

SVA Encapsulation in UVM

enabling phase and configuration aware assertions

Mark Litterick
 Verilab GmbH.
 Munich, Germany
 mark.litterick@verilab.com

Abstract— Complex protocol checks in Universal Verification Methodology Verification Components are often implemented using SystemVerilog Assertions; however, concurrent assertions are not allowed in SystemVerilog classes, so these assertions must be implemented in the only non-class based "object" available, the interface construct. This creates problems of encapsulation (since the verbose assertion code clutters the interface definition) and isolation (since the assertions depend on aspects of class configuration and operation). This paper demonstrates several pragmatic solutions for encapsulation and operation of assertions including mechanisms to make the assertions aware of the configuration and phases of the class-based verification environment.

Keywords—UVM; SVA; SystemVerilog; Interface;

I. INTRODUCTION

Complex protocol checks in Universal Verification Methodology (UVM) Verification Components (UVC) are often implemented using SystemVerilog Assertions (SVA) [1]; however, concurrent assertions are not allowed in SystemVerilog classes [2], so these assertions must be implemented in the only non-class based "object" available - the UVC interface. While this is not a disaster, since the protocol checks are usually closely related to interface signal activity, it does represent pretty poor software encapsulation.

Interfaces can get very cluttered with clocking blocks, signal declarations, modport definitions, interface access methods and so on. Adding what are usually quite verbose and specialized property definitions and assertion statements can overload the interface file content. In addition this assertion code is not really intended for the interface user, since the primary function of the interface is to encapsulate signal communication between the testbench classes and the Device Under Test (DUT) in a clear and concise manner – so the assertion code represents noise in that respect since it is really just part of the overall UVC checker functionality.

Furthermore, for the assertions to be constantly in tune with the rest of the class-based verification environment settings, the actual assertion code may need to be aware of configuration settings (such as `checks_enable` and

configuration object field settings) and UVM phases (in order to use the configuration settings correctly).

This paper presents a thorough overview of these requirements for powerful and adaptive SVA encapsulation and demonstrates several practical solutions based on the SystemVerilog interface construct that can be made to be UVM phase aware and automatically make use of the UVM configuration settings in the associated class-based environment.

II. UVM ENVIRONMENT

A. Testbench Overview

In order to put the checker responsibilities into context, let us first consider the generic setup for a UVM verification environment as shown in Figure 1.

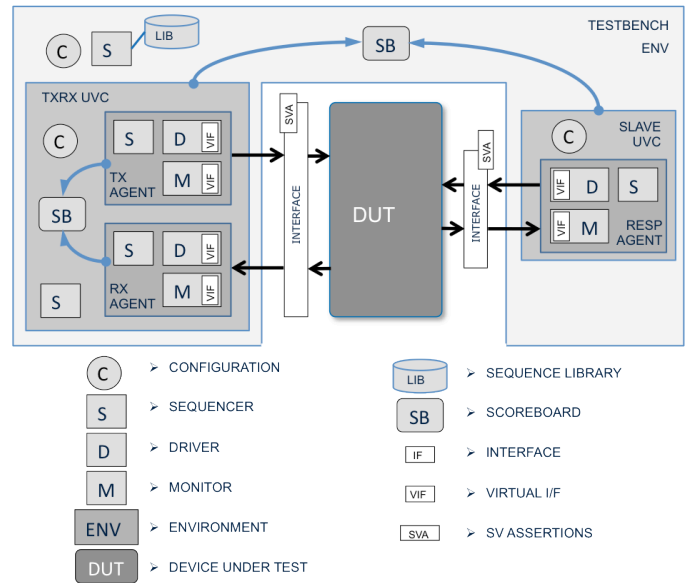


Figure 1. UVM Verification Environment

Figure 1 illustrates a basic verification environment, which comprises of two interface verification components (the actual protocol for which is not specified here since it is not relevant

to the discussion). In this case the sub-components have the following responsibilities:

- *stimulus* is provided by sequencers and drivers
- *functional checks* are performed by scoreboards, monitors & interfaces
- *functional coverage* is collected by monitors
- *debug messages* are generated by all components

B. Check Requirements

In addition to providing constrained-random stimulus, functional coverage and debug mechanisms such as messaging, UVCs must also provide automatic functional checking in order to meet corresponding verification requirements and validate DUT behavior. There are three types of checks typically performed in UVCs and each type is handled by a different component in the environment:

- Signal protocol checks using concurrent property assertions written in SVA and located in interfaces
- Transaction content, decoding and functional checks are performed by monitors
- Transaction comparison and relationship checks are done in scoreboards

Note that two scoreboards are shown in Figure 1, one is checking a relationship between transactions published by two interface UVCs, and the other is entirely contained within a single UVC, for example to validate the relationship between the transmit and receive transactions in a full duplex interface.

Local scoreboard transaction comparisons, functional checks for transaction content, signal protocol behavior and timing checks all belong to the UVC even though SystemVerilog language limitations mean that we cannot encapsulate the concurrent property assertions for the protocol timing checks inside the class world. This distribution of check responsibilities is indicated in Figure 2.

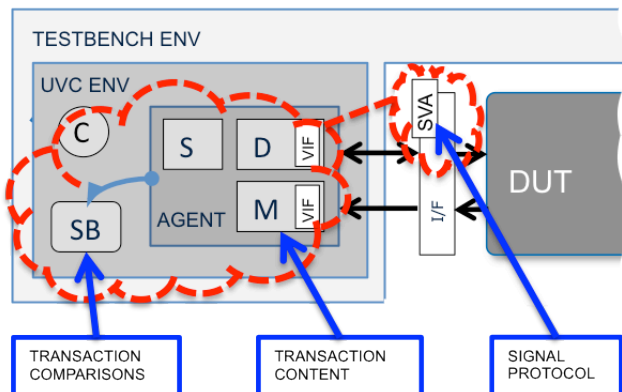


Figure 2. Distributed UVC Checks

C. Configuration and Control of Checks

Since UVM verification components and environments are highly configurable and the signal protocol checks are responsible for implementing part of the verification requirements and capability of the UVC, the protocol checks may also have to adapt to configuration settings in the environment to produce the expected behavior.

In the examples shown in this paper it is assumed that the virtual interface and configuration object are created and added to the configuration database by some higher-level components, for example enclosing environment or base-test component, as shown in Figure 3.

```

class my_base_test extends uvm_test;
  ...
  virtual my_interface vif;
  my_config cfg;
  ...
  function void build_phase(...);
    super.build_phase(phase);
    ...
    // propagate vif from tb to env
    if (!uvm_config_db#
        (virtual my_interface)::get
        (this, "", "TBIF", vif))
      `uvm_fatal("VIFERR",
        "no my_interface TBIF in db")
    uvm_config_db#
        (virtual my_interface)::set
        (this, "*", "vif", vif);

    // create cfg using the factory
    cfg = my_config::type_id::create
        ("cfg", null);
    if (!cfg.randomize())
      `uvm_warning("RNDFLD", "...");

    // propagate cfg down hierarchy
    uvm_config_db#(my_config)::set
        (this, "*", "cfg", cfg);
  endfunction
  ...
endclass
  
```

Figure 3. Base-Test Architecture

In addition the *checks_enable* flag may be added to the configuration database by some higher-level component, such as an actual test that requires to do stress testing or error injection. The corresponding database configuration code would be similar to that shown in Figure 4.

```

class my_stress_test
  extends my_base_test;
  ...
  function void build_phase(...);
    super.build_phase(phase);
    ...
    // disable checks for this test
    uvm_config_db#(bit)::set
      (this, "*", "checks_enable", 0);
  endfunction
  ...
endclass

```

Figure 4. Example Test Override

III. SVA ENCAPSULATION

A. Why Encapsulate?

The primary reasons for encapsulation are to co-locate all the SVA code in a single place and to avoid cluttering the UVC interface specification with the homeless protocol checks that cannot live elsewhere in the class hierarchy. Rather than use tick-include for the assertion code it is better to put it in a code construct that can be instantiated inside the interface.

B. Interface Encapsulation

In order to instantiate the checker inside the interface construct, the checker itself must be implemented as a SystemVerilog interface and not a module, since modules cannot be instantiated inside interfaces but other interfaces can.

The SVA checker template shown in Figure 5 defines only the required ports for the checker and has placeholders for additional attributes that are discussed later in the paper. Bear in mind that normally there are many sequences, properties and assertions definitions as well as supporting code to make the assertions easier to write, debug and maintain.

```

interface my_protocol_checker(
  // signal port definitions
  input logic    CLK,
  input logic    REQ,
  input logic    ACK,
  input logic [7:0] DATA
);
  // local variable definitions
  // support code for assertions
  // property definitions
  // concurrent assertions
endinterface

```

Figure 5. Protocol Checker Template

If the ports for the checker are defined as inputs using the same names as the host interface logic definitions, then the checker can be instantiated using implicit port connections (i.e. the "*" notation). Note that the protocol checker ports can

be a subset of the local signal definitions in the enclosing interface and implicit port connection still works – the actual subset used remains hidden from the enclosing interface definition. The interface definition for the UVC can therefore focus on signal and *modport* definitions and instantiate the checker in a single line as shown in Figure 6.

```

interface my_interface;
  // local signal definitions
  logic    CLK;
  logic    REQ;
  logic    ACK;
  logic [7:0] DATA;
  logic    OTHER;
  ...
  // modport signal groups and directions
  modport driver
    (... , output REQ, input ACK, ...);
  modport monitor
    (... , input  REQ, input ACK, ...);

  // instantiate protocol checker
  my_protocol_checker
    protocol_checker(. * );
endinterface

```

Figure 6. Interface With Protocol Checker Instance

IV. CONFIGURATION AWARENESS

A. Configuration & Control Fields

In reality, many protocol checkers will also need access to factory controlled UVC knobs (such as *checks_enable*) and *configuration object* fields in order to behave correctly. Specifically we need to use these configuration settings inside property definition and assertion statements. Notice however that concurrent assertions cannot directly use class member variables or methods (but supporting code can) so we need to copy the required configuration object fields to local variables in the interface and use these variables in the properties.

In order to illustrate the use of configuration object fields we will make use of the configuration object class declaration (partially) shown in Figure 7. In this case there are four fields in the configuration object, but we only intend to use a subset of these. It is assumed that the configuration object is randomized with additional constraints or an appropriate factory type overload in order to constrain the configuration object field values appropriately for a particular instance of the UVC.

```

class my_config extends uvm_object;
  // configuration fields
  rand my_speed_enum speed_mode;
  rand int unsigned max_value;
  rand bit data_en;
  rand bit etc;
  // constraints
  constraint c_max_value {
    max_value <= 255;
  }
  // utility and field macros
  `uvm_object_utils_begin(my_config)
    `uvm_field_enum(.., speed_mode, ...)
    `uvm_field_int (max_value, ...)
    `uvm_field_int (data_en, ...)
    `uvm_field_int (etc, ...)
  `uvm_object_utils_end
endclass

```

Figure 7. Configuration Object

The corresponding local configuration variables in the protocol checker interface code are shown in Figure 8, with two sections still to be clarified. In this case the protocol checker makes use of only some of the fields from the configuration object, i.e. the required fields. This level of detail is correctly encapsulated within the protocol checker construct and is hidden from whatever external constructs are communicating with the checker – the external code does not care which subset of the configuration fields are used.

```

interface my_protocol_checker(...);
  // control knobs
  bit checks_enable = 1;
  // local vars for required cfg fields
  my_speed_enum cfg_speed_mode;
  int unsigned cfg_max_value;
  bit cfg_data_en;

  // (1) properties and assertions

  // (2) update local variables from cfg

endinterface

```

Figure 8. Protocol Checker Local Variables

B. Example Concurrent Assertions

In order to illustrate the use of configuration settings inside the assertions, let us also assume the following property specifications are part of the verification requirements for the verification component:

- REQ always gets ACK before another REQ is observed and the ACK must come within a specified range depending on the configured speed mode.
 - fast-mode = ACK must be between 1 and 4 CLK delays

- slow-mode = ACK must be between 3 and 10 CLK delays
- DATA must never exceed the configurable max_value during ACK if data_en is configured. When data_en is negated, or at anytime other than ACK, there are no restrictions on DATA.

In addition to the functional requirements, we will disable all assertions if the *checks_enable* flag is negated in the same way we do for other checks in the UVC.

For the first of these protocol checks, we require different timing behavior based on the current configuration settings. Note however that range values in cycle delay (*##N*) and repetition [**A:B*] operators must be constants. One possible solution here is to specify different timing sequence definitions and choose the appropriate one based on the *cfg_speed_mode* condition as shown in Figure 9.

```

sequence s_fast_transfer;
  REQ ##1 !REQ[*1:4] ##0 ACK;
endsequence

sequence s_slow_transfer;
  REQ ##1 !REQ[*3:10] ##0 ACK;
endsequence

property p_transfer;
  @(posedge CLK)
  disable iff (!checks_enable)
  REQ |->
    if (cfg_speed_mode ==
        MY_SPEED_FAST)
      s_fast_transfer;
    else // MY_SPEED_SLOW
      s_slow_transfer;
endproperty

a_transfer:
  assert property (p_transfer)
  else $error("illegal transfer");

```

Figure 9. Property Assertion For Transfer Check

Note that the concurrent property definition is using the local variables from the interface and not an actual configuration object class since dynamic class members are not allowed in SystemVerilog Assertions. There is more detail on this in subsequent sections of the paper.

The second protocol check requires us to validate the data range but only during ACK cycles and when data responses are enabled. This can be implemented by using the *cfg_max_value* field in the property *consequent* (the right-hand-side of implication operator “|->”) to check the range values, and by adding the *cfg_data_en* field to disable the property as shown in Figure 10.

```

property p_data_max;
  @(posedge CLK)
  disable iff (!checks_enable ||
              !cfg_data_en)
  ACK |->
    (DATA <= cfg_max_value);
endproperty

a_data_max:
  assert property (p_data_max)
  else $error("illegal ACK data");

```

Figure 10. Property Assertion For Data Check

C. Updating Configuration

Now we need to consider how to get the configuration information out of the class world and into the protocol checker interface variables. Care is required here since the interfaces themselves are not UVM phased-components, and we need to access the configuration database after the necessary build phases have completed in order to get the correct values. Several possibilities exist to achieve this including applying a software API to set the values of the fields hierarchically after the build phase is complete and also adding code to the interface to control the phasing automatically - both of which are described in the following sections.

V. METHOD API

A. API for SVA Configuration

The simplest approach to ensuring that the configuration and control fields in the interface SVA checker are accurate after the factory controlled build of the UVM environment is to provide a method-based API. The UVC class-based components have a responsibility to maintain the checker configuration at appropriate stages in the lifetime of the test.

As shown in Figure 11, each of the checker components (e.g. monitor and scoreboard), agents and the UVC environment has access to configuration object fields and control knobs, shown by the © symbol. During factory-controlled build of the class-based components these configuration fields are setup using factory creation, type overrides, and the configuration database tables. Each component calls *get()*, either implicitly via the field macros, or explicitly using the *uvm_config_db*, in order to get a handle to the correct configuration settings shown by the thin (red) dashed arrows. Once build is complete, the class-world can push the configuration over to the corresponding instance of the SVA protocol checker in the module domain, as shown by the thick (blue) arrow in Figure 11, and discussed in detail in the following sections.

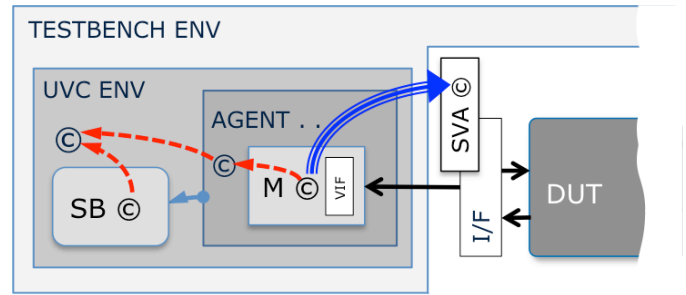


Figure 11. SVA Configuration Using API

B. API Implementation

Initializing the checker knobs and configuration objects using the software API is straightforward; the first step is to add *set_config* and *set_checks_enable* methods to the protocol checker as shown in Figure 12.

```

import ovm_pkg::*;
import my_pkg::*;

interface my_protocol_checker(...);
  ...
  // set required local fields
  function void set_config(my_config cfg);
    cfg_speed_mode = cfg.speed_mode;
    cfg_max_value  = cfg.max_value;
    cfg_data_en    = cfg.data_en;
  endfunction

  // set checks_enable flag
  function void set_checks_enable(bit en);
    checks_enable = en;
  endfunction
  ...
endinterface

```

Figure 12. Property Checker With Method API

Note that it is not enough to just add a declaration of the configuration object and to procedurally reference it to an external configuration object since the *uvm_object* class fields cannot be used directly in the assertions; we need to tell the checker when to update its local configuration fields based on completion of the UVM build phase in the class domain. Note also that by passing the complete configuration object in the method call we effectively isolate the checker-specific subset of configuration fields inside the API, which is desirable for software encapsulation.

Likewise, although we have a *checks_enable* field in the protocol checker, this is not registered with the factory and will not automatically update based on any *set_config_int()* method calls in the parent UVC environment. Hence we need to tell the checker interface when to update the value after build phase is complete.

We also require similar methods in the actual UVC interface as shown in Figure 13, since this is the virtual interface assigned in the parent verification component (i.e. the UVC does not know about the SVA encapsulation in a lower level sub-interface).

```

interface my_interface;
...
function void set_config(my_config cfg);
    protocol_checker.set_config(cfg);
endfunction

function void set_checks_enable(bit en);
    protocol_checker.set_checks_enable(en);
endfunction
...
endinterface

```

Figure 13. Interface API

C. Using API to Control SVA Configuration

Then we are in a position to set the checker configuration field values anytime after the build phase is complete and the virtual interfaces are connected (but before any related signal activity occurs) - for UVM this is most logically done in the *end_of_elaboration_phase* for the associated monitor as shown in Figure 14.

Note that the monitor has some additional work to do in order to propagate the configuration settings into the interface via the virtual interface handle (“vif”).

D. Handling Configuration Changes With API

If the configuration object is not static throughout the run-phase, but contains some pseudo-static configuration fields which are updated in response to run-time stimulus, then we also need to adapt the local configuration fields in the checker accordingly. With a method API approach, the monitor has responsibility for calling the corresponding *set_config* method in the interface when required.

Note that if the monitor developer forgets to update the SVA configuration, or does not do it at the right point in time, then the resulting false assertion failures can be very hard to debug.

```

class my_monitor extends uvm_monitor;
    my_config cfg;
    bit checks_enable = 1;
    virtual my_interface vif;

    `uvm_component_utils(my_monitor)

function void build_phase(...);
    ...
    // virtual interface must be provided
    if (!uvm_config_db#
        (virtual my_interface)::get
        (this, "", "vif", vif))
        `uvm_fatal("VIFERR",
            "no my_interface vif in db")

    // config must be provided
    if (!uvm_config_db#(my_config)::get
        (this, "", "cfg", cfg))
        `uvm_fatal("CFGERR",
            "no my_config cfg in db")

    // checks_enable may be overloaded
    void' (uvm_config_db#(bit)::get
        (this, "",
            "checks_enable", checks_enable));
endfunction

function void end_of_elaboration_phase(..
    ...
    // set interface config after build
    vif.set_config(cfg);
    vif.set_checks_enable(checks_enable);
endfunction
...
endclass

```

Figure 14. API Use-Case

E. API Summary

The above approach is easy to implement and debug but can get quite messy if there is a lot of configuration objects, individual control knobs or many layers of hierarchy to deal with. In addition it forces the higher-level classes to perform additional tasks and be aware of the lower-level checker interface API requirements.

VI. PHASE AWARE CONFIGURATION

A. Automatic SVA Configuration

An alternative approach is to construct the SVA protocol checker such that the interface is phase-aware and can control its own settings from the UVM environment with no additional demands placed on the higher-level classes. This achieves good encapsulation of the checker and isolation of related checker requirements.

In Figure 15, the © symbols represent configuration object fields and control knobs as before, but in this case the SVA protocol checker automatically updates its own configuration object based on the actual factory controlled build of the class-based environment using phase-aware constructs in the interface. The mechanism for achieving this is discussed in the following sections.

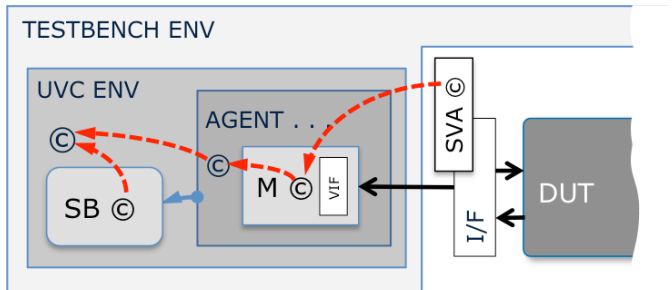


Figure 15. Automatic SVA Configuration

B. Phase-Aware Interface

The SystemVerilog interface construct is not a UVM phased-component; however, we can make a class declaration for a phased-component inside the interface as shown in Figure 16. Since this class is defined inside the interface it can see all the member variables of the interface (such as *cfg*, *cfg_speed_mode* and *checks_enable*). The phase methods for this embedded class will execute along with other components' phase methods in the UVC class world since it is derived from *uvm_component*.

Note however that since this class is declared inside an interface and the interface is ultimately instantiated in the testbench module, then this class forms a top-level component in the UVM instance hierarchy, under *uvm_top*, and in parallel with the main testbench class hierarchy. Normally we rely on the *build_phase* working in a top-down manner; this means we can configure all of the child components in any order and the current parent environment is completely built before child components are created. However, the net result of this setup is that the UVM phases in the two class environments under *uvm_top* execute their phases in parallel and this can result in a race scenario in that the interface class can complete the *build_phase* before the associated UVC completes its *build_phase*.

```

import ovm_pkg::*;
import my_pkg::*;

interface my_protocol_checker(...);
...
my_config cfg;
...

class checker_phaser
  extends uvm_component;

function new(
  string name="",
  ovm_component parent=null
);
  super.new(name, parent);
endfunction

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

  // config must be provided
  if (!uvm_config_db#(my_config)::get
    (this, "", "cfg", cfg))
    `uvm_fatal("CFGERR",
      "no my_config cfg in db")

  // checks_enable may be overloaded
  void'(uvm_config_db#(bit)::get
    (this, "",
      "checks_enable", checks_enable));

  // copy required cfg fields
  cfg_speed_mode = cfg.speed_mode;
  cfg_max_value  = cfg.max_value;
  cfg_data_en    = cfg.data_en;
endfunction

endclass

// create unique instance of phaser class
// (at the top-level under uvm_top)
checker_phaser m_phase =
  new($psprintf("%m.m_phase"));
...
endinterface

```

Figure 16. Phase-Aware Protocol Checker

Two possible workarounds for this parallel phasing problem are available; the first is to configure the database from the testbench module prior to calling the *run_test* task (which starts the UVM phases) – but this is not really a good encapsulation since we want to control settings from inside class components not the testbench module. The alternative is to wait until the class world has completed its build and

connect phases before the protocol checker class updates its own configuration fields and control flags.

Since each of the top-level *build_phases* execute in parallel and they all complete, albeit in an unspecified order, before the *connect_phase* is started in any component, we can choose to update the configuration after *build_phase* but before the time-consuming *run_phase* (when we need the checks to be active). A logical solution therefore is to use *end_of_elaboration_phase* in the protocol checker class to do the configuration updates based on any class based modifications that occurred during the *build_phase* or *connect_phase* of the associated UVC as shown below.

In order to support multiple instances of the associated UVC and interface, it is necessary to provide a unique name for each *checker_phaser* class in the *uvm_top* structure. This can be achieved by constructing the *checker_phaser* class with a unique name derived from the full hierarchical pathname for the enclosing interface using the *%m* format specifier as shown at the bottom of Figure 16. This also means that any UVM messages generated by the *checker_phaser* class have an accurate hierarchical pathname in the corresponding log files, which improves debug capability.

C. Controlling SVA Configuration

Since the *uvm_component* classes inside the checker interfaces are not inside a parent *uvm_component*, but are located at the top-level of the UVM hierarchy, the base-test (or derived tests) need to increase the scope of the configuration database lookup strings by using “null” rather than “this” for the *cntxt* parameter when setting configuration object and *checks_enable* flags as shown in Figure 17.

```
uvm_config_db#(my_config)::set
  (null, "*", "cfg", cfg);

uvm_config_db#(bit)::set
  (null, "*m_phase", "checks_enable", 0);
```

Figure 17. Example *uvm_config_db::set*

D. Handling Configuration Changes Automatically

If the configuration object is not static throughout the run-phase, but contains some pseudo-static configuration fields which are updated in response to run-time stimulus, then we need to add a process to the protocol checker interface which automatically detects changes in the configuration class and updates the local fields accordingly.

Since the configuration object is not yet constructed during elaboration of the static module code where the interface is instantiated, it is not possible to monitor the configuration object, *cfg*, directly in an *always* or *always_comb* construct (i.e. “*always_comb cfg_speed_mode = cfg.speed_mode;*” results in a run-time error). An alternative solution is to perform a similar operation inside the *run_phase* task for the

checker_phaser class as shown in Figure 18. Note that since this code in the checker interface is not part of a concurrent assertion, it can directly look at the configuration object class fields to determine if any of the required fields have been modified and transfer these updates to the local variables for use in the assertions.

```
class checker_phaser ...;
  ...
  task run_phase(uvm_phase phase);
    super.run_phase(phase);
    // monitor required cfg for updates
    forever begin
      @(cfg.speed_mode or
         cfg.max_value or
         cfg.data_en)
      begin
        cfg_speed_mode = cfg.speed_mode;
        cfg_max_value  = cfg.max_value;
        cfg_data_en    = cfg.data_en;
      end
      `uvm_info("CFGUPDATE",
               "cfg update detected", UVM_LOW)
    end
  endtask
  ...
endclass
```

Figure 18. Example Run-Time Configuration Snooping

E. Phase-Aware SVA Summary

With this technique no additional external API is required and the UVM components in the corresponding UVCs do not have to communicate with the protocol checker directly. In fact the protocol checker functionality is well encapsulated and hidden from the class-based components.

VII. CONCLUSION

This paper presented practical suggestions for encapsulating SVA-based protocol checkers in such a way that they can be instantiated into the UVM verification component interface and illustrated some suitable methods to allow the assertions to be aware of configuration and factory generated values in the class-based verification component. The techniques discussed in this paper have been successfully used on several OVM and UVM projects at different clients and using different simulation tools. Constructing SVA protocol checkers using these mechanisms results in much better software encapsulation for these important checks and provides no drawbacks over the other ad-hoc methods observed in operation at clients.

REFERENCES

- [1] Universal Verification Methodology (UVM), www.uvmworld.org
- [2] SystemVerilog, IEEE Std 1800-2009, www.ieee.org