Motivation: The description of sampled value functions $past, $rose, $fell and $stable is insufficient as it does not describe how these functions are updated when called from active region (e.g. HDL), observed region(e.g. assertions) and reactive regions(e.g. action blocks), and what should be their synthesizable RTL equivalent model. Reference text assumes draft 4 and the changes from Mantis 1734 and 1731.

KEY POINTS:

1. Synthesis is not discussed since the standard does not address this. What is important is that the functions retain their value between updates. Implementation is not discussed since any implementation will raise questions about the region in which the update occurs, where in fact, it is irrelevant since sampled value functions use sampled values that do not change throughout the timestep. There can be varying implementations depending on the context from which the sampled value function is used – all that is important is that it is updated prior to the expression in which it is used. So what is stated in the proposal is:
    a. $sampled(x) will always return the value of x from the Preponed region of the timestep. (pre-existing text – not new)
    b. Change value functions like $rose(x,@clk), when triggered by the associated clocking event, will compare the sampled value of the current timestep to the value that $sampled(x) had in the most recent strictly prior timestep in which the clocking event @clk occurred. (this was what Mantis 1731 said)
    c. In the case of $past(x,no_of_timesteps,,@clk), updates occur in the timestep in which the clocking event occurs, and the value is the value that $sampled(x) had in the timestep no_of_timesteps prior in which the clocking event occurred.
    d. Functions retain their value between updates. Should the clocks be the same for the sampled value function and the assignment or expression in which it is used, the sampled value function shall always be updated first.The restriction that the sampled value function outside of assertions must use the same clock as the inferred clock was therefore removed.
2. Clarification was needed relative to inferred clocks section that is referenced.
    a) Existing text implies that the default clocking applies to system value functions in HDL, so I kept that for backwards compatibility, though I don't like it.
    b) It was clarified (based on Jonathan Bromley input and reading about always_ff) that "A clock can be inferred when placed in an always or initial procedure that meets certain requirements. Specifically, there can be one and only one event control and no blocking timing controls. If the event control contains one and only one edge expression and its argument does not appear anywhere else in the body of the procedure, then that edge expression is the inferred clock." Examples were added of more complex legal and illegal code. The new text is compatible with the old text and more consistent with other parts of the standard.
    c) It was stated that it is an error if a clock cannot be inferred and no default or explicit clock exists.
    d) In addition to being able to infer clocks from always and initial procedures, always_ff was added.

REPLACE

16.8.3 Sampled value functions

This subclause describes the system functions available for accessing sampled values of an expression. These functions include the capability to access current sampled value, access sampled value in the past, or detect changes in sampled value of an expression. Sampling of an expression is explained in 16.4. Local variables (see 16.9), and the sequence methods `ended`, `triggered`, and `matched` are not allowed in the argument expressions passed to these functions. The following functions are provided:

```
$sampled(expression)
$rose( expression [, clocking_event])
$fell( expression [, clocking_event])
$stable( expression [, clocking_event])
$changed( expression [, [clocking_event]])
$past( expression1 [, number_of_ticks] [, expression2] [, clocking_event])
```

The use of these functions is not limited to assertion features; they can be used as expressions in procedural code as well. The clocking event, although optional as an explicit argument to the functions, `$past`, `$rose`, `$stable`, `$changed`, and `$fell`, is required for their semantics. The clocking event is used to sample the value of the argument expression.

The function `$sampled` does not use a clocking event. The value of `$sampled` is updated in the Preponed scheduling region in every simulation time step.

The clocking event must be explicitly specified as an argument or inferred from the code where it is used. The following rules are used to infer the clocking event:

— If used in an assertion, the appropriate clocking event from the assertion is used.
— If used in an action block of a singly clocked assertion, the clock of the assertion is used.
— If used in a procedural block, the inferred clock, if any, for the procedural code (see 16.14.5) is used.

Otherwise, default clocking (see 14.12) is used.

The $sampled function returns the value of the expression sampled in the Preponed region of the simulation time step in which the function is called. The value is stable throughout the simulation step.

……

$past can be used in any SystemVerilog expression. An example is shown below.

```
always @(posedge clk)
        reg1 <= a & $past(b);
```

In this example, the clocking event @(**posedge** clk) is applied to $past. $past is evaluated in the current occurrence of (**posedge** clk) and returns the value of b sampled at the previous occurrence of (**posedge** clk).

When *expression2* is specified, the sampling of *expression1* is performed based on its clock gated with *expression2*. For example:

```
always @(posedge clk)
    if (enable) q <= d;

always @(posedge clk)
```

assert property (done |=> (out == $past(q, 2,enable)));

In this example, the sampling of q for evaluating $past is based on the clocking expression

posedge clk iff enable

When the sampled value functions are used in an assertion, the clocking event argument of the functions may be different from the clocking event of the expression in the assertion, as determined by the clock resolution (see 16.15 Note to editor: Subclause ”Clock resolution”).

Consider the following assertions:

```
bit clk, fclk, req, gnt, en;
...
a1: assert property (@(posedge clk) en && $rose(req) |=> gnt);
a2: assert property
(@(posedge clk) en && $rose(req, @(posedge fclk)) |=> gnt);
```

Both assertions a1 and a2 read: whenever en is high and req rises, at the next cycle gnt must be asserted. In both assertions, the rise of req occurs if and only if the sampled value of req at the current **posedge** of clk is 1’b1 and the sampled value of req at a particular prior point is distinct from 1’b1. The assertions differ in the specification of the prior point. In a1 the prior point is the preceding posedge of clk, while in a2 the prior point is the most recent prior posedge of fclk.

WITH

16.8.3 Sampled value functions

This subclause describes the system functions available for accessing sampled values of an expression. These functions include the capability to access current sampled value, access sampled value in the past, or detect changes in sampled value of an expression. Sampling of an expression is explained in 16.4. Automatic variables, such as loop iteraters or local ~~Local~~ variables (see 16.9), and the sequence methods `ended`, `triggered`, and `matched` are not allowed in the argument expressions passed to these functions. The following functions are provided:

```
$sampled(expression)
$rose( expression [, clocking_event])
$fell( expression [, clocking_event])
$stable( expression [, clocking_event])
$changed( expression [, [clocking_event]])
$past( expression1 [, number_of_ticks] [, expression2] [, clocking_event])
```

The use of these functions is not limited to assertion features; they can be used as expressions in procedural code as well.

The clocking event, although optional as an explicit argument to the functions, `$past`, `$rose`, `$stable`, `$changed`, and `$fell`, is required for their semantics. The clocking event is used to sample the value of the argument expression.

Sampled valued functions are evaluated as follows:

— The function `$sampled` does not use a clocking event. The value of `$sampled` is updated in the Preponed scheduling region in every simulation time step.

— Change value functions like `$rose(x,@clk)`, are updated when the associated clocking event occurs. The sampled value of the current timestep is compared to the value that

`$sampled(x)` had in the most recent strictly prior timestep in which the clocking event `@clk` occurred.

— In the case of `$past(x,number_of_ticks,,@clk)`, updates occur in the timestep in which the clocking event occurs, and the value is the value that `$sampled(x)` had in the timestep `number_of_ticks` prior in which the clocking event `@clk` occurred.

All sampled value functions retain their value between updates. Should the clocks be the same for the sampled value function and the assignment or expression in which it is used, the sampled value function shall always be updated first.
~~The function `$sampled` does not use a clocking event. The value of `$sampled` is updated in the Preponed scheduling region in every simulation time step~~.

The clocking event must be explicitly specified as an argument or inferred from the code where it is used. The following rules are used to infer the clocking event:

— If used in an assertion, the appropriate clocking event from the assertion is used.
— If used in an action block of a singly clocked assertion, the clock of the assertion is used.
— If used in an action block of a Multiclocked assertion, the leading clock of the assertion is used
— If used in a procedural block, the inferred clock, if any, for the procedural code (see 16.14.5) is used.

Otherwise, default clocking (see 14.12) is used.

…….

Sampled value functions ~~`$past`~~ can be used in any SystemVerilog expression. An example is shown below:

```
always @(posedge clk)
       reg1 <= a & $past(b);
```

In this example, the inferred clocking event @(**posedge** clk) is applied to `$past`. `$past` is evaluated in the current occurrence of (**posedge** `clk`) and returns the value of `b` sampled at the previous occurrence of (**posedge** `clk`).

When *expression2* is specified, the sampling of *expression1* is performed based on its clock gated with *expression2*. For example:

```
always @(posedge clk)
       if (enable) q <= d;

always @(posedge clk)
assert property (done |=> (out == $past(q, 2,enable)) );
```

In this example, the sampling of `q` for evaluating `$past` is based on the clocking expression

```
posedge clk iff enable
```

~~When the sampled value functions are used in an assertion, the~~ The clocking event argument of ~~the~~ sampled value functions may be different from the clocking event of the expression in which they are used~~the assertion~~, as determined by the clock resolution (see 16.15 Note to editor: Subclause ”Clock resolution”).

Consider the following assertions:

```
bit clk, fclk, req, gnt, en;
...
a1: assert property (@(posedge clk) en && $rose(req) |=> gnt);
```

```
a2: assert property
(@(posedge clk) en && $rose(req, @(posedge fclk)) |=> gnt);
```

Both assertions a1 and a2 read: whenever en is high and req rises, at the next cycle gnt must be asserted. In both assertions, the rise of req occurs if and only if the sampled value of req at the current posedge of clk is 1'b1 and the sampled value of req at a particular prior point is distinct from 1'b1. The assertions differ in the specification of the prior point. In a1 the prior point is the preceding posedge of clk, while in a2 the prior point is the most recent prior posedge of fclk.

As another example,

```
always_ff @(posedge clk1)
    reg1 <= $rose(b,@(posedge clk2));
```

Here, the $rose sampled value function is evaluated on every **posedge** clk2 and compares the sampled value of the LSB of b with the sampled value of the LSB of b in the strictly prior timestep in which the posedge clk2 event occurs. The sampled value function retains its value between events of **posedge** clk2. reg1 is updated on the inferred clock (posedge clk1), using the current value of the system value function. Should the clocks be the same for the sampled value function and the assignment or expression in which it is used, the sampled value function will always be updated first. That is, if the sampled value of b in that time step makes $rose true, then reg1 will be assigned true.

The following two examples are illegal because default clocking is not defined and no clock can be inferred:

```
assign x = $rose(b);     // illegal

always  @(posedge clk) begin
    ...
    @(negedge clk2);
    x = $past(y, 5);   // illegal
  end
```

If default clocking is defined or if there exists an explicit clocking event, then the sampled value functions in the above examples would be updated at that clocking event.

REPLACE in **16.14.5 Embedding concurrent assertions in procedural code**

A clock is inferred if the statement is placed in an **always** or **initial** procedure with an event control abiding by the following rules:

— The clock to be inferred must be placed as the first term of the event control as an edge specifier (**posedge** *expression* or **negedge** *expression*).

— The variables in *expression* must not be used anywhere in the **always** or **initial** procedure.

For example:

```
property r1;
q != d;
endproperty
always @(posedge mclk) begin
q <= d1;
r1_p: assert property (r1);
end
```

The above property can be checked by writing statement r1_p outside the **always** procedure and

declaring the property with the clock as follows:

```
property r1;
      @(posedge mclk)q != d;
endproperty
always @(posedge mclk) begin
      q <= d1;
end
r1_p: assert property (r1);
```

If the clock is explicitly specified with a property, then it must be identical to the inferred clock, as shown below:

```
property r2;
@(posedge mclk)(q != d);
endproperty
always @(posedge mclk) begin
      q <= d1;
      r2_p: assert property (r2);
end
```

In the above example, `(posedge mclk)` is the clock for property `r2`.

WITH

~~A clock is inferred if the statement is placed in an **always** or **initial** procedure with an event control abiding by the following rules:~~

- ~~The clock to be inferred must be placed as the first term of the event control as an edge specifier(**posedge** *expression* or **negedge** *expression*).~~
- ~~The variables in *expression* must not be used anywhere in the **always** or **initial** procedure.~~

A clock can be inferred when placed in an **always, always_ff** or **initial** procedure that meets certain requirements. Specifically, there can be one and only one event control and no blocking timing controls. If the event control contains one and only one edge expression and its argument does not appear anywhere else in the body of the procedure, then that edge expression is the inferred clock.

For example:

```
property r1;
q != d;
endproperty
always @(posedge mclk) begin
      q <= d1;
      r1_p: assert property (r1);
end
```

The above property can be checked by writing statement `r1_p` outside the **always** procedure and declaring the property with the clock as follows:

```
property r1;
      @(posedge mclk)q != d;
endproperty
always @(posedge mclk) begin
      q <= d1;
end
r1_p: assert property (r1);
```

If the clock is explicitly specified with a property, then it must be identical to the inferred clock, as shown below:

```
property r2;
@(posedge mclk)(q != d);
endproperty
always @(posedge mclk) begin
       q <= d1;
       r2_p: assert property (r2);
end
```

In the above example, `(posedge mclk)` is the clock for property `r2`.

Another, more complex example that is legal is as follows:

```
property r3;
       (q != d);
endproperty
always_ff @(posedge clock iff reset == 0 or posedge reset) begin
       r1 <= reset ? 0 : r1 + 1;
       q <= $past(d1);
       r3_p: assert property (r3);
end
```

In the example above, the inferred clock is **`posedge clock iff reset==0`**.

In contrast, a clock cannot be inferred from the following because more than one edge exists in the event control:

```
property r4;
       (q != d);
endproperty
always_ff @(clock iff reset == 0 or posedge reset) begin
       r1 <= reset ? 0 : r2 + 1;
       q <= $past(d1);  // illegal due to lack of inferred clock
       r4_p: assert property (r4);   // illegal due to lack of clock
end
```

The above code shall result in an error because no clock can be inferred, and there is not default clocking or an explicit clock specified in `r4_p`.

In the following example, a clock cannot be inferred due to multiple event controls and delays in the always procedure.

```
property r5;
q != d;
endproperty
always @(posedge mclk) begin
       #10 q <= d1;        // delay prevents clock inferencing
       @(negedge mclk)     // event control prevents clock inferencing
       #10 q1 <= !d1;
       r5_p: assert property (r5);   // illegal
end
```

The existence of a second event control or a timing delay prevents clock inferencing.