Assertion Based Verification Using HDVL

Kausik Datta & P P Das
Interra Systems (India) Private Limited
{kausikd, ppd}@interrasystems.com

Abstract

Over the past several years verification of large designs are becoming more and more complex – both in terms of the maintaining the code size and in keeping parity between the specification written in English; design written in HDL (typically Verilog / VHDL) and the verification models written in HDL or some proprietary verification language. To alleviate this problem, Accellera has come up with the proposal for standardizing an HDVL – a single language that caters to all the needs for Design (as an HDL) as well as Verification (as an HVL – Hardware Verification Language). SystemVerilog [6] standard where the user can model and verify the correctness of the designs using a unified language whose syntax and semantics is already proven and tested in the industry is being projected as a candidate HDVL. SystemVerilog is a set of major enhancements to the Verilog 2001[1] standard and these enhancements are taken from existing industry standard languages and paradigms including Superlog [3], PSL-Sugar [4], OVA [8] and OVL [7]. In this paper we present an overview of the Assertion based Verification methodology in general and explain, with suitable examples, how can one benefit from using an HDVL for the combined purpose of design as well as verification. It attempts to set the right expectations for an engineer from an HDVL and also illustrates the power of the new paradigm.

1. Introduction

Normally the designers and the verification engineers use HDLs like Verilog not only for the design purpose but also for the functional verification of the designs. To achieve effective functional verification, large testbenches are written using the Verilog simulation controls with tool specific "comment based directives" and the monitors or checkers are placed in Designs, testbenches as well as in the simulator tools. But with this approach, the testing is limited to an ad-hoc basis that finds out only the visible bugs, and does not check the correctness of the design. Consequently, the detection of bugs is often possible only at a later stage of the design flow.

Alternates for the above methodology have been sought in terms of formal verification primarily through equivalence checking. This also has capacity limitations for handling large and complex design both in terms of memory as well as time requirements.

Recently, assertion based functional verification methodology has emerged as a new paradigm where the design / verification engineers can accurately specify what is to be tested and verified in the form of well defined temporal language expressions. These are more abstract than what designers used earlier and can be directly mapped from the functional specification of the design. The assertion based technology not only provides a formal language to the verification engineer, but also provides the flexibility of reusing the vendor's verification models (commonly called, Verification IP) to validate the vendor's design components as used as a part of the design. The model checker tools are used to mathematically prove or disprove the assertions generated from the functional specification of the design.

Synopsys's OpenVera™ Assertions (OVA) [8] and IBM's Sugar [9] are examples of such assertion based languages that have been used in the industry for a while to create the verification models. Though use of such languages have aided verification to a certain extent, being proprietary (and separate from the design language) they have added to the costs of learning and working with new languages. Also, the use of different languages for modeling and verification reduces the simulator performance besides creating communication gaps between the design and the verification engineers.

Accellera, the unified body of Open Verilog International and VHDL International, has proposed to bridge this gap through the introduction of an HDVL – a *Hardware Description & Verification Language* – that can serve the dual purpose of design as well as the functional assertive specification of the design. This new language, called SystemVerilog, is an effort to create a paradigm for assertion-based designs with the help of the building blocks that are already well tested and are used in the industry. This is an enhancement over the Verilog 2001 Standard [1] including the donations mainly from the following well-known languages:

- The Co-Design Superlog Language [3]
- The Synopsys VERA-Lite Hardware Verification Language
- The Synopsys VCS™ DirectC Application Programming Interface (API)
- The Synopsys Assertion Application Programming Interface (API)
- The IBM PSL Assertion Technology (Sugar).

In this paper we present an overview of the Assertion based Verification methodology in general and explain, with suitable examples, how can one benefit from using an HDVL for the combined purpose of design as well as verification. Section 2 reviews Assertions and Section 3 highlights some benefits of using them. Advantages of an HDVL over a separate HDL–HVL combine are discussed in Section 4. Few of the currently available assertion based tools are discussed in Section 5. We conclude in Section 6.

2. What is Assertion?

Assertion is a statement, which is used in the implementation to check design's intended behavior. In general assertion doesn't directly contribute anything to the functionality of the design. Its purpose is to keep consistency between the specification of a design and what is actually implemented.

The benefits of the assertion are as follows:

- Improve Observability and Controllability of the Design
- Capture design specification in a formal way within the RTL design
- Improve the verification efficiency via detecting more bugs
- Improve integration and use of third party design
- Improve communication through documentation

2.1. Improve Observability and Controllability of the Design

In the traditional way any design is verified via the use of simulator with the help of testbench and input stimuli. In order to identify a bug it is required to generate a proper input, which will also propagate the wrong value to the output port of the system from the position of the actual problem. This is commonly known as the **Black Box Verification**.

The assertion mechanisms, on the other hand, catch the wrong output at the point of its occurrence and report error. This way it not only improves the observability of the system but also improves the overall control over the system via reducing the search area for finding out the cause of the bug. This requires the use of **White-Box Verification** – a term that stresses the contrast with the traditional black-box verification.

2.2. Capture Design Specification in a Formal Way within the RTL Design

Assertions are developed based on the design specification and are used in the RTL code to check whether designer's intent is properly implemented or not. This way it helps to keep the design specification itself as part of the RTL code.

2.3. Improve the Verification Efficiency via Detecting More Bugs

Assertions can be verified using different formal verification tools like model checking and also in the simulator environment this helps in finding out the corner cases with new set of inputs, without stopping or fixing existing problems.

2.4. Improve Integration and Use of Third Party Design

Assertions supplied with the Vendor's design in the form of verification IP help the developer integrating the design in the proper way without waiting for vendor's feedback on any failure.

2.5. Improve Communication through Documentation

In addition to finding bug, the assertion properties can be used in the form of documentation of the design.

3. Using Assertions

Assertion, as defined, is a statement to check design's intended behavior. The intended behavior of a design can be defined as set of logical and timing relationship called property. Therefore the assertion can be viewed as composition of three distinct layers dealing with properties in three different ways.

The Boolean layer – Presents the Boolean expressions to be checked

The Temporal layer – Describes the relationship between Boolean expressions over time

The Verification layer – Specifies how the properties are to be checked during verification

For example, in

```
assert always (!(en1 & en2));
```

`!(en1 & en2)` depicts the Boolean expressions to be checked;
`always` states the temporal scope; and
`assert` denotes the verification instruction.

In general the property checking can be done in two different ways based on the type of property

1. Assert or check whether the property is true in a specific time
2. The property is defined to use for getting the functional *coverage* information of the design.

3.1. Expressing Assertions – OVL, PSL & SystemVerilog

Here we use two examples to show how the design intents can be verified within the RTL code with Assertions.

As a vehicle of expressing the assertions and to explain their use in the running examples, we use three languages from Accellera (created from donations from various companies):

— Accellera Open Verification Library [7],
— Accellera PSL-1.01 [4] and
— Accellera SystemVerilog 3.1 [6].

In most places the syntax and semantics of the language used is self-explanatory and we do not get into the details for explaining them. These can be looked up from the references. The main target here is to appreciate the use of assertions and not to 'learn' the languages. We however, annotate at places wherever the same is necessary for clarity.

3.2. Bus Mux – A Simple Combinatorial Design

In the example shown in Figure 1, only one out of the four inputs will go to the output based on the value of the `bus_addr`. That means at any point of time only one bit of the `bus_addr` will be selected.

```
module bus_mux
  #( parameter NO_OF_BUS = 4,
     parameter BUS_WIDTH = 8)
   ( input [BUS_WIDTH-1 : 0]
            bus_vector [0 : NO_OF_BUS-1],
     input  bus_addr [0 : NO_OF_BUS-1],
     output [BUS_WIDTH-1 : 0] bus_out);

  generate
    genvar bus_index;
    for (bus_index = 0;
         bus_index < NO_OF_BUS;
         bus_index++)
    begin: tribuf_bus_inst_block
      genvar bit_index;
      for (bit_index = 0;
           bit_index < BUS_WIDTH;
           bit_index++)
      begin: tribuf_inst_block
        bufif1 tribuf_inst (
        bus_out[bit_index],
        bus_vector[bus_index][bit_index],
        bus_addr[bus_index]);
      end
    end
  endgenerate
endmodule
```

Figure 1. Bus-Mux: A simple combinatorial design

While we write this design, we need to ascertain / verify that only one bit of the `bus_addr` should be one at a time and the rest must be zero (that is, checking for an **'one_hot' scenario**). For example, it will be an error if the value of `bus_addr` is like "`1001`" or "`1110`" etc.

Next we show how this property can be asserted and checked in each of the three languages.

3.2.1. Using OVL. This property can be checked in the following way

```
assert_one_hot #(0, NO_OF_BUS-1)
       single_bit_one_at_a_time (clk,
              reset_n, f(bus_addr));
```

3.2.2. Using PSL. This property can be checked in the following way

```
assert always
       ((f(bus_addr) != 0) && ((f(bus_addr) &
        (f(bus_addr) -1 )) == 1'b0 )
```

3.2.3. Using SystemVerilog 3.1. This property can be checked in the following way

```
property single_bit_one_at_a_time;
    ($onehot(f(bus_addr)));
```

```
endproperty

assert property (single_bit_one_at_a_time);

function [0:NO_OF_BUS-1] f;

/* This is a conversion function, to convert
bus_addr from an one dimensional array to a
variable of width of the dimension size. This
is required to check the value of the variable
using $onehot system function. */

input bus_addr [0:NO_OF_BUS-1];
reg [0:NO_OF_BUS-1] temp;
integer i;
begin
    for(i=0; i<NO_OF_BUS; i = i + 1)
        temp[i] = bus_addr [i];
    f = temp;
end
endfunction
```

3.3. Stack – A Simple Sequential Design

In the example shown in Figure 2, we implement a stack using a one dimensional array (`stack_data`). Each entry of the stack is eight bit wide and the maximum stack size is sixteen. That means for every push / pop operation eight bit value will be set / reset in the array. Another array (`valid`) is used to store the information of the point to which the stack is filled up.

The following properties are part of the design of the stack that needs to be verified here. When we talk of the stack, we imply that these hold all through. Assertions will check help them automatically.

1. *Push and pop may take place simultaneously*
   This means in the next cycle of the push and pop operation, the current value of `data_in` should be the value of the `top_of_stack` and the variable `valid` will be unchanged.

2. *Overflow condition*
   No push operation should take place if the value of MSB of `valid` is `1'b1`.

3. *Underflow condition*
   No pop operation should take place if the value of LSB of `valid` is `1'b0`.

4. *The flush operation should not be done with push or pop*
   If the input `flush` is `1`, other two inputs `push` and `pop` should be always `0`.

5. *Stack size is not over-designed*
   This is coverage checking for the design. The size of the stack should not be over-designed so that a part of it will always be empty.

6. *Flush operation cleans up the storage properly*
   This means in the next cycle of the `flush` operation both the variables `valid` and `stack_data` should be reset.

```
module stack
  #(parameter DEPTH  = 16, /* Max size of
stack */
    parameter WIDTH  = 8   /* 8 bit wide */ )
   (input wire [WIDTH - 1 : 0] data_in,
    input wire push, pop, flush, clk, rst_n );

  localparam STKWIDTH       = DEPTH*WIDTH;
  reg [DEPTH-1 : 0] valid;
  reg [STKWIDTH-1 : 0] stack_data;
  wire [WIDTH-1 : 0] top_of_stack =
stack_data[WIDTH-1 : 0];

  always @(posedge clk or negedge rst_n)
  begin
    case ( {push, pop, flush} )
      3'b100:
      begin
        valid <= { valid[DEPTH-2 : 0], 1'b1 };
        stack_data <=
           { stack_data[STKWIDTH-WIDTH-1 : 0],
             data_in };
      end
      3'b001:
      begin
        valid <= { DEPTH{1'b0} };
        stack_data <= { STKWIDTH{1'b0} };
      end
      3'b010:
      begin
        valid <= { 1'b0, valid[DEPTH-1 : 1] };
        stack_data <= { {WIDTH{1'b0}},
             stack_data[STKWIDTH-1 : WIDTH] };
      end
      3'b110:
      begin
        stack_data[WIDTH-1 : 0] <= data_in;
      end
    endcase
  end
endmodule
```

Figure 2. Stack: A simple sequential design

Next we show how the above properties can be asserted and checked in each of the three languages.

3.3.1. Using OVL. These properties can be checked in the following way

1. Push and pop may take place simultaneously

```
assert_next push_and_pop_good (clk, rst_n,
        (push & pop),
        (prev_value(data_in)== top_of_stack &&
         prev_value(valid) == valid));
```

Here the function `prev_value` is not part of the OVL language. This should be implemented by the user to get the value of the input of the previous cycle.

2. Overflow condition

```
assert_never overflow (clk, rst_n,
       (push & !pop &
       valid[DEPTH-1] == 1'b1));
```

3. Underflow condition

```
assert_never underflow (clk, rst_n,
       (!push & pop & valid[0] == 1'b0));
```

4. The flush operation should not be done with push or pop

```
assert_always flush_only (clk, rst_n,
       (flush & !push & !pop));
```

5. Stack size is not over-designed

Coverage checking is not supported in OVL. Hence this property cannot be checked for in OVL.

6. Flush operation cleans up the storage properly

```
assert_next flushing_everything (clk, rst_n,
       (flush & !push & !pop),
       ((valid == DEPTH'b0) &&
       (stack_data == STKWIDTH'b0)));
```

3.3.2. Using PSL. These properties can be checked in the following way

1. Push and pop may take place simultaneously

```
assert always (push & pop |=>
       prev(data_in) == top_of_stack &&
       prev(valid) == valid)@(posedge clk) ;
```

2. Overflow condition

```
assert never (push & !pop &
       valid[DEPTH-1]== 1'b1) @(posedge clk);
```

3. Underflow condition

```
assert never (!push & pop &
   valid[0] == 1'b0) @(posedge clk);
```

4. The flush operation should not be done with push or pop

```
assert always
       (flush & !push & !pop) @(posedge clk);
```

5. Stack size is not over-designed

```
cover forall N in { 0 : DEPTH } :
   { rose(valid[N]) };
```

6. Flush operation clean up the storage properly

```
assert always ((flush & !push & !pop) |=>
       ((valid == DEPTH'b0) &&
       (stack_data == STKWIDTH'b0)) );
```

3.3.3. Using System Verilog 3.1. These properties can be checked in the following way

1. Push and pop may take place simultaneously

```
property push_and_pop_good;
  @(posedge clk)
    (push & pop |=>
       $past(data_in) == top_of_stack &&
       $stable(valid));
endproperty

assert property (push_and_pop_good);
```

Here we have used two system functions `$past`, which returns value of the input of the previous cycle and `$stable`, which returns true if the value of the input is unchanged. If this property fails the assert statement will raise an error message.

2. Overflow condition

```
property overflow;
  @(posedge clk)
    (push & !pop & valid[DEPTH - 1] == 1'b1);
endproperty

assert property (overflow) $error("Overflow");
```

Here if the overflow condition passes the assert message will raise the error.

3. Underflow condition

```
property underflow;
  @(posedge clk)
    (!push & pop & valid[0] == 1'b0);
endproperty

assert property (underflow)
       $error("Underflow");
```

Similar to the overflow condition, here also the error message will come if the condition is true.

4. The flush operation should not be done with push or pop

```
property flush_only;
  @(posedge clk)
    (flush & !push & !pop);
endproperty

assert property (flush_only) else
       $error("Illegal flush operation");
```

Here the error message will come if the condition fails as the message is give in the else block.

5. Stack size is not over-designed

```
property check_push;
  @(posedge clk)
    (push & !pop |=> $onehot($past(valid) ^
valid));
endproperty

property not_over_designed;
    @(posedge clk)
        ($rose(valid[DEPTH - 1]));
endproperty

assert property (check_push);

cover property (not_over_designed);
```

This coverage checking is done in two phases. First via using an assert message which will raise error if and only if every push operation doesn't change only one bit of the variable `valid`. That means here the coverage checking is okay if the MSB of `valid` is never set to `1` and the stack is never filled up to its maximum capacity.

6. Flush operation cleans up the storage properly

```
property flushing_everything;
    @(posedge clk)
        ((flush & !push & !pop) |=>
                ((valid == DEPTH'b0) &&
                (stack_data == STKWIDTH'b0)) );
endproperty

assert property (flushing_everything);
```

4. HDVL = HDL + HVL

In the above examples we looked at the use of assertions in design and observed their use through three languages – two currently in use and one being proposed. The examples show that barring a few scenarios (like OVL does not support cover property) all the languages have similar expressibility for assertions. However, they still differ in terms of finer details and in their support mechanisms in various tools. We take a comparative look at the support for assertions between OVL, PSL and SystemVerilog 3.1 in Table 1.

The language OVL [7] is designed as a library comprising a set of predefined modules which can be instantiated within the design to get the desired assertion effect. As it is not an enhancement of any language this library based mechanism has of both syntactic and semantic limitations. The assertion statements are basically module instantiation where the actual definition of the assertion is written in the module. So the future enhancement of the assertion feature will be limited to only addition of new port and parameter and also it is not very easy to understand the meaning and usage of these ports and parameters without the help of the language reference manual.

Table 1. Comparative Assertion Support

Parameters	OVL	PSL	SV 3.1
Property Declaration	Y*	Y	Y
Property Checking	Y	Y	Y
Constraint specification	N	Y	Y
Sequence Mechanism	Y	Y	Y
Functional Coverage	N	Y	Y
Concurrency (Cycle based)	Y	Y	Y
Immediate Assertion	N	N	Y
Declarative Assertion	N	Y	Y
Procedural Assertion	Y	N	Y
Pragma Syntax	N	Y	N
Built-In Function	N	Y	Y
Ease of use	Not so Easy	Fairly Easy	Fairly Easy
Tool Development	Easy	Not so Easy	Easy

(*) Done through Module Instantiation

PSL [4], on the other hand, is defined based on erstwhile proprietary, later donated, language Sugar [8]. As it is a totally different language it was not easy to directly encapsulate the whole language within Verilog and though it is permitted to use the PSL constructs directly in the RTL code, it is always used by lot of people using pragma mechanism (as an embedded comment that starts with "PSL") to keep the RTL design tool independent of the assertion support.

In this scenario the SystemVerilog 3.1 is defined as an extension of Verilog language including the features of assertion and functional coverage. This would help both the designers and the verification engineers to write tool independent design and verification modules in the same language without adding any tool specific pragma or semantics. And moreover this has made both the modules understandable to everyone. It is a pure HDVL where both the Hardware Description and its associated Verification specification (read 'design intent') are covered together.

5. Assertion Based Tools

Though SystemVerilog, the first HDVL of its kind, is still in its infancy with only its draft LRM available and standard possibly quite a few quarters away, assertions based on property specification have been widely supported in various tools including @Verifier / @Designer (@HDL), CheckerWare (0-In Design Automation), SpyGlass® (Atrenta), Verification Cockpit & NC-Sim (Cadence), RuleBase & FoCs (IBM), Jeda-X (Jeda Technologies), Safelogic Monitor / Verifier (Safelogic), MMAV / PureSpec (Denali Software), Specman Elite (Verisity), Verity-Check (Veritable) and BlackTie (Verplex). A more comprehensive list of such tools that already supports PSL (or an earlier version of it) or are planning to do so within another quarter can be found in [10].

6. Conclusions

In this paper we have taken a look at assertions as an aid to design, with examples in OVL, PSL and SystemVerilog. We have also discussed how assertion based verification can be efficiently used through an HDVL like SystemVerilog. This unified language will, expectedly, greatly increase the ability to model large designs, and verify that these designs are functionally correct.

7. Acknowledgement

We thank Anurag Gujral, Nitin Agarwal, Vikas Grover and Sridhar Gangadharan for reviewing earlier drafts of this paper.

8. References

[1] "IEEE Std. 1364-2001 standard for the Verilog Hardware Description Language", IEEE, Pascataway, New Jersey, 2001.

[2] "Assertion Based Design", by Harry Foster, Adam Krolnik, David Lacey, Kluwer Academic Publishers, Boston, Dordrecht, London, 2003.

[3] "SUPERLOG ® Extended Synthesizable Subset Language Definition", Draft 3, May 29, 2001, © 1998-2001 Co-Design Automation Inc.

[4] "Property Specification Language 1.0, Reference Manual", Accellera, California, 2003. Also, URL: http://www.accellera.org/pslv101.pdf

[5] "SystemVerilog 3.0, Accellera's Extensions to Verilog", Accellera, California, 2002.

[6] "SystemVerilog 3.1, draft 2: Accellera's Extensions to Verilog", Accellera, California, 2003. Also, URL: http://www.systemverilog.org

[7] "Open Verification Library Reference Manual", Accellera, California, 2003. Also, URL: http://www.verificationlib.org

[8] "OVA – OpenVera™ Assertions". URL: http://www.open-vera.com

[9] "Sugar – The Specification Language for Functional Properties of Logic Designs". URL: http://www.haifa.il.ibm.com/projects/verification/sugar/index.html

[10] "Sugar based Verification Tools". URL: http://www.haifa.il.ibm.com/projects/verification/sugar/tools.html