinate the delivery of multiple stimulus streams to a single consumer. In either approach, the hierarchical scenario's ability to deliver stimulus to a dynamically chosen channel permits both the content and the timing of such interrupt-like activity to be managed within a single scenario.

The author is actively pursuing these approaches at the time of publication, and hopes to be able to report on them shortly. It is noteworthy that [2] specifies a somewhat independent mechanism, known as *XVC* (Extensible Verification Components), to address such requirements in the context of large-system verification projects.

8. Conclusions and further work

This paper has indicated the motivation for enhanced scenario generation, and described a prototype hierarchical scenario generator that has provided useful initial results.

For this mechanism to be widely useful, further work is required to integrate it more closely with other standard VMM mechanisms that have been ignored or sidestepped in this prototype implementation. In particular, it is necessary to respect VMM's conventions concerning callbacks, notifications and message logging. All this can be integrated in the new base classes without much difficulty, and the author aims to report on a more robust implementation in due course.

Finally, it is clear that the new base classes need to be specialized for transaction data type. In the VMM Standard Library, such specialization is undertaken by macros that write the base classes corresponding to a user's transaction data type. It is clear that similar macros need to be created to support the proposed new techniques.

9. Acknowledgements

The author is indebted to his employer Doulos Ltd for providing the time, facilities and encouragement that made this work possible.

Hans van der Schoot of Xtreme EDA Corporation reviewed the paper and made valuable suggestions. Thanks are due to the authors of the VMM book, and particularly to Janick Bergeron of Synopsys, Inc. who has on many occasions courteously and patiently corrected the author's misunderstandings of it.

Any trademarks or other proprietary names mentioned in this paper are acknowledged as the property of their respective owners.

10. References

[1] Std.1800-2005: IEEE Standard for SystemVerilog – Unified Hardware Design, Specification, and Verification Language. IEEE, Piscataway, NY 2005.

[2] Bergeron J, Cerny E, Hunter A, Nightingale A: Verification Methodology Manual for SystemVerilog. Springer 2005.

[3] Verisity: eVC Reuse Methodology (eRM) Developer Manual version 2.0. Verisity Design, Inc. 2002.

[4] Cadence: uRM SystemVerilog Class-based Library Reference. Cadence Design Systems, Inc. 2007.

[5] Glasser M (ed): Advanced Verification Methodology Cookbook version 2.0. Mentor Graphics Corporation, 2006.

[6] Mintz M, Ekendahl R: Teal Users Manual. Trusster, 2006. `www.trusster.com`

[7] ARM: AMBA3 APB Protocol Specification v1.0. ARM Ltd, 2004.

[8] Girodias P, van der Schoot H, Sultan A: Stepping Up to Scenarios in VMM. SNUG San Jose 2007.