SVA in a UVM Class-based Environment

This article demonstrates how SVA complements a UVM class-based environment. It also demonstrates how the UVM severity levels can be used in all SVA action blocks instead of the SystemVerilog native severity levels.

SystemVerilog

Ben Cohen Last Updated Feb 2013

Classes, Integrated Environment, SystemVerilog, UVM, February 2013 - Volume 9 Issue 1, Verification Horizons, Beginner

Mark as completed

INTRODUCTION

Verification can be defined as the check that the design meets the requirements. How can this be achieved? Many verification approaches have been used over the years, and those are not necessarily independent, but often complementary. For example, simulation may be performed on some partitions while emulation in other partitions. The verification process in simulation evolved throughout history from simple visual (and very painful) examination of waveforms with the DUT driven by a driver using directed tests, to transaction-based pseudo-random tests and checker modules.

This has led to the development of class-based frameworks (e.g., e, VMM, OVM, UVM) that separate the tests from the environment, are capable of generating various concurrent scenarios with randomness and constraints and are capable of easily transferring data to class-based subscribers for analysis and verification. In parallel, designers and verification engineers have improved the detection and localization of bugs in their design code using assertion-based languages (e.g., PSL, SVA).

Those recent verification technologies need not be independent. Specifically, verification approaches consist of several methodologies that work well, particularly when combined in symphony. UVM is a class-based methodology that, because of its nature, tends to stress the use of scoreboarding, which is supported by the analysis ports. Though scoreboarding solves many verification tasks, SystemVerilog Assertions (SVA) provide another complementary and supporting set of solutions that can speed up the verification and model building processes and can provide additional verification data to UVM for further actions.

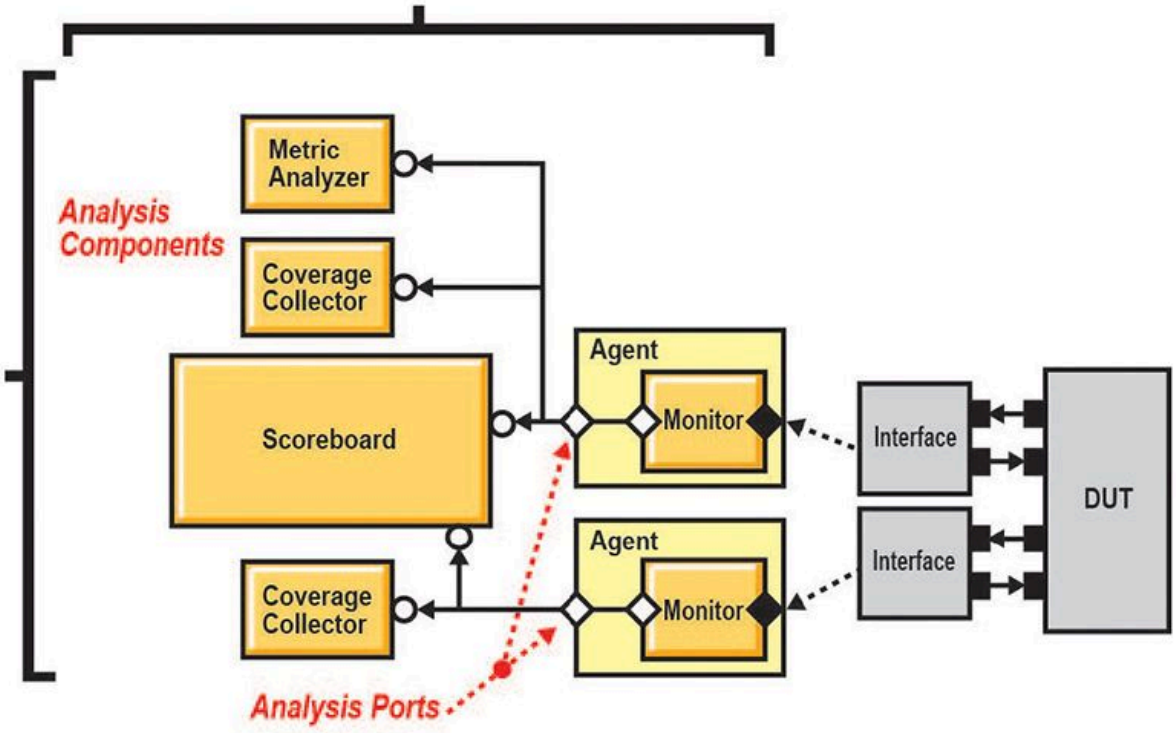
Specifically, SVA provides a simpler definition of temporal design properties for verification and coverage; for the support of covergroup sampling-controls that are based on a sequence of events; and for localized detection of errors and effective debug via on-demand waveform/ transaction recording.

This article demonstrates how SVA complements a UVM class-based environment. It also demonstrates how the UVM severity levels can be used in all SVA action blocks instead of the SystemVerilog native severity levels.

SCOREBOARD IN UVM

Figure 1.0 demonstrates a typical verification model. The scoreboard is part of the analysis group and is most often used for the verification of the design behavior.

Figure 1.0 The UVM Analysis Layer



Scoreboards are analysis components that collect the transactions sent by the monitor and perform specific analysis activities on the collection of transactions. Scoreboard components determine whether or not the device is functioning properly. The best scoreboard architecture separates its tasks into two areas of concern: prediction and evaluation. The prediction is based on the requirements. A predictor model, sometimes referred to as a "Golden Reference Model", receives the same stimulus stream as the DUT and produces known good response transaction streams. The scoreboard evaluates the predicted activity with actual observed activity on the DUT. Thus, in essence, a scoreboard is a piece of checking code that keeps track of the activities of the design and calculates the expected calculated responses based on scenarios and compares those against actual.

SVA AS COMPLEMENTARY/SUBSTITUTION APPROACH

The concept of assertions is not new; any verification methodology or code "asserts" that the design meets the properties of the requirements. Thus, broadly speaking, one can state that the UVM scoreboarding approach provides assertions about the correctness of the design, and in its own rights is a coding style/ language/methodology with its macros, data structures, and libraries. Assertion languages, such as PSL and SVA, evolved to address the need to specify the temporal requirement properties of a design in an executable language that is easy to write and read, and yet meets the needs of the design space for both simulation and formal verification. Thus, in a manner similar to frameworks, SVA is a language that expresses

behavioral properties and supports directives to do something with those properties, such as assert for their correctness, assume for the input space constraints, and cover for the coverage of those properties.

The unique capabilities of SVA in a verification environment

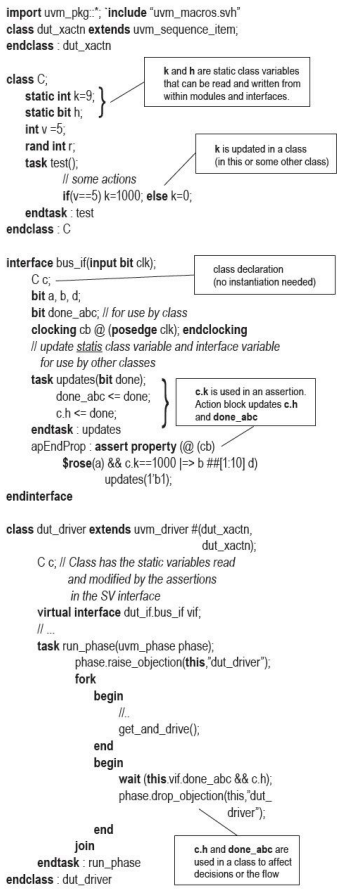
SVA provides two types of assertions: immediate and concurrent. Immediate assertions are executed like a statement in a procedural block and do not need a clocking event to trigger the assertion, as that is controlled by the procedural block when the code reaches the assertion. An example of an immediate assertion commonly used in UVM is something as the following:

```
assert($cast(rsp_item, req_item.clone()) else
`uvm_error("driver",$sformatf("%m : error in
casting rsp_item")));
```

Concurrent assertions are temporal, are based on clock semantics, and use sampled values of static variables. Assertions can also access static variables defined in classes; however, access to dynamic or rand variables is illegal. Concurrent assertions are illegal within classes, but can only be written in modules, SystemVerilog interfaces, and SystemVerilog checkers2. As will be shown in the next subsections, assertions provide many benefits in a verification environment, and if values of class variables are needed for use in concurrent assertions, those variables can be copied onto class-static variables or in SystemVerilog virtual interfaces (to be connected later to actual interfaces), and then used in the assertions. Assertions provide several unique capabilities, as described in the following subsections.

Detection of temporal violations and use of action block to modify flow SVA handles cycle accurate checks more effectively than with an application using scoreboarding because the verification of the signals is done at every cycle rather than at the completion of the data gathering process. The action block of an assertion can modify SystemVerilog interface variables or static class variable on either a pass or a fail action of the assertion. Those variables can then be used by a class instance to drop an objection or to modify the flow of the class. Figure 2.0 demonstrates these concepts.

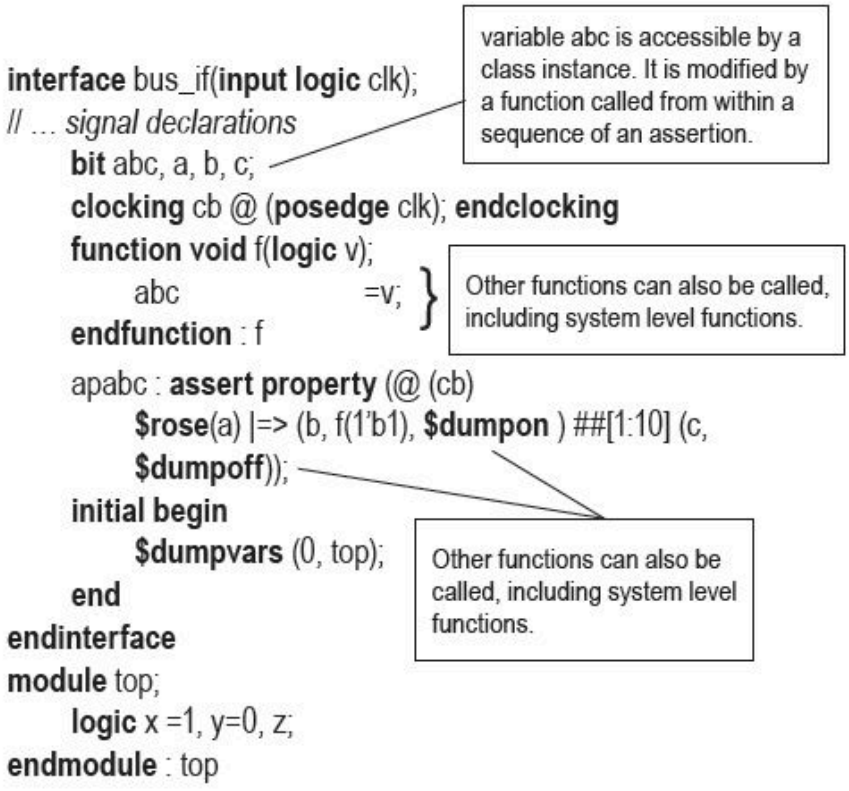
Figure 2.0 Using the results of an assertion to stop the simulator3
http://SystemVerilog.us/verizons/sc2uvm.sv



Sequence match to modify interface variables or trigger system level functions

Assertions can be used to call functions or system functions during any cycle of a sequence upon the success of an evaluation during that cycle. This capability also allows for changes to the SystemVerilog interface variables or static class variable for use by the class. For example, in Figure 3.0:

Figure 3.0 Variables and function calls from assertion statements



Minimizing use of scoreboarding with assertions

In class-based model, the scenario may modify class variables (e.g., the rand variables) that are used in the verification properties of the DUT. Those variables are often transferred to the scoreboard for the necessary computations and comparisons. However, assertions offer another, easier to use alternative that in some cases may also work in conjunction with scoreboards. For example, if a new occurrence of signal a occurs, then DUT signal b must remain at 0 for a number of cycles, as determined by the scenario (the class). Signal a should then go to a logical 1 within 5 cycles after that.

To achieve this with assertions, a task in the class copies the value of that number of cycles into a SystemVerilog interface (or a static class variable, as discussed previously). The assertion in the interface makes use of that copied variable and other signals of the interface to determine compliance to the requirements. Any detected error is reported by the assertion; that error can also be reported back to the class via an interface variable (or a static class variable). That variable is set or reset by the action block. Below is a snapshot of the key elements of this concept and assertion in figure 4.0.

Figure 4.0 Use of values of class variables and assertions in interfaces
(Code and simulation results are in
http://SystemVerilog.us/verizons/test_if_assertions.zip)

```
interface b_if (input logic clk);
     logic a, b, ab_error; // <- ab_error to be used
                                by the class
     clocking cb @ (posedge clk); endclocking
     int count; // <- Value set from the class
     // If $rose(a), then b==0 for count cycles,
                    count >= 1.
     // This is then followed by b==1 within 5 cycles
                                    after that.
     property p_ab_min;
          int count_v;
          ($rose(a), count_v=count) |=>
          (!b, count_v-=1'b1)[* 1:$] ##1 count_v==0
          ##[1:5] b;
     endproperty : p_ab_min
     ap_ab_min: assert property(@ (posedge clk)
      p_ab_min)
        ab_error <= 1'b0; else ab_error <= 1'b1;
endinterface : b_if
```

For more complex problems, a scoreboard data structure can keep track of conditions (driven and expected), and the assertions in the SystemVerilog interfaces can then determine the correctness of DUT's responses. For example, assume that a scoreboard keeps track of the expected caching of data in the variable abc is accessible by a class instance It is modified by a function called from within a sequence of an assertion. Other functions can also be called, including system level functions. DUT's memory based on a caching algorithm.

The scoreboard can keep track of an expected miss or hit (cache_hit), and transfer that information, along with the value of any expected data (expected_data) into a SystemVerilog interface. Assertions in that interface can then verify the correct response from the DUT and log in any detected errors (cache_hit_error, cache_miss_error), which can then be read by the class instance.

Figure 5.0 Use of scoreboard values from class variables for use in assertions.

```
interface c_if (input logic clk);
        bit rd, wr, data_valid;
        bit cache_hit_error, cache_miss_error; // <- for
             use by class-based verification unit
        bit cache_hit; // <- for this read address,
                supplied by scoreboard
                in cycle after the rd
        logic [15:0] mem_data;
        logic [15:0] expected_data; // <- supplied by
                scoreboard
        logic [31:0] addr;
        clocking cb @ (posedge clk); endclocking
        ap_cache_hit_access: assert property(@ (cb)
                                rd |=> cache_hit |->
                            mem_data==expected_data)
            cache_hit_error <= 1'b0; else cache_hit_
                        error <= 1'b1;
       ap_cache_miss_access: assert property(@ (cb)
                        rd |=> !cache_hit |->
                !data_valid[*4] ##1 data_valid)
            cache_miss_error <= 1'b0; else cache_
                        miss_error <= 1'b1;
endinterface : c_if
```

Enhanced coverage sampling

SystemVerilog assertion statements enable the control of transaction recording from action blocks or sequence match item, i.e., from any clocking event when a variable is true. This can be used in the setting of values within a module or interface, or to call any function, including the call of the sample() function to sample group coverage. The following example demonstrates the concepts.

Figure 6.0 Use of assertions for covergroup sampling

```
typedef enum {NOP, ADD, SUB, MULT, JMP, MOV,
              READ, WRITE,
    IDLE} instr_e;
typedef enum {FETCH, DECODE, EXECUTE} mode_e;
typedef enum {PC, IR1, IR2, REGA, REGB, MAR,
              REG_FILE, STK} resource_e;
interface cpu_if(input logic clk);
    clocking cb @ (posedge clk); endclocking
    instr_e instr;
    resource_e resource;
    mode_e mode;
    logic[15:0] addr;
    covergroup instr_cg;
        InstXrsc : cross instr, resource;
    endgroup : instr_cg
    instr_cg instr_cg1 =new();
    covergroup addr_cg @(clk);
        addr_cp : coverpoint addr
        { bins instruction = {[0:255]};
        bins data = {[256:32767]};
        }
    endgroup : addr_cg
    addr_cg addr_cg1 =new();
    sequence q_fetch; @ (clk)
        mode==FETCH ##1 (1, instr_cg1.sample(),
                           addr_cg1.sample());
    endsequence : q_fetch
    cover property (q_fetch); // ← Enables the sampling
                in cycle following the mode==FETCH
endinterface : cpu_if
```

Sampling of cover group from a sequence

UVM SEVERITY LEVELS IN SVA

UVM provides several macros that resemble the SystemVerilog severity levels, but provide more options, verbosity, and consistency with the UVM verification environment. In our SystemVerilog Assertions Handbook, 3rd Edition, we recommend the use of the UVM severity levels in all SVA action blocks instead of the SystemVerilog native severity levels. For example,

```
string tID="UART ";

ap_MEDIUM: assert property(a) else

     `uvm_info(tID,$sformatf("%m : error in a %b", a),

               UVM_MEDIUM);

ap_handshake : assert property ($rose(req) |=>

                 ##[0:4] ack) else

     `uvm_error(tID, $sformatf("%m req = %0h,

          ack=%0h", $sampled(req),

          $sampled (ack)));
```

See http://systemverilog.us/uvm_hi_low3.pdf for a discussion on this topic along with simulation results.

CONCLUSIONS AND RECOMMENDATIONS

Verification consists of several methodologies that work well, particularly when combined in symphony. UVM is a class-based methodology that tends to stress the use of scoreboarding to compare expected results against actual observations. Though the scoreboarding approach solves many verification tasks, SVA provides another complementary approach that works with UVM to speed up the model building process, to support the class-based environment, and to quickly localize errors at the cycle level.

SVA provides complementary advantages for UVM. This includes the simpler definition of temporal design properties for verification and coverage; the support of covergroup sampling control based on sequence of events; the localized detection of errors and effective debug via on-demand waveform/transaction recording; the control of system level tasks based on sequence of events; and the reporting of errors back to the UVM environment for flow control.

For bus protocol types of verification properties (signal level verification), we recommend SVA at the interfaces instead of using scoreboarding with compare as a verification technique. If a tool supports SystemVerilog checkers (1800-2009 and 2012), use checkers bound to interfaces. Assertion results can be written into the interface variables or class static variables for control and access by the UVM environment. For coverage of temporal properties, use assertions in SV interfaces.

For covergroups that require the completion of temporal sequences as the sampling trigger, use SVA sequences that can transfer the sampling triggers to class variables. Use UVM severity levels in all SVA action blocks instead of the SystemVerilog native severity levels. Use SVA immediate assertions to flag unsuccessful randomization and terminate the simulation run.

END NOTES

1. MENTOR GRAPHICS UVM/OVM documentation verification methodology online cookbook
2. See Chapter 17 of SystemVerilog 1800-2012.
3. SystemVerilog Assertions Handbook, 3rd Edition with IEEE 1800-2012, ISBN 978-0-9705394-3-6 http://SystemVerilog.us

Back to Top

February 2013 Articles

Using Formal Analysis to Block and Tackle

Formal V... Article

This article will explain how we applied formal analysis at the block level, extended this to full chip and describe how we significantly reduced verification time at both the block and chip level. Just like a block and tackle provides a mechanical advantage, the formal connectivity flow provides a verification advantage

Bringing Verification and Validation under One Umbrella

Accelerat... Article

The standard practice of developing RTL verification and validation platforms as separate flows, forgoes large opportunities to improve productivity and quality that could be gained through the sharing of modules and methods between the two. Bringing these two flows together would save an immense amount of duplicate effort and time while reducing the introduction of errors, because less code needs to be developed and maintained.

System Level Code Coverage using Vista Architect and Syste...

Covera... Article

SoC are constantly becoming more and more complex forcing design teams to eke out as much performance as possible just to stay competitive. Design teams need to get it right from the start and can't wait until it's built to find out how it truly performs. This is where System Level Modeling and SystemC/TLM shine.

The Evolution of UPF: What's Next?

Low Power Article

Usage of the Unified Power Format (UPF) is growing rapidly as low power design and verification becomes increasingly necessary. In parallel, the UPF standard has continued to evolve. A previous article1 described and compared the initial UPF standard, defined by Accellera, and the more recent IEEE 1801-2009 UPF standard, also known as UPF 2.0. The IEEE definition of UPF is the current version of the standard, at least for now, but that is about to change.

Top Five Reasons Why Every DV Engineer Will Love the Late...

System... Article

SystemVerilog has become the most widely deployed Verification language over the last several years. Starting with the early Accellera release of 3.1a standard, the first IEEE 1800-2005 standard fueled the widespread adoption in tools and user base. Since 2005 there is no look-back to this "all encompassing" standard that tries to satisfy and do more for RTL Designers and Verification engineers alike.

SVA in a UVM Class-based Environment

SystemVe... Article

This article demonstrates how SVA complements a UVM class-based environment. It also demonstrates how the UVM severity levels can be used in all SVA action blocks instead of the SystemVerilog native severity levels.

The Formal Verification of Design Constraints

Formal V... Article

There are two approaches to the verification of design constraints: formal verification and structural analysis. Structural analysis refers to the type of analysis performed by a static timing tool where timing paths either exist or not based on constant settings and constant propagation.

OVM to UVM Migration, or There and Back Again: A Consultant'... U... Article

This article presents an interesting OVM to UVM migration story where we successfully translated a whole family of verification components from OVM 2.1.2 to UVM, assessed the impact, and then reworked the original OVM code, which was still live in a series of ongoing derivative projects, to make the ongoing translations totally automatic and part of the project release mechanism.

View all articles >