Using the New Features in VMM 1.1 for Multi-Stream Scenarios

Jason Sprott *
JL Gray *
Sumit Dhamanwala *
Cliff Cummings †

* Verilab, Inc Austin, TX, USA www.verilab.com	† Sunburst Design Beaverton, OR, USA www.sunburst-design.com

March 17, 2009

ABSTRACT

Today's verification solutions often require complex concurrent streams of stimulus controlled from higher level transactors or scenarios. The VMM 1.1 library has been enhanced to add this capability, and support the management of access to the resources shared by different stimulus streams, i.e. multiple streams providing stimulus to the same transactor. This paper describes the challenges faced in developing these new features, and takes a detailed look at how they are used in a VMM "multi-stream scenario" environment.

Table of Contents

Table of Figures

Table of Tables

1 From VMM Scenarios to Multi-Stream Scenarios

In a VMM 1.0 environment, the user can generate sophisticated sequences of random or directed stimulus using VMM scenarios. Transactions containing the stimulus information are randomized in a scenario and driven by scenario generators, connected to the transactors they are driving via VMM channels. Each scenario generator operates independently and can be loaded with its own library of scenarios (these are now known as single-stream scenarios in VMM 1.1). Figure 1 shows a typical configuration, where each generator is dedicated to providing independent stimulus for a specific transactor.

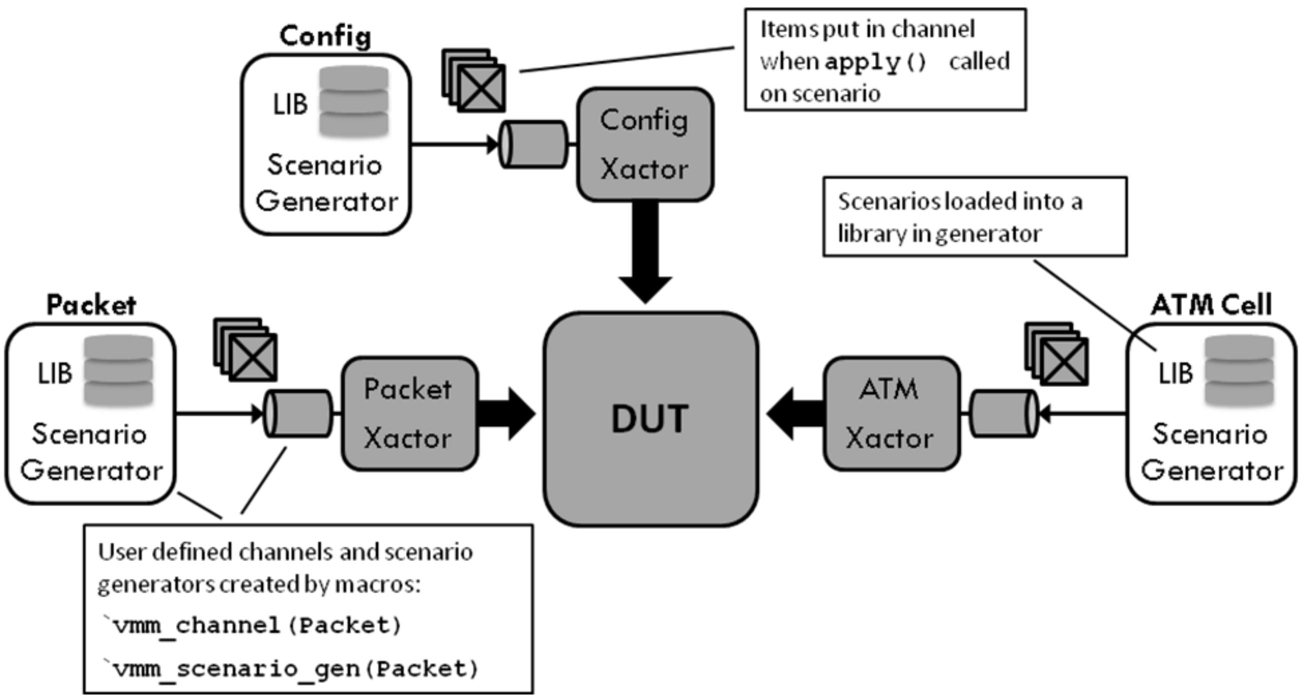
Config
LIB
Scenario Generator
Items put in channel when apply() called on scenario
Config Xactor
Scenarios loaded into a library in generator
Packet
LIB
Scenario Generator
Packet Xactor
DUT
ATM Xactor
ATM Cell
LIB
Scenario Generator
User defined channels and scenario generators created by macros:
`vmm_channel(Packet)
`vmm_scenario_gen(Packet)

Figure 1: Example of Single-stream Scenarios from VMM 1.0

Single-stream scenario generators are type specific and created using the `'vmm_scenario_gen` macro, which also creates a base scenario type. The base scenario type was designed to be connected to one output channel. By default the single-stream scenario generator has the ability to generate random scenarios. More sophisticated stimulus can be created by extending the basic single scenario class, which can then be added to a library of scenarios the generator can choose from.

Although single-stream scenarios can encapsulate procedural stimulus by overriding the `apply()` method, the default behaviour is to create a randomized list of transactions based on constraints. More than one single-stream generator can be connected to the same channel, but in VMM 1.0 data sent by different generators simultaneously would simply be interleaved, which isn't always desirable.

The problem with single-stream scenarios is that in today's complex testbenches, it's not enough to be able to create *independent* sophisticated scenarios, we also need to have some control over the bigger picture: the way the scenarios operate together. We need to control the sequencing of each generator's stimulus stream and allow scenarios to control access to the resources they are using. In the VMM 1.1, specific features have been added to address these needs.

Table 1 provides an overview of the VMM features that have been added, or modified, to provide a new multi-stream scenario capability.

Std Library Class	Feature Description
vmm_ms_scenario (NEW)	Multiple channel grabbing to avoid deadlock, parent/child relationship and use execute() instead of apply() to run multi-stream scenarios
vmm_ms_scenario_gen (NEW)	Registries for vmm_channel, vmm_ms_scenario and vmm_ms_scenario_gen
vmm_ms_scenario_election (NEW)	Selection scheme for multi-stream scenarios from registry
vmm_scenario (modified)	Reference to parent scenario added (used by channel grab). Single and multi-stream scenarios now extend from this class.
vmm_channel (modified)	Grab/Ungrab

Table 1: VMM 1.1 Multi-stream Scenario Related Feature Overview

Multi-stream scenarios provide the user the same ability as single-stream scenarios to build flat and hierarchical stimulus that can be organized into libraries. Multi-stream scenarios add the capability to drive and control access to more than one channel. The multi-stream generator adds the concept of registries which provide a database, indexed by string name, of resources that can be used by the scenarios.

Figure 2 is an overview of a multi-stream scenario version of the example in Figure 1, where a top level scenario controls stimulus to multiple transactors. The scenario is being used to drive configuration reads/writes, packets and ATM cells as stimulus. The multi-stream generator has a library of multi-stream scenarios, each capable of creating stimulus by using single-stream scenarios for Config, Packet and ATM Cell drivers, or injecting transactions directly into channels. Also, although not shown in the diagram, multi-stream scenarios can call other multi-stream scenarios.

The key concepts of multi-stream scenarios are:

- Generator automatically picks and executes a multi-stream scenario from a library of scenarios
- The `execute()` task in the scenario implements stimulus control
- Multi-stream scenarios can access and control multiple channels within the generator's scope
- Execute one multi-stream scenario from another, even if it's in a different generator's library
- Single and multiple-stream scenarios have a concept of hierarchy, i.e. they know their parent scenario

- Although not exclusive to multi-stream scenarios, or part of the scenarios classes themselves, the ability to grab channels and use scenario hierarchy in the grabbing rules, is fundamental to multi-stream operation

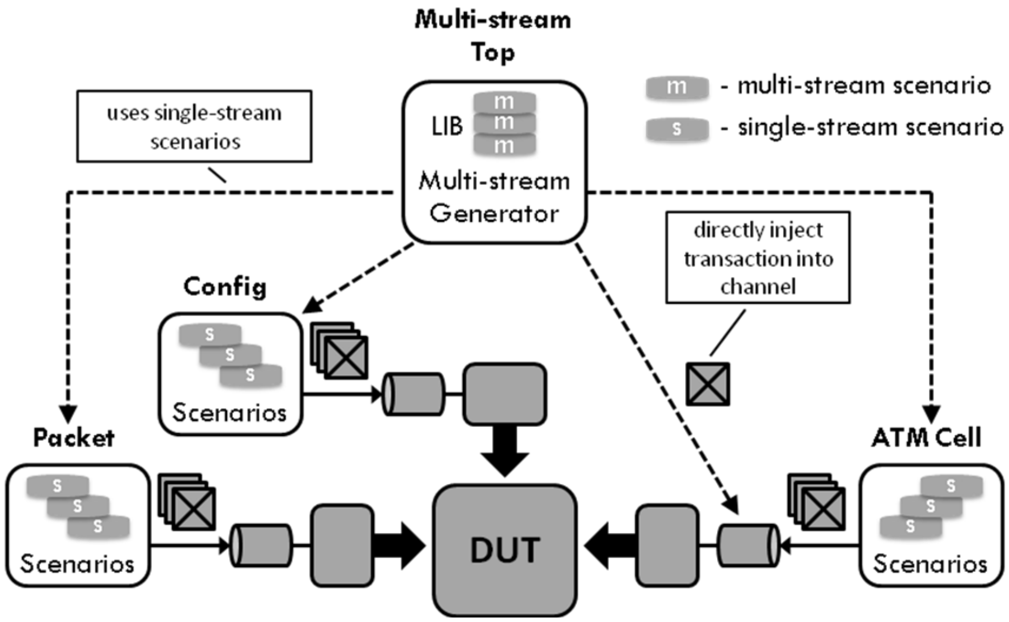
Multi-stream Top
m - multi-stream scenario
s - single-stream scenario
uses single-stream scenarios
LIB
m
m
m
Multi-stream Generator
directly inject transaction into channel
Config
s
s
s
Scenarios
Packet
s
s
s
Scenarios
DUT
ATM Cell
s
s
s
Scenarios

Figure 2: Overview of Multi-stream Scenarios

In Figure 2, the Config, Packet and ATM are each independent flows. A multi-stream scenario can be developed to coordinate the activities of these generators by ensuring the configuration phase from the Config generator runs first, and then turn on specific traffic patterns. Additionally, due to the channel locking capability, these scenarios can run concurrently with other traffic being sent to the same channels from other sources. The concurrent traffic could be sent from other multi-stream scenarios.

Multi-stream stimulus can be added to a legacy single-scenario environment, providing an additional and more controllable layer of stimulus. Alternatively, single-stream scenarios can be used without modification in a multi-stream scenario environment.

2 Grab Features in vmm_channel

One of the limitations in previous releases of the VMM is that there is no built-in way for multiple streams of stimulus to exclusively lock accesses to a given channel. In a VMM testbench we use a channel to pass stimulus information to a transactor. Data sent to a transactor's channel from different streams, i.e. multiple threads calling `put()` or `sneak()` on the same channel, could be interleaved with one another. This is not always desirable. For example, an algorithm might call for an uninterrupted sequence of commands. It may also be desirable to lock multiple resources (channels) for a given set of operations, such that other stimulus streams do not interfere with the flow of data to the transactors. For example, imagine an arbitration scenario where the activities on multiple interfaces are coordinated to create specific traffic timing patterns for a period of time.

In the previous release of the VMM, resources could be shared between stimulus streams, but there was no *standard* technique to guarantee exclusive channel access. Developers typically invented their own custom techniques to work around this limitation. When the multi-stream scenario capabilities were added, this limitation had to be resolved.

The problem cannot be resolved simply by having individual scenarios grab channels for exclusive use. Hierarchical stimulus, where a scenario (single or multi-stream) can have any number of child scenarios, makes it necessary to have a scheme that allows the children to obtain a grabbed channel for use, from their parent or another ancestor scenario. To enable full control over channel access, the following requirements were devised:

- Single and multi-stream scenarios should be able to grab exclusive access to channels
- When a channel is grabbed only the owner can add items
- When a channel is grabbed only the owner can ungrab
- A child scenario can be granted the grab if its parent is the current owner
- Once a child grabs the parent's channel, the parent can no longer access it until the child ungrabs
- Siblings may not access channels grabbed by other siblings
- Ownership will be returned to the parent when the child releases its grab
- A blocking and non-blocking interface is provided
- Multiple channels can be grabbed simultaneously to help manage potential deadlock situations

`vmm_scenario`, `vmm_channel` and `vmm_ms_scenario` were designed, or modified, to address the above requirements. The `vmm_scenario` class was modified to include hierarchical information, i.e. a pointer to the parent scenario (or null). The information is used by `vmm_channel` to decide on the outcome of a `grab()` call, where the grabber is passed as an argument in the call. This is compared to the channel's record of who currently owns the channel. The channel has access to all the information (via the handle to the grabber), necessary to decide if the grabber is a sibling, or a child, of the current grab owner. The information is also used to decide if an `ungrab()` call is legal. A summary of the new channel grab methods are shown in Table 2.

Method	Description

`task vmm_channel::grab(`
`   vmm_scenario grabber)`
	Blocks until channel is grabbed

`function void vmm_channel::ungrab(`
`   vmm_scenario grabber)`
	Ungrabs channel or generates an error if grabber is not the owner

`function bit vmm_channel::try_grab(`
`   vmm_scenario grabber)`
	Returns 1 if the channel grab is success, 0 otherwise. No grab request will be queued.
	If grabber is current owner, a warning will be issued, but the function will still return 0.

`function bit vmm_channel::is_grabbed()`	Returns 1 if the channel is currently grabbed by someone, 0 otherwise.

`task vmm_ms_scenario::grab_channels(`
`   ref vmm_channel channels[$])`
	Returns when all specified channels have been grabbed. Used to avoid deadlock.

Table 2: Summary of Channel Grab Methods

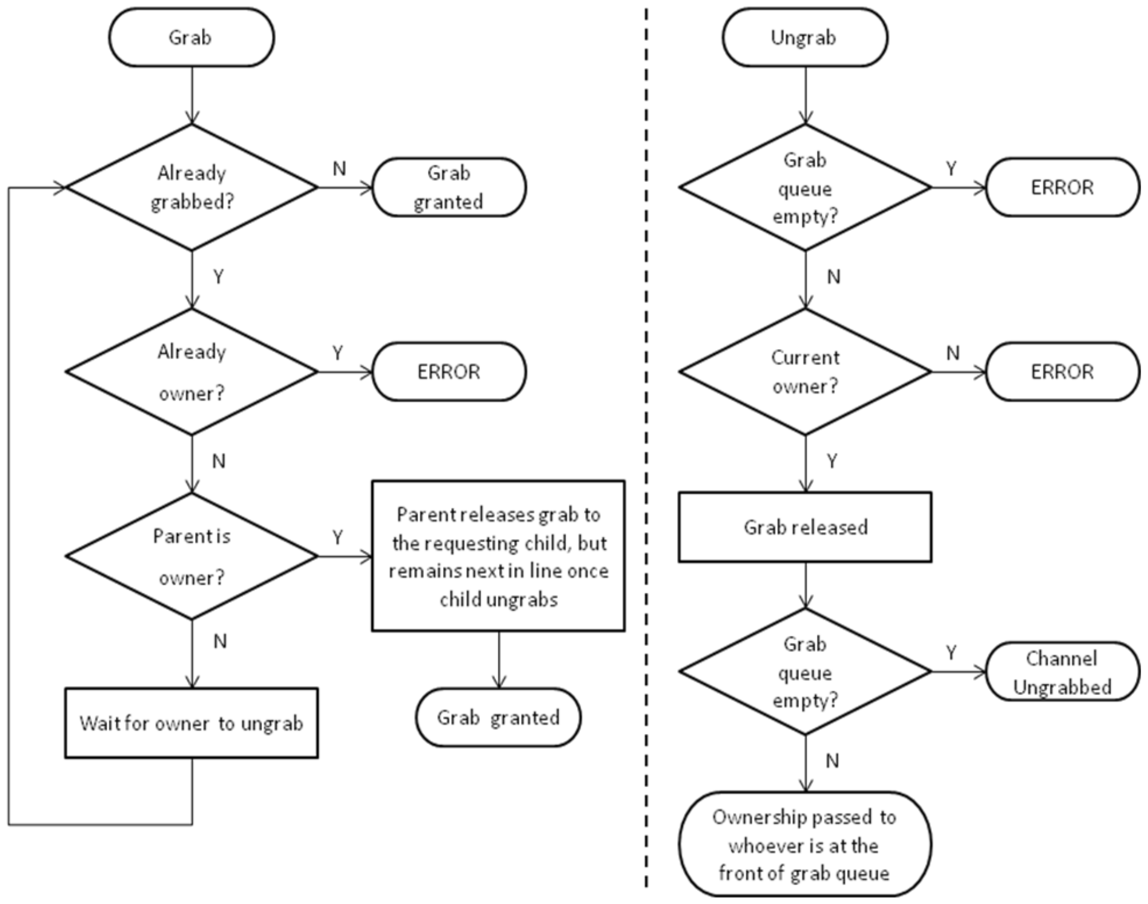

Figure 3: Grab/Ungrab Functionality

Figure 3 shows the basic operation of grab/ungrab in a flow chart. The channel keeps track of *grab owners*, where parent grabbers can be pushed onto a stack if a child grabs ownership from it. Grab requests are not stored in the channel. Threads blocked when calling `grab()` will be granted ownership based on grab rules and SystemVerilog thread scheduling order respectively. Trying to grab a channel already owned by the scenario, is illegal and will result in an error. It is up to the user to ensure that this condition does not occur.

It is possible legacy scenarios will access channels without grabbing them. This is allowed, but if a channel has been grabbed, access to it will be blocked until the channel is free (not grabbed by anyone). Threads blocked waiting on a grab for the channel will take priority over an attempted access not using grab. This could cause a subtle change in a legacy scenario's behavior, as compared to VMM 1.0, if the original assumption was that the scenario stimulus would be interleaved with other traffic on the channel.

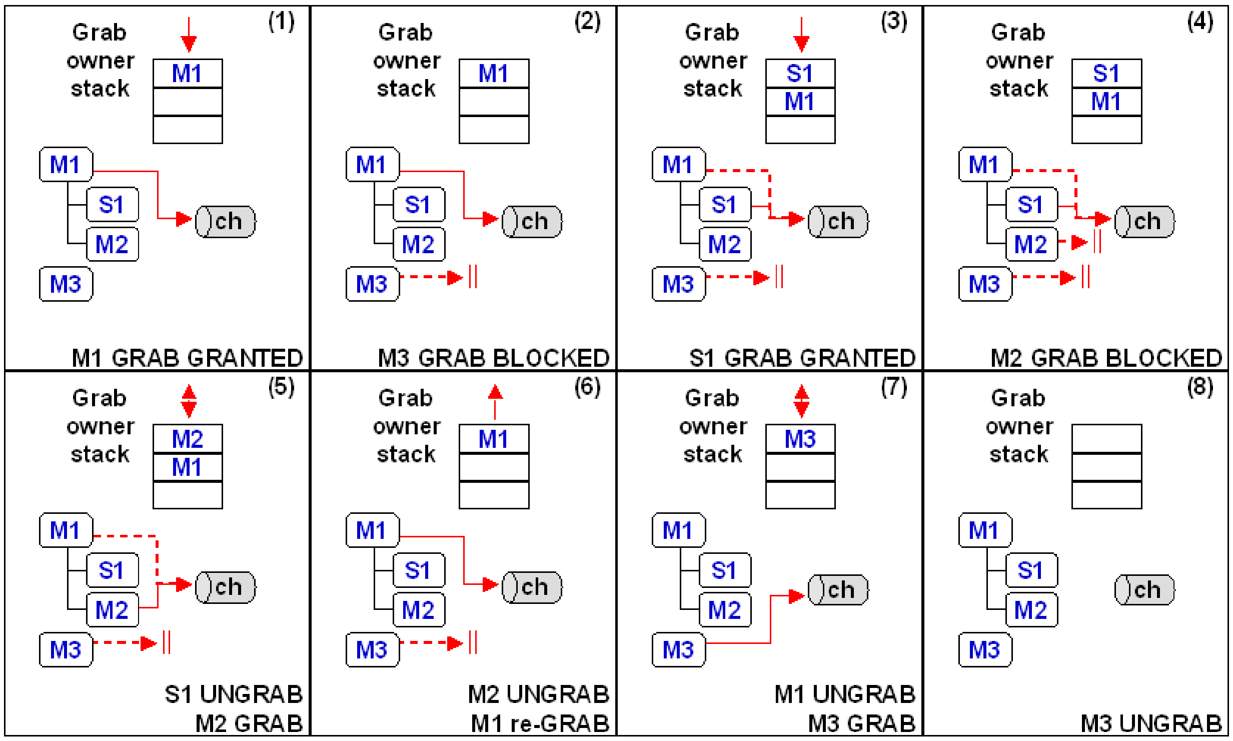

Figure 4: Channel Grabbing Example

Figure 4 shows an example of channel grabbing behavior. Multi-stream scenario M1 has two child scenarios: M2 (multi-stream) and S1 (single-stream). Multi-stream scenario M3 is a sibling to M1. The following describes the behavior in the numbered boxes:

1. Nobody owns the channel. M1 performs `grab()` which is granted. M1 is now the owner.
2. M3 performs `grab()` which is blocked because it is not a child of the owner. M3's request is retried each time someone ungrabs the channel.
3. S1 performs `grab()` which is granted as M1 is the parent. S1 is now the owner and M1 is pushed onto the owner stack, to be restored as owner when S1 ungrabs.

4. M2 performs `grab()` which is blocked since S1 is the current owner and is not the parent of M2. M2's request is retried when the owner ungrabs the channel.
5. S1 performs `ungrab()` which is allowed since it is the current owner. Ownership is passed back to M1, but is immediately passed to M2. M2 is a child of M1. M1 is pushed onto the owner stack, to be restored as owner when M2 ungrabs.
6. M2 performs `ungrab()` passing ownership to the parent M1.
7. M1 performs `ungrab()` allowing M3, who has being trying to grab every time the channel is ungrabbed by someone, to take ownership.
8. M3 performs `ungrab()` and since there are no more requests or stacked owners, the channel is free again.

A deadlock situation can arise if multiple streams try to grab an overlapping set of the same channels, at the same time. For example, multi-stream scenario M1 might try to grab channels C, B and A at the same time as another scenario M2 tries to grab B, C and D. If the channels are grabbed sequentially by each scenario using `grab()`, neither will be able to complete the sequence.

To fix this, a mechanism to grab multiple channels was devised. If any of the channels failed to grab, the request backs off all channels and tries again from the start. This solves the deadlock problem, but since the mechanism spans multiple channels, the `grab_channels()` feature had to be implemented in `vmm_ms_scenario`. The `grab_channels()` method takes in a list of channels to grab and blocks until all channels have been grabbed.

3 Multi-Stream Scenarios

3.1 Overview of Multi-stream Scenarios

There are two main components to multi-stream scenarios: the multi-stream scenario generator and the multi-stream scenario itself. Unlike single-stream scenarios, the generator and basic type of the scenario are not created by a macro; they are extended from their respective std_lib base classes. The multi-stream class hierarchy is shown in Figure 5.

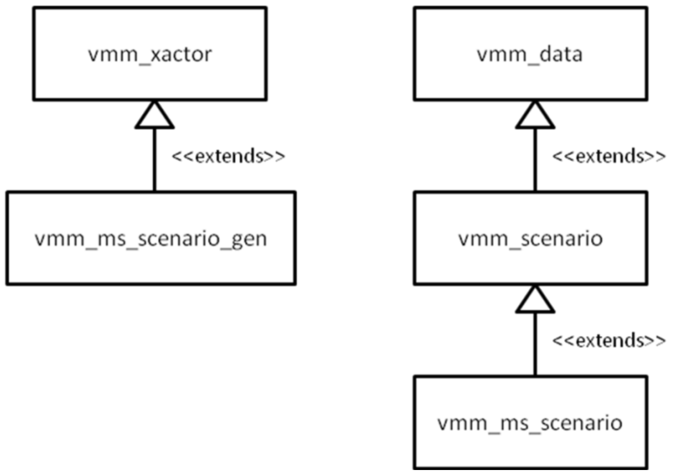

Figure 5: Multi-stream Scenario Class Diagram

The `vmm_ms_scenario_generator` class is a VMM transactor. The generator contains three registries:

- Multi-stream scenario registry: stores multi-stream scenario handles and acts as the library of scenarios available for selection by the generator.
- Channel registry: stores channel handles that are used by the scenarios
- Multi-stream generator registry: stores handles to other multi-stream generators. This can be used to get access to scenarios in another generator's registry.

The registries themselves are private member variables of the class and are managed via methods in the generator. Methods are provided to add, remove, replace, and search entries in the registries.

The election scheme used to select from the `scenario_set[$]` queue can also be changed, by replacing the variable `scenario_select` with an instance extended from `vmm_ms_scenario_election`. This is discussed in Section 3.4.

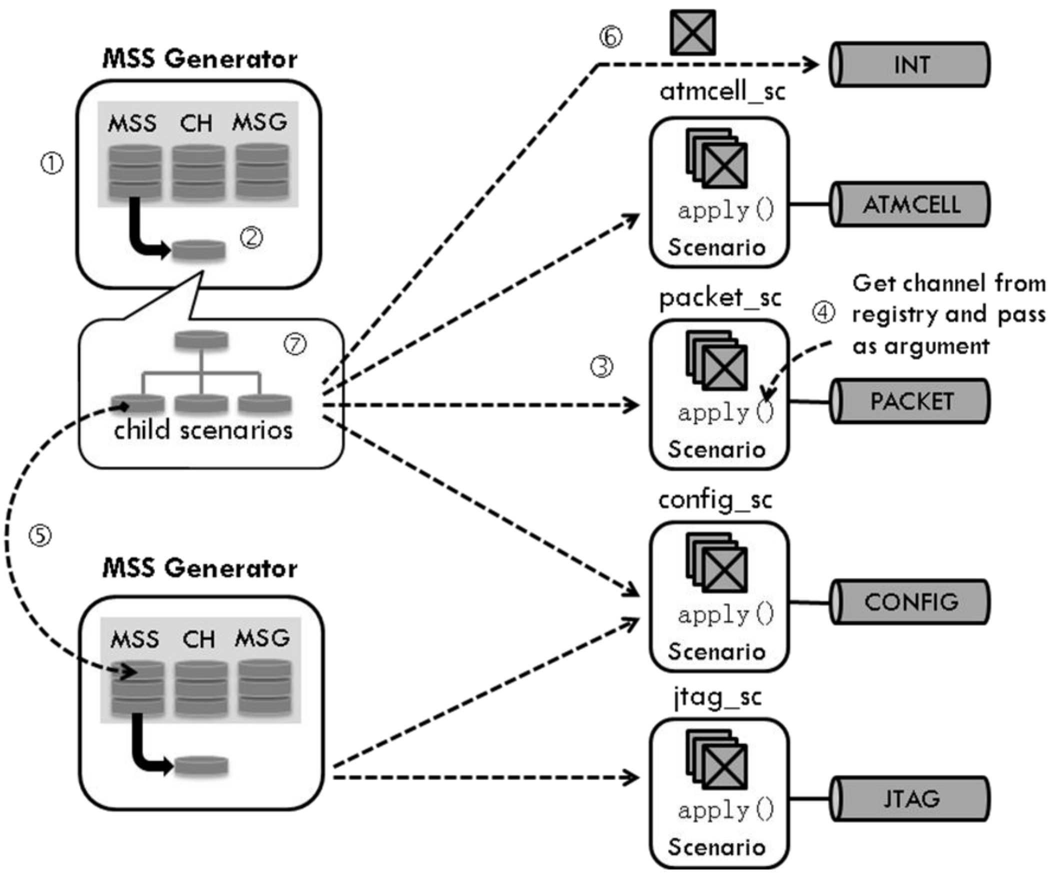

Figure 6: Overview of Multi-Stream Scenarios

Figure 6, provides an overview of multi-stream scenario operation, showing a hierarchy of generators, which either use multi-stream scenarios containing single-stream scenarios to create stimulus, drive items directly into channel, or call multi-stream scenarios located in other generators. The following notes refer to the numbered points in Figure 6:

1. Scenarios, channels and other multi-stream generators are loaded into their respective registries. The entries in the channel registry store the handles for channels used by any of the scenarios stored in the generator. Entries in the generator registry allow scenarios to look up other scenarios in a different generator.
2. The generator automatically selects a multi-stream scenario from the library, using the election mechanism. This will be executed by calling the scenario's execute() method. There is no default scenario to select, so *the user must register at least one scenario* with the generator.
3. A multi-stream scenario can run a single-stream scenario by instantiating and randomizing it in the multi-stream scenario's execute() method, then calling apply(). There is no default mechanism to randomize and run single-stream scenarios.
4. A channel is fetched from the registry and passed as the argument to the single scenario's apply() method. In a multi-stream scenario, the single-stream scenarios can put data into any

compatible channel that the multi-stream scenario generator has in its registry. It is not bound to a single output channel.

5. A multi-stream scenario can call other multi-stream scenarios from its own, or other generators. It is possible to look up other multi-stream scenario generators in the multi-stream generator registry. Once a handle to the other generator is obtained, the scenario can look up and execute multi-stream scenarios in the other generator's registry.
6. A multi-stream scenario can send transactions directly to a channel without using a single-stream scenario by instantiating and randomizing a transaction directly in the execute() method. The transaction can be sent to any compatible channel in the registry.
7. The hierarchical relationship is between the scenarios (single or multiple), not the generators. A multi-stream scenario can have single or multi-stream scenarios children. Scenarios from different generators can have a parent-child relationship with one another.

3.2 Building Multi-stream Scenarios

The following examples demonstrate how to setup a multi-stream scenario generator and implement the `execute()` method. The example has two multi-stream generators, `msg0` and `msg1`. Some multi-stream scenarios and channels are registered with the generators.

The `ms_jtag_debug_cmds` scenario, used in the example, is a child of `ms_traffic_with_jtag`. The scenarios are in different generators. The `ms_traffic_with_jtag` scenario also has a single-stream scenario and another multi-stream scenario from the same generator as child scenarios. The `execute()` task runs the three scenarios in separate threads concurrently.

Figure 7 shows the code for instantiating and registering multi-stream scenarios with their respective generators. An important point to note is that *instances* are registered with the generators. It's only safe to allow one thread to use the instance at any given time. If multiple threads need to run the same scenario, e.g. the scenario is selected and run by its own generator, and in parallel is run as child of a scenario in another generator, an instance of the scenario for each thread is required. This is best done by ensuring that when a scenario is fetched from the registry a copy of the scenario instance is made, such that the thread uses its own instance. This copy is not currently built into `get_ms_scenario()` so it has to be done by the user (see Figure 10).

```
// instantiating multi-stream scenarios
traffic_with_jtag_debug_ms_scenario   ms_traffic_with_jtag = new();
jtag_debug_cmds_ms_scenario           ms_jtag_debug_cmds   = new();
...
// register multi-stream scenarios with generator
msg0.register_ms_scenario("TRAFFIC_WITH_JTAG",  ms_traffic_with_jtag);
msg0.register_ms_scenario("TRAFFIC_NO_INTS",    ms_all_no_ints);
msg0.register_ms_scenario("TRAFFIC_WITH_INTS",  ms_all_with_ints);
// register multi-stream scenarios with generator
msg1.register_ms_scenario("JTAG_READ_STATUS",   ms_jtag_read_status);
msg1.register_ms_scenario("JTAG_DEBUG_CMDS1",   ms_jtag_debug_cmds);
msg1.register_ms_scenario("JTAG_RESET_CMDS",    ms_jtag_reset_cmds);
```

Figure 7: Registering Multi-stream Scenarios

In Figure 8, the channels used by the scenarios, or procedural code, are registered in the generator under logical names.

```
// register channels used by any of the scenarios or procedural code in the generator
msg0.register_channel("INT",    int_channel);
msg0.register_channel("CONFIG", config_channel);
msg0.register_channel("ATM",    atm_channel);
msg0.register_channel("PACKET", packet_channel);
msg1.register_channel("JTAG",   jtag_channel);
msg1.register_channel("CONFIG", config_channel);
msg1.register_channel("ATM",    atm_channel);
```

Figure 8: Registering Channels

Figure 9 illustrates how a multi-stream generator is registered with another generator. This allows the scenarios in `msg0` to access other scenarios in `msg1`.

```
// If a scenario in a generator uses a scenario registered in another generator
// the handle to that generator should be registered in the multi-stream generator
// registry. This can be used to lookup and execute the remote scenario
msg0.register_ms_scenario_gen("JTAG", msg1);
```

Figure 9: Registering a Multi-stream Generator

In Figure 10, the `execute()` task for the scenario encapsulates the stimulus generation and flow control. In this case the three threads generate stimulus using different types of scenarios. In the single-stream thread, the channel is grabbed for exclusive access. The other scenarios leave grabbing (if any), to their own `execute()` tasks.

Although the multi-stream scenarios are shown as being randomized, this is only useful if there are some random variables. Scenarios may not have random data, in which case they do not need to be randomized in `execute()`. When the generator itself picks and runs a scenario, it is always randomized. The remote multi-stream scenario (`ms_jtag`), is run in the context of `msg1`, where it was registered. It is not necessary to register the channels etc. `ms_jtag` uses with `msg0`.

```
class traffic_with_jtag_debug_ms_scenario extends vmm_ms_scenario;
. . .
virtual task execute(ref int n_insts);
   // channel used by single-stream scenario
   config_channel   config_ch;
   // create instance of single-stream scenario
   config_scenario  config_sc   = new(this);
   // get multi-stream scenarios (make local copies of instances for thread safety)
   vmm_ms_scenario ms_jtag     = get_ms_scenario("JTAG_DEBUG_CMDS","JTAG");
   vmm_ms_scenario ms_traffic  = get_ms_scenario("TRAFFIC_NO_INTS");
   $cast(ms_jtag,    ms_jtag.copy());
   $cast(ms_traffic, ms_traffic.copy());

   // setup hierarchy. The following are all children of this scenario
   config_sc.set_parent_scenario(this);
   ms_traffic.set_parent_scenario(this);
   ms_jtag.set_parent_scenario(this);

   // run various stimulus streams
   fork
   begin:single_stream
      int unsigned n=0;
      $cast(config_ch,  this.get_channel("CONFIG"));
      // we want to have exclusive access to the channel
      config_ch.grab(this);
      config_sc.randomize() with { ... };
      config_sc.apply(config_ch, n);
      config_ch_ungrab(this);
      n_insts += n;
   end:single_stream
   begin:local_multi_stream
      int unsigned n=0;
      ms_traffic.randomize() with { ... };
      ms_traffic.execute(n);
      n_insts += n;
   end:local_multi_stream
   begin:remote_multi_stream
      int unsigned n=0;
      ms_jtag.randomize() with { ... };
      ms_jtag.execute(n);
      n_insts += n;
   end:remote_multi_stream
   join
endtask:execute
...
endclass:traffic_with_jtag_debug_ms_scenario
```

Figure 10: Implementing execute() in a Multi-stream Scenario

```
class jtag_debug_cmds_ms_scenario extends vmm_ms_scenario;
  ...
  function vmm_data copy(vmm_data to = null);
    jtag_debug_cmds_ms_scenario s = new();
    // it is essential to copy across the context generator, or the scenario will
    // not know where to run from. We need to use these "unofficial" methods
    // to do that, as the member is local and there is no copy() in vmm_ms_scenario
    s.Xset_context_genX(this.Xget_context_genX());
    ...
    return(s);
  endfunction
  ...
endclass
```

Figure 11: Implementing copy() Method for a Multi-stream Scenario

Figure 11 shows how the copy method is implemented in a multi-stream scenario. It is recommended that copy is implemented in every multi-stream scenario, such that the value of the `context_scenario_gen` variable (the generator the scenario is registered with), can be copied. This allows other multi-stream scenarios to take a copy of the scenario more easily, which is required for thread safety, and shown in Figure 10.

The copy functionality is not built into the `vmm_ms_scenario` class. Since the `context_scenario_gen` variable in `vmm_ms_scenario` is local, `Xset_context_genX()` and `Xget_context_genX()` functions must be used to access it. These functions are *intended* to be local (this is apparent in the source code comments), but that is not enforced and they can be called as a public functions. There is no other way to access `context_scenario_gen`.

3.3 Wrapping a Single-stream Scenario as Multi-stream

A project may have a library of previously developed single-stream scenarios, either from another level of testing (e.g. module), or legacy from another project. These scenarios, and the original single-stream scenario generators, can still be used in the multi-stream environment; however, the intended behaviour will dictate if this is the best solution.

Single-scenarios themselves can be useful stimulus containers in a single or multi-stream environment. They have useful functionality to encapsulate transaction data item randomization, e.g. randomizable list of transaction items. It is the single-stream *generators* that can become redundant in a multi-stream environment.

For independent stimulus streams, that do not require coordination with other streams, using the original single-stream generators may be fine. However, if a scenario is to be coordinated, and made accessible to other scenario generators, it might be useful to make it look like a multi-stream scenario. A multi-stream scenario generator provides the features to support these requirements, and wrapping a single scenario as a multi-stream scenario can be useful for the following reasons:

- Removes the need for the single-stream generator, meaning only one type of generator is used
- `vmm_ms_scenario_gen::get_ms_scenario()` allows other multi-stream scenarios in the same generator to use the scenario
- Multi-stream scenarios in other generators can also access the wrapped scenario via their generator registry

The scenario can be wrapped by instantiating and running it inside a multi-stream scenario. This would involve writing a multi-stream scenario to select, instantiate and run, potentially many single scenarios. Alternatively, it could require writing one multi-stream scenario for each single-stream scenario. The advantage of the latter is that it is possible to use the generator to automatically select (according to election constraints), and run each wrapped single-stream scenario. This emulates the behaviour of a single-stream scenario generator.

Wrapping each scenario could require a significant amount of work, so some form of generic wrapper would be useful. Unfortunately, due to the way the single-stream scenarios are created using the `` `vmm_scenario_gen `` macro, a completely generic wrapper is not possible without another macro.

VMM 1.1 has changed the class hierarchy to make all scenarios extend from the `vmm_scenario` class, but there is no virtual method for `apply()` in this class, which would be required to write the generic code. The `apply()` method is actually implemented in the concrete class generated by the `` `vmm_scenario_gen `` macro, and has a specialized type for the channel in the arguments.

However, it is possible to write a wrapper for a "family" of single-stream scenarios generated by the `` `vmm_scenario_gen `` macro, using the class types created by the macro. Figure 12 shows an example of a wrapper that can be used for any scenario derived from `atm_cell_scenario`. When `execute()` is called by the multi-stream generator, the wrapper will randomize the original single-stream scenario and apply its items to the appropriate channel.

The wrapper should not perform a grab/ungrab of the channel, as this could change the behaviour of the original scenario, which might be designed to interleave transactions with other scenarios. Of course, the behaviour could be changed anyway, by the surrounding channel grabbing in the multi-stream environment.

The other reason not to put a grab in the wrapper is that it can cause the `apply()` to be blocked from its channel, stopping it from being able to complete. This can happen if the legacy scenario implements its own `apply()` method, using the VMM 1.0 style of channel `put()`, where no pointer to the grabber is passed to the channel. This nested call will not be able to obtain access to the channel as the wrapper is the owner and the channel knows of no relationship between the two. If modifying the original scenario is an option, it is very easy to retrofit the `apply()` method to pass the pointer in any `put()` or `sneak()` calls used. This will result in the scenario having the desired child grabbing privileges.

```
// macros creating single-stream related classes
`vmm_channel(atm_cell)
`vmm_scenario_gen(atm_cell, "ATM_CELL")

// wraps any scenario derived from atm_cell_scenario class
class atm_cell_ms_wrapper extends vmm_ms_scenario;
  atm_cell_scenario scenario;
  atm_cell_channel  out_ch;
  ...
  function new(atm_cell_scenario s, atm_cell_channel ch, string name);
    super.new(null);
    define_scenario(name, 0);
    scenario = s;
    /* scenario.set_parent_scenario(this); */ // needed if channel grab used
    out_ch = ch;
  endfunction:new

  task execute(ref int n);
    int unsigned nn = 0;
    scenario.randomize();
    /* out_ch.grab(this); */     // could change behavior of scenario
    scenario.apply(out_ch, nn);  // run child scenario
    /* out_ch.ungrab(this); */
    n += nn;
  endtask:execute
endclass:atm_cell_ms_wrapper

program test;
   ...
   // instance of a class that extends atm_cell_scenario
   my_atm_cell_scenario atm_sc = new;
   // wrap instance in a multi-stream scenario
   atm_cell_ms_wrapper my_ms_atm = new(atm_sc, atm_ch, "ATM_MS");
   ...
   initial begin
     // register wrapped scenario in multi-stream registry
     gen.register_ms_scenario("ATM_WRAPPED_SCENARIO", my_ms_atm);
     ...
  end
endprogram
```

Figure 12: Example Multi-stream Wrapper for a Single-stream Scenario

If however, you do really want the wrapper to grab, it is possible as long as the default `apply()` method is being used. The code generated by `` `vmm_scenario_gen `` using the new VMM 1.1 macro performs a channel `put()` passing a pointer to the grabber. All that is required for correct operation would be to set the parent scenario (so the grab can be inherited), using `set_parent_scenario()`. This can be seen in the `new()` method, commented out, in the Figure 12 listing.

There is a big *disadvantage* to this style of wrapping when in-line constraints (i.e. `randomize () with { ... }`) are required during randomization of the scenario. If the `randomize()` call is made inside the wrapper, there is no scope for applying in-line constraints.

So on the one hand, many scenarios of the same family can be wrapped quite quickly; but on the other, it's only really useful if no in-line constraints are needed. In many cases the solution could be inferior to using the original single-scenario generator in parallel with the multi-stream generator(s).

3.4 Changing Generator Scheme for Scenario Selection

The selection of multi-stream scenarios from the registry is done using the `vmm_ms_scenario_election` class. There is no default scenario in a multi-stream generator, so at least one scenario has to be registered with the generator. The default behaviour of this class is to use a round-robin scheme. An index value is randomized, based on constraints, which then points to a location in an array of scenarios to choose from. This is the same technique used for selecting items in single-stream scenarios.

Changing the behaviour of the election scheme is quite simple. A new class is extended from the `vmm_ms_scenario_election` class, instantiated in the testbench and programmed as the election scheme for the multi-stream generator.

One thing to note is that the generator's `select_scenario` variable will be restored to its original state after construction, when `reset_xactor()` is called with `rst_typ == HARD_RST`. The user will have to ensure that the new election scheme is reprogrammed in this case. Figure 13 shows an example of a typical way to redefine the election scheme, using a distribution.

```
class my_election extends vmm_ms_scenario_election;
  function new();
    // turn off unwanted constraint block in parent class
    this.round_robin.constraint_mode(0);
  endfunction:new

  // add new constraints to distribute selection between
  // index values 0, 1 and 2 with different probabilities
  constraint new_dist {
    select dist {0:=3, 1:=1, 2:=1};
  }
endclass:my_election
...
m_election sel = new;       // new election scheme
                            // assign new election scheme to generator
gen.select_scenario = sel;  // Warning: may be overwritten after a reset_xactor()
```

Figure 13: New Election Scheme Example

Referring to index values in the multi-stream scenario set is not very informative and relies on knowing the position of specific scenarios in list. It would be much nicer if it were possible to refer to the logic names of the scenarios, as they appear in the registry. That information is in the generator class, so to make this possible some way to access the generator class from the election code would be required.

```
class my_election extends vmm_ms_scenario_election;

  // for access to scenario registry methods
  vmm_scenario_gen gen;

  // stores index of scenario in set against names
  protected int scenario_lookup[string];

  // required to workaround constraints not being able to call non-static functions
  function pre_randomize();
    super.pre_randomize();
    scenario_lookup.delete; // create new table each randomize
    foreach(scenario_set[i]) begin
      scenario_lookup[gen.get_ms_scenario_name(scenario_set[i])] = i;
    end
  endfunction: pre_randomize

  // new constraints use array to lookup index values from names
  constraint new_dist {
    select dist {
      scenario_lookup["ATM_ONLY"]    :=3,
      scenario_lookup["CONFIG1"]     :=1,
      scenario_lookup["BIG_PACKETS"] :=1
    };
  }
endclass:my_election
...
my_election sel = new;      // new election scheme
my_election.gen = gen;      // pointer back to scenario generator
gen.select_scenario = sel; // Warning: may be overwritten after a reset_xactor()
```

Figure 14: Using Scenario Names in Election Scheme Example

Figure 14 illustrates an implementation of the same election redefinition as Figure 13, but instead of using the raw index values in the constraint, the logical names of the registered scenarios are used. This requires the new election class to store a pointer to the generator, and build its own lookup, mapping scenario name to the index in the `scenario_set[$]` queue. The lookup is built each time the election scheme is randomized, in case scenarios are added or removed from the registry. However, if this is guaranteed *not* to happen, it need only be done once and does not need to be in `pre_randomize()`.

The lookup is required to work around a limitation in VCS (2008.10), where only static functions can be used in constraints like this. Using an array of dynamic data is fine though. Otherwise the code in Figure 15 *could* have been used. The `scenario_lookup` array is not required, because the generator registry functions are being called directly, where the final return value is the index to the matched scenario in the `scenario_set[$]` queue.

```
// This code does NOT work
constraint new_dist {
  select dist {
    gen.get_ms_scenario_index(gen.get_ms_scenario("ATM_ONLY"))     := 3,
    gen.get_ms_scenario_index(gen.get_ms_scenario("CONFIG1"))      := 1,
    gen.get_ms_scenario_index(gen.get_ms_scenario("BIG_PACKETS")) := 1
  };
}
```

Figure 15: Functions in Constraints (not currently working)

There is a better solution to avoid having to store a reference to the generator in the election class, but it involves calling the function `vmm_ms_scenario::Xget_context_genX()`. As described in Section 3.2, this function is *intended* to be local, but there is no other way to access the associated variable. The function returns a reference to the generator the multi-stream scenario is registered with. Assuming all scenarios in the `scenario_set[$]` queue (a member of the election class), are registered with the same generator, the function would only have to be called on the first item in the set.

Using this function is a neater solution, but is *not recommended*, purely because it goes against the intent for the function and there is a risk this interface could change in the future. This issue has been reported as a potential bug.

Another way of avoiding the need for a reference back to the generator is to iterate around the election class `scenario_set[$]` queue and use the name information stored in the scenarios themselves. These are the names originally defined for the multi-stream scenarios, not the logical names used in the generator registry, which can be different. In this case, queue locator methods can be used to find an index that matches a particular expression. The expression can match, for example, the scenario kind name(s).

It's still not possible to use this technique directly in the constraint, as the locator method return results in a queue. Wrapping the search in a function runs into the same static function limitation as before, so it will still be necessary to build some sort of lookup. It is also still necessary to refresh the lookup values for each randomize in case scenarios have been added or removed. Figure 16 illustrates an example of using the locator lookup.

In the context of a multi-stream scenario generator, it is still usually more useful to make the election scheme select on the registry's logical names. It is a bit more work, but operates at the appropriate level of abstraction.

There is of course a potential performance impact using the basic implementation described in Figure 14, especially if the there are lots of scenarios registered with the generator. However, the number of multi-stream scenario selections is typically small, compared to the number of data items generated within a scenario, so the impact overall may be trivial.

```
class my_election extends vmm_ms_scenario_election;
  ...
  int result[$];
  int scenario_lookup[string];
  ...
  function pre_randomize();
    super.pre_randomize();
    scenario_lookup.delete;
    // find index matching expression. Note that there may be more than
    // one kind defined for a scenario
    result = scenario_set.find_index(i) with (
               i.scenario_name(my_ms_scenario::ATM_ONLY) == "SC_ATM_ONLY");
    scenario_lookup["SC_ATM_ONLY"] = result[0];
    ...
  endfunction:pre_randomize
  ...
endclass:my_election
```

Figure 16: Finding Index in scenario_set for Constraints Using Locator Methods

3.5 Do Multi-stream Scenarios Replace Single Stream?

Now that there is a choice to use multi or single-stream scenarios, it's reasonable to ask if single-stream scenarios should be deprecated. Single and Multi-stream scenario generators are very similar.

Although multi-stream scenarios can operate down at the data item level, single-stream scenarios already have a built-in list of data items, and are randomized at that level when using a single-stream scenario generator. This does not happen by default in a multi-stream scenario. The behaviour has to be implemented in the `execute()` method. This is slightly more work.

The real reason for using a multi-stream scenario is to be able to coordinate stimulus going to multiple channels. If this is not required, single stream scenarios work well and can still be used later in a multi-stream environment. Both single and multi-stream generators can exist together in the same testbench and can even share channels. Single-stream scenarios can now also grab channels.

Single-stream scenarios still offer a quicker solution when multiple stimulus stream control is not required. Atomic scenarios and the built-in item list randomization of single-stream scenarios offer a fast way to create randomized transactions of a specific type.

4 Conclusions

The VMM has been upgraded to include the capability to generate and control multiple stimulus streams. The new features allow users to generate more sophisticated stimulus, and more precisely guide the flow of randomized transactions throughout the testbench. This level of control over stimulus is required, as the interaction between the multiple interfaces, present in today's designs, contribute significantly to the functional space that needs to be verified, and is where many of the bugs are lurking.

The new features are backward compatible with the previous single-scenario scheme, and also provide a way to use legacy single-stream scenarios in a multi-stream environment, where channel grabbing is deployed. This means that current VMM testbenches have an easy migration path to the new features. In some cases the original single-stream generators remain as-is, with additional multi-stream scenario stimulus being layered on top.

The new features also bring the VMM up to a comparable level with other multi-stream capable methodologies. This is useful, since interoperability between the methodology libraries is becoming more important. The SystemVerilog user base is growing rapidly, as is the requirement for interoperable third party Verification IP.

5 References

[1] VMM Standard Library User Guide (VMM 1.1 release)

[2] VMM 1.1 std_lib source code

[3] "Understanding Layered Stimulus Generation", Verification Now 2008 Conference (Verilab Internal)

[4] "Multi-stream Scenarios, Enhancing Stimulus Generation in the VMM", Verilab, BSNUG2008