
Introduction

The guidelines covered in this document are organized according to the different aspects of OVA Checker IP construction. They are organized as follows:

- Modeling strategy, including IP structure
- Naming conventions
- Coding guidelines, including general, modeling, and encapsulation guidelines.

Examples are provided throughout this document based on the PCI OVA Checker.

OVA IP is intentionally packaged so that you can easily use IP with both VHDL and Verilog designs by binding the appropriate OVA unit from the IP to the design module or instance. This is the preferred method for connecting the checker to the design or testbench.

Modeling Strategy

This section covers the recommended modeling strategy for using the Checker IP. The following topics are covered:

- List of Properties
- Layered Structure for the Checker IP

List of Properties

The first task is to identify a set of rules governing the IP protocol from a specification set of correctness properties. You identify these rules, which represent required behavior, based on timing diagrams, state-machines, and text descriptions in the specification.

It is important that you avoid introducing more constraints on the behavior than the specification imposes. This often occurs when interpreting timing diagrams in which actual causal dependencies are not clearly identified. For example, event B may be shown 2 to 6 cycles after event A, and event C may occur 3 to 8 cycles after event A. Unless there is a timing constraint between B and C, or the description implies such a constraint, no temporal precedence between events B and C should be required.

You will need to subdivide the rule into subsets according to the signal direction they control. For system-level tests, all these assertions may be used. However, when verifying an individual device (DUT), only the appropriate subset is applicable to the DUT, and all the other subsets represent assumptions on the behavior of the other devices in its environment. That is, the assumptions can be used as a model of the design environment when using formal property checking tools.

For example, in a single-master/single-slave system, there are signals from master to slave (M-S) and from slave to master (S-M). The rules should be subdivided into two subsets according to the direction they characterize, M-S and S-M. Let FRAME be a signal from master to slave and READY a signal from slave to master. If the rule says, “Whenever FRAME is asserted then within 1 to 8 cycles READY should be asserted”, the rule indicates a required behavior of READY and thus it belongs to the S-M subset. It is thus an assertion about the behavior of a slave or an assumption about the behavior of a slave when verifying a master. Note that the “whenever” or “if” condition may refer to signals from either set. It is generally the signals in the conclusion that determine the intent and thus the subset.

Certain rules may refer to signals in the device before it drives a shared bus. Such rules may not be always verified by just observing the bus, since the bus value is determined by the combination of all drivers. In this case, the rules should be put in a separate sub-category of the rules that deal with the same signal direction.

For example, a rule may state: “Master never starts a transaction cycle unless GNT# is asserted.” This means that you should check that, when there is no grant to a master request, the master does not start a transaction. However, if there are other masters on the same bus, then it may not be possible, by observing just the bus, to identify which master started a transaction on the bus (unless no grant is given at all).

In the case of a PCI master protocol, to check the fast back-to-back transfer, you will need to know the *frame* (output from master), *irdy* (output from master), *devsel* (input to master), *trdy* (input to master), and *stop* (input to master). The assertion then verifies if the frame is asserted by the master. This rule thus belongs to the subset related to master. In OVA, this could be represented as follows:

```
event SnpsPciEvMP16_frame:
```

```
if (SnpsPciFrame_deassert_and_transact_not_over &&  
    (!SnpsPci_fast_back_to_back_transfer))  
    then (#1 (SnpsPci_frame_asserted));
```

The identifiers in the above event refer to defined OVA booleans.

You will need to identify a minimal set of non-redundant and non-overlapping properties to reduce the number of assertions that must be implemented and checked.

Consider the following additional cases from the PCI protocol specification:

Rule 1: If PERR# is enabled and a data parity error is detected by a master during a read transaction, the master must assert PERR# two clocks after a completion of a data phase in which a parity error occurs. (Section 3.7.4.1 of PCI 2.2 Specification)

Rule 2: Master always drives PERR# (when enabled) for a minimum of 1 clock for each data phase that a parity error is detected. (Section 3.8.2.1 of PCI 2.2 Specification)

Once Rule 1 has been implemented, it checks for the first occurrence of PERR# assertion and thus Rule 2 is also verified. This is because once PERR# is asserted it is for at least one cycle which means that Rule 2 is also covered. There is no need to have a separate assertion for Rule 2.

If possible, break rules into smaller assertions that jointly are equivalent to the original complex rule. Simpler rules are easier to understand and to implement, which helps in debugging both the assertions and eventually the design. In addition, in formal property checking tools, the verification time of a complex property is usually greater than the verification time of a set of simpler ones.

Consider the following rule:

If not already de-asserted, TRDY#, STOP#, and DEVSEL# must be de-asserted the clock following the completion of the last data phase and must be three-stated the next clock.

The “If not already de-asserted ...” can be coded by logical combinations of events due to signals TRDY#, STOP#, and DEVSEL#, and it is thus possible to put checks for these signals in one rule. However it is simpler to state the rule as follows:

Rule a: If not already de-asserted, TRDY# must be de-asserted the clock following the completion of the last data phase and must be three-stated the next clock.

Rule b: If not already de-asserted, STOP# must be de-asserted the clock following the completion of the last data phase and must be three-stated the next clock.

Rule c: If not already de-asserted, DEVSEL# must be de-asserted the clock following the completion of the last data phase and must be three-stated the next clock.

You should try to eliminate any contradictions in the rules and the resulting assertions. This is a difficult task and you may prefer to test more complex assertions on contrived simple models.

Often the protocol specification is accompanied by so-called *compliance* statements that state what sequences of transactions or operations on the bus must be exercised during simulation so as to verify the design to sufficient degree. These compliance statements can be easily expressed using OVA events and then used in functional coverage evaluation (for example, as supported by the VCS Verilog simulator).

To observe if Write followed by Read to a different peripheral occurs, you may define a coverage statement as follows:

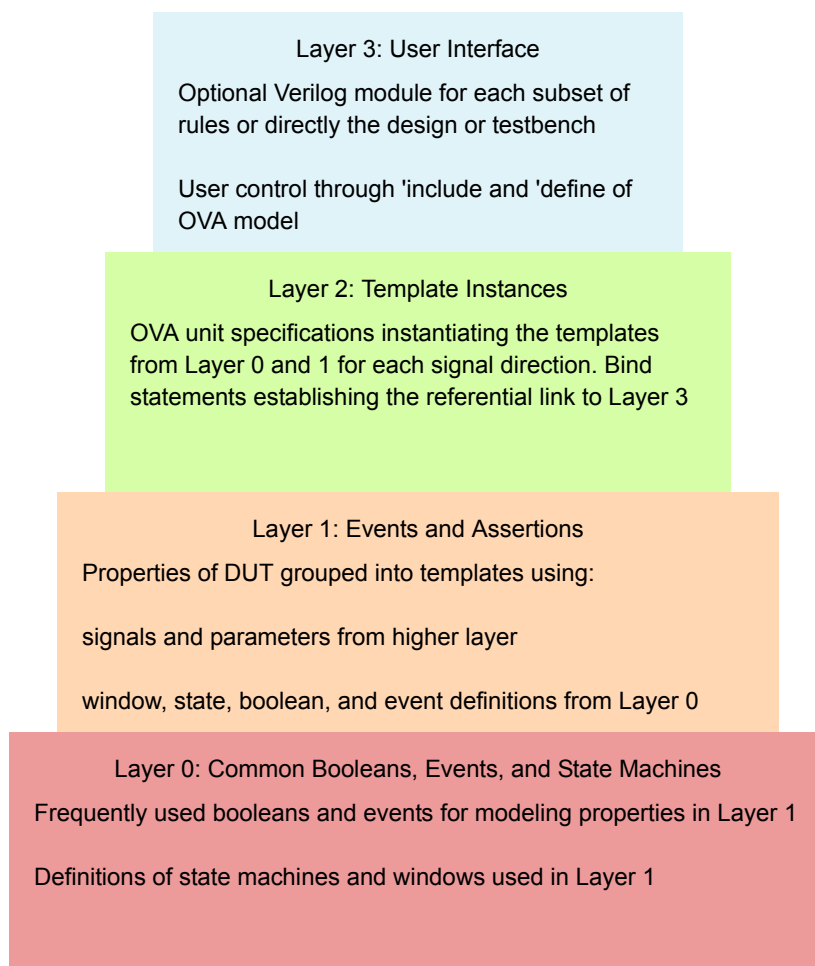
```
event SnpsApbEv_wr_rd_dper:  
    (SETUP_WR_SELECT #1 ENABLE_WR_DSEL #1 IDLE #1  
     SETUP_RD_COV #1 ENABLE_READ);
```

In the previous example, each of the identifiers represent a particular phase of the protocol expressed using an OVA bool or event.

Layered Structure for the Checker IP

IP implementation should be based on a layer architecture, using Verilog module, and OVA unit, template, and bind constructs, as illustrated in Figure 1-1.

Figure 1-1 Layer architecture in OVA Checker IP Models



The roles of the individual layers are described in the following sections.

Layer 0 — Common Booleans

This layer contains common bools and events that are used in multiple templates in the higher layers. This avoids duplication and helps to identify contradictions. The following example shows a code segment of a common template for the OVA PCI IP. The common template will be instantiated with a suitable name (see “Naming Conventions” on page 1-12) in Layer 2. Fully qualified names (for example, `SnpsPci_event_bool_name`) are then used to access the objects in the template instance from another template.

```
template SnpsPciBoolsAndEvents( clk,
                               frame_,
                               irdy_,
                               ad,
                               cbe_,
                               devsel_,
                               trdy_,
                               stop_,
                               par,
                               perr_,
                               rst_
                               ) : {

clock posedge clk {
    bool reset_asserted : ( rst_ == 0);
    bool perr_asserted : ( perr_ == 1'b0);
    bool frame_asserted : ( frame_ == 0);
    bool frame_deasserted : ( frame_ == 1);
    bool no_frame_change : !( edge frame_);
    bool frame_asserted_first : ( frame_asserted &&
                                (negedge frame_));
    ...
    ...
//end template
```

Layer 1 — Events and Assertions

The rules of the IP protocol are coded in the OVA language and encapsulated in templates, one template per rule. The statements inside the templates can refer to the variables, booleans, and events that are placed inside the common template instance (see Layer 2 — Template Instances") using fully qualified names. They can also define local objects of this type. The following example shows a code segment for layer 1:

```
// SnpsPciMP01a:- FRAME#, a Sustained Tri-state signal must be driven high
//                for one clock before being Tri-stated

template SnpsPciMP01a(clk): {
  clock posedge clk {
    event SnpsPciMP_ev_CoverageMP01a : SnpsPciMP_frame_asserted;
    event SnpsPciMP_ev_MP01a_frame: if( SnpsPciMP_frame_asserted)
                                   then( #1 SnpsPciMP_frame_not_z_or_x);
  }
  assert SnpsPciMP01a : check( SnpsPciMP_ev_MP01a_frame, "FRAME#, a Sustained Tri-state
signal must be driven high for one clock before being tri-stated");
`ifdef SnpsPciCoverage
  assert SnpsPciCoverageMP01a : forbid( SnpsPciMP_ev_CoverageMP01a);
`endif
}

// SnpsPciMP01b:- IRDY#, a Sustained Tri-state signal must be driven high for
//                one clock before being Tri-stated

template SnpsPciMP01b(clk): {
  clock posedge clk {
    event SnpsPciMP_ev_CoverageMP01b : SnpsPciMP_irdy_asserted;
    event SnpsPciMP_ev_MP01b_irdy: if( SnpsPciMP_irdy_asserted)
                                   then( #1 SnpsPciMP_irdy_not_z_or_x);
  }
  assert SnpsPciMP01b: check( SnpsPciMP_ev_MP01b_irdy," IRDY#, a Sustained Tri-state
signal must be driven high for one clock before being tri-stated");
`ifdef SnpsPciCoverage
  assert SnpsPciCoverageMP01b : forbid( SnpsPciMP_ev_CoverageMP01b);
`endif
}
...
...
```

Layer 2 — Template Instances

In this layer, templates are instantiated for each signal direction inside an OVA unit specification. There is also a file containing an OVA bind statement to the Verilog module defined in Layer 3 for the particular signal direction. This file is optional and needed only if the Verilog module shell is provided in Layer 3. Otherwise, the user can directly bind the OVA units in Layer 2 to the specific design module or instance by either inlining the unit in the Verilog code or by using a separate OVA file with the bind statement. The following example shows a template instances in an OVA unit for signals driven by a PCI Master (direction from Master to Slave and to Arbiter). The example also shows the OVA bind statement file that ties it to the Verilog module shell of Layer 3.

```
`ifndef HEADER
`else
`define HEADER
`include "SnpsPci.ovah"
`endif

unit SnpsPciOvaMasterUnit(  logic clk,
                           logic rst_,
                           logic frame_,
                           logic irdy_,
                           logic devsel_,
                           logic trdy_,
                           logic stop_,
                           logic perr_,
                           logic par_,
                           logic [31:0] ad,
                           logic [3:0] cbe_,
                           logic req_pin,
                           logic gnt_pin,
                           logic idsel,
                           logic lock_
                           );

SnpsPciBoolsAndEvents SnpsPciMP( .clk( clk),
                                  .frame_( frame_ ),
                                  .irdy_( irdy_ ),
                                  .ad( ad ),
                                  .cbe_( cbe_ ),
                                  .devsel_( devsel_ ),
                                  .trdy_( trdy_ ),
                                  .stop_( stop_ ),
                                  .par( par ),
                                  .perr_( perr_ ),
                                  .rst_(rst_)
                                  );
```

```

    SnpsPciMP02(.clk( clk));
    SnpsPciMP03(.clk( clk));
    ...
    ...
end unit

bind module SnpsPciOvaMasterInterface : SnpsPciOvaMasterUnit
    SnpsPciOvaMasterUnitInst( clk,
                               rst_,
                               frame_,
                               irdy_,
                               devsel_,
                               trdy_,
                               stop_,
                               perr_,
                               par_,
                               ad,
                               cbe_,
                               req_pin,
                               gnt_pin,
                               idsel,
                               lock_
                               );

```

Layer 3 — User Interface

This layer contains the optional Verilog module shell declarations, one per group of assertions related to one signal direction. The ports of each module are those from the protocol specification as sampled by the assertions. The direction of all the ports is “input”. There may be parameters that allow parameterizing the OVA templates. These modules can be instantiated in the testbench or design to insert the checker. As mentioned earlier, the preferred way of linking the checker to the design is the use of a bind statement that is non-intrusive as far as the design is concerned.

This layer also contains a file of 'define of symbols that control the configuration of the active checkers. This file is included ('include) in all OVA specifications in Layer 2.

```

/-- FILE: SnpsPci.ovah
/--
/-- OVA PCI checker header file that can be customized for a particular
// PCI implementation under test
/--
/-- ABSTRACT: This file is the header file that User can customize through
/-- macro definitions. The purpose of each macro defined in this
/-- file is explained in the User Guide. A brief description is

```

```

//--          also located with each item below.
//--          Comment out the definition if the respective feature
//--          is not required.
//--
//-- Release Version: 0.1
//--
//--

// Macro definitions are given below

`define SnpsPciCoverage
`define SnpsPci_X_Valid
//`define SnpsPciOvaCompiler
`define SnpsPciLock
`define SnpsPciMP10
`define SnpsPciMP20
`define SnpsPciTP04
`define SnpsPciTP15
`define SnpsPciTP34a
`define SnpsPciTP34b

// The above macros may need to comment out if the user does not want to
// have the corresponding feature explained below.

```

Naming Conventions

This section provides general guidelines for naming and coding OVA events and assertions.

It is recommended that you make use of naming conventions to prevent name conflicts. A simple rule is to name all OVA objects as follows:

<group_prefix><assertion_group_prefix>_<item_name>

For example, a Synopsys-generated event in the OVA PCI checker would be named as: SnpsPci_ev_MP01a_frame.

This minimizes the chance of name conflicts with user-generated PCI assertions. It also simplifies interpreting and sorting the simulation report, as it is possible to collect all related assertions just by using the name prefix.

Select the names of booleans and events to make clear correspondence with the original specification.

For example, to check that the last two bits of signal AD are zero as in linear burst ordering, the corresponding bool could be named as follows:

```
bool SynPci_linear_burst_ordering:  
    (ad[1:0] === 2'b0);
```

Tag assertion names with the item number in the test plan and user guide.

For example, MP39 is an item number in the PCI Master protocol. The assertions is then named as:

```
assert SnpsPciMP39: check (...
```

Add a short but meaningful message to each assertion. It should indicate the failure of the rule. The message is a quoted string passed as the second argument to the directives `check` and `forbid`. For example:

```
assert SnpsPciMP39: check(SnpsPciEvMP39,  
    "Master must ignore its GNT# if RST# is asserted.");
```

General Coding Guidelines

Use clocked events and assertions instead of unlocked assertions, because the latter uses simulation time as the clock, which is inefficient on synchronous designs. Also, it cannot be used with some formal tools.

Correct clock:

```
clock posedge clk {  
    event e1 = ...;  
}  
assert a: check (e1 || b1);
```

Inefficient clock and not usable with some formal tools:

```
event e1 = ...;  
assert a: check (e1 || b1);
```

Minimize the use of edge expressions on signals (other than in a sampling clock specification where an edge is required) as it adds overhead during verification and in many cases the edge detection is not needed.

For example, to check that a pulse on signal x is only one clock cycle wide can be written as:

```
if (posedge x) then #1 negedge x;
```

However, a simpler and equivalent check can be stated as:

```
if (x) then #1 !x;
```

Similarly, instead of using

```
if (posedge request) then #[2..10] posedge ack;
```

it is often possible to state it as follows:

```
if (posedge request) then #[2..10] ack;
```

Placing top-level “if” statements into the assertion may simplify reuse of events in other event definitions.

For example, we wish to verify that when a occurs then the sequence b #1 c must hold, and then if a is followed by b #1 c, then some other sequence e2 must hold.

One way to code it is as follows:

```
event e2: ... ;
event e_if: if a then #1 b #1 c;
event e_a_e1: a #1 b #1 c;
event e_next: if ended e_a_e1 then e2;
assert c_if: check (e_if);
assert c_next: check (e_next);
```

It is simpler and possibly easier to understand the conditions in the assertions to express the same checks as follows:

```
event e2: ... ;
event e_abc: a #1 b #1 c;
assert c_if: check (if a then e_abc);
assert c_next: check (if ended e_abc then e2);
```

Note that the sampling clock for the booleans a and ended e_abc is inferred from the (unique) clock of the events e_abc in c_if and e2 in c_next.

Avoid large bounded intervals in time shift and repetition operators. Use bounds that are as tight as is expected from the design and if large then consider expressing the property in some other way, such as using OVA variables. (See “Window Checks” on page 1-22.)

For example:

```
event e: xyz #[1..4600] abc;
```

Could take a long time to compile and verify.

Use the open-ended interval $[n ..]$ with caution, because failures in some assertions involving a time shift with an open-ended interval cannot be detected using simulation and by many property checkers that do not handle unbounded liveness.

For example, the check

```
if a then #[1..] b;
```

cannot fail in a finite simulation, because b could occur right after the end of simulation. Nevertheless, an indication that the missing b is an error could be deduced from the fact that the evaluation attempt of the assertion has not completed by the end of simulation. By looking at the start time of the attempts and the time of the end of simulation could reveal that something is amiss.

Note, however, that open-ended intervals are useful in pattern matching or if constrained by the length operator. The latter is particularly useful in cases exemplified by the following abstract example:

```
if y then #1 length [10..20] in a #[1..] b #[1..] c;
```

It states that after y , a must be followed by b that must be followed by c , and the total extent of the sequence $a \rightarrow b \rightarrow c$ is at least 10 and at most 20 clock ticks.

OVA supports automatic detection of meaningless successes in the case of top-level “if” statements without the “else” clause. Therefore, it may be preferred to code implications using “if” statements rather than burying them in a boolean expression.

For example, `req` cannot be asserted unless `ack` is de-asserted can be coded as:

```
!ack || !req
```

But if we are verifying the component that generates `req`, it is better to express this property as:

```
if ack then !req;
```

In the latter case the situation when `ack` is never asserted during simulation (and so, `req` de-asserted in that situation is never tested) can be detected by OVA functional coverage.

If the edge operators (`posedge`, `edge`, and `negedge`) are applied to a `bit_vector` then only the least significant bit is tested. Therefore, if you need to detect a change of value on a `bit_vector` expression `exp`, do not use `edge` but instead define a `bool` as follows:

```
bool change(exp): past(exp) != exp;
```

`change` can be instantiated like `edge` whenever we need to detect a change of value on `exp`, except that the argument must be in parentheses. For example: `change(data)`.

Placing an “if” statement without an else clause under the `forbid` assertion directive usually does not produce the desired result.

For example, suppose we wish to verify that if `a` occurs then it is not followed by `b` in the next clock cycle. This could be coded as:

```
event e1: if a then #1 !b;  
assert c1: check (e1);
```

Suppose now that we wish to express the incorrect sequence and then forbid it. The temptation may be to write it as:

```
event e2: if a then #1 b;  
assert f2: forbid(e2);
```

The problem is that whenever a is false then e2 succeeds and hence forbid will fail. This is clearly not the intent of the verification. It is required that only the sequence a followed by b should fail. But this is exactly how it should be written using the OVA syntax. Namely:

```
event e3: a #1 b;  
assert f3: forbid(e3);
```

Here, f3 will report a failure only when the sequence a b is detected in the simulation trace.

To summarize: if bool then good_conclusion is usually in a check assertion, while the form bool && bad_conclusion is placed under a forbid assertion.

An example where a particular writing style can lead to efficiency problems yet one that can be easily avoided is as follows:

Let

```
clock posedge clk {  
    event e1: if a1 then #1 b1;  
    event e2: if a2 then #1 b2;  
    event e3: if a3 then #1 b3;  
    event e4: if a4 then #1 b4;  
}
```

where ai and bi are some boolean expressions.

To verify that all these events hold on a design, you might write a conjunction of all these events in one assert statement:

```
assert c1: check(e1 && e2 && e3 && e4);
```

Since the compiler cannot see if there is any relationship between the boolean expressions a_1 , a_2 , a_3 , and a_4 , to simplify the checker, the smallest machine that can be generated will contain $O(2^n)$ states, where n is the number of atomic sequences.

In the example, for each combination of a_1 , a_2 , a_3 , and a_4 , there is a different combination of b_1 , b_2 , b_3 , and b_4 in the next clock tick, i.e. it will be equivalent to $2^4 = 16$ transitions:

```
    ((a1 && a2 && a3 && a4) #1 (b1 && b2 && b3 && b4))
|| ((a1 && a2 && a3)      #1 (b1 && b2 && b3))
|| ((a1 && a2 && a4)      #1 (b1 && b2 && b4))
... (other 12 combinations of satisfied preconditions)
|| ((a1)                #1 (b1));
```

If n is larger this can represent a significant blow-up of the checker and compilation time.

The simplest way to avoid it is not to form this conjunction. Instead, place an assert statement on each of the component events separately. This is always better in the case of formal tools like FormalVera.

Another solution is to use knowledge that the preconditions satisfy some relations. For example, that they are mutually exclusive. In that case a much simpler checker can be constructed. In the above example, if the a_i are mutually exclusive (and you do not want several separate assert statements) then a more efficient assertion can be written as follows:

```
clock posedge clk {
    event e1: if a1 then #1 b1
             else if a2 then #1 b2
             else if a3 then #1 b3
             else if a4 then #1 b4;
}
assert c1: check(e1);
```

Conclusion: Do not form long top-level conjunctions of events. Either decompose into separate smaller assertions or use knowledge of the desired behavior to simplify the sequence expression.

Modeling Guidelines

The section covers modeling guidelines for:

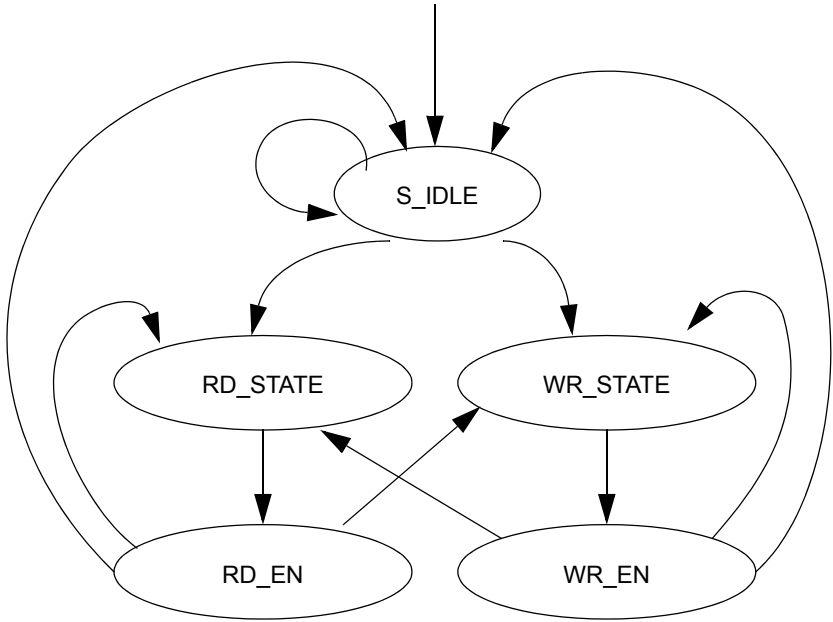
- State Machines
- Window Checks
- Encapsulation

State Machines

A state machine may be provided in the specification of the bus protocol. This state machine can be useful in simplifying the OVA assertions. It can be easily implemented using OVA variables and placed in the shared template, such that events and assertions can access the state variable.

The following example is a simple state machine.

Figure 1-2 Simple State Machine



The following example is a model for a simple state machine

```

var[2:0] machine_state;
init machine_state = `S_IDLE;

// Machine state encoding
machine_state <= (!resetn ? `S_IDLE :
  ((idle_idle) ? `S_IDLE :
  ((idle_readsetup) ? `RD_STATE :
  ((idle_pwritesetup) ? `WR_STATE :
  ((readsetup_readenable) ? `RD_EN :
  ((writesetup_pwriteenable) ? `WR_EN :
  ((readenable_idle) ? `S_IDLE :
  ((readenable_readsetup) ? `RD_STATE :
  ((readenable_pwritesetup) ? `WR_STATE :
  ((writeenable_idle) ? `S_IDLE :
  ((writeenable_pwritesetup) ? `WR_STATE :
  ((writeenable_readsetup) ? `RD_STATE :
  machine_state)))))))));
  
```

Each line in the above definition corresponds to one transition. The conditions in the above statements are thus booleans, each representing the enabling condition for the specific transition. For instance, `idle_idle` would be defined as:

```
bool idle_idle: (machine_state == `IDLE) && !psel;
```

A rule that verifies that `PENABLE` is not active while in the `IDLE` state can then be written as follows:

```
event no_enable_in_idle:
    if (machine_state == `IDLE) then !penable;
```

Similarly for other signal values that must have a specific value in a state of the protocol. Note that if a signal value is used to make a choice between two or more transitions in the state machine then (of course) it cannot be constrained as shown above. If a rule requires this to happen, then there may be something wrong either with the rule or the state machine.

Window Checks

There are times when checks are valid in a time window. A possible way to do this is by using an OVA variable to indicate when the window is active. It is set or reset by the occurrence of events (through the "ended" or "matched" operators) or by booleans.

For example, it is possible to define the write data phase in PCI as follows:

```
var SynPciWrite_dataPhase_bit;
init SynPciWrite_dataPhase_bit = 0;
SynPciWrite_dataPhase_bit <=
    (((SynPciNegedge_irdy &&
        (!SynPciWrite_dataPhase_bit)) ||
```

```

SynPciLast_dataphase) ||
SynPci_bus_idle) &&
SynPciWrite_bit) ?
(! (SynPciLast_dataphase || SynPci_bus_idle)) :
SynPciWrite_dataPhase_bit;

```

The variable is set at the beginning of the data phase and reset at the end.

Use OVA variables to compute and store expected results for data checking.

For example, use of OVA variables to keep and compute expected values that can be later compared with monitored signals and events. To check the parity in the PCI bus, the expected parity can be precomputed in an OVA variable.

```

bool SnpsPci_parity_error : (par != SnpsPciParity);
var SnpsPciParity;
init SnpsPciParity = 1;
SnpsPciParity <= (^ {ad[31:0], cbe_[3:0]});

```

Note, however, that if the design under verification is pipelined, the stored value could be overwritten by the next input data before it is checked. In such situations, a FIFO or a shift register must be built using OVA variables to preserve the order and value until the time when the data check can be performed.

Make use of macros for readability.

For example, the state of the bus may be defined as macros:

```

`define S_IDLE    3'b001
`define RD_STATE  3'b010
`define WR_STATE  3'b011
`define RD_EN     3'b100

```

Use 'ifdef conditional compilation controls.

For example, formal verification tools work with two states only, hence checks that require four states (such as HiZ detection) can be conditionally compiled under user control as follows:

```
`ifdef SnpsPciOvaCompiler
    // Do nothing (used in this form instead of `ifndef)
`else
    bool SnpsPciPar_not_tristated : par !== 1'bz;
    bool SnpsPciData_driven_during_read :
        ((ad[31:0] !== 32'bz) &&
         (SnpsPciRead_bit &&
          (SnpsPciRead_dataPhase_bit ||
           SnpsPci_trdy_asserted)));
`endif
```

Use parameterized bools and events to improve readability and reusability of the design.

For example,

```
bool SnpsPci_asserted(signal) : (signal == 0);
```

The bool can be used as follows:

```
event SnpsPciEvMP01b_irdy:
    if (SnpsPci_asserted(irdy_))
        then (#1 SnpsPci_irdy_not_z_or_x);
```

Similarly,

```
event SnpsPciEvMP17_master_abort :
    if (SnpsPci_asserted(devsel_) &&
        (SnpsPci_data_transfer_count == 0) &&
        (!SnpsPci_asserted(stop_)) &&
        (!SnpsPci_asserted(trdy_)))
        then (#1 (!SnpsPci_bus_idle));
```

Using `bool` definitions, it is possible to encode all needed enabling conditions for a state machine. It is then much simpler and understandable to code the state machine transitions using the `? :` expression.

Use `event` instead of a `bool` if that event is to be observed or controlled from a Vera testbench or if coverage information is to be collected regarding the occurrence of the boolean event. In Vera, events and assertions can be accessed and controlled by the Vera testbench but booleans and macros cannot.

If an event `e` is to be used in the condition of an if statement in a more complex event, then the `ended e` (or `matched e`) construct must be used in the condition of the complex event. Recall that any time the event `e` completes successfully (matches), the `ended e` boolean expression becomes true.

Encapsulation

Use units and templates to encapsulate and reuse OVA assertions.

The difference between a template and units, and events and booleans is that a template or a unit defines a parameterized, reusable set of property checkers. It is a set because a template or a unit can have one or more assertions. In that case, the assertions will all be related to the verification of a specific design object. For example, a FIFO or a bus protocol.

The difference between units and templates is that only units can be bound to a Verilog or VHDL design and the ports and parameters of units must have type specifications. Templates on the other hand behave more like macros in that they can be instantiated within units or templates, the parameters are typeless, and the bodies are just expanded in the place of instantiation (all internal names are prefixed by the template instance name if any).

Units allow the construction of general purpose property checker libraries and application-specific checker libraries that can be written once and easily reused across multiple designs. Templates allow building reusable modular components of such checkers.

Use a template for common objects, and use a separate template for each rule. It may contain more than one assertion pertaining to different aspects of the rule.

See Example 1-3.

Example 1-3 Modular Rule for PCI Template

```
// SnpsPciMP01b:- IRDY#, a sustained tri-state signal
// Must be driven high for one clock before being tri-stated.

template SnpsPciMP01b(clk): {
    clock posedge clk {
        event SnpsPciEvCoverageMP01b : SnpsPci_irdy_asserted;
        event SnpsPciEvMP01b_irdy: if (SnpsPci_irdy_asserted)
            then (#1 SnpsPci_irdy_not_z_or_x);
    }
    assert SnpsPciMP01b: check (SnpsPciEvMP01b_irdy,
        "IRDY#, a Sustained Tri-state signal, must be driven high
        for one clock before being tri-stated.");
}
```