SYNOPSYS®

OpenVera™ Assertions

March 2003

©2003 Synopsys, Inc.

Introduction

The amount of time and manpower that is invested in finding and removing bugs is growing faster than the investment in creating the design. In addition, the raw dollar cost of a chip re-spin is also increasing. Problems with chip functionality are caused by incomplete or unclear specifications or just simple human coding errors. Current verification techniques are inadequate at catching these errors and eliminating them.

Consider the case where you have embedded a core in a system-on-a-chip (SoC). If you are just checking signals at the chip level, then you may not catch a bug that is caused by an improper protocol to the embedded core. If the protocols were constantly being monitored, then when the violation occurs the simulation would immediately respond with an assertion failure, and the real cause of the problem can be investigated.

Assertion-based verification methodologies[1] are emerging in which assertions are created to describe design specifications. Assertions can describe undesirable behaviors or specific behavior that are required to complete a test plan. These assertions can then be checked in either dynamic simulation or formal verification.

OpenVera™ Assertions (OVAs) provides a clear mechanism to describe sequences of events and to test for their occurrence. This method is more concise and human-readable than the procedural descriptions provided by hardware description languages such as Verilog. In addition OVAs have built-in functions to minimize the amount of code that you need to write. With the expressive power of OVAs, complex protocol assertions can be described in much fewer lines of code than with Hardware Description language (HDL)—based assertions. With clear definitions and less code, verification is more productive.

OVAs can be checked dynamically during simulation, and they can be targeted for proofs by formal verification tools. With this unified support, designers can specify once and use in multiple verification environments. Additionally, OVAs can be used to specify functionality to simulate and measure functional coverage.

OVAs provide language capabilities to build and reuse libraries of pre-built assertions. This macro capability provides a mechanism to build a reusable library of assertions, which can be shared within groups or among the OpenVera community. With a library of assertions, designers will be able to reuse the prior specifications and raise the level of abstraction of the specification.

OVAs are part of the OpenVera open source standard. The open source model has been demonstrated to provide a path for fast time to market with innovation and contribution from multiple sources.

OVAs Features

OVAs are declarative with semantics that are formally based on the theories of regular expression and linear temporal logic. These two theories provide a powerful combination for expressing common hardware activities, such as sequencing, invariants and finite state machine operations.

OVAs are implicitly and concurrently evaluated during all of verification. In dynamic verification, the assertions would be evaluated continuously. Each assertion is evaluated at every time step and its status is updated by the state of simulation. If an assertion completes, then its result can be logged and displayed in a GUI environment. With this implicit evaluation, any overhead of starting, processing and evaluating assertions is a natural feature and does not require any action by the user.

[[1] Principles of Verifiable RTL Design, Author: Lionel Bening, Harry Foster, Published by Kluwer Academic Publishers, Date Published: 06/2001 ISBN: 0792373685]

©2003 Synopsys, Inc.

The language includes features to express the following:

- Basic events and sequencing to specify the order of events
- Time bounded sequences with references to past and future
- Composite sequences using logic connectives (such as and or)
- Repetition of sequences
- Conditional sequences with conditions to hold during sequences
- User specified single or multiple clocks
- Data storing and checking during sequences
- Parameterized library of specifications

OVA has been designed to support hierarchical verification. Any assertion can be either checked as an assertion or assumed as a given property of the design. With this capability assumptions that are used to verify a sub-block, can be reused as checks when that sub-block is instantiated at a higher level. This forms a hierarchical proof to enable sub-block verification independent of the enclosing design.

OVA provides capabilities to refer to data values in the past, present or future. This enables checks of data values at specific time points. An example of this usage is to check that fifo output corresponds to input after several operations. In addition, OVA provides capability to have a non-deterministic variable called a free variable specification, which enables formal verification tools to verify for all possible variable assignments. This provides a shorthand method of asserting all possible values.

Language Hierarchy

The OVA language is broken down into five main sections or levels. The first level is *Bindings* and is used to help define the scope of the assertions. the next level is *Units* and defines the assertion name and port list and the sampling time used. The next level *Directives* is used to specify what properties are to be monitored or checked. The next level consists of *Boolean Expressions*. The last level is *Event Expressions* that describe temporal sequences.

Bindings

OVA provides 2 mechanism to bind or instantiate an assertion to a device under test (DUT). Both mechanisms use the “bind” keyword. To bind an assertion to all instances of a module use the "bind module" keywords. To bind an assertion to a specific instance or set of instances use the “bind instances” keywords.

Units

Units are how assertions are grouped together to make a generic checker. You can think of a unit as a library for assertions. When units are defined, a port list also must be defined, as in a module definition in Verilog. Parameters are also allowed in unit definitions. The allowable signals types in the unit port list are of types “logic”.

Let’s now look at a simple example. Figure 1, on page 3, is the Verilog code of a small 8-bit counter with the corresponding OVA code. The counter starts at 0 and increases until it reaches 255, and then 0 again. We want to create OVA code that catches an overflow condition (cnt changing from 255 to 0) during simulation.

The OVA code shown in Figure 2, on page 3, starts with a unit specification “counter_checker.“ The next line is “clock negedge(clk)”. This specification implies that all events within this grouping should sample their design signals at the falling edge of clk. The next line declares an event “e_overflow” as follows: At some falling edge of clk at time t, cnt is equal to 8’hff, followed by cnt equal to 8’h00 at time t+1. The “#1” specifies that there is a one clk cycle gap between when cnt equals 8’hff and then 8’h00. After the event declaration, the clock context section is terminated with a closed bracket “}”. Also, the last line of the assertion will bind or instatiate the assertion “counter_checker” to the HDL module counter_8bit, and will pass in signals clk and cnt. All instances of counter_8bit will inherit this assertion.

©2003 Synopsys, Inc.

The next line uses the "assert" directive, and the "forbid" construct, so this statement

```
assert a_overflow : forbid(e_overflow);
```

is used to assert that this overflow condition should never happen during simulation. The forbid construct is used whenever it is necessary to describe an illegal condition.

The timing diagram in Figure 3 below shows the output cnt and the clock, clk, and the event e_overflow. The signal values are sampled on the negative edge of the clock. At time 2 cnt=255, and at time 3, cnt=0 which is an overflow condition. The event e_overflow has a starting time at time 2 (represented by the up arrow) and a finish time at time 3 (represented by the second up arrow). The overflow condition will be flagged during simulation, because we assert that this event should never occur.

```
module counter_8bit(rst, clk, cnt);
input rst, clk;
output [7:0] cnt;
reg    [7:0] counter;

always @(rst or posedge clk)
if (rst)
  counter <= 8'b0;
else
  counter <= counter + 1;

assign cnt = counter;
endmodule
```

Figure 1. Verilog Code of Counter

```
unit counter_checker (logic clk, logic [7:0] counter) ;
clock negedge (clk) {
    event e_overflow :  (cnt==8'hff)
#1 (cnt==8'h00);
  }
  assert a_overflow : forbid(e_overflow);
endunit
bind module counter_8bit : counter_checker (clk, cnt) ;
```

Figure 2. OVA Code of Counter

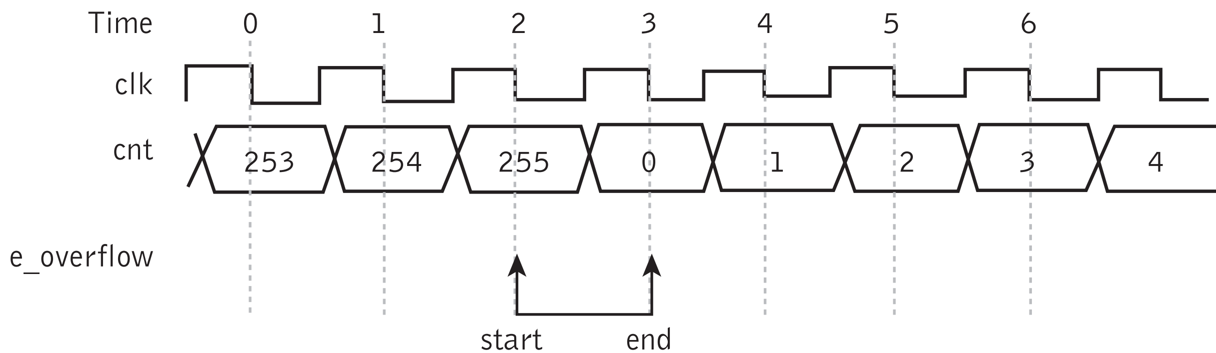

Figure 3. Waveform of Counter

©2003 Synopsys, Inc.

Directives

Directives are statements that define a property that is to be monitored during simulation. Directives need to be placed within the context of a module or an instance, and cannot be placed within a clock grouping. There are two general types of directives that can be used. The first type is:

```
assert a_overflow : forbid(e_overflow);
```

This directive asserts that an overflow condition should never happen. A positive directive may also be written; that is, a property that should always be true:

```
unit mutex_check (logic clk, logic a, logic b) ;
  clock posedge (clk) {
    event e_mutex : !(a && b) ;
 }
 assert a_mutex : check(e_mutex);
endunit
bind module test : mutex_check (clk, a,b)
```

Figure 4. OVA Code for Mutex

In Figure 4 below, a mutex (mutually exclusive) property has been defined. Signals a and b should not be asserted at the same time. The directive:

```
assert a_mutex : check(e_mutex);
```

asserts that the mutex property should always be enforced during the entire simulation.

Boolean Expressions

Boolean expressions are used as building blocks to create properties. The expressions are evaluated to be either True or False. OVA supports 4-state values, 0,1,x,z, and these values can be used in Boolean expressions. All logical operators from the Verilog language are supported within OVA expressions.

Consider the Verilog example below (Figure 5) of a memory definition, and we want to ensure that the memory index is valid. The OVA code shown in Figure 6, on page 5, declares a Boolean valid_address which is true whenever the index of the memory is less than the maximum value, nmax.

```
module test (clock,rst, wen, wa, din, ra,
dout);
  parameter databits = 8;
  parameter addrbits = 8;
  parameter addrmax = (1<<addrbits) - 1;
  input  wen, clock,rst;
  input [addrbits-1:0] wa, ra;
  input [databits-1:0] din ;
  output [databits-1:0] dout;
  reg [databits-1:0] mymem [0:addrmax];

// Other Code Here

endmodule
```

Figure 5. Verilog Memory Definition

©2003 Synopsys, Inc.

```
unit memcheck #(parameter  nbits=8)
       (logic clk, logic [nbits-1 : 0] wa, logic [nbits-1:0] nmax);
 clock posedge clk {
   event e_valid_address : (wa < nmax)
;
 }
 assert a_valid_address : check(e_valid_address);
endunit
```

Figure 6. OVA Code for Memory

Event Expressions

Event expressions are very useful for describing sequence of events over time, as illustrated in the overflow example in Figure 6. One can write very powerful descriptions of sequences of events using OVA. It is usually desirable from a debugging perspective to break complex checkers into smaller parts that are check separately, but jointly imply the original sequence of events. Consider the following example:

Whenever signal "a" goes high, signal "b" must go high within 1-4 clocks cycles later, and signal "a" should stay high until signal "b" goes high.

In this example, whenever "a" is a logical 1, then "a" should stay high until "b" is a logical 1. The *istrue* construct is useful in this instance. Figure 7 shows the OVA code.

```
clock posedge clk {
  event chk : if (a) then
               #1 istrue (a==1) in (#[0..3] b) ;
}
assert a_chk : check(chk);
```

Figure 7. OVA Code

Matched Expressions

In the previous example, the event chk was gated by the rising edge of request. It is sometimes useful to gate an event with another event. Since an event is not a boolean datatype, it is necessary to use the matched function. This function will return either true or false, and will become true when the event succeeds. The matched function is also used to transfer the match of an event from one clock domain to the next. Figure 8 is an example of OVA code for protocol.

If ready is True intermittently or continuously for 3 clock cycles then after that transmit must be True within 4 clock cycles, unless reset happened in the meantime.

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

Figure 8. OVA code for protocol

The initial sequence that we want to check for is “ready is True intermittently or continuously for 3 clock cycle. Lets first look at:

```
(ready #[1..] ready #[1..] ready)
```

This sequence will succeed whenever ready is high in 3 clocks cycles, which do not need to be consecutive. Figure 9 shows 2 valid sequences.

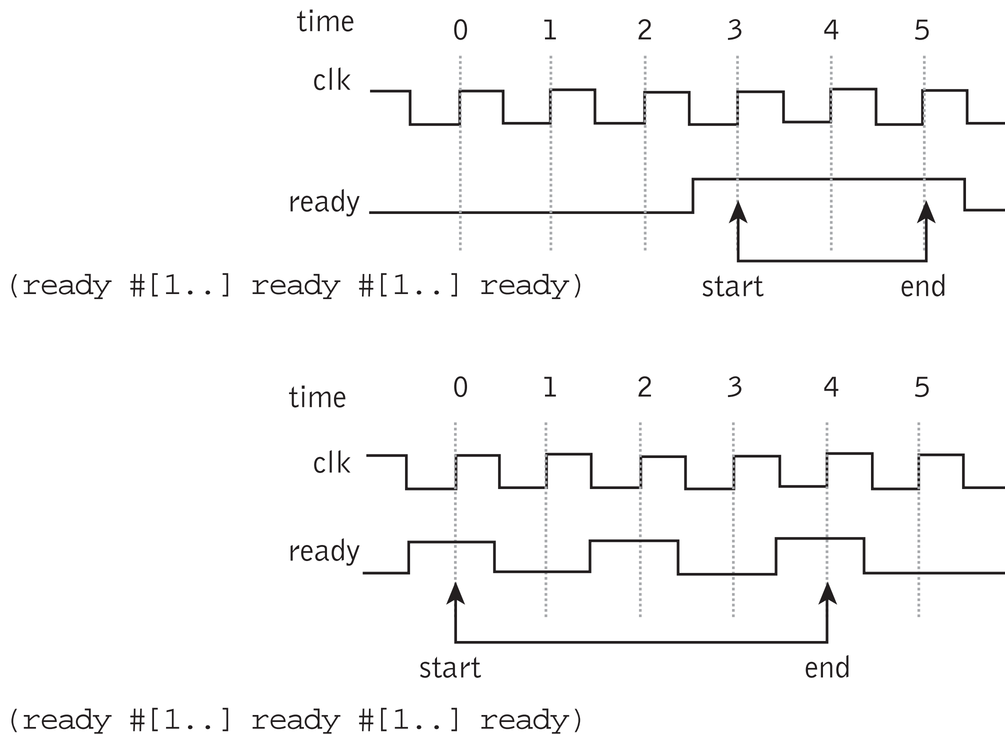

Figure 9. Waveform Diagram for Protocol

In the first waveform, ready is high for 3 consecutive clock cycles. The start and end time of the event is shown. In the second waveform, ready is high in every other cycle, which is also a valid sequence. This condition is necessary but not sufficient. We must also constrain the reset from triggering as well as transmit from going high. In order to constrain these signals the *istrue* construct is used.

```
istrue (!transmit || !reset) in (ready #[1..] ready #[1..] ready);
```

Now, both transmit and reset are inactive during the ready sequence. Whenever this event succeeds or is matched, we want to check that transmit goes high. We use the matched construct, which is True at the end of the event ready3.

```
if (matched ready3) then #[1..4] (transmit || reset);
```

Whenever event ready3 succeeds, then between 1 and 4 clock cycles later either transmit goes high or reset goes high. If ready3 never matches, then we do not check the then condition. Only if ready matches do we perform this check. Finally, we use the assert directive to enforce this check during simulation.

Advanced Features of OVA 2.3

In addition to an intuitive syntax for expressing basic temporal sequences and events, OVA includes a rich set of language features for advanced usage and application.

Complex sequence checks can be written using temporal formulas. Temporal formulas are useful to assert how temporal sequences must occur in relation to one another. The temporal formulas can be composed using the following operators:

- followed_by (ex. X followed_by Y), at least sequence X must be followed by sequence y
- triggers: (ex. X triggers Y), all sequences of X must be immediately followed by at least one sequence of Y
- until, wuntil: (ex. X until Y), sequence X must hold until sequence y is successful
- next, wnext: (ex. next Y), at least one sequence of Y must hold at the next clock cycle

Asynchronous set and resets can be added to any expression by prepending ACCEPT or REJECT with a triggering Boolean expression. This provides mechanism to abort or complete assertion when an asynchronous condition occurs.

Summary

OpenVera Assertions is a high-level language for expressing temporal sequences of events. There are powerful constructs available for describing and checking complex protocols for verification in both dynamic and formal verification environments. With OVAs, verification and design engineers can increase productivity and the ability to find design bugs by writing less code to check assertions on their designs.

OpenVera is an open source language, accessible under the OpenVera open source license. OVA 2.3 includes key enhancements to increase expressive power and formal verification support.

OpenVera Assertions provide a language platform on which an assertion—based methodology will be supported. Libraries of assertion intellectual property (IP) can be developed and exchanged to increase verification abstraction and productivity.

©2003 Synopsys, Inc.

SYNOPSYS®

700 East Middlefield Road, Mountain View, CA 94043 T 650 584 5000 www.synopsys.com
For product related training, call 1-800-793-3448 or visit the Web at www.synopsys.com/services

Synopsys, and the Synopsys logo are registered trademarks and OpenVera is a trademark of Synopsys, Inc.. All other products or service names mentioned herein are trademarks of their respective holders and should be treated as such. All rights reserved. Printed in the U.S.A.
©2003 Synopsys, Inc. 3/03.TM WO