Subject: Scheduling Region Questions and Problems of new SystemVerilog commands

I have read and re-read sections 14-17 of the SystemVerilog 3.1 Standard multiple times and am still confused about exactly when events are scheduled. I think part of the problem is that some of the descriptions apply to program variables and other descriptions apply to RTL-design signals (and the distinction between the two is not well delineated).

Please see notes and questions below. Note: all sections and page numbers are with respect to the SystemVerilog 3.1 final document.

When are output drives updated?

This is very confusing. I believe section 16.4 holds the key (??)
All "design signals" assigned from a program block are assigned using nonblocking assignments and are scheduled into the NBA region (??)
All program variables assigned from a program block are assigned using blocking assignments and are scheduled into the Reactive region (??)

See the following passages:

Section 14.3, 2nd to last paragraph at the bottom of page 126
Code specified in the program block, and pass/fail code from property expressions, are scheduled to occur in the Reactive region.

15.14.1 Drives and nonblocking assignments - 1st paragraph - page 140
Synchronous signal drives are processed as nonblocking assignments.

15.14.2 Drive value resolution - last paragraph - page 140
Clock-domain outputs driving a net (i.e. through different ports) cause the net to be driven to its resolved signal value. When a clock-domain output corresponds to a wire, a driver for that wire is created that is updated as if by a continuous assignment from a register inside the clock-domain that is updated as a nonblocking assignment.

16.1 Introduction (informative) - 3rd numbered item - page 141
The program block serves three basic purposes:
...
3) It provides a syntactic context that specifies execution in the Reactive region.

16.4 Eliminating testbench races - 2nd paragraph - page 143 - (I believe this paragraph held the key)
To avoid the races inherent in the Verilog event scheduler, program statements are scheduled to execute in the Reactive region, after all clocks in the design have triggered and the design has settled to its steady state. In addition, design signals driven from within the program must be assigned using nonblocking assignments. Thus, even signals driven with no delay are propagated into the design as one event. With this behavior, correct cycle semantics can be modeled without races; thereby making program-based testbenches compatible with clocked assertions and formal tools.

If my interpretation is correct, I could propose clarifications (most clarifications are proposed below).

The SystemVerilog 3.1 Standard lacks a good simple example that shows: RTL combinational always block (blocking assignments in the Active events region), RTL sequential always block (nonblocking assignments in the NBA region), a program block, and a clocking domain with @(posedge clk) clocking event and default inputs (no delay) and negedge outputs making assignments to both design signals (NBA region) and program variables (Reactive region) and an assertion to sample inputs in the Preponed region, evaluate the inputs in the Observed region and do pass/fail assignments in the Reactive region.

Section 15.3 - 4th paragraph starts with the following sentence:
An input skew of **`#0`** forces a skew of zero. ...
I now believe that after this sentence, most if not all other references to "zero skew" and "zero delay" are actually referring to "explicit #0-delays." This will be confusing to most ASIC & FPGA RTL designers. RTL designers often refer to 0-delay RTL designs to mean RTL designs with no specified delays.

I propose that we modify the appropriate sections of the LRM with the following terminology:

"zero delay" or "0-delay" should refer to a design with no delay specified.

"explicit #0-delay" should refer to making explicit #0 procedural assignments, explicit #0 clocking domain input skews and explicit #0 clocking domain output skews.

I think we need to clarify certain sections and I have proposed clarification wording (but I am not sure my clarifications are correct - I need committee review and feedback).

Other minor typos are noted in red.

Regards - Cliff

Section 14 - Scheduling Semantics

14.3 The stratified event scheduler

The Observed and Reactive regions are new in the SystemVerilog 3.1 standard, and events are only scheduled into these new regions from new language constructs.

The Observed region is for the evaluation of the property expressions with inputs that were sampled in the Preponed region (see section 17.3) when they are triggered. It is essential that the signals feeding and producing all the clocks to the property expressions have stabilized, so that the next state of the property expressions can be calculated deterministically. A criterion for this determinism is that the property evaluations must only occur once in any clock triggering time slot. During the property evaluation, pass/fail code shall be scheduled to be executed in the Reactive region of the current time slot.

(Hence: Preponed sampled - Observed assertion evaluation - Reactive pass/fail code)

The sampling time of sampled data for property expressions is controlled in the clock domain block. The new **`#1step`** sampling delay provides the ability to sample data immediately before entering the current time slot, and is a preferred construct over other equivalent constructs because it allows the **`1step`** time delay to be parameterized. This **`#1step`** construct is a conceptual mechanism that provides a method for defining when sampling takes place, and does not require that an event be created in this previous time slot. Conceptually this **`#1step`** sampling is identical to taking the data samples in the Preponed region of the current time slot.

*** The following paragraph seems to conflict with section 15.14.1 and is a key point of confusion. I think my wording is what the intention is supposed to be but I am very unsure of my interpretation!! ***

~~Code specified in the program block,~~ Assignments made to program block variables (not design signals) and pass/fail code from property expressions, are scheduled to occur in the Reactive region.

(Add the following after the previous paragraph?)
Output signals in a clocking domain (design signal inputs) are scheduled to execute in the NBA region. (??)

NOTE: there are no examples showing NBA region signal assignments to RTL models mixed with Reactive region assignments to program variables anywhere that I could find. It seems this type of clarification may be important.

The Pre-active, Pre-NBA, and Post-NBA are new in the SystemVerilog 3.1 standard but support existing PLI callbacks. The Post-observed region is new in the SystemVerilog 3.1 standard and has been added for PLI support.

The Pre-active region is specifically for a PLI callback control point that allows for user code to read and write values and create events before events in the Active region are evaluated (see Section 14.4).

The Pre-NBA region is specifically for a PLI callback control point that allows for user code to read and write values and create events before the events in the NBA region are evaluated (see Section 14.4).

The Post-NBA region is specifically for a PLI callback control point that allows for user code to read and write values and create events after the events in the NBA region are evaluated (see Section 14.4).

The Post-observed region is specifically for a PLI callback control point that allows for user code to read values after properties are evaluated (in Observed or earlier region).

BUG IN Figure 14-1 — The SystemVerilog flow of time slots and event regions
There should not be a feedback path from the Observed region to the active region. Assertions are tested in the Observed region but the assertions react and make procedural assignments in the Reactive region.

The Active, Inactive, Pre-NBA, NBA, Post-NBA, Observed, Post-observed and Reactive regions are known as the *iterative* regions.

The Preponed region is specifically for a PLI callback control point that allows for user code to access data at the current time slot before any net or variable has changed state. The Preponed region is also where assertions sample inputs for evaluation in the Observed region.

The Active region holds current events being evaluated and can be processed in any order.

The Inactive region holds the events to be evaluated after all the active events are processed.

An *explicit zero delay* (#0) requires that the process be suspended and an event scheduled into the Inactive region of the current time slot so that the process can be resumed in the next inactive to active iteration. A nonblocking assignment creates an event in the NBA region, scheduled for current or a later simulation time.

The Postponed region is specifically for a PLI callback control point that allows for user code to be suspended until after all the Active, Inactive and NBA regions have completed. Within this region, it is illegal to write values to any net or variable, or to schedule an event in any previous region within the current time slot.

Section 15 - Clocking Domains

15.2 Clocking domain declaration

The syntax for the `clocking` construct is:

The *delay_control* must be either a time literal or a constant expression that evaluates to a positive integer value.

The *clocking_identifier* specifies the name of the clocking domain being declared.

The *signal_identfier* identifies a signal in the scope enclosing the clocking domain declaration, and declares the name of a signal in the clocking domain. Unless a `hierarchical_expression` is used, both the signal and the `clocking_item` names shall be the same.

The *clocking_event* designates a particular event to act as the clock for the clocking domain. Typically, this expression is either the **`posedge`** or **`negedge`** of a clocking signal. The timing of all the other signals specified in a given clocking domain are governed by the clocking event. All **`input`** or **`inout`** signals specified in the clocking domain are sampled when the corresponding clock event occurs. Likewise, all **`output`** or **`inout`** signals in the clocking domain are driven when the corresponding clock event occurs. Bidirectional signals (**`inout`**) are sampled as well as driven.

The *clocking_skew* determines how many time units away from the clock event a signal is to be sampled or driven. Input skews are implicitly negative, that is, they always refer to a time before the clock, whereas output skews always refer to a time after the clock (see Section 15.3). When the clocking event specifies a simple edge, instead of a number, the skew can be specified as the opposite edge of the signal *. A single skew can be specified for the entire domain by using a **`default`** clocking item.

* Does this mean the following? If so, perhaps these examples could be added to this section.

LEGAL:
```
clocking ck1 (posedge clk);
  default input #1step output negedge;
  // outputs driven on the negedge clk
  input  ... ;
  output ... ;
endclocking
```

ERROR??:
```
clocking ck2 (clk); // no edge specified!
  default input #1step output negedge; // syntax error??
  input  ... ;
  output ... ;
endclocking
```

The *hierarchical_identifier* specifies that, instead of a local port, the signal to be associated with the clocking domain is specified by its hierarchical name (cross-module reference).

Unless otherwise specified, the default **`input`** skew is `1step` and the default **`output`** skew is **`0`** (in the NBA region if an assignment is made to a design signal or in the Reactive region if an assignment is made to a program variable). A **`step`** is a special time unit whose value is defined in Section 18.6. A `1step` input skew allows input signals to sample their steady-state values in the time step immediately before the clock event (i.e., in the preceding Postponed region). Unlike other time units, which represent physical units, a step cannot be used to set or modify either the precision or the timeunit.

15.3 Input and output skews

An input skew of 1**`step`** indicates that the signal is to be sampled at the end of the previous time step. That is, the value sampled is always the signal's last value immediately before the corresponding clock edge.

~~An input skew of **`#0`** forces a skew of zero.~~ Inputs with zero skew (I would change this to "explicit #0 skew") are sampled at the same time as their corresponding clocking event, but to avoid races, they are sampled in the Observed region. Likewise, assignments to design signals (assigned as clocking outputs) with zero skew (is this *really* "explicit #0 skew" or "with no specified delay" - if it is the former, I would replace "zero" with "explicit #0") are driven at the same time as their specified clocking event, as nonblocking assignments (in the NBA region).

Skews are declarative constructs, thus, they are semantically very different from the syntactically similar procedural delay statement. In particular, an explicit #0 skew, does not suspend any process nor does it execute or sample values in the Inactive region.

15.12 Input sampling

All clocking domain inputs (input or inout) are sampled at the corresponding clocking event. If the input skew is non-zero (for clarity, I would change this to "non-#0" because if no delay is specified, then sampling happens in the Postponed region just prior to the current time-step), then the value sampled corresponds to the signal value at the Postponed region of the time step skew time-units prior to the clocking event (see Figure 15-1 in Section 15.3). If the input skew is zero (again for clarity, I would change this to "#0"), then the value sampled corresponds to the signal value in the Observed region.

Samples happen immediately (the calling process does not block). When a signal appears in an expression, it is replaced by the signal's sampled value, that is, the value that was sampled at the last sampling point.

When the same signal is an input to multiple clocking domains, the semantics are straightforward; each clocking domain samples the corresponding signal with its own clocking event.

15.13 Synchronous events

The values used by the synchronization event control are the synchronous values, that is, the values sampled in the postponed region immediately before ~~at~~ the corresponding clocking event (except clocking inputs with explicit #0 delays, which are sampled in the Observed region of the corresponding clocking event).

15.14 Synchronous drives

Clocking domain outputs (**`output`** or **`inout`**) are used to drive values onto their corresponding signals, but at a specified time. That is, the corresponding signal changes value at the indicated clocking event as modified by the output skew.

The syntax to specify a synchronous drive is similar to an assignment:

The *clockvar_expression* is either a bit-select, slice, or the entire clocking domain output whose corresponding signal is to be driven (concatenation is not allowed):

```
dom.sig // entire clockvar
dom.sig[2] // bit-select
dom.sig[8:2] // slice
```

The expression can be any valid expression that is assignment compatible with the type of the corresponding signal.

The *event_count* is an integral expression that optionally specifies the number of clocking events (i.e. cycles) that must pass before the statement executes. Specifying a non-zero `event_count` blocks the current process until the specified number of clocking events have elapsed, otherwise the statement executes at the current time. The `event_count` uses syntax similar to the cycle-delay operator (see Section 15.10), however, the synchronous drive uses the clocking domain of the signal being driven and not the default clocking.

The second form of the synchronous drive uses the intra-assignment syntax. An intra-assignment `event_count` specification also delays execution of the assignment. In this case the process does not block and the right-hand side expression is evaluated when the statement executes.

Examples:

```
// drive data in the NBA or Reactive region of the current cycle
bus.data[3:0] <= 4'h5;

##1 bus.data <= 8'hz; // wait 1 (bus) cycle and then drive data

##2; bus.data <= 2; // wait 2 default clocking cycles, then drive data

bus.data <= ##2 r; // remember the value of r and then drive
// data 2 (bus) cycles later
```

Regardless of when the drive statement executes (due to event_count delays), the driven value is assigned to the corresponding signal only at the time specified by the output skew.

15.14.1 Drives and nonblocking assignments

Synchronous signal drives to design signals are processed as nonblocking assignments. (see section 14.3, 2nd to last paragraph at the bottom of page 126 to see my confusion)

A key feature of **`inout`** clocking domain variables and synchronous drives is that a drive does not change the clock domain input. This is because reading the input always yields the last sampled value, and not the driven value.

15.14.2 Drive value resolution

When more than one synchronous drive is applied to the same clocking domain output (or inout) at the same simulation time, the driven values are checked for conflicts. When conflicting drives are detected a runtime error is issued, and each conflicting bit is driven to X (or 0 for a 2-state port).

The variable `j` is an output to two clocking domains using different clocking events (**posedge** vs. **negedge**). When driven, the variable `j` shall take on the value most recently assigned by either clocking domain.

Clock-domain outputs driving a net (i.e. through different ports) cause the net to be driven to its resolved signal value. When a clock-domain output corresponds to a design input wire, a driver for that wire is created that is updated as if by a continuous assignment from a register inside the clock-domain that is updated as a nonblocking assignment. (see section 14.3, 2nd to last paragraph at the bottom of page 126 to see my confusion)

Section 16 - Program Block

16.1 Introduction (informative)

The program block serves three basic purposes:
...
3) It provides a syntactic context that specifies execution in the Reactive region.
(Again, it seems that program variables are updated in the Reactive region and design variables are updated in the NBA region ??)

16.4 Eliminating testbench races

There are two major sources of non-determinism in Verilog. The first one is that active events are processed in an arbitrary order. The second one is that statements without time-control constructs in behavioral blocks do not execute as one event. However, from the testbench perspective, these effects are all unimportant details. The primary task of a testbench is to generate valid input stimulus for the design under test, and to verify that the device operates correctly. Furthermore, testbenches that use cycle abstractions are only concerned with the stable or steady state of the system for both checking the current outputs and for computing stimuli for the next cycle. Formal tools also work in this fashion.

I think this was the key paragraph to help me understand Reactive -vs- NBA region updates ???

To avoid the races inherent in the Verilog event scheduler, program statements are scheduled to execute in the Reactive region, after all clocks in the design have triggered and the design has settled to its steady state. In addition, design signals driven from within the program must be assigned using nonblocking assignments and are updated in the NBA region. Thus, even signals driven with no delay are propagated into the design as one event. With this behavior, correct cycle semantics can be modeled without races; thereby making program-based testbenches compatible with clocked assertions and formal tools.

Since the program executes in the Reactive region, the clocking domain construct is very useful to automatically sample the steady-state values of previous time steps or clock cycles. Programs that read design values exclusively through clocking domains with non-zero input skews (is this supposed to mean "non-#0" input skews??) are insensitive to read-write races. It is important to note that simply sampling input signals (or setting non-zero (ditto ??) skews on clock domain inputs) does not eliminate the potential for races. Proper input sampling only addresses a single clocking domain. With multiple clocks, the arbitrary order in which overlapping or simultaneous clocks are processed is still a potential source for races. The program construct addresses this issue by scheduling its execution in the Reactive region, after all design events have been processed, including clocks driven by nonblocking assignments.

16.4.1 Zero-skew clocking domain races

When a clocking domain sets both input and output skews to `#0` (see Section 15.3) then its inputs are sampled (in the Observed region) at the same time as its outputs are driven (in the NBA region). This type of ~~zero~~ explicit #0-delay processing is a common source of non-determinism that can result in races. Nonetheless, even in this case, the program minimizes races by means of two mechanisms. First, by constraining program statements to execute in the Reactive region, after all ~~zero~~ explicit #0-delay transitions have propagated through the design and the system has reached a quasi steady state. Second, by requiring design variables or nets to be modified only via nonblocking assignments. These two mechanisms reduce the likelihood of a race; nonetheless, a race is still possible when skews are set to ~~zero~~ explicit #0.

16.5 Blocking tasks in cycle/event mode

Calling program tasks or functions from within design modules is illegal and shall result in an error. This is because the design must not be aware of the testbench. Programs are allowed to call tasks or functions in other programs or within design modules. Functions within design modules can be called from a program, and require no special handling. However, blocking tasks within design modules that are called from a program do require explicit synchronization upon return from the task. That is, when blocking tasks return to the program code, the program block execution is automatically postponed until the Reactive region. (I have to think about this some more??) The copy out of the parameters happens when the task returns.

Calling blocking tasks in design modules from within programs requires careful consideration. Expressions evaluated by the task before blocking on the first timing control shall use the values after they have been updated by nonblocking assignments. In contrast, if the task is called from a module at the start of the time step (before nonblocking assignments are processed) then those same expressions shall use the values before they have been updated by nonblocking assignments.

TYPOS

```
module ...
```

```
  task T;
     begin
       S1: a = b; // might execute before or after the Observed region
       #5;
       S2: b <= 1'b1; // always executes before the Observed region
     end
  endtask
endmodule
```

If task `T`, above, is called from within a module, then the statement `S1` can execute immediately when the Active region is processed, before variable `b` is updated by a nonblocking assignment. If the same task is called from within a program, then the statement `S1` shall execute when the Reactive region is processed, after variable `b` might have been updated by nonblocking assignments. Statement `S2` always executes immediately after the delay expires; it does not wait for the Reactive region even though it was originally called from the program block.

Section 17 - Assertions

17.2 Immediate assertions

Typo in the example on page 146?? To use the %t format specifier, I think you first need to declare a $timeformat command. Is %0t even legal (%t with leading spaces deleted?)

```
time t;

always @(posedge clk)
  if (state == REQ)
    assert (req1 || req2)
    else begin
       t = $time;
       #5 $error("assert failed at time %0t",t);
    end
```

17.3 Concurrent assertions overview

TYPO
The values of variables used in assertions are sampled in the Preponed region of a time slot and the assertions are evaluated during the Observed region. This is explained in Section 14, Scheduling Semantics.

Here is where we finally learn that assertions that are evaluated in the Observed region had sampled their inputs in the Preponed region.

...

A *clock tick* is an atomic moment in time and implies that there is no duration of time in a clock tick. It is also given that a clock shall tick only once at any simulation time, and the sampled values for that simulation time are used for evaluation. In an assertion, the sampled value is the only valid value of a variable at a clock tick. Figure 17-1 shows the values of a variable as the clock progresses. The value of signal `req` is low at clock ticks 1 and 2. At clock tick 3, the value is sampled as high and remains high until clock tick 6. The sampled value of variable `req` at clock tick 6 is low and remains low until clock tick 10. Notice that, at clock tick 9, the simulation value transitions to high. However, the sampled value is low.

The above paragraph is true because sampling happened in the Preponed region.

Figure 17-1 — Sampling a variable on simulation ticks
TYPO

An expression used in an assertion is always tied to a clock definition. The sampled values are used to evaluate value change expressions or **B**oolean sub-expressions that are required to determine a match with respect to a sequence expression.

17.4.1 Operand types

Typo - 3rd paragraph

The following example shows some possible forms of comparison ~~of~~ over members of structures and unions:

17.4.2 Variables

The variables that can appear in expressions must be static design variables or function calls returning values of types described in Section 17.4.1. The functions should be automatic (or preserve no state information) and pure (no output arguments, no side effects). Static variables declared in programs, interfaces or clocking domains can also be accessed. If a reference is to a static variable declared in a task, that variable is sampled as any other variable, independent of calls to the task.

Is this true for both RTL variables and program variables?

17.12 Concurrent assertions

The *action_block* shall not include any concurrent **`assert`** or **`cover`** statement. The *action_block*, however, can contain immediate assertion statements.

Note: The pass and fail statements are executed in the Reactive region. The regions of execution are explained in the scheduling semantics section, Section 14.

Since assertions are evaluated in the Observed region, does this mean that both RTL-DUT actions and program block-variable actions are both executed in the Reactive region???