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.

KEY POINTS:

1. Regardless where $past(x,,,@(posedge clk)) is called from (e.g. from the active, reactive, or observed regions), the RTL equivalent is:

   ```
   always @(posedge clk) past_variable <= $sampled(x);
   ```

   The value of the sampled value function is updated in accordance with the clock associated with it, using the Preponed value, and retains its value between updates.

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 example below illustrates the use of `$rose` in SystemVerilog code outside assertions.

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

In this example, the clocking event `@(posedge clk)` is applied to `$rose`. `$rose` is true whenever the sampled value of `b` changed to `1` from its sampled value at the previous tick of the clocking event.

…

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. Sampled value functions may be evaluated in the active region (e.g. in an always block), observed region(e.g. in an assertion), or reactive region (e.g. an assertion action block). In all cases, the expression evaluation uses the sampled value from the Preponed scheduling region as the reference value for the current time step. Regardless where `$past(x,,,@(posedge clk))` is called from (e.g. from the active, reactive, or observed regions), the RTL equivalent is:

```
always @(posedge clk) past_variable <= $sampled(x);
```

A sampled value function is updated in accordance with the clock associated with it, using the Preponed value, and retains its value between updates.

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 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.

…….

The example below illustrates the use of `$rose` in SystemVerilog code outside assertions.

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

In this example, the inferred clocking event `@(posedge clk)` is applied to `$rose`. `$rose` is true whenever the LSB of the sampled value of `b` changed to `1` from its sampled value at the previous tick of the clocking event.

As another example,

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

Here, the `$rose` sampled value function is evaluated on every `posedge clk2` using the sampled value of `b`, but `reg1` is updated on the `posedge clk1`, using the current value of the system value function.

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 clock, then the sampled value functions in the above examples would be updated at that clock.

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.