Reset Testing Made Simple with UVM Phases

Brian Hunter, Cavium
Ben Chen, Cavium
Rebecca Lipon, Synopsys, Inc.

Cavium
San Jose, California, USA

www.cavium.com

Synopsys, Inc.
Mountain View, California, USA

www.synopsys.com

ABSTRACT

Reset testing is a crucial element of functional sign-off for any chip. However, correctly synchronizing architectural components of the verification environment to different reset conditions is difficult. There has been no standard method for making scoreboards, drivers and monitors enter and exit reset states cleanly, or kill complex stimulus generation processes gracefully; it is common to see reset testing that does not achieve self-checking autonomy forcing engineers to rely on inefficient techniques such as visual inspection.

Handling these complexities with a company-wide framework requires a well-coordinated effort by all team members. UVM's phase jumping capabilities and its native ability to kill phase-related threads and sequences now allows companies to deploy a standard model for reset testing. This paper will explore UVM-compliant methodologies and best practices for idle, active, soft, and multi-domain reset testing based on experience deploying UVM in the networking domain.

Table of Contents

Table of Figures

1. Introduction

Reset testing is a crucial element of functional sign-off for any chip. The architectural components of the entire verification environment need to be correctly synchronized to be made aware of the reset condition. Scoreboards, drivers and monitors need to be tidied up, and the complex stimulus generation needs to be killed gracefully.

Handling these complexities with a company-wide framework requires a well-coordinated effort by all team members and is often incompatible with externally developed IP.

Now with UVM's phase jumping capabilities and its native ability to kill phase-related threads and sequences, we can deploy an industry-wide standard model for reset testing. This paper will explore UVM-compliant methodologies and best practices for idle, active, soft, and multi-domain reset testing based on experience deploying UVM in the networking domain.

2. Resetting Components

Before exploring how phase jumps can be used in reset testing, you must first prepare the many UVM components that will be affected.

The Reset Driver

Developing a driver that drives a reset signal high or low is a trivial task. Many test benches skip this step and use a simple `initial` block instead. Using a UVM driver, however, permits it to be in sync with the system wide phases. The following driver drives the active-low reset signal to which it is attached during the `reset_phase`, waits a random time, takes the signal out of reset, and releases its objection.

```
// class: rst_drv_c
class rst_drv_c extends uvm_driver;
   `uvm_component_utils_begin(rst_drv_c)
      `uvm_field_string(intf_name,           UVM_ALL_ON)
      `uvm_field_int(reset_time_ps,          UVM_ALL_ON | UVM_DEC)
   `uvm_component_utils_end

   // var: intf_name
   string intf_name = "rst_i";

   // var: reset_time_ps
   // The length of time, in ps, that reset will stay active
   rand int reset_time_ps;

   // Base constraints
   constraint rst_cnstr { reset_time_ps inside {[1:1000000]}; }

   // var: rst_vi
   // Reset virtual interface
   virtual rst_intf rst_vi;

   function new(string name="rst_drv", uvm_component parent=null);
      super.new(name, parent);
   endfunction : new
   virtual function void build_phase(uvm_phase phase);
      super.build_phase(phase);
      // get the interface
      uvm_resource_db#(virtual rst_intf)::read_by_name("rst_intf", intf_name, rst_vi)
   endfunction : build_phase

   virtual task reset_phase(uvm_phase phase);
      phase.raise_objection(this);
      rst_vi.rst_n <= 0;
      #(reset_time_ps * 1ps);
      rst_vi.rst_n <= 1;
      phase.drop_objection(this);
   endtask : reset_phase
endclass : rst_drv_c
```

Figure 1. Reset Driver Example

One could further generalize the driver by making it configurable for active-high resets, by having it drive X during the `pre_reset_phase`, or in other ways that suit your organization's needs.

Resetting Monitor Components

Components such as monitors that attach to signaling interfaces should be designed to be phase-independent because they are intended to mimic other real devices in the system. These components should watch the reset signal associated with their interface and reset themselves accordingly. A typical pattern for this is shown in the example monitor's `run_phase` below.

```
class mon_c extends uvm_monitor;
   `uvm_component_utils(mon_c)

   ...

   virtual task run_phase(uvm_phase phase);
      forever begin
         @(posedge my_vi.rst_n);

         fork
            monitor_items();
         join_none

         @(negedge my_vi.rst_n);
         disable fork;
         cleanup();
      end
   endtask : run_phase

  virtual task monitor_items();
    forever begin
      …
    end
  endtask : monitor_items()

endclass : mon_c
```

Figure 2. Reset-Aware Monitor Example

The run phase first waits for the positive edge of the `rst_n` signal to indicate that reset is complete. The `monitor_items` task and any other supporting tasks are designed to run in forever loops. Upon seeing the negative edge of the active-low reset signal, the disable fork statement will kill the `monitor_items` task and the cleanup function will reset any of the class's fields which track state. The whole task is wrapped in a forever block so that it loops back and is able to monitor more items once the reset event is finished.

Coordinating the Reset of Drivers and Sequencer

You may find that the driver, the sequencer, and their currently running sequences will squawk with errors if they are not synchronized properly. UVM requires that the sequencer first stop its sequences and then the driver must be certain to not call item_done on any outstanding sequences. However, the order that a simulator executes threads in the various components is indeterminate. To synchronize these operations, the containing agent has a pre_reset_phase such as the following:

```
class agent_c extends uvm_agent;
   `uvm_component_utils(agent_c)
   sqr_c sqr;
   drv_c drv;
   ...
   virtual task pre_reset_phase(uvm_phase phase);
      if(sqr && drv) begin
         sqr.stop_sequences();
         ->drv.reset_driver;
      end
   endtask : pre_reset_phase
endclass : agent_c
```

Figure 3. Pre-reset Phase in UVM Agent

In this design pattern the driver contains an event called `reset_driver` that *immediately* kicks out of the driver's item-handling task(s). The driver's `run_phase` is similar to that of the monitor, but waits on the `reset_driver` event instead:

```
class drv_c extends uvm_driver;
   `uvm_component_utils(drv_c)

   event reset_driver;
   ...
   virtual task run_phase(uvm_phase phase);
      forever begin
         @(posedge my_vi.rst_n);
         fork
            driver();
         join_none

         @(reset_driver);
         disable fork;
         cleanup();
      end
   endtask : run_phase

   virtual task driver();
      forever begin
         ...
      end
   endtask : driver
endclass : drv_c
```

Figure 4. Example of Handling Reset in a Driver

Avoiding the Letter X

One thing to note about all reset testing is that while it is perfectly reasonable to set the reset and the clock to X at time zero, you may get into trouble if you do this during subsequent reset phases. Cavium's global environment artificially suppresses all RTL errors and warnings at time zero and keeps them suppressed until the end of the pre-reset phase. When you begin setting clocks and/or resets to X, these X values propagate throughout the design at an indeterminate rate and the assertions throughout the design whose errors were previously suppressed may now suddenly trigger, causing tests to fail.

3. Reset Testing with Phase Jumps

Idle Reset Testing

The simplest form of reset testing is idle testing. When all stimulus has drained out of the device, all scoreboards are quiet, and everything has quiesced, send the device back into reset and do it all over again.

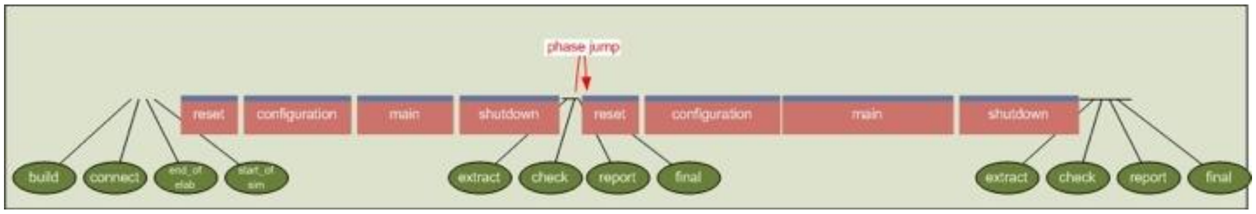

Figure 5. Diagram of Idle Reset Testing with Phase Jump

This testing is made easy because the manner in which the DUT reacts should be highly predictable. Just about all testbenches should be able to implement this manner of testing with a test that looks similar to the one shown here:

```
class idle_reset_test_c extends basic_test_c;
   `uvm_component_utils(idle_reset_test_c)

   // field: run_count
   // The number of times the test has run so far
   int run_count;

   function new(string name="idle_reset", uvm_component parent=null);
      super.new(name, parent);
   endfunction : new

   virtual function void phase_ready_to_end(uvm_phase phase);
      super.phase_ready_to_end(phase);

      if(phase.get_imp() == uvm_shutdown_phase::get()) begin
         if(run_count == 0) begin
            phase.jump(uvm_pre_reset_phase::get());
            run_count++;
         end
      end
   endfunction : phase_ready_to_end
```

```
endclass : idle_reset_test_c
```

Figure 6. Idle Reset Test Example

During the final_phase, the test executes a phase jump back to the pre_reset_phase. This triggers the reset driver shown above (rst_drv_c) to apply the reset signal and all run-time phases are run through again.

Idle Reset Testing with Run Count

At Cavium, we provide a global environment that provides facilities that must be present in all test benches. Cavium's global environment has been outfitted with an automatic way to perform idle reset testing. The test writer must merely set its run_count variable to a value higher than 1, and the test bench automatically runs through all the run-time phases that number of times.

Active Reset Testing

Applying a reset signal while stimulus traffic is flying throughout the DUT is also fairly straightforward due to UVM's phase jumping technique. The complexity lies in how each UVM component reacts to reset.

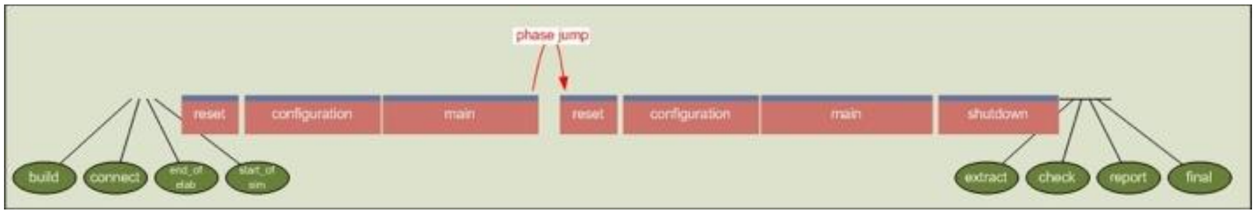

Figure 7. Diagram of Active Reset Testing with Phase Jump

First, here is an example active reset test:

```
class active_reset_test_c extends basic_test_c;
   `uvm_component_utils(active_reset_test_c)

   // field: hit_reset
   // Clear this after the reset event to ensure that it only happens once
   bit hit_reset = 1;

   // field: reset_delay_ns
   // The amount of time, in ns, before applying reset during the main phase
   int unsigned reset_delay_ns;

   function new(string name="active_reset", uvm_component parent=null);
      super.new(name, parent);
   endfunction : new

   // Ensure that the register block is reset
   virtual task reset_phase(uvm_phase phase);
      reg_block.REG_BLOCK.reset("HARD");
   endtask : reset_phase

   virtual task main_phase(uvm_phase phase);
      fork
         super.main_phase(phase);
      join_none
      if(hit_reset) begin
         phase.raise_objection(this);
         std::randomize(reset_delay_ns) with { reset_delay_ns inside {[1000:4000]}; };
         #(reset_delay_ns * 1ns);
         phase.drop_objection(this);
         phase.get_objection().set_report_severity_id_override(UVM_WARNING, "OBJTN_CLEAR", UVM_INFO);
         phase.jump(uvm_pre_reset_phase::get());
         hit_reset = 0;
      end
   endtask : main_phase
endclass : active_reset_test_c
```

Figure 8. Active Reset Test Example

The fork..join_none construct allows the basic test's main_phase to run as normal, in case it does anything important.

When a phase jump occurs, all running phase tasks, their children, and all of their local variables will be wiped clean. Also scrubbed away are any running sequences that were *not* launched during the `run_phase`. Components such as scoreboards that retain some state in their fields will need to clear themselves anytime they enter the pre-reset or reset phases.

Because UVM's phase objection emits a warning when a phase jump occurs, the code suppresses this warning using the objection's set_report_severity_id_override function.

Soft Reset Testing

Your company's definition of soft reset may vary but typically it involves register writes that are meant to clear the device under test in some way. Assuming that a device's expected behavior during a soft reset is very much the same as a hard reset, the use of the phase jump to the reset phase is the best course of action as it leads to minimal changes to what works. In that case, the only difference between the soft reset and an active hard reset is that the reset driver must be disabled and replaced by a register write instead.

More challenging is the soft reset that differs from a hard reset in some way. Perhaps packets that are in flight continue to their completion; or interrupts continue to be serviced; or state machines are forced to idle but buffers are not emptied. There are so many ways that a soft reset might differentiate itself from a hard reset that it is impossible to provide a foolproof design pattern for all occasions.

However, if a phase jump is not the desired solution then the use of an analysis port to broadcast the event to interested components is best. The following is a scoreboard that marks all outstanding packets as unpredictable upon receipt of a soft reset event via its `write_soft_reset` method. The use of a ternary prediction algorithm in such cases can be very useful as it may be impossible to predict what will happen to in-flight traffic, but the environment still must check that future traffic completes successfully.

```
class sb_c extends uvm_scoreboard;
   `uvm_component_utils(sb_c)

   uvm_analysis_imp_soft_reset #(bit) soft_reset_imp;
   uvm_analysis_imp_rcvd_pkt #(pkt_c) rcvd_pkt_imp;
   pkt_c exp_pkts[$];

   ...

   function void write_soft_reset(bit _reset);
      // mark all outstanding packets as unpredictable
      foreach(exp_pkts[num])
         exp_pkts[num].unpredictable = 1;
   endfunction : write_soft_reset

   function void write_rcvd_pkt(pkt_c _pkt);
      pkt_c exp_pkt = exp_pkts.pop_front();
      if(exp_pkt.unpredictable)
         return;
      else if(exp_pkt.compare(_pkt) == 0)
            `uvm_error(get_full_name(), "Packet Miscompare.")
   endfunction : write_rcvd_pkt
endclass : sb_c
```

Figure 9. Scoreboard with Analysis Port for Soft Reset

Multiple Reset Domains

Just as the RTL can have multiple reset domains, so too can testbench components. By establishing different domains and assigning them to different components, you can jump one domain's phases without changing others.

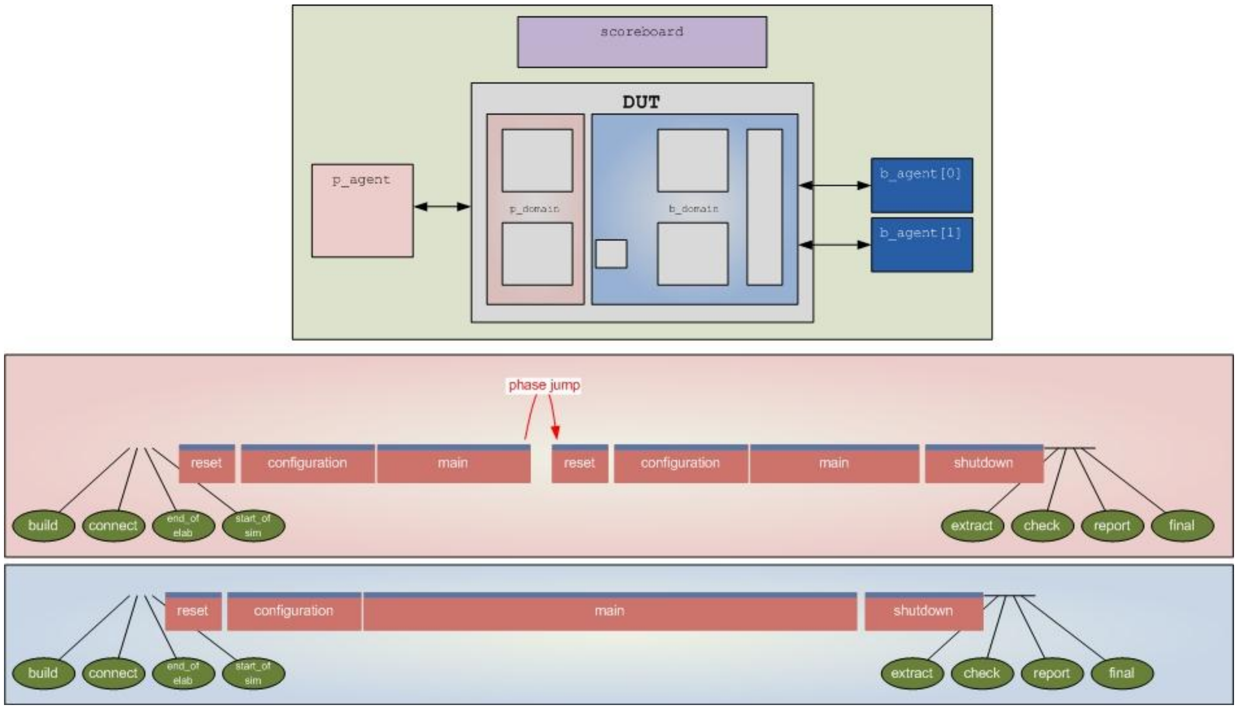

Figure 10. Diagram of Multi-Domain Resets

The diagram above shows how the agent and RTL blocks in the pink "P" domain can undergo an active reset, while the agents corresponding to the blue "B" domain continue on their merry way.

How much domain resets affect the entries in the scoreboard depends upon the architecture of the environment. It is likely that the reset event would need to be communicated to the scoreboard in some fashion.

Setting up and assigning domains is a snap. The following code shows how it would be done for the above scenario.

```
class domain_reset_test_c extends basic_test_c;
   `uvm_component_utils(domain_reset_test_c)

   // field: p_domain
   // A UVM domain that will undergo reset in the middle of the main phase
   uvm_domain p_domain;

   // field: reset_delay_ns
   // The amount of time, in ns, before applying reset during the main phase
   rand int reset_delay_ns;
   constraint delay_cnstr { reset_delay_ns inside {[100:1000]}; }

   function new(string name="domain_reset", uvm_component parent=null);
      super.new(name, parent);
   endfunction : new

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

      p_domain = new("p_domain");

      // assign the p_agent, and all its sub-components, to the p_domain
      p_agent.set_domain(p_domain, .hier(1));
   endfunction : build_phase

    virtual task main_phase(uvm_phase phase);
      fork
         super.main_phase(phase);
      join_none

      if(run_count == 0) begin
         phase.raise_objection(this);
         randomize();
         #(reset_delay_ns * 1ns);
         phase.drop_objection(this);
         phase.get_objection().set_report_severity_id_override(UVM_WARNING, "OBJTN_CLEAR", UVM_INFO);
         p_domain.jump(uvm_pre_reset_phase::get());
         run_count++;

         // tell scoreboard that a reset occurred
         -> scoreboard.p_domain_reset;
      end
   endtask : main_phase
endclass : domain_reset_test_c
```

Figure 11. Multi-domain Reset Example

This test resembles the active reset test except for a few important differences. By default, all components (including this test) are assigned to the *UVM* domain. This test creates a second domain (`p_domain`) and assigns the `p_agent` and all its sub-components to this new domain. The remaining components stay within the UVM domain.

As the simulation progresses through to the main phase, both domains remain synchronized, until the test tells the `p_domain` to jump back to the pre-reset phase. As with soft resets, an analysis imp can alert interested components such as scoreboards.

```
class sb_c extends uvm_scoreboard;
   `uvm_component_utils(sb_c)

   uvm_analysis_port_p_reset#(bit) p_reset_port;

   function new(string name, uvm_component parent=null);
      super.new(name, parent);
      p_reset_port = new("p_reset_port", this);
   endfunction : new

   function void write_p_reset(bit _reset);
     // "clean up" entries in the p_domain
   endfunction : write_p_reset
...
```

Figure 12. Example of Scoreboard Analysis Port for Multi-domain Reset

Layered stimulus may also merit adding a new phase domain to your environment. One such example is the layered stimulus inherent to a MAC in a resilient system. The testbench may simulate a link-level reset but keep intact those packets that are in-flight at the logical level. Or, your environment might try to maintain TCP connections in a self-healing mesh network.

Re-Randomizing After Reset

Upon resetting the device, you will probably want to re-randomize the DUT's configuration registers to re-simulate in a different mode. However, if you are tempted to re-randomize and re-create the component hierarchy, then you are out of luck. The UVM1.1 environment cannot jump back to each component's build phase. If your testbench builds components based upon random conditions, then the first run-through must have an identical architecture as all subsequent runs. Likewise, if the environment architecture is somehow dependent on the DUT's configuration registers, then any re-randomization must be constrained accordingly.

By identifying and separating those random variables that must maintain their values across resets you can effectively re-randomize the system over and over again. We call these variables *structural* variables.

In the example environment below, the mode variable in the configuration object is a structural variable because the type of agent that is built depends upon its value.

```
class env_c extends uvm_env;
   `uvm_component_utils_begin(env_c)
      `uvm_field_object(cfg, UVM_REFERENCE)
   `uvm_component_utils_end

   // var: agent
   // Base class of the agent that will be created
   agent_c agent;

   // var: cfg
   // Configuration object holding many random variables
   cfg_c cfg;

   function new(string name="env", uvm_component parent=null);
      super.new(name, parent);
   endfunction : new

   virtual function void build_phase(uvm_phase phase);
      super.build_phase(phase);
      case(cfg.mode):
         XAUI: agent = xaui_agent_c::type_id::create("agent", this);
         RXAUI: agent = rxaui_agent_c::type_id::create("agent", this);
         SGMII: agent = sgmii_agent_c::type_id::create("agent", this);
      endcase
   endfunction : build_phase

   ...
endclass : env_c
```

Figure 13. Example of Environment dependent on Structural Variable from Configuration Class

The base test that creates both the environment and the random configuration class stabilizes the cfg class's structural variable after its initial randomization, but permits the re-randomization of the cfg class whenever a pre_reset_phase occurs. This allows the other controls and environment knobs that populate the cfg class to have different values during every pass without affecting the environment architecture. We choose to perform this functionality in a virtual function, so that descendent tests may vary constraint modes as necessary.

```
class base_test_c extends uvm_test;
   `uvm_component_utils(test_c)

   rand cfg_c cfg;
   env_c env;
   reg_block_c reg_block;

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

      // create the reg_block
      reg_block = reg_block_c::type_id::create("reg_block", this);

      // create and randomize cfg class
      cfg = cfg_c::type_id::create("cfg");
      cfg.reg_block = reg_block;
      randomize_cfg();

      // create env and populate its reference to the cfg class
      env = env_c::type_id::create("env", this);
      uvm_config_db#(uvm_object)::set(this, "env", "cfg", cfg);
   endfunction : build_phase

   // randomize cfg, but afterwards tell structural variables to be stable
   function void randomize_cfg();
      randomize();
      cfg.mode.rand_mode(0);
   endfunction : randomize_cfg

   // re-randomize all of the test knobs during a reset
   virtual task pre_reset_phase(uvm_phase phase);
      randomize_cfg();
   endtask : pre_reset_phase
endclass : base_test_c
```

Figure 14. Example of Base Test dependent on Structural Variable from Configuration Class

The configuration class contains both the random mode variable and the DUT's CSR register block, given to it by the test.

```
class cfg_c extends uvm_object;
   `uvm_object_utils(cfg_c)

   rand reg_block_c reg_block;
   rand mode_e mode;
   // ensure that the device’s mode always lines up with the chosen mode
   constraint mac_mode_cnstr {
      cfg.reg_block.MAC_MODE.value == mode;
   }

   function new(string name=”cfg”);
      super.new(name);
   endfunction : new
endclass : cfg_c
```

Figure 15. Example of Configuration Class with Structural Variables

With this framework, the DUT can be reset and re-configured multiple times in a single simulation.

4. Conclusion

UVM’s phase-jumping and native ability to kill phase-related threads and sequences has enabled a standard methodology for reset testing. In this paper we detailed a UVM-compliant framework for how to make scoreboards, drivers and monitors enter and exit reset states cleanly, as well as kill complex stimulus generation processes gracefully. This paper also proposed a UVM-compliant method for idle, active, soft, and multi-domain reset testing with examples from the networking domain. We hope to see these methods deployed on internally and externally developed IP for easier integration and validation of chips.

As with any open standard, UVM will continue to evolve. The Accellera committee has identified that the phasing mechanism may change in future releases. While nothing formal is proposed, we hope adherence to backward compatibility will ensure that jumps backward in time, and jumps forward (to the extract phase at a minimum) are maintained. We view this functionality as critical as detailed throughout this paper. Of course if the UVM standards committee does deprecate functionality pertaining to run-time phases and phase jumping, it will be because a different mechanism has been added to the standard and if that occurs we look forward to revisiting the topic of reset testing utilizing the state-of-the-art methodologies at that time.

5. References

[1] All code was simulated on VCS F-2011.12-3 and G-2012.09-2 versions, with UVM version 1.1b.