Practical Asynchronous SystemVerilog Assertions

Doug Smith
Doulos, San Jose, CA 95119

***Abstract*—Nearly all digital designs have asynchronous behaviors. For example, designs often have asynchronous resets or asynchronous inputs like interrupts or ready signals. Some RTL designs are inherently asynchronous in nature as in the case of power management modules receiving off-chip boot up signals. Asynchronous behaviors also appear in the form of asynchronous handshaking protocols for peripheral devices or in the case of synchronizers between clock domain crossings. SystemVerilog assertions (SVA) provide a great way of testing and describing design behaviors. However, using SVA to capture asynchronous behavior is not always straightforward due to the scheduling semantics of SystemVerilog. While triggering on an asynchronous event is easy enough, the sampling of the assertion inputs is either dependent on its context as in the case of immediate assertions or synchronous by nature as in the case of concurrent assertions. Often, asynchronous events occur before the design has updated its state, requiring the checking of the RTL to be delayed. Further, the timing of asynchronous events may be hard to predict, making it harder to describe using an assertion. In this paper, eight common asynchronous scenarios are presented and SVA solutions for checking them in simulation. In additional, alternative approaches using a fast clock or functional coverage are presented as both a portable simulation solution and a working solution for other verification flows.**

I. Introduction

The scheduling semantics of SystemVerilog is the cause for a number of interesting behaviors during simulation. For example, the checking of glitchy logic may result in false failure messages for logic that eventually resolves to passing values. How a signal is updated using a blocking or non-blocking assignment also affects the order that a signal resolves and the values available to checkers. To complicate evaluation further, some SystemVerilog constructs evaluate during different regions of the scheduler.

Figure 1 illustrates a simplified view of the SystemVerilog simulation scheduler. Combinational logic is typically described using blocking assignments, which are updated in the Active region unless delayed with a #0 to schedule in the Inactive region. Synchronous logic is typically described using non-blocking assignments, which are updated later in the scheduler during the NBA region.

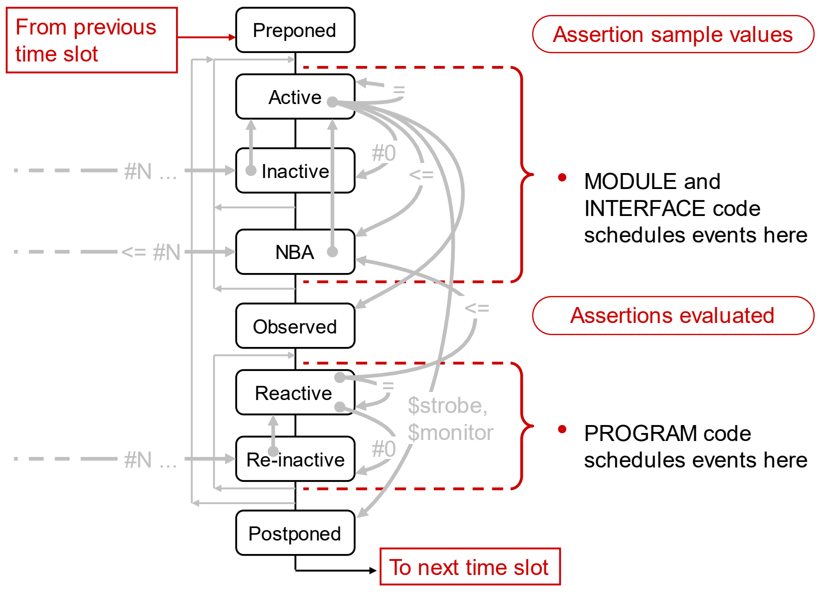

Figure 1: Simplified SystemVerilog scheduler illustration.

Where the difficulty arises in writing asynchronous assertions is when to sample the assertion inputs. Concurrent assertions sample their inputs in the Preponed region, which occurs before any of the asynchronous behavior has occurred (typically in the Active region). Immediate assertions sample their values in the context that they occur, which is typically the Active region. As a result, to really make asynchronous assertions work properly, their inputs need sampled after the RTL is updated, i.e., after the NBA region in one of the later regions like the Observed or Reactive regions.

While there are a number of ways to accomplish a delayed input sampling, simulator support for SystemVerilog syntax still varies across the major simulators even though SystemVerilog has been used in the industry for several decades. Ambiguity in the standard and differences in interpretation may account for some inconsistencies, but for many constructs the standard is quite clear how and when they should evaluate. Even constructs like delayed immediate assertions, which are intended to handle glitchy behaviors, are not evaluated per the standard across all simulators. For a detailed look at many ways to delay assertion input sampling, see [1]. This paper, however, focuses only on the well-supported SystemVerilog syntax portable across the major simulators[1].

Delaying input sampling is only an issue for *asynchronous controls*. Asynchronous behaviors occur in two forms: (1) *asynchronous controls* and (2) *asynchronous communication*. Asynchronous communication occurs for bus protocols like UART and SCSI where no clock is transmitted so the transfer is asynchronous even though the components may be transmitting synchronously internally. In the case of asynchronous communication, the handshaking protocol is easily matched using multi-clocked properties, which are well-defined in the SystemVerilog standard; therefore, the author refers the reader to an earlier paper explaining the handling of these scenarios [1]. For this discussion, only asynchronous control scenarios will be considered.

II. Common Methods for Handling Asynchronous Checking

Asynchronous events are commonly checked in one of two ways—either oversampling using a fast clock, or through an event-based method.

A. *Fast clock / Oversampling*

The easiest way to test asynchronous behaviors is to use a fast clock and oversample. If the clock is fast enough, even glitchy behavior should be detectable, but at the cost of being able to only approximate any timing window checks.

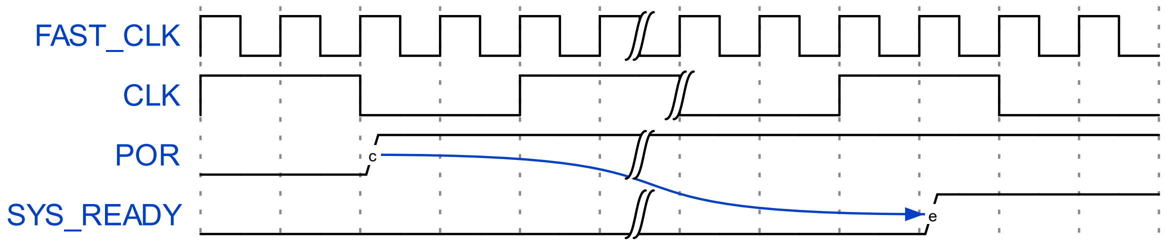

Figure 2: Using fast clock oversampling to capture asynchronous events.

Using a fast clock, asynchronous assertions become just a form of a synchronous concurrent assertions. For example,

```
default clocking cb @(posedge FAST_CLK); endclocking

assert property ( $rose(POR) |-> ##[0:$] SYS_READY );
```

For most applications, this is sufficient. In cases of very glitchy RTL behavior, oversampling may be not adequate, though in theory, a fast enough clock should be able to catch about anything. The advantage of this approach is that it works for other verification flows as well such as emulation, prototyping, and even formal verification if the design is synthesizable.

[1] While the author considers Riviera Pro a major simulator, some of the multi-clocked solutions do not work with Riviera at the time of this writing. Most solutions in this paper will work with Riviera and the alternative coverage approach in C.1 offers a multi-clock solution for Riviera.

B. Event-based methods

Simulators are event driven so capturing asynchronous behaviors using events is very straightforward. It is sufficient to just wait on an asynchronous signal and then perform the necessary checking. However, the difficultly occurs depending on the type of behavior being checked.

For checking *overlapping* behaviors that occurs on the same timestep as the asynchronous event, the ordering of events is important. For example, RTL updated in the Non-Blocking Assignment (NBA) region will not get updated until a later region in the scheduler than the asynchronous event, requiring the checking to be delayed. When checking *non-overlapping* behavior, the checking can be treated as a multi-clocked SVA. Unfortunately, describing some non-overlapping behavior is complicated, and the major simulators have inconsistent and often incomplete support for multi-clocked property syntax.

A general technique that is very portable and easy to implement is using a simple form of coverage. For each asynchronous event, assign a variable or flag to indicate that it has occurred:

```
bit cover_por = 0;
cover property ( @(posedge por) 1 ) cover_por = 1;
```

Here, a cover property is used to detect the asynchronous power-on-reset and set a coverage flag. An *always*, *initial*, or even *continuous assignment* could be used as well. Likewise, an associative array can be used to store all the coverage events in one place; however, some simulators do not like the use of associative arrays within SystemVerilog concurrent assertions. Once the asynchronous event is captured in the coverage flag, writing assertions to check if an event has occurred is very simple:

```
assert property ( @(posedge system_ready) cover_por );
```

Where the coverage method breaks down is in the case of *overlapping* events on the same timestep. Since SystemVerilog does not guarantee the ordering of events, the coverage flag must be set before it is checked. This paper will explore in detail different ways to delay the setting and checking of these overlapping events in the next section.

The strength of the coverage method is that it makes writing more sophisticated scenarios much easier to write. For example, suppose a design requirement says that once an asynchronous signal occurs, it must not occur again within 10 clock cycles. This behavior could be captured using a fork and a coverage flag that toggles like this:

```
bit cov_event;

always
begin
   cov_event = 0;
   @(posedge myevent) cov_event = 1;
   fork
      @(posedge myevent) cov_event = 0;
      repeat (10) @(posedge clk);
   join_any
   disable fork
end

assert property ( @(posedge clk) cov_event |-> cov_event throughout ##10 );
```

An alternative approach to using a simple coverage method is using a *multi-clocked* property. Each asynchronous event is treated just as another clocking event. The SystemVerilog standard has well-defined this scenario and it generally works well across the simulators. However, where it breaks down is with overlapping events and in some of the partial and inconsistent implementation in the simulators. For example, the *.triggered* method does not work on multi-clocked sequences on all simulators, nor does the overlapping zero-delay clock hand-over operator (##0), which tends to give various and unexpected results. Given these challenges, the following section discusses a number of common scenarios and several practical solutions for implementing working asynchronous assertions.

III. Common Asynchronous Control Scenarios

When writing assertions, it is important to consider the thoroughness of the checking—both the cause-and-effect and the effect-and-its-cause. In this section, common asynchronous scenarios are described for both along with recommended approaches for writing the assertion checks.

C. Cause-and-effect scenarios

C.1 Asynchronous signal causes another asynchronous behavior

The simplest asynchronous control scenario is when an asynchronous behavior triggers another signal as shown in Figure 3.

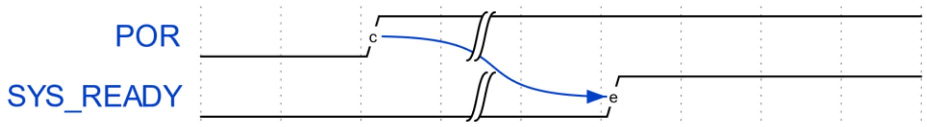

Figure 3: Async control causes another async behavior.

This scenario is simply a version of a multi-clocked assertion where the asynchronous signals become the sampling events. For example,

```
assert property ( @(posedge POR) 1 |-> @(posedge SYS_READY) 1 );
```

While this assertion will match the asynchronous behavior, it is a *weak* property, meaning it will not flag an error if the SYS_READY signal never occurs, which is the whole point of the assertion check. To fix this, the assertion can be made strong by using the *strong*() or *s_eventually*() keyword:

```
assert property (@(posedge POR) 1 |-> @(posedge SYS_READY) strong(1[*1:$]));

// Recommended solution – most portable
assert property (@(posedge POR) 1 |-> @(posedge SYS_READY) s_eventually(1));
```

With the strong attribute, simulation will flag an error if SYS_READY never occurs. While both versions should work, the first assertion using only the *strong()* keyword has limited support across simulators; whereas, all major simulators have good support for *s_eventually()*.

An alternative approach is to set a flag indicating the asynchronous event occurred and then check the flag. As mentioned in the previous section, the flag becomes a form of functional coverage since it keeps track of the number of occurrences. An easy implementation is using an associative array as follows:

```
    bit coverage[string];
    cover property ( @(posedge POR      ) 1 ) coverage["POR"      ]++;
    cover property ( @(posedge SYS_READY) 1 ) coverage["SYS_READY"]++;
```

When the cover property triggers, an entry into the array is set. A final block is used to assert the expected asynchronous event occurs before simulation finishes:

```
    final begin
      if ( coverage["POR"] ) begin
         assert( coverage["SYS_READY"] == coverage["POR"] ) else
            $error( ... );
      end
    end
```

By comparing the number of coverage counts, multiple asynchronous event pairs can be checked to see if every cause received its corresponding effect. Since this checking is done at the end of simulation, the timing or ordering of the events cannot be checked (unless the simulation time is recorded) so this is most useful in the case of one-off occurring events like power-up signals that occur once during simulation.

C.2 Asynchronous signal causes RTL updates

The most common scenario for checking asynchronous behaviors is when the RTL updates from an asynchronous event occur as in the case of an asynchronous reset (Figure 4). Using concurrent assertions is particularly difficult in this case because the inputs for SVA are the RTL values *before* the asynchronous signal occurs (e.g., reset).

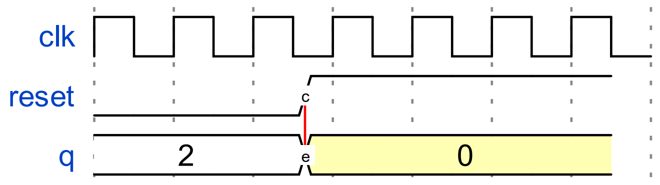

Figure 4: Asynchronous signal causes RTL update.

The IEEE-1800 standard defines concurrent assertion input sampling in the Preponed region of simulation, which occurs before the asynchronous event is scheduled in the Active region. Therefore, the following assertion will sample the value of *q* as "2" in Figure 4 instead of "0":

```
assert property ( @(posedge reset) q == 0 );  // Fails: q == 2
```

There are a number of possible ways to create a working asynchronous assertion, but results vary across the major simulators. However, here are a several proven and portable solutions for handling this scenario:

(1) *Program blocks*
SystemVerilog *program* blocks execute in the later regions of the scheduler. Programs sample their inputs in the Observed region, which runs after the RTL has been updated in the Non-Blocking Assignment region. A program's code executes in the Reactive region so using an immediate assertion, which executes in the context it is invoked, delays the checking until after the RTL has already been updated:

```
program async_asserts(input bit async_event, input bit [7:0] rtl_signal);
  initial
    forever
      @(posedge async_event)
          assert ( rtl_signal == 1 )
             else $error( ... );
endprogram
```

The program block just needs instantiated in the testbench with the appropriate inputs:

```
async_asserts asserts1 ( reset, q );
```

The program block can also be nested within the testbench without passing arguments and it will be implicitly instantiated (provided the simulator supports it):

```
program async_asserts;
  initial
    forever
      @(posedge reset)
          assert ( q == ... );
endprogram
```

Using a *program* block is perhaps the most sure-fire way of correctly sampling asynchronous behaviors. It also serves as a useful block for collecting all of the immediate assertions in one place and binding into the design under test. For checking at the absolute last possible point during a timestep, a *final deferred immediate assertion* (assert final) can be used to sample in the Postponed region.

One drawback to using an immediate assertion is that the temporal syntax of SVA is unavailable. However, SystemVerilog provides a way to combine the SVA temporal syntax along with immediate assertions by using the *expect* statement. With *expect*, a named sequence is not required and the temporal syntax can be written in one line similar to a concurrent assertion. The SystemVerilog standard says that the action block of the expect statement is scheduled in the Observed region ([2], 16.17); therefore, the *expect* statement, which uses SVA temporal syntax, can be used to describe behavior as with an *assert property* statement, but using an immediate assertion in its action block as follows:

```
program async_asserts(input bit async_event, input bit rtl_signal);
    initial
      forever
        expect ( @(posedge async_event) 1 /* Temporal syntax here */ )
            ast_expect: assert ( rtl_signal == 1 );
endprogram
```

Note, some simulators may issue a warning with this example about placing an assertion within the action block of another assert, in which case an 'if' statement could be used instead to remove any warnings.

(2) *Sequence events*
Another way to combine the SVA temporal syntax with immediate assertions is through the use of *sequence events*. The standard says that a process executing a sequence event will block until the sequence reaches its end point and then it resumes execution in the Observed region ([2], 9.4.2.4). Using this approach, the *program* block's delayed execution can be emulated:

```
  sequence seq_async_event;
    @(posedge async_event) 1;
  endsequence

  always
    @(seq_async_event) assert ( rtl_signal == ... );
```

In this example, the immediate assertion is delayed until after the RTL has already been updated, making this an easy solution to implement without the use of a program block.

(3) *Procedural concurrent assertions*
Similar to using a sequence event is embedding a concurrent assertion into a process, making it a *procedural concurrent assertion*. Procedural concurrent assertions allow temporal sequences and properties as regular concurrent assertions, but they infer their sampling event from the context where they are embedded. Since they may be included in combinational logic, the simulator maintains a *procedural assertion queue* so that the last evaluation of the assert is used in order to avoid false negatives from glitchy combinational behaviors. To accomplish this, the procedural concurrent assertion matures in the Observed region so that all design updates have already been completed. Therefore, using a procedural concurrent assertion, an immediate assertion can be delayed until the Observed region just as waiting on a sequence event:

```
always @(posedge async_event)
   assert property ( 1 )
      assert( rtl_signal == ... );
```

The assertion can be re-written to move the asynchronous event inside the procedural concurrent assertion by using an *always_comb* process:

```
  always_comb
      assert property ( @(posedge async_event) 1 )
          assert( rtl_signal == ... );
```

Placing the check inside the procedural concurrent assert would be ideal and is technically possible. The SystemVerilog standard ([2], 16.14.6.1) specifies a way to delay the sampling of the procedural concurrent assertion's input values by using an automatic variable or a *const'()* cast; however, this functionality still remains unavailable in most major simulators.

(4) *A timing delay*
The simpliest solution is to delay the asynchronous control signal and then use it in an assert property as follows:

```
    assign #1 delayed_event = async_event;
    assert property ( @(posedge delayed_event) rtl_signal == ... );
```

The major simulators have good support for this solution since delays have been part of Verilog since its beginning. Of course, delaying by 1 timestep means that the checking is not occurring on the same timestep as the asynchronous event, which may concern those trying to test glitchy combinational logic. Perhaps a better alternative is to use a time delay of *#1step*:

```
    assign #1step delayed_event = async_event;
    assert property ( @(posedge delayed_event) rtl_signal == ... );
```

The *#1step* causes the asynchronous signal to be delayed until essentially the Postponed region of the current timestep (or technically, the Active region of 1 time precision step away) so that the RTL signal is checked at the last possible moment. Major simulators have good support for this solution, but may issue a message about using the local time precision.

C.3 Asynchronous signal causes another asynchronous event resulting in RTL updates

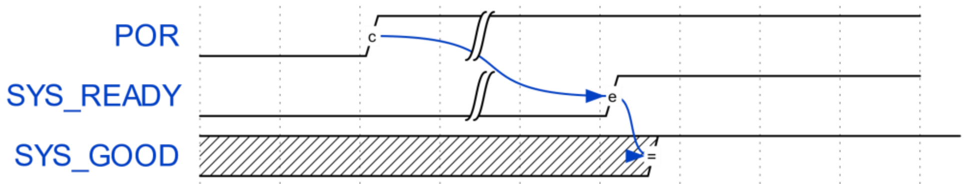

Figure 5: Asynchronous signal causes another asynchronous event and updates the RTL.

In this scenario, one asynchronous event causes another asynchronous event, resulting in the RTL coming up to a known state. If the RTL requires time to settle and update, then this is simply a multi-clock scenario. Modifying the solution from *C.1*, this behaviour can be matched by chaining implications[2]:

```
  assert property (@(posedge POR) 1 |-> @(posedge SYS_READY) s_eventually(1)
|-> @(posedge SYS_GOOD) s_eventually(1));
```

If the RTL updates immediately on the second event, then the delayed evaluation of the assertion action block can be used to check the RTL value:

```
  always_comb
    assert property ( @(posedge POR) 1 |->
                         @(posedge SYS_READY) s_eventually(1))
                             assert(SYS_GOOD == ...);
```

[2] Note, both |-> and |=> implications work; however, the non-overlapping implication (|=>) causes errors with Riviera Pro.

C.4 Asynchronous timing window cause-and-effect

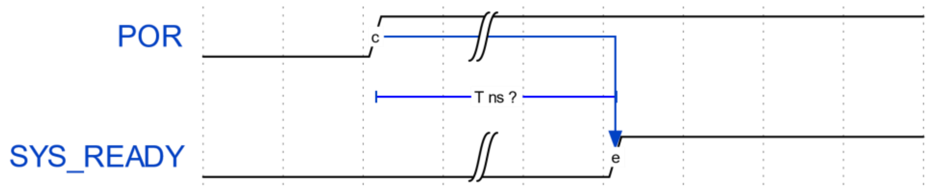

Figure 6: Asynchronous timing window cause-and-effect.

Sometimes applications arise where the timing window needs checked between asynchronous events. This can be accomplished using a named property and a couple local variables:

```
property prop_check_timing;
   realtime start;
   realtime finish;

   @(posedge POR)       (1, start  = $realtime) |->
   @(posedge SYS_READY) (1, finish = $realtime) ##0
   (finish - start) <= timing_window;
endproperty

assert property ( prop_check_timing );
```

In property prop_check_timing, when the POR event occurs, the current simulation time is captured. Upon SYS_READY occurring, the simulation time is captured again and the timing window calculated for comparison. If the timing window is less than or equal to the design requirement, then the assertion check passes. Of course, the timing window check only occurs if the second asynchronous event occurs; therefore, an additional cause-effect check (using a *strong* property as in C.1) should also be included to ensure that the SYS_READY actually occurs.

D. Effect and its cause

It is only natural to think in terms of cause and effect. However, what if the effect happens without the cause? In fact, the assertions above would not evaluate if the preconditions were never met. In order to be completely thorough, assertions need written looking in both directions, which requires assertions for both cause-and-effect and effect-and-its-cause.

D.1 Asynchronous signal caused by another asynchronous signal

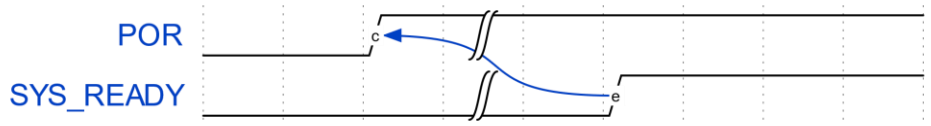

Figure 7: Asynchronous signal caused by another asynchronous signal.

The simplest effect-cause scenario is when an asynchronous signal occurs and the cause should have occurred. The simplest way to implement this is with a cover property:

```
bit cov_por;
cover  property ( @(posedge POR)   1  ) cov_por = 1;
assert property ( @(posedge SYS_READY) cov_por );
```

A coverage associate array could also be used, but not all simulators support those inside of an assertion. The solution shown here assumes that both the effect and the cause occur on different timesteps. If there is a possibility of both events occuring simultaneously, then a solution that sets the coverage bit sooner than a cover property (which evaluates in the Observed region) may be needed. Such an implementation is discussed in the next section.

A slightly less portable solution is using a named sequence and the trigger method:

```
sequence seq_past_por;
   @(posedge POR) 1 ##1 @(posedge SYS_READY) 1;
endsequence

assert property ( @(posedge SYS_READY) seq_past_por.triggered );
```

This solution works quite well for checking effects and their cause. If it is expected that both effect and cause will occur on the same timestep, the sequence could be modified to use the sequence *fusion* operator instead to handle the overlapping timestep:

```
sequence seq_past_por;
   @(posedge POR) 1 ##0 @(posedge SYS_READY) 1;
endsequence
```

Unfortunately, support and behavior of the zero-delay overlapping clock-handover operator within multiclocked sequences is quite varied and inconsistent across the major simulators, meaning it probably should be avoided and the *.triggered* approach will not suffice for overlapping events.

D.2 RTL update caused by an asynchronous event

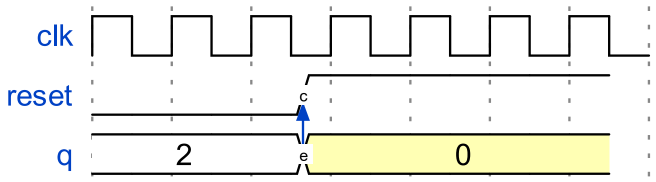

Figure 8: RTL update caused by an asynchronous event.

Writing an assertion for this scneario is particularly tricky. A coverage bit can be used to record the asynchronous event; however, a cover property will not work because the action block evaluates in the later Observed region. In other words, the cover bit will never be updated in time for the assertion check if the asynchronous event occurs on the same timestep. Instead, the coverage bit needs updated in the Active region, and the assertion checking needs deferred until the Observed region. The solution is to use an always block to update the coverage bit in the Active region, and the action block of a procedural concurrent assertion to delay the checking to the Observed region:

```
bit cov_reset;
always @(posedge reset) cov_reset = 1;

always_comb
    assert property ( @(q) 1 )
       assert ( cov_reset ) else $error ( … );
```

It is probably best to use @(q) for any change in q versus a posedge or negedge since the IEEE-1800 standard says that only the least significant bit is used in edge detection.

One drawback with this solution is that it only works once. If it needs checked repeatedly, the always process needs modified to the following:

```
bit cov_reset;
always @(posedge reset) cov_reset = 1;
```

```
// Continued on next page ...
always @(q)
   assert property ( 1 )
      assert ( cov_reset )
         cov_reset = 0;
      else
         $error( ... );
```

The *always_comb* does not like two processes modifying the *cov_reset* coverage bit so this needs replaced with a simple *always* statement.

D.3 Multiple causes for a sequence of asynchronous events

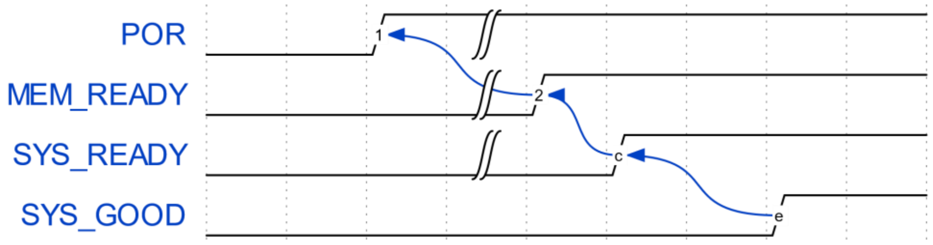

Figure 9: Multiple causes for a sequence of asynchronous events.

When multiple causes need checked for a particular effect, a simple effect-cause assertion as shown in D.1 can be implemented for each effect and its cause. Provided that none of the causes occur on the same timestep as their effects, either solution of D.1 could be used individually for each signal combination or all events combined into one assertion as follows:

```
bit cov_por, cov_mem_ready, cov_sys_ready;
cover  property ( @(posedge POR)       1) cov_por = 1;
cover  property ( @(posedge MEM_READY) 1) cov_mem_ready = 1;
cover  property ( @(posedge SYS_READY) 1) cov_sys_ready = 1;

 assert property ( @(posedge SYS_GOOD)   cov_por &&
                                         cov_mem_ready &&
                                         cov_sys_ready );
```

Provided your simulator supports it, triggering on a named sequence is easier to construct:

```
sequence seq_multi_cause;
   @(posedge POR      ) 1 ##1 @(posedge MEM_READY) 1 ##1
   @(posedge SYS_READY) 1 ##1 @(posedge SYS_GOOD ) 1;
endsequence

 assert property ( @(posedge SYS_GOOD) seq_multi_cause.triggered );
```

If the effect could occur on the same timestep as the cause, then the solution presented in D.2 should be used to handle setting the coverage bit sooner during the Active region and avoid using the *.triggered* approach.

D.4 Asynchronous timing window effect-and-cause

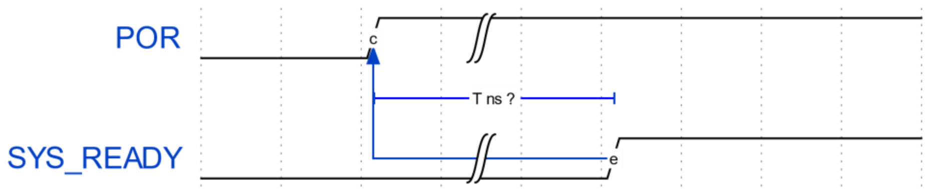

Figure 10: Asynchronous timing window effect-and-cause.

Another scenario you may need to test is if the effect-cause window meets a certain timing window. This can easily be written by modifying the solution of C.4 as follows:

```
bit cov_por;
realtime start;
cover property ( @(posedge por) 1 ) begin
   cov_por = 1;
   start = $realtime;
end

assert property ( @(posedge SYS_READY) cov_por &&
                  (($realtime - start) <= timing_window ));
```

In this example, when the cause is detected, power-on-reset (POR), both a coverage bit and start time are recorded. When the effect occurs, the start time is subtracted from the current time ($realtime) and then compared with the timing window. If either the cause is missing or the timing window is not met, the assertion will fire. Note, this scenario assumes that both the effect and cause do not occur on the same timestep. In such a case, a solution similar to D.2 would need to be implemented to delay the setting of the coverage bit.

V. SUMMARY

While checking asynchronous events should be rather straightforward with SystemVerilog assertions, the scheduling semantics of a SystemVerilog simulator often results in either useless assertions or unexpected behaviors. Accurately matching asynchronous behavior is best accomplished when understanding the region that different language constructs evaluate during a timestep. The task is further complicated by partial and inconsistent support for SystemVerilog constructs across the major simulations. However, using the approaches and templates presented in this paper will guarantee not only that your asynchronous assertion captures the desired behavior, but it will compile and run as expected across all major simulators. While there exists a number of other possible approaches like using *final deferred immediate assertions* or *const'()* casted inputs for example, we must wait for better support in the industry from simulator vendors. In the meantime, the best methodology for checking asynchronous behaviors is keeping them simple, breaking them into cause-and-effect and effect-and-its-cause, and using the robust methods as presented in this paper.

REFERENCES

[1] D. Smith, "Asynchronous Behaviors Meet Their Match With SystemVerilog Assertions," *DVCon San Jose*, February 2010. https://www.doulos.com/knowhow/systemverilog/asynchronous-behaviors-meet-their-match-with-systemverilog-assertions/.

[2] IEEE Standard for SystemVerilog—Unified Hardware Design,Specification, and Verification Language, 2017.