PARADIGM® WORKS

Top Dos and Don'ts for Writing OpenVera Assertions

Min-Hun Kim

Paradigm Works, Inc.

Min-Hun.Kim@paradigm-works.com

ABSTRACT

Assertions are a compelling verification methodology that is the cornerstone of new, cutting edge verification techniques. Assertions can catch bugs in HDL during a simulation as a checker. In addition they can help the designer to encapsulate concepts as embedded specifications, which are useful to formal property checking tools. Although assertions have existed in one form or another for many years, they have only been recently utilized to their full potential.

This paper will share a set of coding guidelines that are useful in OpenVera Assertion (OVA) checker development. Coding guidelines presented here come from a real-life of OVA checker development.

Table of Contents

List of Figures

List of Examples

1 Introduction

ASIC design and verification engineers have been using Assertion-Based Verification (ABV) to some extent over the past decade. ABV tasks from past design and verification experiences can be divided into two areas. The first area is the HDL-based checker. This type of checker is needed to observe the transaction at the interface of a system or a complex state-machine where most functional bugs occur. The second area is HVL-based functional coverage or scoreboard. These checkers or scoreboards give us important information regarding the efficiency of the test bench and behavior of the design. However, we just haven't seen it get much support from many of our verification colleagues. What is keeping us from using ABV in a real verification effort? What more do we need to make it really happen?

Such an effort using HDL or HVL does not give the full range of assertion capability due to limitations inherent to these languages. Traditionally, the main reason the checkers or scoreboards are built with HDL or HVL-type of languages is that it is easy to combine with an existing design or test bench environment. It is not easy to learn a new language and methodology in a reasonable time. This barrier keeps us from using ABV, and we need to break this barrier. It may be a yet another learning curve to traverse; nevertheless, it is certainly worth going through.

Fortunately a number of EDA vendors, including Synopsys, have been developing tools for ABV. We now have tools and methodology in place with assistance of major EDA vendors. We now need to have a clear direction of how to use it to accelerate our practical verification tasks.

2 OpenVera Assertions

2.1 OpenVera Assertions Defined

OpenVera Assertions (OVA) is a high-level language that contains powerful declarative constructs for accurately capturing design specification that are useful in both dynamic simulation and formal verification environments. With this language, design and verification engineers describe the target application environment including complex protocols and data objects at a high level of abstraction. This high-level approach significantly improves productivity, readability and reusability.

Synopsys announced the Vera Open Source Initiative and the availability of OpenVera™ as an open hardware verification language that included assertions (known as OVA). Later, Intel added their ForSpec formal verification language to OVA.

In conjunction with this effort, the following companies have been developing Assertion tools focused on ABV with OVA.

- Synopsys – VCS, VCS-MX, OVAsim and Magellan
- Verisity – Specman Elite
- Aldec – Riviera, Riviera-IPT
- 0-in – 0-In ABV Suite
- @HDL - @Designer, @Verifier
- Novas - Verdi

OVAsim is a PLI based application from Synopsys that allows an OVA module to be compiled and run with third party simulators. OVAsim works by compiling the OVA module into a shared library object and by creating a wrapper file in HDL format, which forms the link between the OVA module and design.

The latest version of the OVA Language Reference Manual (LRM) is 1.3. Please note that OVA is an evolving language. It is a good idea to look for the latest OVA LRM from www.open-vera.com.

2.2 OVA Characteristics

According to the OVA LRM, OVA is a declarative language that allows much more concise and easily created checkers than the procedural descriptions provided by traditional HDLs.

OVA is very helpful in the following ways:

- Sequences can specify precise timing or a range of times.
- Descriptions can be associated with specified modules and module instances.
- Descriptions can be grouped as a library for repeated use.

A typical OVA file consists of several temporal expressions and a single temporal assertion. Temporal expressions, the descriptions of the event sequences, will check the value change or the value itself in any Verilog registers, integers or nets. Therefore, they could have two forms of expression, sequential expression or logical expression. The assertion tells the checker what to do if the described temporal expression is successful or violated.

A sequential expression is a descriptive format for a series of certain events. According to the OVA LRM, a sequential expression is defined by combining two or more sequences with temporal operators that specify a range of possibilities and repetitions of sequence. A sequential expression is defined using the *event* clause. The OVA LRM states an event is declared with an identifier to name the expression, and a sequence expression to specify the relationship for monitoring.

With a sequential expression, the right hand side of a temporal expression must occur in order to assign such a value to the left side. Example is shown below:

- event data_started_safely : (req_is_granted) #[1..5] ((start_of_frame) && (data_valid));

A logical expression consists of variables with Boolean operators, and returns TRUE or FALSE as part of their evaluation of the expression. A *bool* can be treated as a text-macro; therefore, this macro can be combined as many times as it is needed. In fact, this is a recommended coding style because it improves the readability of OVA code and decreases the likelihood of logical bugs in the OVA code. However, whenever a new value is visible with a clock transition, a new TRUE/FALSE value will be calculated and assigned to the left hand side of the expression. An example is shown below:

- bool data_transfer : (frame == 1'b1) && (^data !== 1'bx);

The basic instruction for testing is a temporal assertion. An assertion specifies an expression or combination of temporal expressions to be tested. Assertion in OVA comes in two forms, *check* and *forbid*. The *check* assertion is used when a valid sequence of events is defined in the temporal expression. The *forbid* assertion is used when an illegal sequence is defined. The *check* assertion is a success when the simulation matches the expression. The *forbid* assertion is a success, but a functional failure, when the simulation matches the expression.

Temporal expressions, temporal assertions, clocks and bool expressions must be grouped together in the form of a *unit*. This *unit* can be connected to a design or test bench through a *bind* statement. Using *bind*, a *unit* can be connected to a single design module or all of the instantiated design modules. Example 1 shows a sample of an OpenVera Assertion demonstrating several of these keywords.

```
unit protocol_checker (logic clk, logic reset, logic transmit, logic ready);

clock posedge clk  {
   event ready3: istrue (!transmit && !reset) in (ready ->> ready ->> ready);
   event protocol: if (matched ready3) then #[1..4] (transmit || reset);
  }
 assert c1 : check(protocol);
endunit


bind module top.v : protocol_checker u1 (clk,reset,tran,rdy);
```

Example 1: Sample of OpenVera Assertions code syntax

Design examples in this exercise and in practical use have been devised using a layered approach. Checker component development, especially as IP, can benefit greatly from such a coding structure. The following Figure 1 is included as an Example of the layered approach to coding OpenVera Assertions.

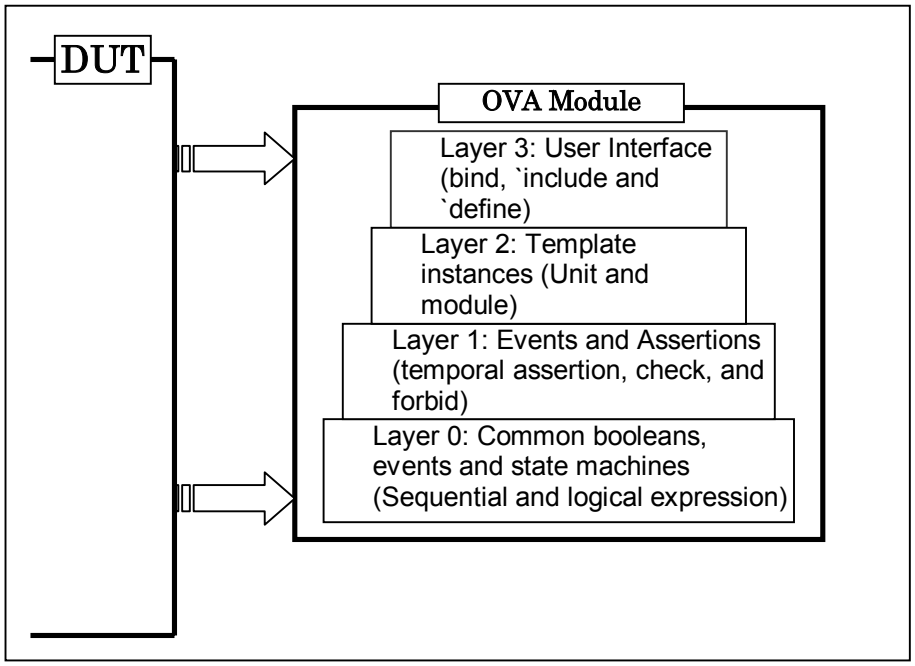

Figure 1: Sample of Layered OVA Code Structure

2.3 The Value of OVA in Verification

Value in verification checker development using OVA can be found in three ways, re-usability, formal verification, and efficient and flexible coding.

An assertion checker developed with OVA can be reusable because it was developed based on an industry standard. In addition, it can be instantiated for each design object as many times as it is needed. It can run with other simulators, when compiled with OVAsim.

Due to the characteristics of OVA, a temporal expression can be expressed in a compact coding style. OVA can do multi-threaded checking if it was written using the proper keyword and coding style. A conventional HDL based checker can only do single-threaded checking. Each assertion can be written as

a functional coverage point. This feature is only available from a handful tool vendors including Synopsys, Inc.

As a part of the formal verification process, a design or verification engineer can capture the design intent in OVA from the beginning. Another practical reason to use OVA checkers is that OVA checkers can be safely delivered as Verification IP to customers with the use of source code encryption technology.

3 Summary of OVA Coding Guideline From Experience

Most EDA vendors are introducing an ABV method as a part of their verification suites. It is getting popular in the Verification field, even being extended to the development of reusable Verification IP for industry standard designs. This document summarizes what was learned from a Paradigm Works R&D project regarding what to do vs. what not to do while OVA code is being written. The revision of OVA LRM referred to in this paper is 1.3.

Although there is an OVA development Guideline published by Synopsys, many of those guidelines did not really help the author very much in a real world implementation of Verification IP. Therefore, this document summarizes not only what was learned from this project but also a few topics from the OVA IP guideline from Synopsys. This will provide the engineer who wants to develop verification IP a more comprehensive guideline based upon real world experience.

Three fundamental pieces of advice are:

1. Get acquainted to with the pre-defined operators and examples.
2. Break down one rule into a small number of temporal expressions
3. Use *check* or *forbid* properly

A big reason for this advice is that OVA coding is not done in a way that is common with how we write code using other high-level of computer languages, like C/C++, Verilog or VHDL. It is an assertion-based language, which means that all of basic booleans and variables will be refreshed at each clock cycle boundary. With this sampling cycle, OVA program behaves like a program with a non-blocking procedural statement and continuous statement.

3.1 What To Do

There is a limit to what OVA can do. OVA code will look for very specific things within a range specified. Therefore, what is provided must be a specification of all legal values for the HDL construct being examined. For instance, an OVA checker to determine if the width of a data bus is 16bits or 32bits wide is not a meaningful task. However, knowing the bus width in advance it would be meaningful to assert if part of bus is not driven when it should be.

3.1.1 Use a Variable, but Use it Wisely

The variable (*var*) will make the OVA code developer very happy. It is the only construct in OVA that can store and manipulate a specific value to be use at different times during a sequence. However, it is important to be aware that OVA is not a procedural language. Any value, which was assigned to the left operand of '<=' sign, is not going to be available to use until the next cycle.

Example 2 shows a conditional statement at line 5 and 6. Each line states that if either start is asserted or *packet_type* is *`WRITE_PACKET*, then assign a portion of *data_bus* information to the left operand of <=sign, otherwise, keep the old value.

```
var [4:0] packet_type;
var [7:0] data_width;
init packet_type = 5'h00;
init data_width = 8'h00;
packet_type <= (start) ? data_bus[4:0] : packet_type;
data_width <= (packet_type == `WRITE_PACKET) ? data_bus[31:24] : data_width;
```

2 Non-Blocking assignments

Example 2: Use of *var* in Temporal Expression

When using a non-blocking assignment to access a data in a variable, the *packet_type* variable at line 6 will have the previous *packet_type* information from line 5 at any given moment. This can limit the usage of a variable. Please be aware of non-blocking assignment. Figure 2 shows the value in each variable and a timing of update of each variable.

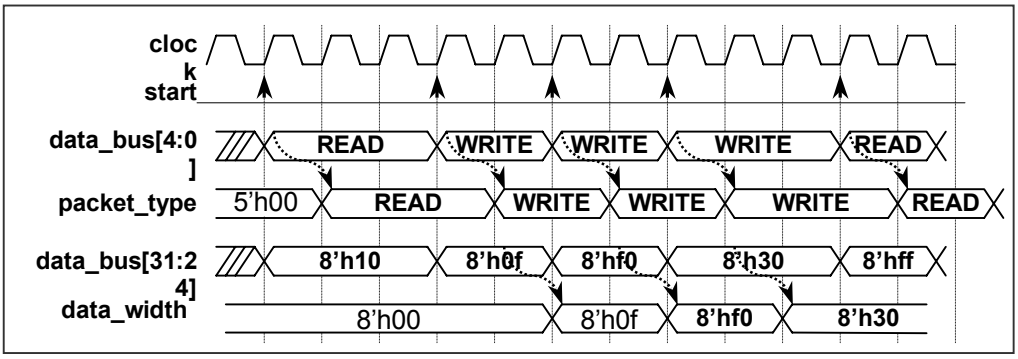

Figure 2: Timing Diagram for Figure 1

Example 3 shows that two non-blocking statements could be combined into one line effectively. This one line of code will do the exact same thing. New information from data_bus[31:24] for the data_width will be latched correctly at the beginning.

```
var [7:0] data_width;
init data_width = 8'h00;
data_width <= ((start) && (data_bus[4:0] == `WRITE_PACKET)) ?
                    data_bus[31:24] :
                    data_width;
```

1 Non-Blocking

Example 3: Updated code with Optimized Timing

Figure 3 shows how sampling timing with variable can be changed.

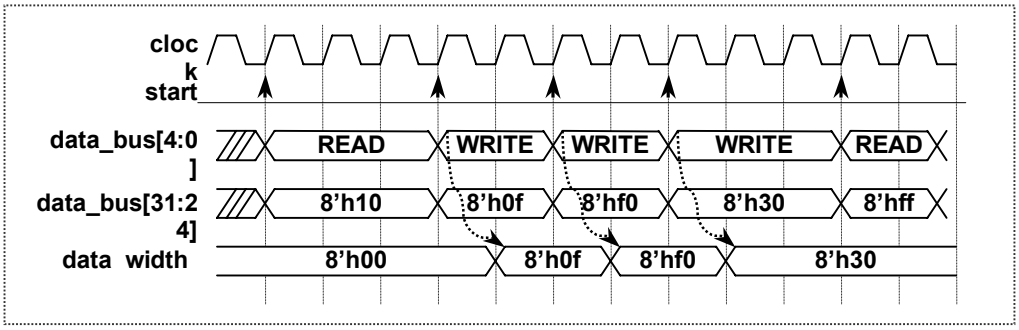

Figure 3: Timing Diagram for Optimized code

3.1.2 Use bool and event as Often as Possible

To overcome the limitations on *var*, two other methods are useful to support OVA coding. They are *bool* (boolean) and *event*. *Bool* is a simple text substitution, just like a text-macro. No matter how many *bool*s were stacked together, they will be replaced at compile-time. No timing is related to *bool* expression.

```
bool valid_data : (^Incoming_data !== 1'bx);
bool valid_parity : ((Incoming_parity ^ 1'b1) !== 1'bx);
bool valid_start : ((incoming_start ~ 1'b1) !== 1'bx);
bool all_valid_input : (valid_data && valid_parity && valid_start);
```

Example 4: Good use of boolean macro substitution

At the line 4 in Example 4, a logical AND operation is performed between the three Boolean expressions. Line 4 can be substituted with:

```
bool all_valid_input : (^incoming_data !== 1'bx) &&
                       ((Incoming_parity ^ 1'b1) !== 1'bx) &&
                       ((incoming_start ~ 1'b1) !== 1'bx);
```

Example 5: When boolean macro substitution was not used

This long line of *bool* conditions is not recommended, because it is hard to understand and maintain in the future.

Event can be used in a similar fashion. Please note that *event* can be observed or controlled from Vera test bench. Therefore if the test bench needs to control the behavior of an OVA checker, or there is a special test plan such as an error test, *event* could help checker to be controlled. *Event* also can be used to collect coverage information, such as functional coverage, regarding the occurrence of the *bool* event. Example 6 shows several ways of using *event*. Event e3 and e4 are identical. Event e5 could be used as a functional coverage statement for PCI bus protocol to detect that a Write followed by a Read to a different peripheral occurred.

```
event e1 : (bus_data == 8'hf7);
event e2 : (bus_data == 8'hbd);
event e3 : (e1 #1 e2);
event e4 : (bus_data == 8'hf7) #1 (bus_data == 8'hbd);

event e5 :(SETUP_WR_SELECT #1
            ENABLE_WR_DSEL #1
              IDLE #1
                SETUP_RD_COV #1
                  ENABLE_READ);
```

Example 6: Good use of event

3.1.3 Know When to Use check

When the correct (or legal) sequence of events is known, then use the *check* assertion. If this sequence of events occurs in the manner as described, no message will be printed in a log file. The assertion will only fire if the stated order of events does not occur. If the *check* fails a predefined assertion message will be printed. The code in Example 7 and corresponding flowchart in Figure 4 describe how the check assertion behaves.

```
event parity_check : if (data_valid) then (parity_valid);
assert event_for_parity : check (parity_check, "parity is not valid");
```

Example 7: Use of 'check' formula

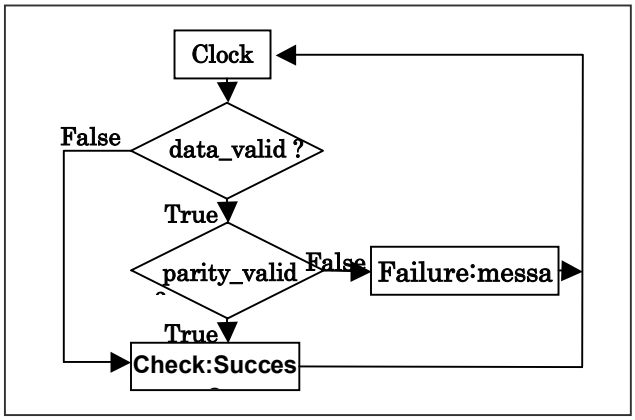

Figure 4: Flowchart for check formula

The *check* will only succeed if *data_valid* is TRUE, and then *parity_valid* is TRUE. *Event parity_check* will be triggered for the assertion statement. If *data_valid* is false, *parity_check* will not be considered at this clock cycle. The check formula does not print any assertion failure message. We will call it a vacuous success because it gives us the same result as if this assertion was triggered successfully. There is no way we can distinguish a vacuous success from a true success; however, that is also the nature of check formula.

A failure message will be displayed only if *data_valid* is TRUE then *parity_valid* is FALSE

3.1.4 Know When to Use forbid

When the exact wrong (or illegal) sequence of events known then use of *forbid* is recommended. The *forbid* assertion is used to catch a sequence that must not happen. Quite often the use of *forbid* is much easier and clearer than the use of *check*. It is important to avoid any a vacuous successes from a *forbid*. One suggested method is a sequential expression. By combining two or more sequences with temporal operators, a vacuous success can be avoided.

The following code in Example 8 and corresponding flowchart in Figure 5 describe how the *forbid* assertion behaves.

```
event not_allowed_events : (sequence_a) #1 (sequence_b);
assert not_allowed : forbid (not_allowed_events, "It is a prohibited sequence.");
```

Example 8: Use of 'forbid' formula

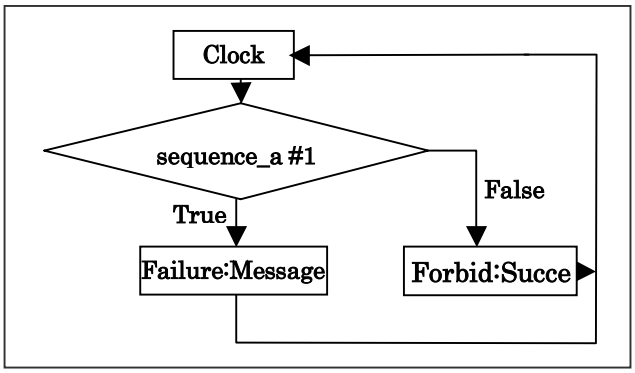

Figure 5: Flowchart for forbid formula

Two clocks of a timing window are used to look for a certain sequence of events, "*sequence_a*, then, one clock later *sequence_b*". It does treat this set of sequences as one group. If this sequence of events occurs, a named event, *not_allowed_events*, will be asserted. This is an event that is not supposed to occur with *forbid*. The *forbid* formula will print the optional assertion failure message as a result of the assertion triggering.

Example 9 also detects a sequence of events. Due to a vacuous success whenever sequence_a is FALSE, *forbid* formula will print the assertion failure message.

```
event not_allowed_events : if (matched sequence_a) then (#1 sequence_b);
assert not_allowed : forbid (not_allowed_events, "It is a prohibited sequence.");
```

Example 9: Incorrect use of 'forbid' formula

3.1.5 Use a Pre-defined Operator as Often as Possible

Concatenation ({}), replication ({{}}), bit-wise operators (~, &, |, ^ and ~^) and reduction operators (&, |, ~|, ^ and ~^) are very useful. OVA is an assertion-type language that uses non-blocking statements. Those operators are very useful to complete the operation within one line. Example 10 shows the usefulness of replication ({}) and reduction operator (|).

```
bool not_all_data_zeros : (|({data_bus1, data_bus2[15:8]}) === 1'b1);
```

Example 10: Use of pre-defined operator

This code checks a bit in *data_bus1* and *data_bus2[15:8]* to find any *1'b1*. If *1'b1* is in either data_bus1 or portion of data_bus2, it will set *not_all_data_zeros* to TRUE. This can also be achieved by using the logical expression shown in Example 11.

```
bool not_all_data_zeros : (data_bus1[16] | data_bus1[15] | ... | data_bus2[8])
                                    === 1'b1);
```

Example 11: If not use of pre-defined operator

Using the pre-defined operator greatly simplifies the expression. It can be even more complicated if a *for-loop* method with a loop counter is used. Example 12 shows how much more complicated it could be.

```
data_bus1_not_all_zeros[0] : data_bus1[0] === 1'b1;
for (counter = 1; counter <=15 ; counter = counter + 1)
{
    data_bus1_not_all_zeros[counter] : data_bus1[counter]  ||
                                        data_bus1_not_all_zeros[counter - 1];
}

data_bus2_not_all_zeros[0] : data_bus2[8] === 1'b1;
for (counter = 1; counter <=7 ; counter = counter + 1)
{
    data_bus2_not_all_zeros[counter] : data_bus2[counter + 8]  ||
                                        data_bus2_not_all_zeros[counter - 1];
}

bool not_all_data_zeros : data_bus1_not_all_zeros[15] |
                          data_bus2_not_all_zeros[7];
```

Example 12: More complicated case without using pre-defined operator

3.1.6 Know When to Use matched and ended

In a typical system, more than one clock can control a data transfer between modules. When signals from multiple clock domains are used within an OVA module, it is important to use the appropriate operator, *matched* or *ended*, to establish a temporal assertion correctly. According to the OVA LRM, use *matched* operator if the event and operator are from different clock domains to build the sub-expression. Use *ended* operator, if the event and operator are within the same clock domain to build the sub-sequence expressions.

Since the *matched* operator could have an operand either from the other clock domain or the current clock domain, *matched* operator asserts an event at the nearest future clock tick in the current clock domain. As the *ended* operator has its operand from a single clock domain only, *ended* operator asserts an event at the same clock tick in the current clock domain. The following code in Example 13 and corresponding timing diagram in Figure 6 describe a possible usage of *matched* and *ended*, with a timing diagram. It is important to understand the timing of events when both the matched and ended operations are used in the same clock domain and the matched operator is used over a different clock domain.

```
clock posedge clk1 {
    event e0 : sequence_e0;

    event matched_e2 : if (matched e0) then #1 (sequence_e1);
    event ended_e3: if (ended e0) then #1 (sequence_e1);
}

clock posedge clk2 {
    event e5: sequence_e5;

    event matched_e7 : if (matched e0) then #3 (sequence_e5);
}
```

Example 13: Use of matched and ended

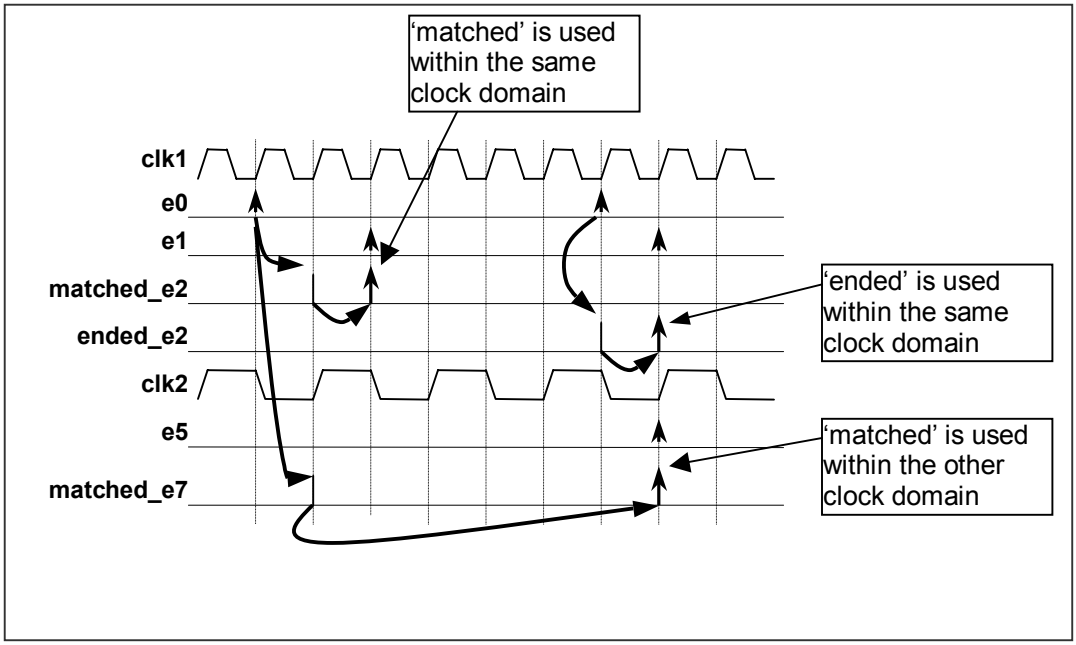

Figure 6: Timing of event for matched and ended operator

If the right hand side contains *matched* or *ended* on an event, it will be evaluated before the expression is evaluated. If either *matched* or *ended* is used the paired event with matched or ended has to be evaluated at that moment.

The following code in Example 14 and corresponding timing diagram in Figure 7 describe two event cases for the use of lexical operator and one event case for the use of *ended* operator. Event *full_e1* and *full_e2* do the same thing. Four clocks after *start_e*, another event expression is invoked to look for *(a==3) && (b==4).* Also note that for event *full_e3* after 4 clock ticks from *start_e*, only a success or failure of *(sub_e)* event will be searched and considered.

```
clock posedge clk {
    event sub_e : (a==3) && (b ==4) #2 (a==5) && (b ==6);
    event full_e1 : start_e #4 sub_e #4 finish_e;
    event full_e2 : start_e #4 ((a==3) && (b==4)) #1 ((a ==5) && (b==6)) #4 finish_e;
    event full_e3 : start_e #4 (ended sub_e) #4 finish_e;
}
```

Example 14: Sample use ended operator

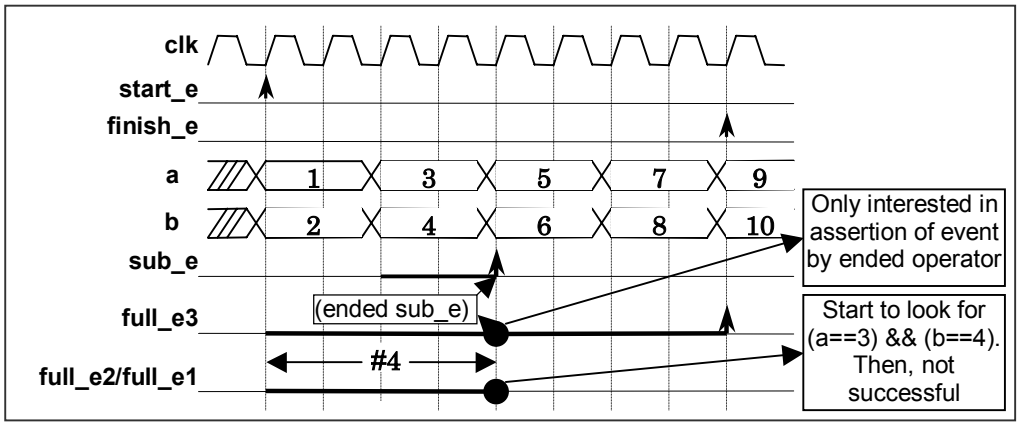

Figure 7: Timing diagram for the use of ended operator

3.1.7 Consider the Formal Verification Case in Advance

In general, the test bench is compiled and run using four states in simulation. If it is planed to use OVA modules as verification IP, please take into account the fact that the formal verification tools use two states. Using compiler directives to resolve two state simulations versus a conventional four state simulations is necessary. The following Example 15 shows that there is no check for 'X' status in 2-state simulation.

```
`ifdef two_state_simulation
    // Do nothing due to 2-state
`else
    bool valid_data : (^Incoming_data !== 1'bx);
`endif
```

Example 15: Preparing a code for a formal verification

A compiler directive, *two_state_verification*, can be specified in a header file. Usually the header file that contains such compiler directives needs to be compiled only once. The following code in Example 16 shows how header files can be included in OVA code once through multiple file structure.

```
`ifdef HEADER
`else
    `define HEADER
    `include "../OVA/csix_l1_32xNbits.ovah"
`endif
```

Example 16: How to include a header file

3.1.8 Use a Meaningful Name for Layer 0 Instantiation

The instantiated name will be used in a hierarchical path with an assertion failure message when an assertion failure message is printed to a log file. Additionally, when a *bool*, *event* and *var* are defined adding a suffix to each variable is recommended. When the OVA module is getting larger, it is hard to know which one is defined as *bool*, *event* and *var* by looking at the name of variable. Therefore, *_b* for a *bool* variable, *_e* for an *event* variable and *_v* for *var* variable are recommended.

The following Example 17 shows how to instantiate a template and how the name can be used throughout a hierarchical structure of OVA code.

```
Layer 0:
template one_bus_check (…,xyz1_b,…,xyz2_e,…);
…
…
Layer 1:
template bus_status (clk) : {
    clock posedge clk {
        event … bus_check1_xyz1_b…;}
    clock posedge clk {
        event … bus_check2_xyz2_e…;}
}
…
Layer 2:
unit  bus_check (…);
one_bus_check bus_check1 (…);
one_bus_check bus_check2(…);
…
```

1. Original template name is one_bus_check (). xyz1_b is bool. xyz2_e is

2. Use xyz1_b and xyz2_e with each instantiated name, bus_check1 and bus_check2. ("_" is attached, automatically.)

3. Both bus_check1 and bus_check2 are is one of instantiated

Example 17: Name change from a module instantiation

3.1.9 Use an Optional Assertion Failure Message

An assertion statement can only print a constant string passed as an optional parameter to the *assert* statement. This string should indicate a clear reason for a failing assertion. A string should have a short but meaningful content with a rule number. Please note that the assert statement should be in one line without any delimiter.

There is a way to print any message with data using some API – PLI code in C, refer to the OVA LRM for details. In general, use the following format shown in Example 18 to print an optional failure message into the log file.

```
assert Rule_P42 : forbid (rule_ev_parity_error,
                          "Rule P42. TxPar vector is a horizontal odd parity");
```

Example 18: An optional assertion failure message in assert

3.1.10 Be Careful When a Shift Operator (either << or >>) is Used

```
buffer <= (reset) ? 0 : (data << counter);
```

Example 19: Incorrect use of shift operator

A left-shift operator << will drop the bits off the left operand, while it is shifting an operand to the left. This is also the way of Verilog/VHDL work. In Example 19, the left most bit will be dropped as the shift operator pushes a bit to the left.

To avoid this situation, the operand has to be a prefixed by a sufficient number of 0's. There is no way of dynamic re-sizing such an appending vector of 0's. It is better to add a large, but not unreasonable, number of 0's. Example 20 is a better way to shift a data into a variable.

```
buffer <= (reset)? 0: {128'b0, data} << pointer);
```

Example 20: Correct use of shift operator

3.2 What Not to Do

This section is written for verification engineers who have experience in high-level HDL/HVL languages but little or none of the assertion-specific language experience, such as OVA. Each of these items could be an easy mistake and a hard-to-find error, because of previous experience of HDL/HVL coding, and making us blind to the assertion coding style. However, as new features are added, and OVA compiler is being enhanced, it can be easier to write. Unless you use an OVA debugger (available from various vendors), it may be hard to locate logical errors.

3.2.1 Do Not Use var in Place of bool

During the initial coding stage, it may be preferable to use *bool* and *event* rather than *var*. If *var* is used, the user must be careful about timing. When a value is assigned to *var*, a newly assigned value will not be available to use until the next clock event.

For instance, examine the following OVA code in Example 21. The code in each column is doing the same thing, i.e. parity checking by comparing a calculated parity with a parity line.

```
bool parity_error : (parity_line !==
                     !(^data_bus));
```

vs.

```
var parity_error;
init parity_error = 0;
parity_error <= (parity_line !=
                 !(^data_bus));
```

Example 21: Comparing of usage between bool and var

The OVA code in left column is using *bool* to check for parity errors and store even parity information. The OVA code in right column can get the same information in a format of one or zero at *var parity_error*. However, it is not possible to have updated *parity_error* information until the next clock tick.

If at all possible, try to break the rule into smaller events and bools so that each OVA line can be a less complicated temporal expression. By combining these smaller events and bools, it is easier to build a clean temporal assertion.

3.2.2 Do Not Use a var to Index a For-Loop

The OVA variable, *var*, cannot be used as a control variable in a for-loop. According to the OVA LRM, all conditional expressions must be constant types with known values during OVA compile. Since a for-loop is processed at compile-time, a parameter and constant are only things that can be used in an initial statement, condition and step assignment of for-loop.

Example 22 shows two examples. Each example has the same concept with a for-loop statement. However, only the right side example can be compiled successfully.

```
var [7:0] start;
var [7:0] max_value;
var [7:0] increment;
init start = 0;
init max_value = 30;
init increment  = 1;

var [max_value-1:0] Memory;
for (i=start; i<max_value; i=i+increment)
{
   Memory[i] <= i * 8;
}
```

VS.

```
`define start          0
`define max_value 30
`define increment   1


Memory[0] : `start * 8;
for (i= `start+1; i<`max_value; i=i+`increment)
{
   Memory[i] <= i * 8;
}
```

Example 22: Two cases of using var and for-loop

3.2.3 Do Not Introduce Duplicate Meanings or Over Specified Rules of Temporal Expressions in Layer 0

During compilation of a compliance list, the duplicate meaning of temporal expressions can be written many times at Layer 0. The reason could be many temporal assertions are required to be started or ended with a certain set of signals.

For instance, one temporal expression using *devsel* and *frame* could provide the necessary indication of starting point of PCI bus transaction. Named temporal expressions can be re-used in multiple places. Performance can be improved by removing any duplicate meaning of temporal expression.

It is important to check that the temporal expressions do not impose any additional rules other than what the design specification states. In the case of assertion-checker development for verification IP, it is crucial not to impose any rules.

3.2.4 Do Not Cascade if-then Statements

Due to the nature of vacuity contained in *if-then* statement, usage of *if-then* statement has to be limited to the last stage of temporal expression. If a conditional statement has to be used, conditional operator, *?-:*, or sequential expression is recommended. Example 23 shows two other ways of describing a conditional statement.

Because of a vacuous success from the *if <boolean>* condition from *if-then* clause, a success from *if <boolean>* will make the cascaded *if <condition>* become true. A true condition in the case from the cascaded *if <condition>* could generate a false negative of an assertion. To avoid such a vacuity with the *if-then* statement, always use the sequence format.

```
a) cascaded if-then statements.
event X : if (posedge a) then (b == 1'b1);
event Y : if (matched X) then #1 (c == 1'b1);

b) Using sequential expression
event X: (posedge a) (b == 1'b1);
event Y: if (matched X) then #1 (c == 1'b1);

c) Using a conditional operator, ?:
event X : (posedge a) ? (b == 1'b1) : (b !== 1'b1)
event Y : if (matched X) then #1 (c == 1'b1)
```

Example 23: Cascaded if-then clause vs. Sequential vs. ?:

3.2.5 Be careful of the open-end format

Longer compile and run times may be experienced when an OVA checker specifies an event with a start and end time in the sequence format. Example 24 uses the open-ended format.

```
event e1 : xyz1 #[t1…] xyz2 #[t1 …];
```

Example 24: Open-ended shift format

Although it is may be necessary to use the open-ended format, try to use a variable to define the start/end of a time sequence as an alternative to this long time shift. Example 25 uses a variable to control the open-end format.

```
t1_valid_period <= ((frame_) && (data_A_valid)) ? `FALSE : `TRUE;
t2_valid_period <= ((frame_) && (data_B_valid)) ? 'FALSE : 'TRUE;

event e1 :  (xyz1 && (t1_valid_period)) #[t1…] (xyz2 && (t2_valid_period)) #[t2…];
```

Example 25: Alternative to Open-ended shift format

Use the *length* operator if an open-ended time specification is needed. *Length* will limit the time period for the match of an expression. Example 26 shows that a sequence, *xyz1* and xyz2, will be only checked from t1 to t3 even though both sequences could happen from t1 time to the end of simulation. If both *xyz1* and *xyz2* do not complete in between time t1 and time t2, *event* e1 will not be asserted.

```
event e1 : length [t1..t3] in xyz1 #[t1…] xyz2 #[t1…];
```

Example 26: Using *Length* to limit match expression period

4 Conclusions and Recommendations

The project mentioned throughout this paper took seven months from the learning of OVA and H/W specification to the complete set of OVA checker development. It was concluded successfully.

Although it only took two months to learn basic concepts of OVA and build a test bench for development, it took about five months to complete OVA coding. The most difficult thing to overcome was a lack of examples and coding guidelines. Functional limitation of the early version of OVA compiler was also an important delay factor of development; however communications with Synopsys technical support were very helpful to find any workarounds or examples.

These coding guidelines will help in the understanding of the OVA language without going through time-consuming learning curve. In addition, guidelines also help to understand and write OVA code better. Using a carefully selected set of these coding guidelines could produce OVA code that is more readable, maintainable and reusable code. The fundamental steps for OVA coding are

- Get acquainted with pre-defined operators and expressions
- Break down one rule into small pieces of temporal expressions
- Follow the layered implementation approach
- Combine those temporal expressions together
- Use *check* and *forbid* properly

Although all of guidelines may not be helpful for some of engineers, most of guidelines are helpful to improve the efficiency of coding and reduce schedule overheard.

5 Acknowledgements

Four engineers gave valuable feedback to this project from the beginning. Many thanks to my colleagues, Bob Ionta and Hermant Mallavaram, at Paradigm Works, Inc. for their time and initiatives. Without a doubt, both Haihui Chen and Edward Cerny at Synopsys, Inc. gave technical advice and examples at every obstacle during this project.

6 References

1. OVA IP Development Guidelines, Synopsys, Inc., August 5, 2002
2. OpenVera ™ Language, Reference Manual: Assertions, Synopsys, Inc., March 2003
3. OVA Customer Training Slide, Synopsys, Inc., June 2002
4. OpenVera website: http://www.open-vera.com
5. Tutorial B, OpenVera Assertions: Synopsys SmartVerification Seminar, May 2002