

Asynchronous Behaviors Meet Their Match with SystemVerilog Assertions

Doug Smith

Doulos

16165 Monterey Road, Suite 109

Morgan Hill, CA USA

+1-888-GO DOULOS

doug.smith@doulos.com

ABSTRACT

Most digital designs inherently possess asynchronous behaviors of some kind. While the SystemVerilog assertion (SVA) language offers some asynchronous controls like *disable iff*, writing concurrent assertions that accurately describe asynchronous behavior is not so straightforward. SVA properties require a clocking event, making them innately synchronous. When describing asynchronous behavior, the behavior of interest typically occurs *after* the asynchronous trigger appears. Unfortunately, SystemVerilog scheduling semantics make this rather difficult to check because the assertion input values are sampled *before* the trigger occurs. This often leads assertion writers to sampling using clocks, which may not guarantee matching and optimal checking in all cases. Alternatively, there are some simple approaches for describing asynchronous behavior using SVA that this paper explores. The SystemVerilog scheduling semantics are described along with the difficulties they pose for checking asynchronous behavior. Traditional approaches are considered such as synchronizing to a clock, but better asynchronous alternatives are suggested and practical examples provided. In addition, some practical solutions are offered for other asynchronous behaviors like asynchronous communication between clock domains or across bus interfaces. Lastly, this paper considers the various changes and additions to the recently published IEEE 1800-2009 standard, which may simplify checking asynchronous behavior.

Categories and Subject Descriptors

B.6.3 [Logic Design]: Design Aids – automatic synthesis, hardware description languages, optimization, simulation, switching theory, and verification.

General Terms

Languages, Verification.

Keywords

Assertion, asynchronous, SystemVerilog, SVA, SystemVerilog Assertions, clock domain crossing, asynchronous handshaking, delay, trigger, clock handover, scheduling semantics, simulation regions, Preponed, Active, NBA, non-blocking assignment, Observed, Reactive, expect, programs, clocking block, immediate assertions, concurrent assertions, procedural concurrent assertions, deferred assertions, checkers, multi-clock sequences, abort properties, disable, property, sequence.

1. INTRODUCTION

Asynchronous behaviors still find their way into almost every design whether it operates synchronously or not. For example, designs use asynchronous reset controls or respond to asynchronous inputs like non-maskable interrupts, enables, or other asynchronous controls. Not uncommonly, interface protocols use asynchronous handshakes, and multiple clocks in a design cause asynchronous communication between clock domains. Therefore, it is just as necessary to adequately test the asynchronous behaviors in a design as it is the synchronous ones.

SystemVerilog assertions (SVA) are an ideal choice for writing checkers given the rich temporal syntax provided by the language. However, they operate synchronously by nature because they sample relative to a sampling event (such as a clock) and because of the SVA scheduling semantics described in the IEEE 1800-2005 SystemVerilog standard[3], making SVA a little tricky to use for describing asynchronous behaviors. Asynchronous behaviors usually fall into two categories: (1) asynchronous control, and (2) asynchronous communication. SystemVerilog assertions can be used for either, but each presents its own set of challenges. In the following section, both types of asynchronous behaviors are considered along with the difficulties of describing them using SVA, and practical examples and solutions to resolve these difficulties. In section 3, the latest additions and modifications to the SystemVerilog 2009 standard[4] that aid asynchronous assertion writing are considered, followed by a brief summary of the recommended practices and solutions presented in this paper.

2. ASYNCHRONOUS BEHAVIORS

2.1 Asynchronous controls

The most common form of asynchronous behavior found in nearly every design is asynchronous control. For purposes of discussion, consider the following up-down counter example:

```
module Counter (input Clock, Reset, Enable,
                 Load, UpDn,
                 input [7:0] Data,
                 output logic [7:0] Q);

    always @ (posedge Reset or posedge Clock)
        if (Reset)
            Q <= 0;
        else
```

```

if (Enable)
  if (Load)
    Q <= Data;
  else
    if (UpDn)
      Q <= Q + 1;
    else
      Q <= Q - 1;
endmodule

```

As one might expect, this counter has an asynchronous reset to initialize the module upon power-up or system reset. The counter's behavior is defined in Table 1.

Table 1. Truth table of up/down counter functionality.

Reset	Clock	Enable	Load	Up1Dn0	Data	next Q
1	-	-	-	-	-	0
0	rise	0	-	-	-	unchanged
0	rise	1	1	-	Data	Data
0	rise	1	0	0	-	Q-1
0	rise	1	0	1	-	Q+1

Using concurrent¹ SystemVerilog assertions, checkers can be easily written to cover the functionality in the counter truth table. For example, several assertions could be written as follows:

```

default clocking cb @(posedge Clock);
endclocking

// Enable
assert property ( !Enable |=> Q == $past(Q) );

// Load of data
assert property ( Enable && Load |=> Q ==
$past(Data) );

// Up counting
assert property ( Enable && !Load && UpDn |=> Q
== $past(Q)+8'b1 );

// Down counting
assert property ( Enable && !Load && !UpDn |=> Q
== $past(Q)-8'b1 );

```

2.1.1 Disable iff

These concurrent assertions are fairly straightforward; however, they neglect the effect of an asynchronous reset during the assertion evaluation. If a reset occurs, these checks may immediately become invalid. A common mistake is to place the asynchronous control signal in the precondition (also referred to as the *antecedent*) of the assertion. Adding the asynchronous control into the assertion's precondition stops the evaluation of any *new* assertion threads, but it fails to affect any *existing* assertion threads. For example, Figure 1 shows how adding the reset to the precondition seems like it will work, but actually results in false failures.

```

assert property ( !Reset && Enable && Load |=>
Q == $past(Data) );

```

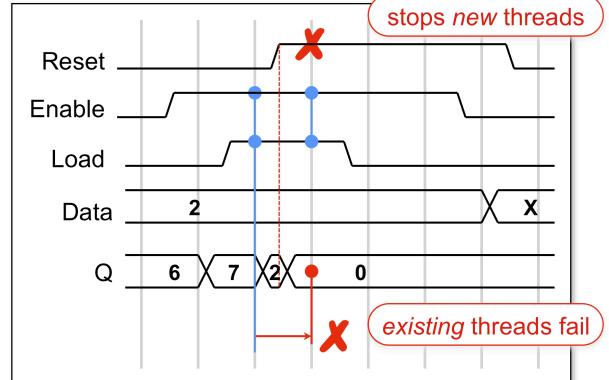


Figure 1. Asynchronous control in a precondition results in false failures.

The most appropriate approach to cope with the asynchronous reset is to use the SystemVerilog *disable iff* construct. *Disable iff* provides a level-sensitive control to automatically stop new assertion evaluations and terminate active threads. To fix the assertions in this example, each assertion should have an abort condition specified by adding a *disable iff* clause in order to work properly in all situations:

```

// Enable
assert property ( disable iff( Reset ) !Enable
|=> Q == $past(Q) );

// Load of data
assert property ( disable iff( Reset ) Enable &&
Load |=> Q == $past(Data) );

// Up counting
assert property ( disable iff( Reset ) Enable
&& !Load && UpDn |=> Q == $past(Q)+8'b1 );

// Down counting
assert property ( disable iff( Reset ) Enable &&
!Load && !UpDn |=> Q == $past(Q)-8'b1 );

```

Guideline 1: Always use disable iff to asynchronously terminate active assertions threads.

2.1.2 Checking asynchronous events

While *disable iff* handles asynchronous assertion termination, what if the asynchronous behavior is the thing to be checked? In the simple counter example, one check is missing—does Q go to 0 immediately upon reset? Since concurrent assertions require a sampling event, one might be tempted to write a concurrent assertion as follows:

```

assert property(@(posedge Reset) Reset |=>
Q == 0 );

```

At first glance, this looks like it works because Q is being checked for 0 *after* the reset signal occurs. However, that is not actually the case. In order to understand what is happening, the SystemVerilog scheduling semantics need to be considered.

¹ SystemVerilog defines two types of assertions: (1) *immediate* and (2) *concurrent*. Immediate assertions are created by using *assert()* in a procedural block of code like *always* or *initial*. Concurrent assertions create their own thread of execution waiting for the particular property or sequence to occur, creating independent checkers.

Before SystemVerilog, a Verilog simulation only had a few scheduling regions: *active*, *inactive*, *non-blocking assignment (NBA)*, and the *monitor/strobe* regions. All blocking assignments and statements are scheduled in the active region, while non-blocking assignments are scheduled to evaluate later after all active assignments and statements have been executed. This gives non-blocking assignments their characteristic behavior of updating at the end of a simulation time slot.

SystemVerilog, on the other hand, has greatly expanded the number of simulator regions in order to evaluate correctly the new constructs introduced like assertions, clocking blocks, and programs. The *Prepared* region was introduced to properly handle assertions. As simulation time advances, the *Prepared* region is at the beginning of each new simulation time slot and proceeds before any events or assignments are generated from always or initial blocks (see Figure 2). The SystemVerilog standard requires that all values used to evaluate concurrent assertions must be sampled during the *Prepared* region ([3], section 17.3). This means that the values are always sampled before any sampling event triggers the assertions to evaluate, making all assertions synchronous to the sampling event and avoiding any changes or metastability issues on the assertion inputs.

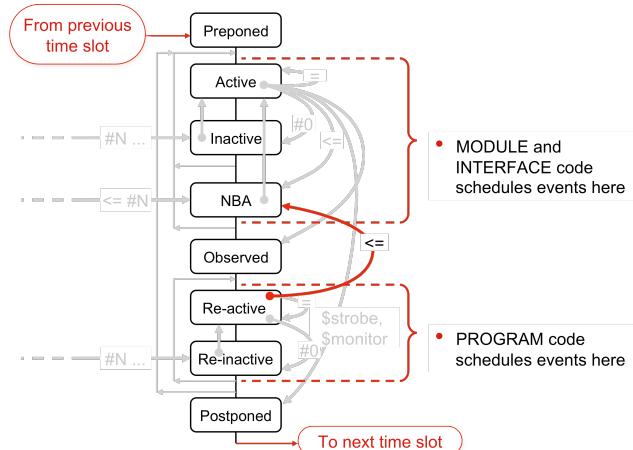


Figure 2. SystemVerilog scheduling regions.

Unfortunately, the way the concurrent assertion semantics are defined makes it difficult to check asynchronous behavior during the same time slot as the asynchronous event. In the simple counter example, the assertion above will never succeed for two reasons. First, the precondition checks to see if Reset is 1, which seems sensible since a posedge Reset triggers the assertion. However, assertions use input values sampled in the *Prepared* region, or the value just *before* the posedge Reset occurs; *i.e.*, a value of 0. Since Reset equals 0, the precondition always evaluates false. A way to work around this would be to set the precondition to true in order to force the check:

```
assert property(@(posedge Reset) 1 |-> Q == 0);
```

Using this precondition causes the assertion to evaluate, but raises a second issue. As with the Reset value, the value of Q will always be the value before the posedge of Reset. Considering the simple up-down counter, Q is reset to 0 using a non-blocking statement, which means that Q does not update until the *NBA* region long after Q is sampled in the *Prepared* region. Figure 3

illustrates this point. The assertion could be re-written to use $\@(\negedge \text{Reset})$ instead, but the behavior of Q during reset might be unstable and that would never be detected by the check.

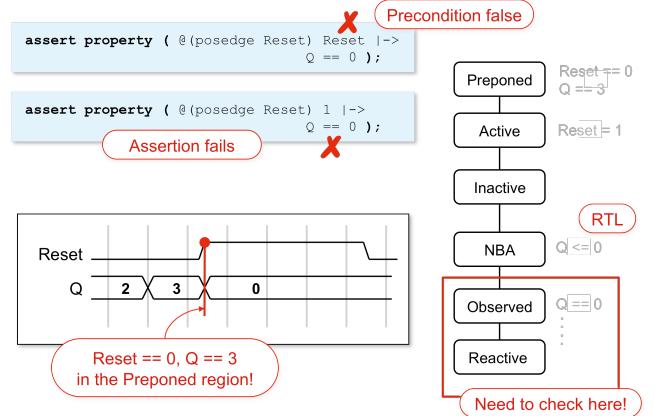


Figure 3. Signals that change asynchronously are not available immediately in concurrent assertions.

The solution to the problem is to sample the input values at a different simulation time. Therefore, the key to checking asynchronous behavior after an asynchronous control is to delay either the evaluation of the check or the sampling of the assertion inputs. There are many ways to accomplish this and the following offers some possible solutions.

Guideline 2: The key to checking asynchronous behavior is to delay either the evaluation of the checking or the sampling of the assertion inputs.

2.1.2.1 Synchronously checking

The most common way to check asynchronous behavior is to synchronously sample the assertion inputs. While this is usually sufficient, it is essentially a “cheat” and lacks the assurance that all behavior has been thoroughly checked (see Figure 4).

```
assert property (@(posedge Clock) Reset |-> Q == 0);
assert property (@(posedge Reset) 1 |=>
    @(posedge Clock) Q == 0);
```

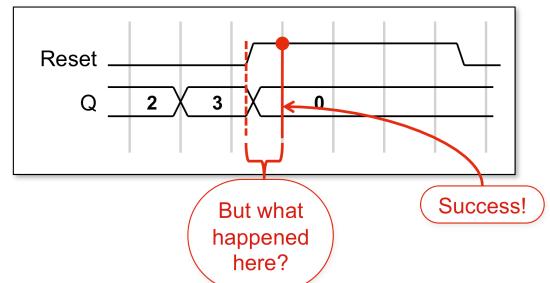


Figure 4. Synchronously sampling asynchronous behavior.

Sometimes, the clock frequency may not be fast enough to sample the asynchronous behavior so an oversampling fast clock could be

used. Often, this is used with hardware emulators or to catch glitches that occur on the asynchronous signals. Even so, it is the author's opinion that synchronous sampling of asynchronous behavior should only be used when necessary since there are better asynchronous ways to write checkers that do not open the possibility of missing spurious transitions between sampling events.

2.1.2.2 Immediate assertions

Immediate assertions have the advantage that they evaluate at the point that they are executed whichever simulation region that may be. Using immediate assertions, the asynchronous reset assertion could be written as:

```
always @(posedge Reset) assert( Q == 0 );
```

As with the earlier concurrent assertion example, this looks like it should work; however, again there is the issue of when the immediate assertion evaluation takes place. Immediate assertions execute by default in the *Active* scheduler region. In the design, Q is being updated by a non-blocking assignment so it is being updated *after* the assertion check evaluates during the *NBA* region.

Considering Figure 2, the only way for the checking of Q to evaluate correctly is for the assertion to execute in either the *Observed*, *Reactive*, *Re-inactive*, or *Postponed* regions since they all evaluate after the design has updated the value of Q in the *NBA* region.² The *Postponed* region is only available to PLI routines (like \$monitor and \$strobe) or clocking block sampling, leaving the *Observed*, *Reactive*, and *Re-inactive* regions to evaluate the assertion. There are several easy approaches to accomplish this: use a (1) *program block*, (2) a *sequence event*, (3) the *expect statement*, (4) a *non-blocking trigger event*, or (5) a *clocking block*.

2.1.2.2.1 Program blocks

Program blocks are designed intentionally to execute after all events are evaluated in the design in order to avoid race conditions between the testbench and the design. A *program* block's inputs are sampled in the *Observed* region, and any initial blocks within a *program* are scheduled to execute in the *Reactive* region ([3], section 16.3). By placing the immediate assertion in a *program*, Q will have the reset value when the check is evaluated.

```
program tester;
    initial forever
        @(posedge Reset) assert( Q == 0 )
            else $error( "Q != 0 after reset!" );
endprogram
```

Note, *program* blocks can be nested inside of modules or interfaces so that they have visibility to signals in the local scope, but not all simulators support nested programs. Thus, one disadvantage to this approach is that it requires hierarchical

² There are also several regions not shown in Figure 2 that are provided for PLI such as the *Post-NBA*, *Post-Observed*, and *Pre-Postponed* regions, which would also suffice. While PLI could be used to check asynchronous behaviors, it is the author's opinion that PLI is considerably more complicated than the simple approaches presented in this paper and are therefore not considered.

references to probe the design signals, unless the *program* is bound (using *bind*) into the design hierarchy.

2.1.2.2.2 Sequence events

A sequence event is an alternate approach to using programs. Sequences define a series of temporal events and can be used to block the execution of statements until the sequence has been matched. The SystemVerilog standard defines that the end-point status for a sequence is set in the *Observed* region of simulation. Therefore, waiting upon the completion of a sequence by either using a sequence method like *.ended*, *.matched*, or *.triggered*, or using *@(sequence)* will delay the execution of subsequent statements until after the *Observed* region ([3], sections 10.10.1 and 17.12.6).

For example, the same reset check could be written as:

```
sequence reset_s;
    @(posedge Reset) 1;
endsequence

always @(reset_s) assert( Q == 0 );
```

The advantage of using sequence events is that each asynchronous control can be defined as a named sequence and then used appropriately in any always or initial block in modules, interfaces, or programs. Sequence events provide a great alternative to delay assertion input sampling, but be aware that not all simulation tools fully support them yet.

2.1.2.2.3 Expect statements

Perhaps providing the best compromise between immediate and concurrent assertions is the SystemVerilog *expect* statement. *Expect* is a procedural statement for use within always or initial blocks, but it also understands the temporal property syntax available to concurrent assertions. For example, an *expect* statement can use properties such as this:

```
initial
    expect( @(posedge clk) a ##1 b ##1 c ) else
        $error;
```

Fortunately, *expect* is a great construct for checking asynchronous behaviors. The SystemVerilog standard states that the statement after *expect* is executed after the *Observed* region ([3], section 17.16). Therefore, using *expect*, immediate assertions can be executed after the *Observed* region as follows:

```
always
    expect(@(posedge Reset) 1) assert( Q == 0 );
```

Unfortunately, not all major simulators execute the *expect* statement and subsequent statements after the *Observed* region. To compensate, the assertion evaluation can be delayed to at least the *NBA* or *Observed* region by waiting for the change to occur on Q:

```
always
    expect(@(posedge Reset) 1) @Q assert( Q == 0 );
```

However, waiting for Q to change may be problematic. For instance, if Q is already 0 when Reset occurs, then the assertion would fail to check until the first non-zero change of Q, resulting in a false failure. Instead, the *expect* statement can be placed inside of a *program block* to delay the sampling of the assertion inputs:

```

program tester;

initial forever
    expect(@(posedge Reset) 1) assert( Q == 0 );
endprogram

```

2.1.2.2.4 Non-blocking event trigger

Another trick to delay the sampling of an immediate assertion's inputs is to use a non-blocking event trigger (`->>`). Traditional Verilog provides an event trigger (`->`) that is evaluated immediately and only for the current simulation time step. The non-blocking event trigger delays its evaluation to the *non-blocking assignment (NBA)* region of the current or future time step. In the counter example, Q is updated in the *NBA* region using a non-blocking assign. By waiting for a non-blocking event trigger, the assertion is also delayed until the *NBA* region. For example,

```

always @(posedge Reset)
begin
    event t;
    ->> #0 t;
    @t assert ( Q == 0 );
end

```

However, this non-blocking trigger evaluates during the same simulation time as the RTL is resetting Q, which essentially creates a race condition depending on the order that the simulator evaluates the two processes. In some simulators, the non-blocking trigger always occurs at the appropriate time after the RTL has been updated; in others, it depends on whether the assertion is co-located with the RTL code or in a separate module.

In order to guarantee the correct input sampling, a `#0` delay can be placed before the assertion to cause the assertion to be further delayed until the RTL has finished evaluation:

```

always @(posedge Reset)
begin
    event t;
    ->> t;
    @t #0 assert ( Q == 0 );
end

```

As the simulator processes the *NBA* events, they are promoted to the *Active* region where they are evaluated as shown in Figure 5:

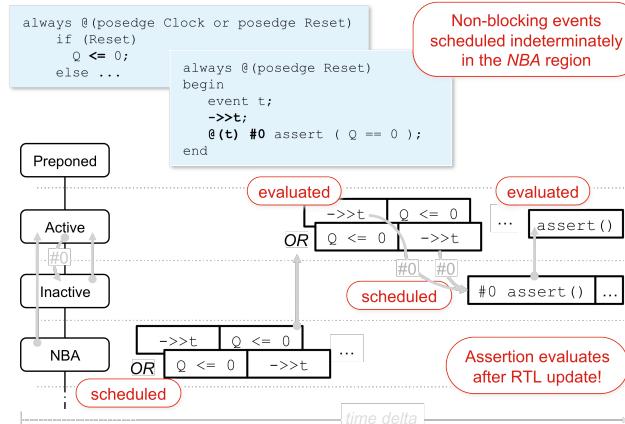


Figure 5. Verilog indeterminacy does not affect the assertion evaluation when using a non-blocking trigger and `#0` delay.

The assertion is further delayed to the *Inactive* region, causing the RTL assignment to Q to always evaluate beforehand regardless of the order that the non-blocking events were scheduled. Using the `#0` delay, the race condition between non-blocking events is eliminated and the assertion can be placed either in the RTL code or elsewhere. Incidentally, until recently not all major simulators supported non-blocking event triggering, but now the latest versions generally have good support.

2.1.2.2.5 Clocking blocks

Clocking blocks can delay the sampling of inputs used in immediate assertions. By default, clocking blocks sample using `#1step`, which is equivalent to the *Postponed* region of the previous time slot or the *Preponed* region of the current. Sampling can be delayed until the *Observed* region by specifying the input delay to be `#0` instead. By using a clocking block input in the immediate assert, the value of Q will already be updated from asynchronous reset:

```

clocking cb @(posedge Reset);
    input #0 Q; // Delay sampling
endclocking

always @(posedge Reset)
    assert( cb.Q == 0 );

```

Notice that the clocking block samples with respect to the asynchronous reset. Because the input delay is specified as `#0`, the value used for Q comes from the *Observed* region after the updated RTL reset value was set in the *NBA* region.

Using the reset as the clocking block's sampling event creates a potential race condition between updating the clocking variable `cb.Q` and sampling `cb.Q` from the assertion the first time Reset occurs. Assuming Q is 4-state and the assertion process evaluates first, then a erroneous value of X will be sampled before the clocking variable `cb.Q` is updated, resulting in a false failure. With some simulators, it is enough to wait on the clocking block before evaluating the assertion like this:

```

always @(posedge Reset)
    @cb assert( cb.Q == 0 );

```

Given the possibility of a false failure, this method should be used with care.

Guideline 3: Immediate assertions can check asynchronous events if evaluated in the Observed or Reactive simulation regions.

2.1.2.3 Concurrent assertions

While concurrent assertions sample synchronously and so have a disadvantage when used for asynchronous checking, there are a few tricks to make them work without resorting to sampling off a clock. As previously discussed, the key to checking asynchronous behaviors is to delay the checking or the sampling of the inputs. Clocking blocks cannot be used as with immediate assertions because the SystemVerilog standard explicitly states that clocking block inputs used by concurrent assertions must be sampled using `#1step`, not `#0` ([3], section 17.3). Nonetheless, there is still a way to delay the input sampling or the assertion checking by delaying the sampling trigger or calling subroutine methods using matched sequences.

2.1.2.3.1 Delaying the asynchronous control

While it may seem like “cheating” just like sampling using a clock, delaying the asynchronous control signal slightly to allow the RTL to finish evaluation is really one of the easiest and simplest approaches to checking asynchronous behavior. In the counter example, the original concurrent assertion can be used but with the Reset signal delayed just enough for the RTL to reset the value of Q:

```
assign #1 trigger = Reset;
assert property( @(posedge trigger) 1 |-> Q==0);
```

Since there is no unit specified with #1, the delay will be 1 time unit delay. Normally, this is sufficient but the smallest precision time unit could also be used. Some simulators allow the use of the `#1step` keyword, which represents the global time precision:

```
assign #1step trigger = Reset;
assert property( @(posedge trigger) 1 |-> Q==0);
```

Using `#1step` guarantees that no action is missed between the time slot that the Reset occurs and when the value of Q is reset. Not all simulators implement `#1step` so a hard coded value is usually adequate.³ Note, using a delay with a continuous assignment statement is easiest, but a separate process could also be used to delay Reset or create a named event to trigger the assertion evaluation.

```
assign #1 trigger = Reset;
assert property( @(posedge trigger) 1 |-> Q == 0 );
```

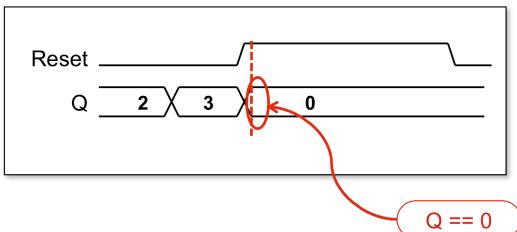


Figure 6. A delayed asynchronous trigger causes Q to be sampled at the correct time.

2.1.2.3.2 Calling subroutines on matched sequences

While the SystemVerilog standard restricts the sampling of concurrent assertion inputs to the *Preponed* region (*i.e.*, the value before the sampling event), it does not restrict the sampling of the inputs used by tasks or functions called by an assertion sequence. In fact, a subroutine called by a sequence is specifically defined to evaluate in the *Reactive* region ([3], section 17.9), allowing the inputs to be sampled after the RTL design has finished updating from any asynchronous events.

For example, consider the following task:

```
task automatic check( ref logic [7:0] data,
                      input logic [7:0] value );
  assert ( data == value );
endtask
```

The `check()` task accepts any 8 bit variable or wire and compares it to `value`. Notice, the `data` is passed using `ref`, which is important because otherwise the *Preponed* value will be passed. Since `ref` is required, the task must be `automatic` to work properly in some simulators. Now the task can be called in a concurrent assertion in the following manner:

```
assert property( @(posedge Reset) 1 |->
                  (1, check( Q, 0 )));
```

or in a named sequence:

```
sequence check_q_eq_0;
  (1, check( Q, 8'b0 ));
endsequence
assert property( @(posedge Reset) 1 |->
                  check_q_eq_0 );
```

Because the task is evaluated in the *Reactive* region, the value of Q is read at that time, giving the updated RTL value after the asynchronous reset occurs. The same could be done with a function, but not all simulators evaluate functions or sample their inputs in the *Reactive* region as the standard specifies. Nonetheless, a generally portable solution is as follows:

```
function bit check_q( input logic [7:0] value );
  return ( Q == value );
endfunction
assert property( @(posedge Reset) 1 |->
                  check_q( 0 ));
```

or using a named sequence:

```
sequence checkq;
  check_q( 0 );
endsequence
assert property(@(posedge Reset) 1 |-> checkq );
```

The drawback to this approach is that the variable (or wire) being checked cannot be passed as an argument but must be hard coded inside the function so that it is sampled at the appropriate time. As a result, using the task version is a more flexible and preferable solution.

Guideline 4: Concurrent assertions can check asynchronous events by delaying the asynchronous control or calling a subroutine from a matched sequence.

2.1.3 Other Considerations

2.1.3.1 Stability checking

In the simple counter example, checking that the RTL outputs the correct value upon an asynchronous control signal is important and deserves consideration. However, there are other assertion checks worth considering as well. For example, a check to test the stability of Q while under reset might also prove useful. Such a check could simply be written as:

```
assert property( @(!posedge !Reset) Q == 0 );
```

This assertion checks if a non-zero change on Q occurs that it does not happen while the device is held under reset. The

³ One major simulator disallows `#1step` outside of a clocking block, but allows the non-standard use of `#step` in an assignment statement to accomplish the same result.

assertion could have been written using `@(posedge Q)`, but simulation tools might interpret this as a non-zero `Q[0]` or the entire vector `Q`. Instead, a more cautious approach is preferred of detecting a negedge transition using the true/false expression result of `(Q==0)`.

2.1.3.2 Timing simulations

Delaying the input sampling on an asynchronous assertion works well as long as there are no timing delays in the design RTL. Most of the methods shown in this section evaluate the assertions at the same simulation time as the asynchronous control signal occurs, which would not work with a design that includes delays such as a gate-level netlist. In this case, several easy options exist that could be used to delayed the assertion checking long enough for Reset to propagate through the design and `Q` to be updated:

- (1) Sample synchronously as shown in 2.1.2.1:

```
assert property (
    @(posedge Clock) Reset |=> Q == 0
);
```

- (2) Delay the asynchronous control signal as shown in 2.1.2.3.1:

```
parameter gatedelay = 10;
...
assign #gatedelay trigger = Reset;
assert property( @(posedge trigger) 1 |-> Q==0);
```

- (3) Delay the assertion checking a fixed amount of time:

```
always @(posedge Reset)
#gatedelay assert( Q == 0 );
```

OR

```
always @(posedge Reset) begin
    event t;
    -> #gatedelay t;
    @t assert( Q == 0 );
end
```

- (4) Delay the assertion checking until the RTL changes:

```
program tester;
    initial forever
    begin      // Signals visible to program
        @(posedge Reset);
        if ( Q != 0 )
            @Q assert ( Q == 0 && Reset );
    end
endprogram
```

Option 1 generally works provided Reset lasts longer than one clock period; a fast sampling clock could also be used. Options 2 and 3 generally work, but may require some trial and error to find the correct sampling delays that work.

Option 4 is essentially immune to timing delays since it triggers on events, but poses its own set of difficulties. First, if `Q` already equals 0, then the assertion never performs its check (this is required if `Q` equals 0 to prevent a false failure occurring when Reset is released and `Q` starts changing). Second, in a gate-level simulation glitches may occur on `Q` resulting in false failures. Third, a concurrent assertion cannot be used for this check since the value of `Q` will be sampled in the *Preponed* region instead of after the RTL has updated `Q`; therefore, the assertion needs to be delayed using a program block or other method previously

discussed in order to correctly sample the assertion's input value(s). Fourth, there is no guarantee that `Q` actually changes on the same Reset that triggered the evaluation! If `Q` fails to change and Reset is de-asserted and re-asserted, then the assertion may not check the value of `Q` until a subsequent occurrence of Reset.

- (5) Create a multi-clocked sequence:

```
parameter TIMEOUT = 2;
...
assert property (
    @(posedge Reset) 1 |=>
        @Clock) ##[1:TIMEOUT] Q == 0 && Reset
);
```

Probably the best compromise is Option 5—sampling using the clock once the Reset triggers the assertion evaluation. Instead of waiting for `Q` to change, a parameterized timeout value can be specified so if `Q` never changes before Reset de-asserts then an error is flagged. This allows the use of a concurrent assertion, and changing the timeout number of clock edges to sample is much simpler than adjusting hard-coded timing delays anytime the gate-level netlist changes. This type of assertion is referred to as a multi-clocked sequence and is discussed in detail in the next section.

Guideline 5: For timing simulations, synchronously checking the design's behavior upon the asynchronous event is probably the best overall solution.

2.2 Asynchronous communication

The second major category of asynchronous behavior is asynchronous communication. In most designs, asynchronous communication commonly occurs between two independent clock domains or with an asynchronous interface protocol. Checking asynchronous communication is usually easier than asynchronous controls because the signals or data being checked are typically setup and ready for sampling before the sampling event occurs. The exception to this occurs between clock domains when the independent clocks happen to occur at the exact same simulation time. While this may be unlikely, even if it does happen it is usually not a problem because sampling is simply delayed to the next clock edge; whereas, with asynchronous controls, there is no following control signal to perform the sampling so other methods are required such as those outlined in the preceding sections.

The SystemVerilog standard has extensively defined the semantics for multi-clock support, which can be used to sample events across clock domains as well as asynchronous protocols. The basic principles will be presented here; however, refer to [3] for more specific and in-depth details.

2.2.1 Clock domain crossing

The first type of asynchronous communication to consider is clock domain crossing. The key to building multi-clocked sequences is using the concatenation operator `##1`. In a singly clocked sequence, the concatenation operator represents one sampling event and joins sequences and expressions together. In multi-clocked sequences, however, the concatenation operator synchronizes between two differently clocked sequences or properties. Consider the following example:

```
@(posedge clk1) sig_a ##1 @(posedge clk2) sig_b
```

Here, `sig_a` is sampled using `clk1` and `sig_b` using `clk2`. The `##1` joins the two differently clocked sequences together, also known as *clock handover* (see Figure 7). In fact, it is illegal to use `##0`, `##2`, or any other cycle delay operator besides `##1`. Likewise, the non-overlapping implication operator can be used between differently clocked sequences, but the overlapping implication operator is not allowed. The clock flows through an implication operator and sequence until another sampling clock is encountered.

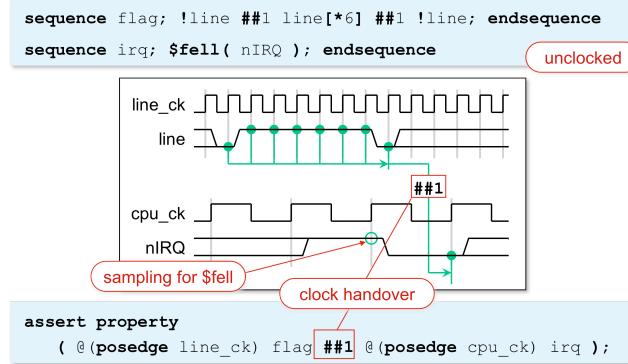


Figure 7. Example of using `##1` clock handover between differently clocked sequences.

The concatenation operator also requires that `clk2` is *strictly subsequent*, i.e., not occurring at the exact same simulation time slot as `clk1`. If it does, then `sig_b` will not be sampled until the next subsequent occurrence of `clk2`.

Using clock handover when crossing clock domains seems rather straightforward, but the duration of `##1` may be arbitrarily short, which may not provide the setup and hold time necessary to avoid timing hazards. Consider the scenario in Figure 8 where the strobe signal is generated in the `src_clk` domain and must be stable for at least 3 cycles in the `dst_clk` domain. An assertion must check that the strobe signal remains stable but also that it has the adequate setup and hold time to be sampled in the `dst_clk` domain.

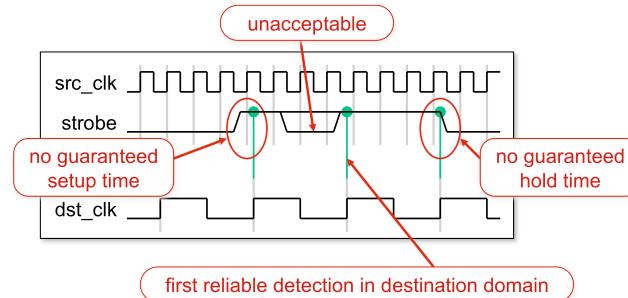


Figure 8. `##1` is arbitrarily short so timing hazards may occur.

One possible solution would be to describe the strobe signal in both clock domains and match the two sequences together. The `intersect` operator can easily accomplish this, but the beginning and end points must occur at the same time or with the same starting and ending clocks. Using `intersect`, the assertion can be described as:

```
assert property (
    @(posedge src_clk) $rose(strobe) |> (
        strobe[*1:$] ##1 1
    ) intersect (
        ##1 @(posedge dst_clk) strobe[*3]
        ##1 @(posedge src_clk) 1
    )
);
```

Since the `intersect` operator requires the same end points, the additional `##1 1` is appended to the `src_clk` sequence so that it can match the end of the `dst_clk` sequence. Likewise, the `dst_clk` sequence switches to the `src_clk` domain to complete its sequence, giving it the same ending point as the `src_clk` sequence. Figure 9 illustrates how the two sequences synchronize together. The assertion satisfies both the stability and timing checks since the sequences combined ensure that the strobe signal remains asserted for the required number of cycles. For a more in-depth discussion on clock domain crossing and jitter as well as an example of a multi-clocked asynchronous FIFO, refer to [5].

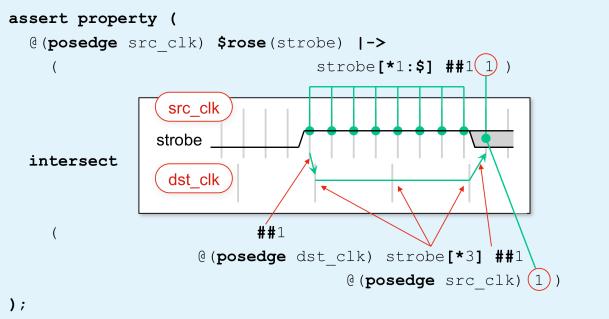


Figure 9. Assertion for clock domain crossing.

Guideline 6: The key to writing assertions for clock domain crossing is proper understanding and handling of the clock handover using `|>` or `##1`.

2.2.2 Asynchronous interface protocols

An asynchronous interface protocol is one that either sends information without an accompanying clock or one that uses an asynchronous handshaking mechanism. While the interface may be asynchronous, the design will still use a clock to sample the asynchronous data or transfer the information over the interface. Because a clock is used, writing assertions for asynchronous interfaces can be as simple as writing a synchronous property, or at worst a multi-clocked sequence since the handshaking signals can be treated like different sampling clocks. With handshaking, the data is setup beforehand so sampling the data is typically not an issue as discussed previously with asynchronous control signals.

2.2.2.1 Serial UART Interface Example

A classic example of an asynchronous protocol is the universal asynchronous receiver / transmitter—commonly known as a UART. More recent UARTs support synchronous transfers, but they still support asynchronously data transfer. The protocol requires that both the receiver and transfer agree beforehand the baud rate to transmit, and then both set their internal clocks to the same frequency.

The protocol is considered asynchronous because no clock signal is transmitted with the data. The sampling of the data is accomplished by using a fast clock to detect the start of the transfer, and then generate a sampling clock from the fast clock that samples in the middle of each data bit. A simple ready-to-send (RTS) and clear-to-send (CTS) handshaking is used to start the data transfer.

Since a sampling clock is used, writing the assertion for the serial data transfer is very straightforward. The beginning of the transfer can be detected by using a sequence to wait for the RTS/CTS handshake:

```
sequence handshake;
  @(posedge rts) 1 ##1 @(posedge cts) 1;
endsequence
```

The ##1 in this sequence performs the clock handover between the two signals. Recall from section 2.1.2 that using `@(posedge rts)` `rts` or `@(posedge cts)` `cts` does not work properly because the value of `rts` and `cts` will be sampled before the rising edge occurs.

To check that the data is sent correctly, a sequence with a local variable is used to capture each bit and then check the parity at the end of the sequence. The internal sampling clock is used to capture the bits:

```
sequence check_trans;
  logic [7:0] tr; // Local variable
  @(posedge sample_clk) 1 ##1 // Skip start bit
    (1, tr[0] = data) ##1 (1, tr[1] = data) ##1
    (1, tr[2] = data) ##1 (1, tr[3] = data) ##1
    (1, tr[4] = data) ##1 (1, tr[5] = data) ##1
    (1, tr[6] = data) ##1 (1, tr[7] = data) ##1
  data === ^tr ##1 // Check parity
  data === 1; // Check stop bit
endsequence
```

With the two sequences defined, the two can be synchronized using the non-overlapping implication operator to handover the clock between the sequences:

```
assert property( handshake |=> check_trans );
```

Notice using multi-clock semantics, both the synchronous and asynchronous behaviors can easily work together as illustrated in Figure 10.

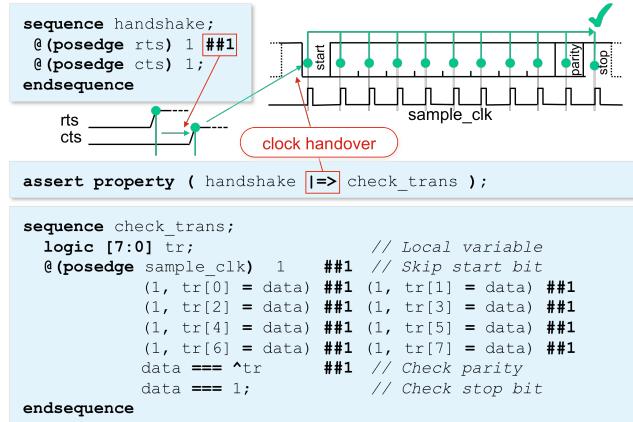


Figure 10. Example using multi-clocked sequences to sample an asynchronous serial transfer.

SystemVerilog also defines the sequence method `.matched` that can be used to detect the end of a sequence sampled in a different clock domain. Using `.matched` instead, the assertion could have been written as:

```
assert property (
  @(posedge sample_clk) handshake.matched |->
  check_trans
);
```

The matched method retains the end state of the sequence until its next evaluation. No clock handover is required because the end state is simply sampled using `sample_clk`. The `.matched` method is often an easier way to describe clock domain crossing than matching sequence end points with the `intersect` method as shown in Figure 9.

2.2.2.2 SCSI Interface Example

While the UART protocol operates asynchronously, data is still sent in a synchronous manner because both sides agree to an transmission frequency. Handshaking is used to start the transfer, but many other protocols use handshaking as their primary way to transfer data. The SCSI⁴ protocol is one such interface used primarily to transfer data to peripheral interfaces like hard disk drives. The SCSI protocol involves both an arbitration handshaking phase and an information transfer handshake (see Figure 11). For purposes of this paper, just the information transfer handshake will be considered.

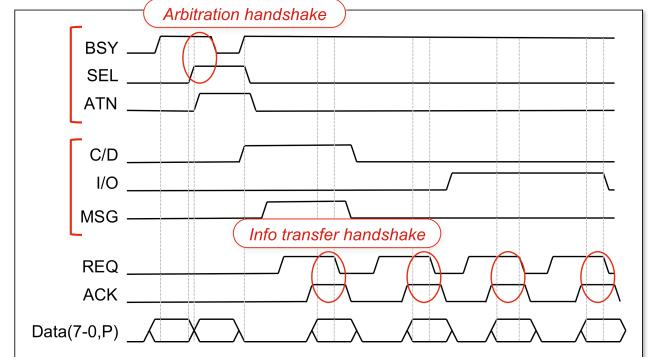


Figure 11. Handshaking in the SCSI interface protocol.

The SCSI protocol passes messages and data, and uses the three signals C/D, I/O, and MSG to define the transfer type. The initiator requests a transfer using REQ and the receiver acknowledges with ACK. The data is transferred over differential pairs and is sampled upon arrival of the ACK signal.

A simple assertion check would be to check that the data sent by the initiator properly appears on the SCSI interface. The assertion should detect when the design is ready to transfer by synchronously sampling the design's FSM and the data to be transferred. While the initiator will assert the REQ signal using its internal clock, it is just as easy to write a multi-clocked property to describe the REQ/ACK handshake. Using a local variable to capture the data, the data is then checked by the assertion on the data bus when the ACK signal arrives:

⁴ SCSI is an acronym for *Small Computer System Interface*.

```

sequence data_cmd; // Valid data command
  !cd && io && !msg;
endsequence

property check_data;
  data_t txdata; // Local variable

  @(posedge clk)
    ( state == TX, txdata == datareg ) |=>
      @(posedge REQ) data_cmd ##1
      @(posedge ACK) databus == txdata;
endproperty

assert property ( check_data );

```

The design's internal clock is used to sample the current FSM state and save the data to be transferred in a local variable. Because the data bus has valid data to sample when ACK arrives, this check is nothing more than a straightforward multi-clocked sequence. The non-overlapping implication and concatenation operators perform the clock handover from the RTL's clock domain to the asynchronous REQ and ACK signals. The flow of this assertion evaluation is illustrated in Figure 12.

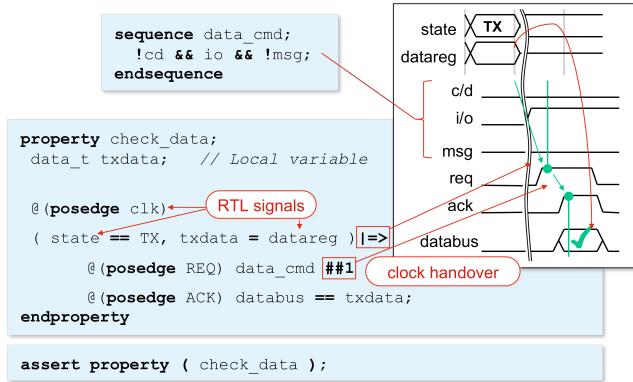


Figure 12. An example handshaking assertion for the SCSI interface.

The SCSI and UART examples demonstrate how communication across an interface with asynchronous handshakes or using sampling is nothing more than a type of a clock (sample) domain crossing, which the SystemVerilog standard has ample support for using multi-clocked properties and sequences. The key to properly written assertions is ensuring proper clock handover between sampling domains.

Guideline 7: An asynchronous interface can be handled using clock handover in the same way as clock domain crossing.

3. SV 1800-2009 ENHANCEMENTS

The Verilog (IEEE 1364-2005) and SystemVerilog (IEEE 1800-2005) standards have been merged into one unified document known as the SystemVerilog 2009 standard[4]. In addition to merging the two languages, many improvements have been made to the assertion language. This section offers a preview of some of the improvements that should have an impact on handling asynchronous behaviors as discussed in this paper (for a very good summary of SVA changes, refer to [1]).

3.1 Asynchronous abort properties

The SVA language defines `disable iff` to disable assertions and their actively evaluating threads. In the new standard, `disable iff` can be applied to assertions globally like the default clocking, making them more concise to write:

```
default disable iff reset;
```

The `disable iff` statement has also been added to the new `cover sequence` statement:

```
cover sequence ( @(event) disable iff ( expr )
  sequence_expr );
```

The `cover sequence` statement is used to record the number of times a sequence is *matched* versus a `cover property` that records the number of times a property *succeeds*.

The new standard has also introduced some new constructs to abort assertions in a similar way as `disable iff` called *abort properties*. There are 2 asynchronous abort properties and 2 synchronous. The asynchronous abort properties take the form:

```
accept_on ( expr ) property_expr
```

```
reject_on ( expr ) property_expr
```

where *expr* represents the *abort condition*. If while the assertion is evaluating the abort condition becomes true, then `accept_on` returns true or `reject_on` returns false; otherwise, the `property_expr` is evaluated. A significant difference with `disable iff` is how the abort condition is evaluated. The expression used with `disable iff` uses the current simulation values (*i.e.*, not sampled but level sensitive); whereas, the abort condition for abort properties are sampled the same property values (*i.e.*, in the *Preponed* region).

3.2 Global clocks

In section 2.1.2.1, synchronously checking using a fast clock was offered as a common solution for checking asynchronous behaviors. The SV-2009 standard defines the semantics for a global assertion clock, which could be used as a fast clock for sampling asynchronous signals. The global clock is defined by using the new keyword `global` with an unnamed clocking block:

```
global clocking @clk; endclocking
```

Once the global clock is defined, many new sample value functions are available:

Past value functions	Future value functions
\$past_gclk(expr)	\$future_gclk(expr)
\$rose_gclk(expr)	\$rising_gclk(expr)
\$fell_gclk(expr)	\$falling_gclk(expr)
\$stable_gclk(expr)	\$steady_gclk(expr)
\$changed_gclk(expr)	\$changing_gclk(expr)

These sample value functions provide new ways to define properties to match asynchronous behaviors.

3.3 Procedural concurrent assertions

Generally, concurrent assertions are written to stand by themselves, but they can also be embedded inside of initial and

always blocks. Inside a procedural block, the event control and enabling conditions are inferred from the context if not declared explicitly. While procedural concurrent assertions are evaluated in the simulator’s *Observed* region ([3], section 17.3), the inputs are sampled as other concurrent assertions in the *Preponed* region. This poses the same issues when checking asynchronous control signals as discussed in section 2.1, and requires the same solutions presented in section 2.1.2.3.

However, the SV-2009 standard has greatly enhanced the semantics for procedural concurrent assertions. All procedural concurrent assertion inputs are still sampled in the *Preponed* region save for one exception—inputs declared as *const* or automatic variables. With *const* and automatic variables, their values are sampled when the assertion is placed on the assertion queue, effectively causing it to sample its inputs as immediate assertions. For example, the asynchronous reset assertion for the counter example could be written as:

```
always @( posedge Reset )
  assert property( const'( Q ) == 0 );
```

An automatic variable will also be treated as a constant and its immediate value used. For example, the assertion could be written as:

```
always @( posedge Reset )
begin
  automatic byte q = Q;
  assert property( q == 0 );
end
```

Unfortunately, this feature does not eliminate the difficulty of sampled values with concurrent assertions. The evaluation of the assertion is delayed until the simulation’s *Observed* region, but the immediate value used for the constant and automatic values is taken when the procedural concurrent assertion is *scheduled*. Inside a module, always and initial blocks execute in the simulator’s *Active* region so the assertion will be scheduled in the *Active* region before the design has updated the value of Q upon reset in the *NBA* region. In order to make this new feature work, either a timing control would be required before the assertion to delay its scheduling—which is explicitly prohibited by the standard—or the procedural concurrent assertion could be used inside *program* block to delay its scheduling until the *Reactive* region (see section 2.1.2.2.1).

Another change that may benefit describing asynchronous behavior is how the enabling condition for procedural concurrent assertions is handled. For example, an *if* statement could be used around a procedural concurrent assertion:

```
always @(posedge clk)
  if ( a )           // Enabling condition
    assert property( check_data );
```

In the current SV-2005 standard, the values used for the enabling condition are sampled in the same manner as the assertion inputs. According to the SV-2009 standard, the enabling conditions will use the immediate values just like constant and automatic inputs. If used in a *program* block, this will provide another means of checking behavior after the asynchronous signal(s) occur.

3.4 Checkers

The SV-2009 standard introduces a new verification block known as a checker. A checker can contain assertions, covergroups,

procedural blocks of code, and generate statements much like a module, but modules, programs, interfaces, or packages cannot be instantiated within a checker. Checkers can be used most anywhere a concurrent assertion is used with the exception of *fork-join* statements. Checkers can also be passed arguments, which will work very much the same as procedural concurrent assertions. If a checker’s arguments are declared as *const* or *automatic*, then the immediate values from the checker’s context will be passed. As with procedural concurrent assertions, checkers could be adapted to check asynchronous behaviors if used in the same manner described in the previous section. For an in-depth look at checkers, refer to [1].

3.5 Deferred Immediate Assertions

Deferred assertions are immediate assertions that are delayed in their reporting. The purpose of a deferred assertion is to avoid false failures due to glitches on combinational circuitry. The SV-2009 standard defines the semantics for deferred assertions so that their evaluation works like an immediate assertion, but their action blocks report in the *Reactive* region. A deferred assertion may be used within or outside of a procedural block and is written as follows:

```
assert #0 ( expression ) pass_action_block else
               failure_action_block;
```

A deferred assertion follows the same form as an immediate assertion but with the additional *#0* notation. Since deferred assertions sample their inputs as regular immediate assertions, the deferred assertions evaluation would need to be delayed to handle asynchronous events as shown previously in this paper (see 2.1.2.2).

4. CONCLUSION

In this paper, SystemVerilog assertions have been examined in detail for their ability to handle describing and checking asynchronous behaviors. The harder asynchronous behavior to check are the asynchronous control signals that immediately affect the design like enables and resets. The difficulty lies in how the assertion inputs are sampled preceding the asynchronous trigger. This difficulty can be overcome by using the following guidelines:

- *Guideline 1: Always use disable iff to asynchronously terminate active assertions threads.*
- *Guideline 2: Delay either the evaluation of the checking or the sampling of the assertion inputs.*
- *Guideline 3: Immediate assertions can check asynchronous events if evaluated in the Observed or Reactive simulation regions.*
- *Guideline 4: Concurrent assertions can check asynchronous events by delaying the asynchronous control or calling a subroutine from a matched sequence.*
- *Guideline 5: For timing simulations, synchronously checking the design’s behavior upon the asynchronous event is probably the best overall solution.*

The second class of asynchronous behaviors is communication between modules. The sender and receiver both operate synchronously, but since they do not pass a clock the communication becomes asynchronous and either the information

must be sampled synchronously or with a handshaking scheme. Checking asynchronously communication can be treated as nothing more than a SVA multi-clocked sequence. SystemVerilog has well-defined semantics for clock handover and clock flow through a sequence so the difficulty lies in synchronizing between the two clock domains. With proper clock handover, writing sequences to check asynchronous communication is a straightforward task. Handling asynchronous communication can be summarized using the following guidelines:

- *Guideline 6: Clock domain crossing is handled using `|=>` or `##1` for clock handover.*
- *Guideline 7: An asynchronous interface can be handled using clock handover in the same way as clock domain crossing.*

Coverage can also be measured using these techniques. While not discussed in this paper, the same assertion properties can be used by *cover property* and the same asynchronous behaviors recorded. By following these simple guidelines, most—if not all—kinds of asynchronous behaviors can be properly handled, increasing overall verification confidence.

5. ACKNOWLEDGEMENTS

I would like to thank and acknowledge the excellent engineers at Doulos who have developed the SystemVerilog and SVA training courses. In particular, I have borrowed diagrams from the SV scheduling and multi-clock assertion materials.

Many thanks to my former colleague, Jonathan Bromley (Verilabs), who has provided incredible insights and invaluable comments in reviewing this paper. As always, Jonathan has an incredible depth of knowledge and understanding of SystemVerilog, and challenges me to research my ideas to much greater depths.

Also, many thanks to Matt Homiller (Sun Microsystems) for his review and comments of this paper, and special thanks to Scott Little and John Havlicek (Freescale) for their many helpful comments and corrections.

6. REFERENCES

- [1] Dudani, S., Cerny, E., Korchemny, D., Seligman, E., and Piper, L. 2009. “Verification case studies: evolution from SVA 2005 to SVA 2009.” Proceedings of DVCon (February 24-25, 2009).
- [2] IEEE P1800/D8 Draft Standard for SystemVerilog—Unified Hardware Design, Specification, and Verification Language. IEEE Computer Society, New York, 2008.
- [3] IEEE Std 1800™-2005. IEEE Standard for SystemVerilog—Unified Hardware Design, Specification, and Verification Language. IEEE Computer Society, New York, 2005.
- [4] IEEE Std 1800™-2009. IEEE Standard for SystemVerilog—Unified Hardware Design, Specification, and Verification Language. IEEE Computer Society, New York, 2009.
- [5] Litterick, M. 2006. “Pragmatic Simulation-Based Verification of Clock Domain Crossing Signals and Jitter Using SystemVerilog Assertions.” Proceedings of DVCon (February 22-24, 2006). http://www.verilab.com/files/sva_cdc_presentation_dvcon2006.pdf http://www.verilab.com/files/sva_cdc_paper_dvcon2006.pdf