

System Verilog 3.1 Donation

Part II: Assertions

Version 1.2, June 2002



Contains proprietary information of Synopsys, Inc.

Copyright © 2002 Synopsys, Inc. Synopsys. All rights reserved. This documentation contains proprietary information that is the property of Synopsys, Inc.

The Synopsys logo and VERA are registered trademarks of Synopsys, Inc. OpenVera is a trademark of Synopsys Inc. All other brands or products are trademarks of their respective owners and should be treated as such.

1

OpenVera Assertions

This chapter describes the language for expressing timing relationships between design objects. Using this language, you can specify one or more expressions, their functional and timing relationships, and a set of criteria for the relationships to fail or succeed. This chapter includes:

- [Evaluating Sequence Expressions](#)
- [Specifying Edge Events and Clocks](#)
- [Specifying Time Shift Relationships](#)
- [Defining Expressions](#)
- [Specifying Temporal Assertions](#)
- [Specifying Assertions for an Instance of a Module](#)
- [Specifying Assertions for A Module](#)

- [Name Resolution](#)
- [Specifying Composite Sequences](#)
- [Specifying Conditional Sequence Matching](#)
- [Matching Repetition of Sequences](#)
- [Specifying Conditions Over Sequences](#)
- [Specifying Unconditional Number of Clock Ticks](#)
- [Grouping Assertions as a Library](#)
- [What Is a Formula](#)
- [Defining Formulas](#)
- [Writing Simple Formula Expressions](#)
- [Specifying Temporality Using Formulas](#)
- [Using Sequences as Conditions to Formulas](#)
- [Specifying Reset Conditions](#)
- [Directives for Formal Verification](#)

Evaluating Sequence Expressions

This section describes how sequence expressions are evaluated. There are two important aspects of expression evaluation: one indicates whether the expression matched the simulation results, and the other explains the start and end time of the evaluation. The concepts of expression evaluation and advancement of time are used in deriving the success/failure of assertions, and are fundamental to understanding the descriptions of language features.

A sequence is a Verilog boolean expression in a linear order of increasing time. These boolean expressions must be true at those specific points in time for the sequence to be true over time. A boolean expression at a point in time is a simple case of a sequence with time length of one unit.

A sequence expression describes one or more sequences by using temporal operators that specify a range of possibilities of and repetitions of sequences. During the sequence expression evaluation, the temporal operators act upon the boolean expressions over those possibilities of time and repetition during the sequence expression evaluation. After such monitoring and evaluation of a sequence expression, one or more sequences can actually satisfy the expression. This section provides several examples to illustrate how evaluation is carried out in time and results computed for assertions.

The variables or operands in sequence expressions are Verilog regs, integers, and all varieties of nets. There is also an OpenVera Assertions event that can be a variable or operand.

Note:

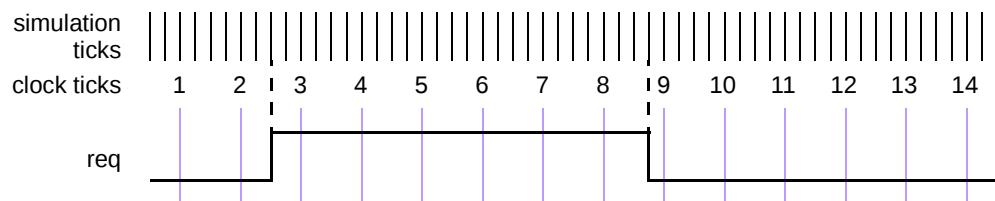
In this manual variable in an expression refers to these types of design objects and not the sense of the term variable in the Verilog-1364-2001 standard.

Timing Model and Edge Events

The timing model employed in this specification is based on clock ticks, and uses a generalized notion of clock cycles. The definition of a clock is explicitly specified by the user, and can vary from one expression to another. In addition, a user can choose to use the simulation time as a clock to express asynchronous events.

A clock tick is an atomic moment in time and implies that there is no duration of time in a clock tick. The value of a variable in an expression at a clock tick is sampled precisely one simulation tick before the clock tick. The sampled value is the only valid value of a variable at a clock tick. [Figure 1-1](#) 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 9. The value of variable `req` at clock tick 9 is low and remains low.

Figure 1-1 Sampling a Variable on Simulation Ticks



Note:

For accessing the value of a variable from Verilog at a simulation time, the value is obtained after all the event computations have been performed at that simulation time and no more changes in the value are expected to occur. The value of a variable one simulation tick before the clock is considered as the sampled value for the variable with respect to its clock.

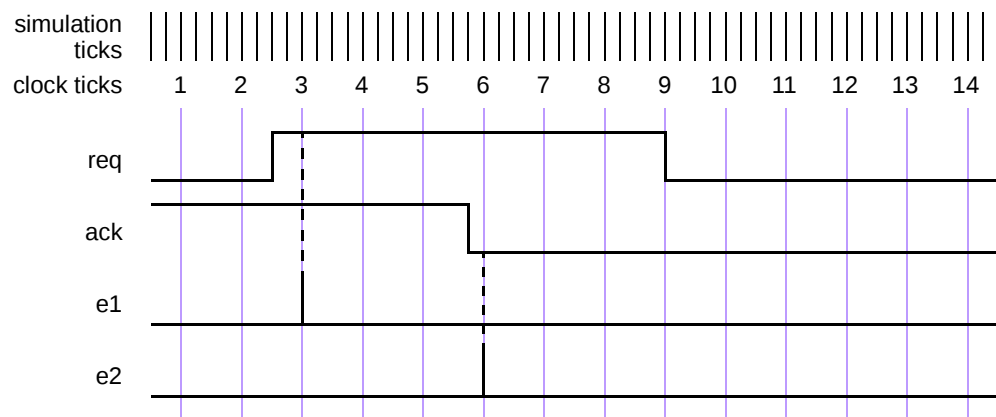
An expression is always tied to a clock definition. The values of variables are sampled only at clock ticks. These values are used to evaluate edge events (such as `posedge` and `negedge`) or boolean sub-expressions that are required to determine a match with respect to a sequence expression.

An edge event at a clock tick changes the value of an expression from the value of that expression at the previous clock tick. Like boolean expressions, an edge event evaluates to true if the event occurs, and to false if the event does not occur.

For example, when a signal changes its value from low to high (a rising edge), it is considered a posedge event. [Figure 1-2](#) illustrates two examples of edge events:

- edge event `e1` is defined as `(posedge req)`
- edge event `e2` is defined as `(negedge ack)`

Figure 1-2 Edge Events



The clock used for sampling the events is `clock`, which is different than the simulation ticks. Assume, for now, that this clock is defined in this language elsewhere. At clock tick 3, event `e1` occurs because the value of `req` at clock tick 2 was low and at clock tick 3, the value is high. Similarly, event `e2` occurs at clock tick 6 because the value of `ack` was sampled as high at clock tick 5 and sampled as low at clock tick 6.

Note:

A vertical bar, in figures like [Figure 1-2](#), without an arrow on the top or the bottom of the bar indicates an occurrence of an edge event.

Matching A Sequence

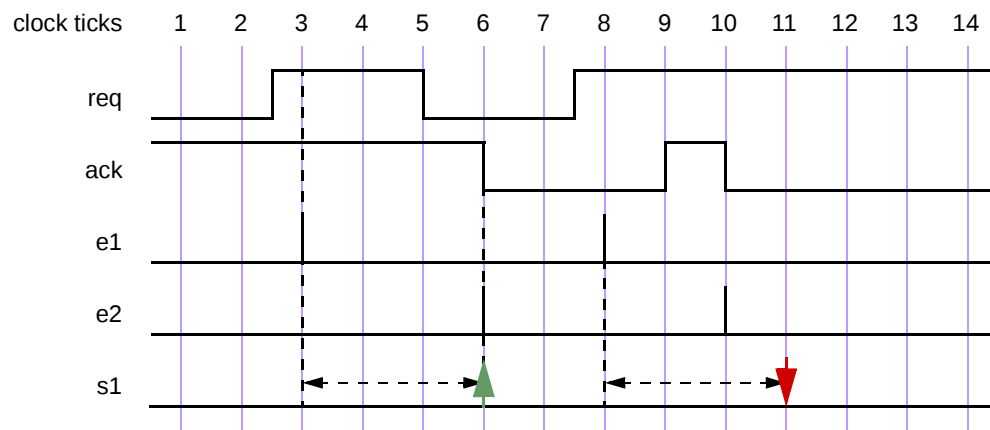
Another way to look at a sequence is that it is a series of checkpoints described by a sequence expression. These checkpoints are dispersed in time from the beginning to the end of evaluation time of the expression. At each checkpoint, a boolean expression or an edge event is evaluated, resulting in a true/false value. A boolean expression is evaluated in the same way as a Verilog expression. To determine a match of a sequence, checkpoints are evaluated at appropriate times to satisfy the expression. If all the checkpoints are satisfied, then a match of a sequence to the simulation results occurs.

A sequence expression that specifies a complete assertion, that is not a sub-expression of a larger expression, typically has a checkpoint at every clock tick to see if it is violated. To test the assertion at a clock tick, a new evaluation attempt for the expression is carried out, independent of any attempt at a previous clock tick. The results of each attempt are also reported separately. Generally, we will be discussing one attempt when we describe the behavior of the language constructs.

For example, consider the sequence of edge events, $s1$, in [Figure 1-3](#). $s1$ is defined as:

```
e1 #3 e2
```

Figure 1-3 Matching a Sequence



The # notation is used to refer to clock ticks. The above example says that e2 is expected to occur at the third clock tick after the occurrence of e1. [Figure 1-3](#) illustrates this process for an attempt starting at clock tick 3 and shows how the time is advanced for the attempt. e1 is evaluated to be true at clock tick 3. The outcome of this result is the continuation of checking the expression for the next checkpoint, which is event e2 at clock tick 6. No evaluation or checking is performed at clock ticks 4 and 5 for this attempt. Thus, variables can take on any values during these clock ticks. Event e2 occurs at clock tick 6, so the expression is said to match for the attempt starting at clock tick 3.

Note:

A sequence match is indicated as an upward arrow and a no match is indicated as a downward arrow. At all other points in time where there is no upward or downward arrow, the expression is in the process of evaluating a match. A time line is shown with a dashed horizontal line $\leftarrow - - - - \rightarrow$ with a left and a right arrow to indicate that an evaluation is in progress during that time period.

Note:

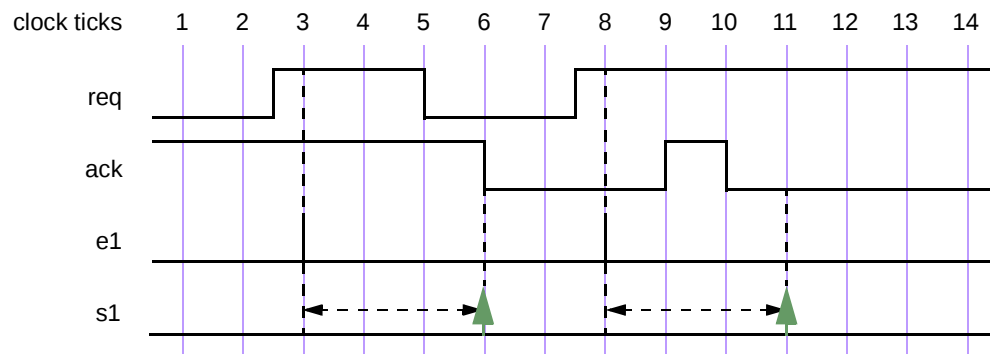
The values of signals shown in diagrams in this manual are the derived sampled values of those signals with respect to their clock, and not the actual simulation values at the corresponding simulation time.

The above example shows the evaluation of events as part of checkpoints in an expression. A checkpoint can also be a variable or a boolean expression that is evaluated to determine if the checkpoint is true or false. Reconsider the same example with a change that signal (`ack==0`) is tested instead of `negedge ack` in the expression as shown below.

```
e1 #3 (ack==0)
```

This example is illustrated in [Figure 1-4](#) for the evaluation attempt starting at clock tick 8. The value of signal `ack` is low, so there is a sequence match at clock tick 11.

Figure 1-4 Matching a Sequence with a Variable



Note:

If only a variable is specified as a checkpoint, then it is implicitly converted to its logical value during the checkpoint evaluation by the rules of Verilog. For the above example, if the expression were written as:

e1 #3 ack

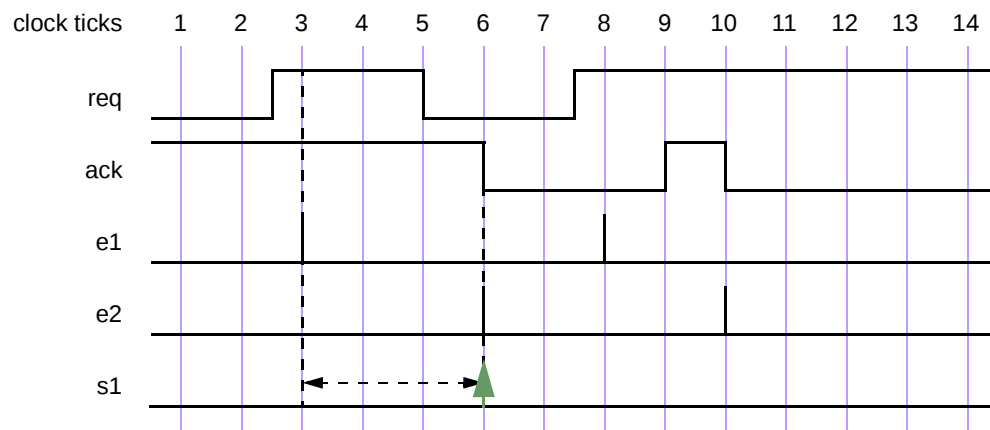
Then, `ack` is converted to its logical value using the above rule and the expression is true if `ack` is true at the proper clock tick.

Start and End Time of A Sequence

Each sequence has a start time and an end time. As seen from the examples in [Figure 1-3 on page 1-7](#) and [Figure 1-4 on page 1-8](#), while monitoring sequences the reference time (current time) is advanced according to the clock ticks between the checkpoints.

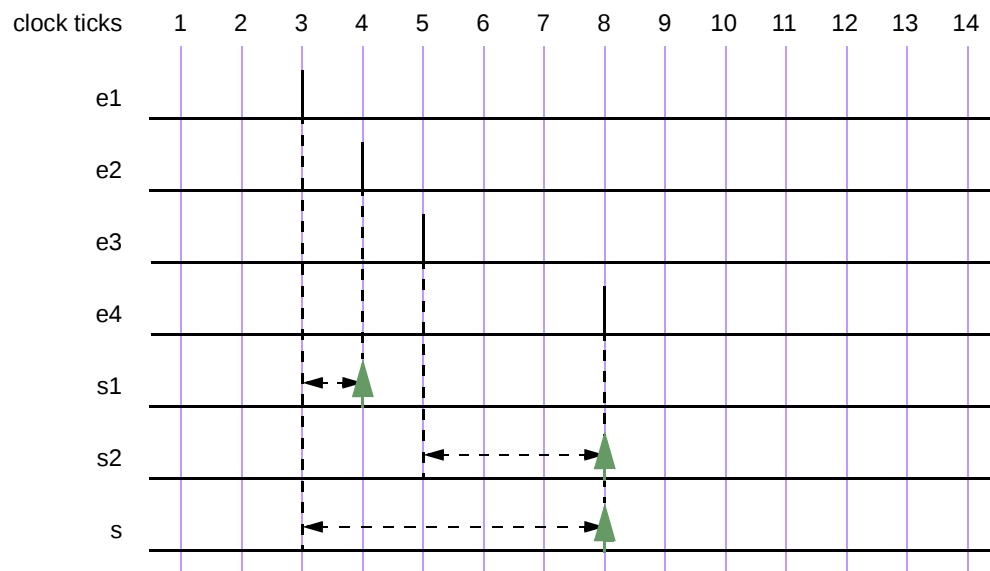
The start time for a sequence match is the time from which a new evaluation attempt of the sequence expression begins. The end time is the time at which a success or a failure for the sequence is detected. Let us examine the start and end times of the evaluation attempt at clock tick 3 for the example illustrated in [Figure 1-5](#). The attempt starting at clock tick 3 matches at clock tick 6, so the start and end times are clock ticks 3 and 6 respectively.

Figure 1-5 Start and End Times of a Sequence



A sequence can consist of sub-sequences, again dispersed in time. Same rules apply to sub-sequences regarding the start time and end time. Now, assume a series of events (e_1 , e_2 , e_3 and e_4) at the corresponding clock ticks (3, 4, 5 and 8). Consider a sequence s consisting of two sub-sequences s_1 and s_2 , where s_1 is ($e_1 \#1 e_2$) and s_2 is ($e_3 \#3 e_4$), and s is defined as ($s_1 \#1 s_2$), and shown in [Figure 1-6 on page 1-11](#). The time clause #1 specifies the expectation of the occurrence of the second operand event in the next clock tick after the occurrence of the first operand event. The time clause #3 specifies the expectation of the occurrence of the second operand event at the third clock tick after the occurrence of the first operand event.

Figure 1-6 Start and End Times of Sub-sequences



The sequence expression is:

```
(e1 #1 e2) #1 (e3 #3 e4)
```

Let us examine the evaluation attempt at clock tick 3 in [Figure 1-6](#).

- The attempt starting at clock tick 3 succeeds for sub-sequence s1 at clock tick 4.
- Next, #1 directs to move to the next clock tick, so the evaluation of s2 begins at the next clock tick after sub-sequence s1, and the start time of sub-sequence s2 becomes 5.
- Sub-sequence s2 terminates when event e4 occurs, resulting in the end time for sub-sequence s2 as clock tick 8.

Single vs. Multiple Sequences of Evaluation

A more complex scenario arises when the expression evaluation branches out to compute all alternative sequences implied by a construct. In such cases, a sequence match is determined for every sequence independent of each other. The expression can result in multiple successful or failed matches. If such a sequence expression is a sub-expression of a larger expression, then the resulting matches are used to determine sequence matches of the enclosing expression. An example of evaluating multiple sequences follows:

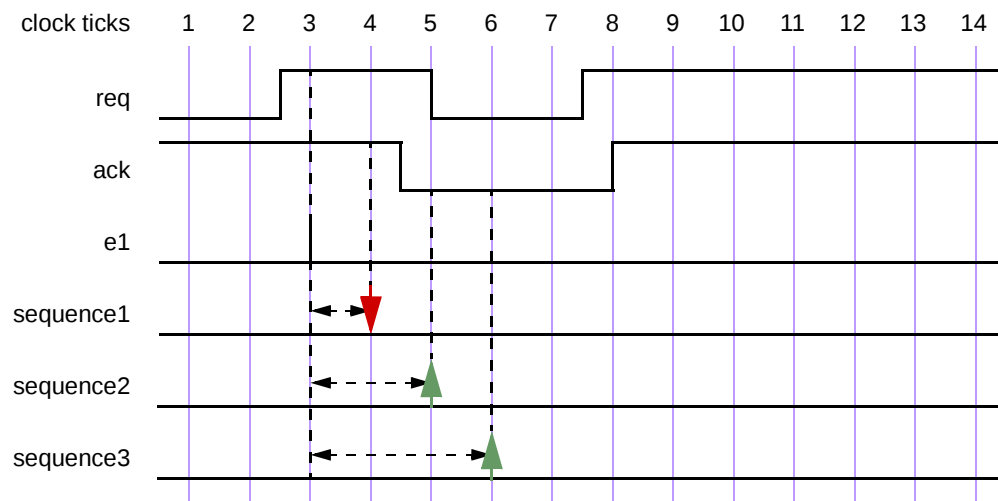
```
e1 # [1..3] (ack==0)
```

Event `e1` is defined as `(posedge req)`.

This statement says that signal `ack` must be low at the first, second, or third clock ticks after the occurrence of event `e1`. To determine a match for each of these three cases, three separate evaluations are started. An example is illustrated in [Figure 1-7](#). The three sequences are:

```
e1 #1 (ack==0)
e1 #2 (ack==0)
e1 #3 (ack==0)
```

Figure 1-7 Evaluating Multiple Sequences



Let us consider an evaluation attempt at clock tick 3:

- At clock tick 3, event `e1` occurs, so three sequences are started.
- Sequence1 fails to match at clock tick 4 as signal `ack` is 1.
- Sequence2 and sequence3 match at clock ticks 5 and 6 respectively, as signal `ack` is 0 at those clock ticks.

Specifying Edge Events and Clocks

Edge Events

The syntax for specifying edge events is as follows:

```
posedge | negedge | edge bit_vector_expr  
matched event_name  
ended event_name
```

One important use of events is being able to describe change in values of variables. In practical situations, most activities in systems are initiated based on detecting a change in value.

bit_vector_expr is an expression that results in a single or multi-bit vector. Three clauses are provided to specify change in values:

`posedge bit_vector_expr`

Is used to express positive edge and generates an event upon 0 to 1 transition on the value of the expression *bit_vector_expr*.

`negedge bit_vector_expr`

Is used to express negative edge and generates an event upon 1 to 0 transition on the value of the expression *bit_vector_expr*.

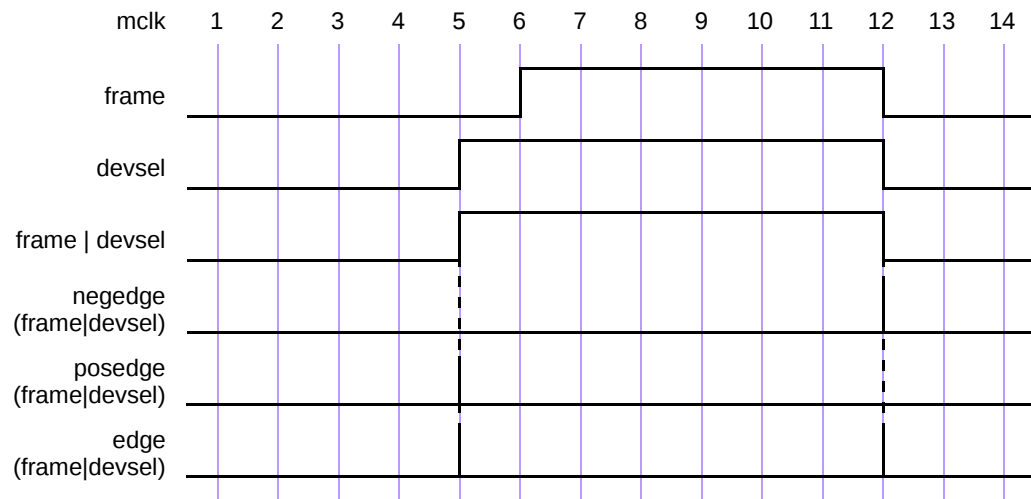
`edge bit_vector_expr`

Is used to express a change in value and generates an event upon either 1 to 0, or 0 to 1 transition on the value of the expression *bit_vector_expr*.

Note that Verilog semantics are used to evaluate edge events. In particular, if *bit_vector_expr* is a vector, then only the least significant bit is considered for determining the result of an edge event.

An example of `posedge`, `negedge` and `edge` is shown in [Figure 1-8](#).

Figure 1-8 Edge Events



A sequence event is another case of a simple event specification and is specified in two ways as shown below.

```
matched event_name
ended event_name
```

The ***matched*** and ***ended*** operators are used to test if the sequence event occurred or not. If the sequence event was generated by event *event_name*, then the result is true, otherwise the result is false. The ***matched*** operator is used for multiple clock domains, while the ***ended*** operator is used for a single clock domain. The *event_name* refers to a sequence expression specified by the ***event*** clause. The ***event***, ***ended***, and ***matched*** clauses are explained in [“Defining Expressions” on page 1-24](#).

Clocks

The syntax for specifying a clock is as follows:

```
clock | sclock edge_expr
{
    statements other than clock
}
```

Each sequence or boolean expression is associated with a clock. The clock determines the sampling times for variable values.

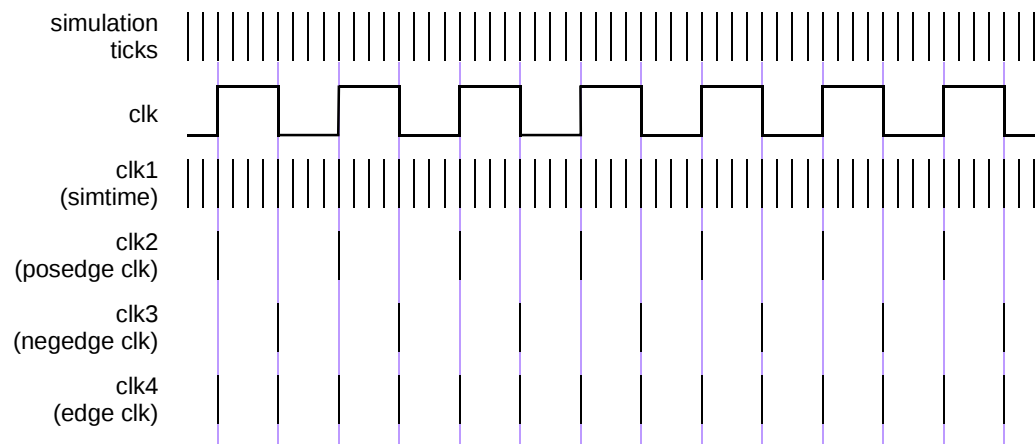
A clock tick occurs whenever the edge event described by *edge_expr* occurs during simulation.

Note that if a clock is not explicitly specified, then the simulation clock is used as the clock for the expression. This is useful for asynchronous events, but can substantially slow the simulation.

[Figure 1-9](#) illustrates the use of event clauses for specifying clocks. Four clocks are shown as follows:

- clk1 as simulation time
- clk2 as posedge clk
- clk3 as negedge clk
- clk4 as edge clk

Figure 1-9 Specifying Clocks with Event Clauses



A clock can be specified for any individual sequence expression, for example:

```
clock posedge clk {
    event tex1: start_sig #1 end_sig ;
}
clock posedge global_clk {
    event tex2: trans #1 trans_end ;
}
```

In the above case, `posedge global_clk` is used as a clock for sequence expression `tex2`, while `posedge clk` is used as a clock for sequence expression `tex1`.

There are two types of clock: weak clock with the keyword **clock**, and strong clock with the keyword **sclock**. The only difference between the two types of clock is that **clock** does not require the `edge_expr` to be true, while **sclock** enforces the clock to be true at least once during simulation. In the case when a weak clock does not tick at all during simulation, the assertion is considered to be true at the end of simulation. On the contrary, a strong clock is required to

occur at least once. The assertion that is clocked by **sclock** is considered to be false at the end of simulation if the clock does not tick.

A strong clock is an actual clock signal in your design. A weak clock is another design object that is not an actual clock but you want to use it as a clock for an assertion anyway.

Specifying Time Shift Relationships

The syntax for time shift relationships is as follows:

```
# int | [int .. int] | [int ..]  
  
->>
```

Time is expressed in terms of clock ticks and is specified using # notation. a #t b, means that a should occur, followed by (t-1) clock ticks, followed by b. In other words, a must occur, followed by b t clock ticks later.

[Table 1-1](#) shows variations of time specifications that can be used.

Table 1-1 Time Specification Syntax

Specification	Meaning
#t	t clock tick delays
#[t1..t2]	A variable time delay between t1 and t2. It defines a period between clock tick t1 and clock tick t2, with t1 and t2 being inclusive. t1 must be less than t2.

Table 1-1 Time Specification Syntax(Continued)

Specification	Meaning
#[0..t2]	A period between the current clock tick and clock tick (t2), with current time and t2 being inclusive.
#[t1..]	A period between clock tick t1 and the end of simulation.
#[1..]	A period between the next clock tick and the end of simulation.

In addition, the following must be noted:

- `->>` is a shorthand notation for expressing `#[1..]`, or eventuality of occurrence.
- `t`, `t1` and `t2` cannot be negative numbers, specifying activities in the past. They can be zero.
- In case of a range specification, `t2` must be greater than `t1`.

The clock that determines the basic unit of time (clock tick) is inferred from the context of the expression in which time is specified. How to specify clock has been described in a previous section.

The syntax for specifying timing sequences is as follows:

`[sequence_expr] time_shift sequence_expr`

This basic time notation is used to express temporal relationships between expressions, and provides the building blocks for sequences. Examples of specification are:

- clock ticks between sequences
- specific clock tick when a sequence is expected to occur
- time period during which a sequence is expected to complete

- eventuality of occurrence of a sequence

Clock ticks between two sub-expressions is specified using:

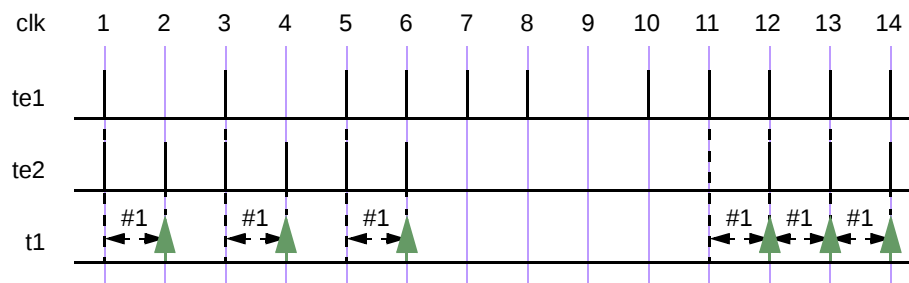
```
sequence_expr time_shift sequence_expr
```

All variations of *time_shift* can be used between the two sequence expressions. Note that the time specified by *time_shift* can take on only a positive value. Consider two expressions that are expected to occur, one followed by the other in the next clock tick. This can be written as:

```
event t1: te1 #1 te2;
```

Here where *te1* and *te2* are events, *te1* must evaluate to true first, then *te2* must evaluate to true in the next clock tick. *t1*, as illustrated in [Figure 1-10](#), for the attempts at clock ticks 1, 3, 5, 11, and 12 matches at clock ticks 2, 4, 6, 12 and 13 respectively because *te2* is true one clock tick after *te1*.

Figure 1-10 Time Specification *event t1: te1 #1 te2;*



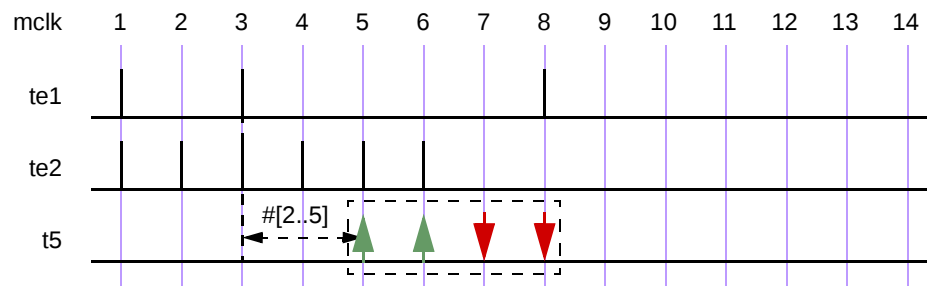
When an activity is allowed to end within a range of time (time window), then the *time_shift* clause with a minimum and maximum time specification is used to express the time window. When a range of time is specified, multiple matches can occur. At each clock tick within the time window, a new evaluation attempt is made to determine a match.

Consider the following example:

```
event t5 :te1 #[2..5] te2;
```

Here, t_{e2} can be true anywhere within a time window starting at 2 clock ticks after t_{e1} becomes true, and ending at 5 clock ticks after t_{e1} becomes true. [Figure 1-11](#) illustrates this example for the evaluation attempt at clock tick 3.

Figure 1-11 Range Time Specification



The time window for attempt starts at 5 and ends at 8. This attempt generates two matches at clock ticks 5 and 6 because t_{e2} is true at those clock ticks. At clock ticks 7 and 8, t_{e2} does not match.

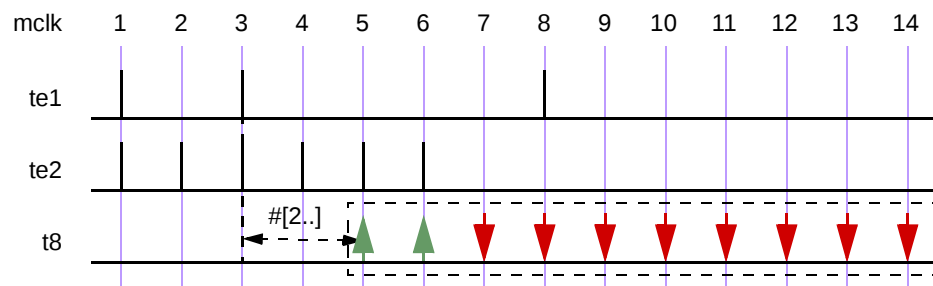
Consider another example:

```
event t8: te1 #[2..] te2;
```

In the expression t_8 , the maximum time is the end of simulation for t_{e2} to evaluate to true. t_{e1} must first evaluate to true, followed by t_{e2} some time after 1 clock tick, but before the end of simulation.

The time window for t_8 is shown in [Figure 1-12](#) for the attempt at clock tick 3. The attempt generates matches at clock ticks 5 and 6, reports failures for the rest of the simulation.

Figure 1-12 Until Simulation End Time Specification



A special case of this range is when the minimum time is one clock tick (next clock tick).

```
event ch9: te1 ->> te2;
```

The above assertion can be written as:

```
event t9 : te1 #[1..] te2;
```

The notation `->>` specifies any subsequent clock tick until the end of simulation and is provided because this type of sequence expression is frequently used.

So far, we have described `time_shift` operation in a binary context, when one sequence expression follows another. `time_shift` operation can also be specified as a unary operator, where it is used as a delay.

The syntax for the time shift unary operator is as follows:

```
time_shift sequence_expr;
```

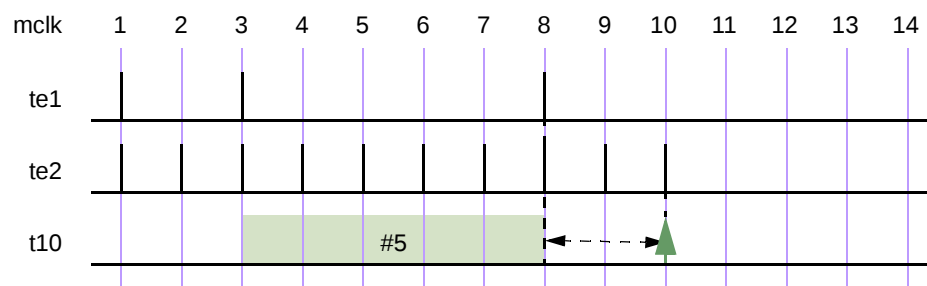
All variations of `time_shift` can be used. Time can be of 0 or positive value.

```
event t10 : (#5 te1) #2 te2;
```

The above sequence expression has an additional requirement for `te1`: four clock ticks must precede `te1`, which implies to reject any `te1` which occur prior to 5 clock ticks from the beginning of simulation. In sub-expression `(#5 te1)`, `time_shift` is used as a unary operator.

This is shown in [Figure 1-13](#).

Figure 1-13 Time Shift as an Unary Operator



To express certain amount of delay following an expression, ***any*** clause is used as follows.

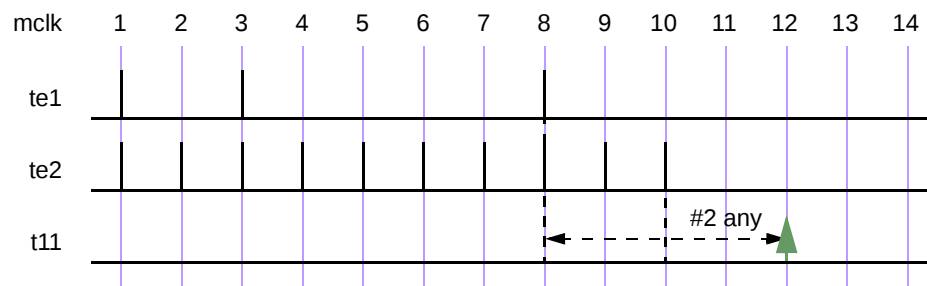
```
sequence_expression time_shift any ;
```

any denotes an empty expression that always evaluates to true (see [“Specifying Unconditional Number of Clock Ticks” on page 1-76](#)). In the above expression, after satisfying `sequence_expression`, time is advanced unconditionally by an amount specified in the `time_shift`.

```
event t11 : (#5 te1) #2 te2 #2 any;
```

[Figure 1-14](#) illustrates the above expression. Notice that 2 clock ticks are required at the end of the expression because it ends with `#2 any`. At clock tick 10, the expression `(#5 te1) #2 te2` is matched. Two clock ticks later, at clock tick 12, `t11` is matched.

Figure 1-14 *any at the End of a Sequence*



Defining Expressions

The syntax for defining expressions follows:

```
bool name [(param1, ..., paramN)] : boolean_expr ;
event name [(param1, ..., paramN)] : sequence_expr ;
```

Expressions are categorized as boolean or sequence. A boolean expression consists of variables with boolean operators as defined in Verilog, and returns true or false as the result of the evaluation of the expression. A boolean expression is defined using the **bool** clause.

A **bool** clause declares a boolean expression with an identifier to name the expression. Optionally, a list of parameters, separated by commas, can be declared for the expression, in which case, the parameters supplied at the time of instantiation replace the corresponding variables. Regardless of whether the declaration is made under an explicit clock or without a clock, no clock is associated with the expression until it is instantiated in an expression. Upon its usage, the **bool** definition is expanded in the

expression where used and becomes part of the expression. The values of the signals of the *bool* expression are sampled according to the clock associated with the expression where instantiated.

For example,

```
bool b1: !req && ack;
clock posedge sysclk {
    bool b2: b1?(addr[3:2]==0):(addr[3:2]!=0);
}
```

In this example, boolean expression *b1* gets replaced by its expression in *b2*, as follows:

```
clock posedge sysclk {
    bool b2: (!req && ack)?(addr[3:2]==0):(addr[3:2]!=0);
}
```

b2 evaluates its expression at each posedge of clock *sysclk* using the sampled values of *req*, *ack* and *addr*.

An example with parameters is shown below.

```
bool b3(ad[3:0]): (!ad[0])&&ad[1]&&!ad[2]&&ad[3]);
clock posedge sysclk {
    bool b4: (req&&pack1)?b3(addr[6:3]):b3(addr[10:7]);
}
```

In the above example, boolean expression *b3* is declared with a parameter *ad*. *b3* is instantiated twice in boolean expression *b4* with different parameters. In the first instantiation, *b3* is evaluated with variable *addr[6:3]*, while in the second instantiation, *b3* is evaluated with variable *addr[10:7]*.

A sequence expression uses boolean as well as temporal operators, and returns sequences that matched the expression. A sequence expression is defined using the *event* clause. An *event* is

declared with an identifier to name the expression, and a sequence expression to specify the relationship for monitoring. An explicit clock may be specified for sampling values/events. If the clock is not specified, the simulation clock is assumed as a clock.

Like the *bool* clause, an *event* can be defined with parameters, such that multiple instantiations of the sequence expression can be made with different variables as arguments.

There are two ways in which an *event* is used:

- To decompose a complex sequence expression into simpler sub-expressions. The sub-expressions are used as part of the expression by simply referencing their names. The evaluation of a sequence expression that references an *event* sequence expression is performed the same way as if the *event* sequence expression was a lexical part of the expression. In other words, the *event* sequence expression is “invoked” from the expression where it is referenced. An example is shown below:

```
clock posedge sysclk {  
    event seq: a #1 b #1 c;  
    event rule: if (trans) then  
        start_trans #1 seq #1 end_trans;  
}
```

This is equivalent to:

```
clock posedge sysclk {  
    event rule: if (trans) then  
        start_trans #1 a #1 b #1 c #1 end_trans;  
}
```

- To use the *event* sequence expression to generate a simple event called sequence event. In this case a sequence event is generated each time the sequence expression succeeds. The

occurrence of the event can be tested in any sequence expression by using the clause ***matched*** or ***ended***. An example is shown below:

```
clock posedge sysclk {
    event e1: posedge rdy #1 proc_1 #1 proc_2;
    event rule: if (reset)
        then inst #1 matched e1 #1 branch_back;
}
```

In this example sequence expression e1 must end successfully one clock tick after inst. If the keyword ***matched*** wasn't there, sequence expression e1 starts one clock tick after inst.

As described above, ***event*** clause can be used to specify a sequence expression as a sequence event. So far the expressions are described as monitors that use variable values to check for edge events or boolean expressions at specified times. A sequence event is different from an edge event. The main difference between an edge event and a sequence event is that when the sequence expression attached to ***event*** succeeds, a sequence event is generated. However, if the sequence expression fails, then the results are discarded and no event is generated. So, while an edge event is said to occur if a change in value at a clock tick compared to the previous clock tick is detected, a sequence event occurs when its corresponding ***event*** clause succeeds.

Consider the following:

```
clock posedge sysclk {
    event t1: te1 #1 te4 #1 te7;
    event t2: te2 #1 te5 #1 te8;
    event t3: te3 #1 te6 #1 te9;
    event rule: if (matched t1) then
        matched t2 #1 matched t3;
}
```

The sequence events `t1`, `t2` and `t3` define sequence expressions that are used in `rule`. When the sequence expression `t1` succeeds, a sequence event is generated for `t1`. Similarly, sequence events `t2` and `t3` are generated. The ***event*** rule ensures that these sequence events occur in a specific sequence. The ***event*** clause is used for building sub-expressions.

The testing of an event occurrence is performed using either the ***ended*** or ***matched*** operator. ***matched*** is a more general operator than ***ended***. ***matched*** can be used when the event is generated at one clock and tested in a sequence expression with the same or a different clock. The ***ended*** operator, on the other hand, can only be used when the same clock is used for generating the event and testing the event in a sequence expression.

```
clock posedge sysclk {
    event e1: posedge rdy #1 proc_1 #1 proc_2;
    event rule: if (reset)
        then inst #1 ended e1 #1 branch_back;
}
```

In the above example, `sysclk` is used for generating `e1`, and testing `e1` in `rule`. Both, `e1` and `rule` use the same clock. The same example is modified below where `rule` is based on a different clock, thus operator ***ended*** is replaced by ***matched***.

```
clock posedge sysclk {
    event e1: posedge rdy #1 proc_1 #1 proc_2;
}

clock posedge uclk {
    event rule: if (reset)
        then inst #1 matched e1 #1 branch_back;
}
```

The **event** clause enables you to specify sequence expressions at different clocks and use their results in another sequence expression.

Consider the following:

```
clock posedge clk1 {
    event t1: sequence_expr1 ;
}
clock posedge clk2 {
    event t2: sequence_expr2 ;
}
clock posedge clk3 {
    event rule2: if (matched t1) then matched t2 ;
}
```

Event `t1` is defined with a clock `clk1` and event `t2` is defined with a clock `clk2`, while `rule2` is defined with a clock `clk3`. `rule2` uses the sequence event `t1` and `t2` to construct a more complex expression `rule2`.

To illustrate the generation and use of sequence event, consider the following example. A master device issues a transaction request using master clock. The device examines the request and issues a device selection signal within 3 clock ticks of posedge `dclk`. Note that signals `mclk` and `dclk` are different signals and may possess no timing relationship.

```
clock posedge mclk {
    event trans: negedge frame ;
}
clock posedge dclk {
    event rule3: if (matched trans)
        then #[1..3] negedge devsel;
}
```

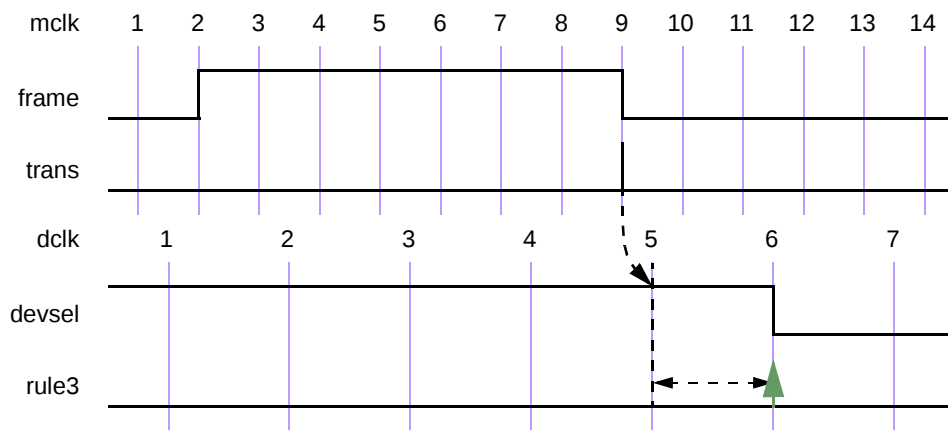
The semantics of the *if then* clause is discussed in [“Specifying Conditional Sequence Matching” on page 1-58](#). A brief description of its usage follows.

```
event event_name : if boolean_cond then  
    sequence_expression
```

The expression *boolean_cond* is evaluated. If *boolean_cond* evaluates to false, then that particular evaluation of *event_name* is considered successful with a matched sequence of just *boolean_cond*. Consequently, *boolean_cond* acts as a precondition to evaluating expression *sequence_expression*. Whenever *boolean_cond* evaluates to true, then *sequence_expression* is evaluated that results in sequence matches for the event *event_name*.

In [Figure 1-15](#), at clock tick 9 for clock *mclk*, *negedge frame* occurs. [Figure 1-15](#) illustrates this evaluation attempt. As a result, a sequence event *trans* is generated and latched for clock *dclk*. This sequence event *trans* is available at the clock tick 5 for clock *dclk*. *negedge devsel* occurs at clock tick 6 of clock *dclk*, matching *rule3*.

Figure 1-15 *if-then Example*



Note that the sequence event is only generated when its associated sequence expression succeeds. At all other times, the sequence event does not occur. Furthermore, the sequence event gets latched, in the sense that it can be tested for its occurrence, until the next clock tick of the sequence expression where it is used. The occurrence of a sequence event is tested simply by its reference as a variable in an expression. The test returns true if the sequence event occurred, and false if it did not occur. For a particular success of the associated sequence expression of the sequence event, the sequence event will test as true only for one clock tick, after which it will test as false. In [Figure 1-15](#), signal `trans` gets latched to clock tick 9 for clock `dclk`. At clock tick 9, signal `trans` is true, but false at all other clock ticks.

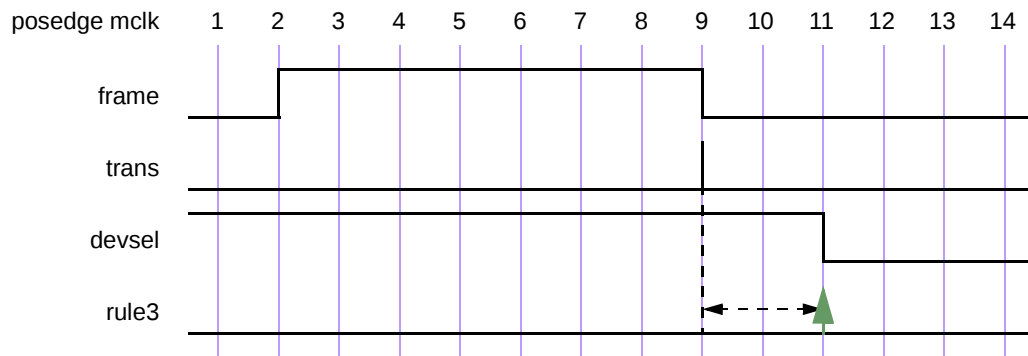
When the same clock is used for both **event** sequence expressions, the sequence event coincides with the sampling clock, and will be available in the same clock tick. This is illustrated in [Figure 1-16](#). The waveform shown for signal `frame` is the result of sampled values of the design signal `frame` with respect to the clock `mclk`.

```

clock posedge mclk {
    event trans: negedge frame;
    event rule4: if (trans) then #[1..3] negedge devsel;
}

```

Figure 1-16 Events Using the Same Clock



Finally, if multiple sequences of evaluation are required for an expression, the time at which the sequence event is generated is determined by the following rule:

- Every time a match is recognized for a sequence, a sequence event is generated at that time.
- Every time there is a match failure for a sequence, no sequence event is generated.

Resolving Clock for Event Definitions

This section describes the rules governing the resolution of the clock for an event expression, when event definitions are instantiated in the expression. As we saw from the previous section, any number of event expressions can be used as sub-expressions in the definition of an event expression. The instantiated sub-expressions may or

may not be bound to a clock. This gives rise to conflicting situations where a sub-expression may be bound to a clock different than the clock where it is instantiated.

Following rules with examples describe the clock resolution scheme. Consider the following code example:

```
event t1: a;
clock edge clk {
    event t2: t1;
    event t3: b #1 c;
    event t4: t2 #1 t3;
}
clock negedge clk {
    event t5: e #1 f;
}
event t6: t2;
event t7: t3 #1 t5;
event t8: g #1 f;
event t9: t1 #1 t8;
```

1. When an unclocked event is instantiated in a clocked event, it inherits the clock of the clocked event. In the example, event t1 will be evaluated with respect to clock “edge clk” when used in t2.
2. When a clocked event is instantiated in the definition of an unclocked event, the definition will inherit the clock of the clocked event. In the example, t6 will be evaluated with respect to clock “edge clk”.
3. When two events are instantiated that are bound to different clocks, an error is reported. In the example, event definition t7 is an error because event t3 and t5 are bound to different clocks. While event definition t4 is correct as both events t2 and t3 are on the same clock.

4. When an unlocked event is instantiated in an unlocked definition, the definition remains unlocked. In the example, event t9 is unlocked as both events t1 and t8 are unlocked.
5. After resolution of clocks with respect to the sub-expression, if the event remains unlocked, it assumes the simulation time as clock.

Specifying Temporal Assertions

The syntax for specifying a temporal assertion is as follows:

```
assert event_name ;  
    | name : sequence_expr | event_name ;  
    | name : check | forbid (sequence_expr|event_name) ;
```

Assertions are expressed as ***assert*** directives. An assertion defines a property of a system that is monitored to provide the user with a functional validation capability. Properties are specified as temporal expressions, where complex timing and functional relationships between values and events of the system are expressed. The ***assert*** directive can be specified in four different ways:

- `assert name : sequence_expr ;`

An ***assert*** directive is declared with an identifier *name* to name the assertion and a sequence expression *sequence_expr* to specify the relationship for monitoring. While reporting, the name serves to identify the results for that specific assertion. No explicit clock can be associated with the ***assert*** directive.

sequence_expr is evaluated at the first clock tick only, and the results of this evaluation generate a set of sequence matches

and failed sequences. The assertion succeeds if the results include at least one matched sequence. Otherwise the assertion fails.

- `assert name : event_name ;`

In this case, *event_name* is an identifier of an event that specifies the sequence expression. The sequence expression specified by *event_name* is evaluated according to its clock at the first clock tick. The success criteria of the assertion is the same as described for the first case.

- `assert event_name ;`

This is similar to the second case, except that the name of the assertion is taken to be the *event_name*.

- `assert name : check(event_name);`
`assert name : check(sequence_expr);`
`assert name : forbid(event_name);`
`assert name : forbid(sequence_expr);`

For the first three cases, the sequence expression is evaluated only once, that is at the first clock tick. But, in general, the assertions are required to hold for the entire simulation. To accomplish that objective, two built-in functions are provided: *check* and *forbid*. These functions start a new evaluation attempt at every clock tick and determine if the assertion succeeds. The *check* function ensures that every evaluation attempt results in at least one matched sequence, while the *forbid* function ensures that no sequences are matched. The *check* function is specified with the expectation that the sequence expression will hold true for all attempts. On the other hand, the *forbid* function is specified to ensure that a certain condition never occurs. The clock is determined in the same way as for the first three cases. That is, if the argument to these functions is a sequence expression, then

the simulation clock is used as the clock to evaluate the expressions. If an event name is used as the argument, if there is a clock specified for the event that clock is used, if no clock is specified then the simulation clock is used.

Whenever a clock tick occurs, the values and events are examined to determine if the sequence expression for each assertion succeeds or fails.

For example, in a bus read transaction, assume that there is a turn-around time of one clock cycle. After the signal named `frame` toggles low, the signal named `trdy` must not get toggled low in the next clock tick. Also assume that signal `trdy` gets toggled low eventually at some point in time in compliance with the bus operation rules. An assertion can be written for this as follows:

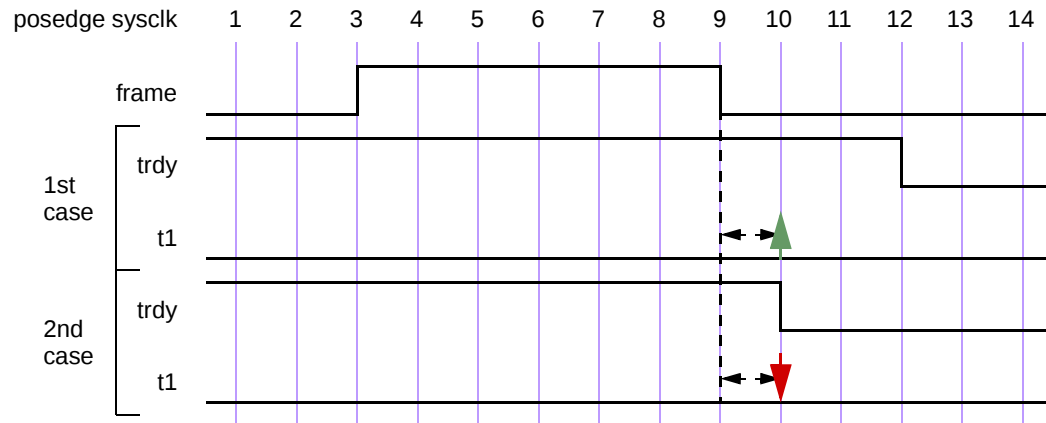
```
assert prop_t1: check(t1);
clock posedge sysclk {
    event t1: if (negedge frame) then
        (#1 !(negedge trdy));
}
```

In this example, `sysclk` is used as a clock to sample the values for the sequence expressions. If `(negedge trdy)` occurs in the next clock tick, `t1` will fail, otherwise `t1` will succeed. In [Figure 1-17](#), these two cases are illustrated by showing two different waveforms for signal `trdy`. Please note that the values shown for the signals are sampled values with respect to their clock. Consider the evaluation attempt when `(negedge frame)` occurs at clock tick 9. All evaluation attempts at other clock ticks succeed as preconditions `(negedge frame)` does not occur.

- In the first case, `(negedge trdy)` does not occur at clock tick 1 resulting in a match at clock tick 10. Thus, the attempt at clock tick 9 succeeds.

- In the second case, (negedge trdy) occurs at clock tick 10, resulting in a failed match at clock tick 10. Thus, the attempt at clock tick 9 fails.

Figure 1-17 assert Example



Specifying Assertions for an Instance of a Module

The syntax for specifying assertions for an instance is as follows:

```
scope instance_name
{
    assert, clock, and scope statements
}
```

The **scope** construct is used to bind a list of assertions to a specific instance of the design. A **scope** is declared with a name that must match a full hierarchical instance name in the design under consideration.

Note:

A *scope* does not instantiate a Verilog design object, but refers to a Verilog instance already declared within the design.

A **scope** declaration helps to organize assertion specifications for the design. All assertions specified within a **scope** specification belong to that instance. For example, checker `tc2` belongs to the instance `tb.a`:

```
scope tb.a {
    assert checker_tc2: check(tc2);
    clock edge(clk) {
        event tc2: if (posedge req) then
            ([1..11]posedge req_end);
    }
}
```

A **scope** declaration provides scoping of names for Verilog variables referred in the expressions specified for that scope. Any Verilog variable referenced in an expression automatically assumes the scope of its **scope** name. If a variable is not in the specified instance, OVA does not search up the hierarchy for it. In the above example, variables `clk`, `req` and `req_end` are assumed to be declared for Verilog instance `tb.a`. A variable in another instance can be referenced using the same conventions as Verilog for hierarchical cross-referencing. For example, below, assertion `tc2` in instance `tb.m1` refers to variable `req` in instance `tb.m2` as `tb.m2.req`, while variable reference `req_end` refers to a variable in instance `tb.m1`.

```
scope tb.m1 {
    assert checker_tc2: check(tc2);
    clock edge(clk) {
        event tc2: if (posedge tb.m2.req)
            then ([1..11]posedge req_end);
    }
}
```

A **scope** declaration provides local scoping of **event** and **assert** identifiers. Two expressions can be declared with the same name in two different scopes. The naming conventions are the same as

hierarchical names used in Verilog. For example, below, expression `tc4` in instance `top.m2` refers to a sequence event `te1` in instance `tb.m1` as `tb.m1.te1`.

```
scope tb.m1 {
    clock posedge sysclk {
        event te1: b && c ;
        event te2: posedge s ;
        event te3: if (matched te1) then te2 ;
    }
}
scope tb.m2 {
    clock edge(clk) {
        event tc4: if (matched tb.m1.te1)
            then ([1..11] posedge end_trans);
    }
}
```

A ***scope*** declaration can be nested. This is provided so that a hierarchy that is defined in the design can be referenced in the same way, without explicitly defining assertions for each flat scope.

A ***scope*** declaration can be nested with or without a clock specified for the nested scope. In the case where a clock is specified for the nested scope, there are two restrictions on the specifications within that scope:

- The clock applies for all the expressions in the nested scope and the recursively nested scopes.
- No other clocks may be specified within the nested scopes and recursively nested scopes. In other words, there is no nesting of clocks allowed.

An example of scope nesting with and without a clock is illustrated below:

```

scope tb {
    // tb.m1 is an unclocked nested scope
    scope m1 {
        clock posedge sysclk {
            event te1: b && c ;
            event te2: posedge s ;
            event te3: if (matched te1) then te2 ;
        }
    }
    clock edge(clk) {
        // tb.m2 is a clocked nested scope. No clock
        // separation is allowed for check tc4 and tc5.
        scope m2 {
            event tc4: if (matched tb.m1.te1)
                then ([0..10] posedge end_trans) ;
        }
        event tc5: if (start_test)
            then (start_test #[1..] end_test) ;
    }
}

```

Specifying Assertions for A Module

The syntax for specifying an assertion for a module is as follows:

```

module instance_name
{
    assert, clock, and scope statements
}

```

Assertions or expressions for a module definition can be defined using the **module** construct. **module** is declared with a name that must be a module definition declared in the design. Like the **scope** construct, **module** is also a feature to provide scoping of referenced variables. While a **scope** refers to a specific instance, a **module** refers to a module definition, and thereby refers to all instances of that module definition.

Note:

A *module* does not define a Verilog design object. A *module* name must refer to a Verilog module already declared within the design.

Note these differences between a ***scope*** and a ***module*** declaration:

- Assertions declared in a ***module*** apply to all its Verilog instances. Assertions specified for a ***scope*** apply to that Verilog instance only.
- No ***module*** may be nested within a ***module***. On the other hand, a ***scope*** declaration can be nested with other ***scope*** declarations.

An example of a *module* declaration is shown below. *event* *te1* and *te2*, and *assert* *c1* and *c2* belong to *module* *cpu*, while *event* *tc3* and *assert* *c3* belong to *module* *iop*.

```
module cpu {
    assert c1: check(tc1);
    assert c2: check(tc2);
    clock posedge sysclk {
        event te1: b #1 c ;
    }
    clock edge(dclk) {
        event te2: pipe1 #1 pipe2 ;
    }
    clock edge(eclk) {
        event tc1: if (e1) then matched te1 ;
        event tc2: if (e2) then matched te2 ;
    }
}
module iop {
    assert c3: check(tc3);
    clock edge(clk) {
        event tc3: if (posedge mem_req)
            then ([0..10]posedge mem_end) ;
    }
}
```

A *module* declaration provides scoping of names for Verilog variables referred in the assertions specified for that module. Any Verilog variable referenced in an expression automatically assumes the scope of its module. If a variable is not in the specified instance, OVA does not search up the hierarchy for it. In the example above, variables `sysclk`, `dclk`, `eclk`, `b`, `c`, `pipe1`, `e1`, and `e2` are assumed to be declared in module `cpu` in the design, while variables `clk`, `mem_req` and `mem_end` are assumed to be declared in Verilog module `iop`. A variable in another module can be referenced using the same conventions as Verilog for hierarchical cross-referencing.

Assertions declared within a **module** are applied to all instances of the Verilog module. This feature is illustrated below:

```
module bus {
    assert ce4: check(e4);
    clock edge(clk) {
        event e4: if (posedge req) then
            ([1..10]posedge end);
    }
}
```

In the following code example, assertions have been specified for module definition `bus`. In the Verilog description, three instances of `bus` as `bus1`, `bus2` and `bus3` are declared with the full hierarchical name of `tb.bus1`, `tb.bus2` and `tb.bus3` respectively. The assertion specification is equivalent to:

```

scope tb.bus1 {
    assert ce4: check(e4);
    clock edge(clk) {
        event e4: if (posedge reg) then
            ([1..10]posedge end);
    }
}
scope tb.bus2 {
    assert ce4: check(e4);
    clock edge(clk) {
        event e4: if (posedge reg) then
            ([1..10]posedge end);
    }
}
scope tb.bus3 {
    assert ce4: check(e4);
    clock edge(clk) {
        event e4: if (posedge reg) then
            ([1..10]posedge end);
    }
}

```

Name Resolution

This section specifies the rules regarding the name resolution whenever there is a conflict between a name declared in the sequence expression specification and a design object name from Verilog. There can be two kinds of name conflict as follows:

1. A design object name is also a sequence expression language keyword such as **inv** or **in**.
2. A design object name is also an identifier declared in the sequence expression language such as for an event or bool.

To resolve the first conflict, use the escape mechanism using **v'**name to denote a design variable. An example is shown below.

```

clock posedge clk {
    event seq_e: if (trans) then v'inv;
}

```

The design variable name is `inv`, but as it conflicts with the keyword **`inv`**, it is used as `v'inv`.

For the second kind of conflict, the compiler gives a warning that a name is shadowing a design variable name, and resolves it to the name declared for the sequence expression. To force the compiler to use the design variable instead of the sequence expression name, use the same escape mechanism using **`v'`**. For example:

```

assert rule: if (matched start) then reset_end #1 v'start;
clock posedge clk {
    event start: if (reset) then int_sequence;
}

```

In the above example, `start` is declared as a sequence expression. In the declaration `rule`, `start` is referenced twice, once to refer to the sequence expression, and second to the design object `start` using `v'start`.

Specifying Composite Sequences

Sequences can be combined with functions such as AND and OR. Sequences can also be modified with the invert function. The syntax for specifying composite sequences is as follows:

```

sequence_expr && sequence_expr
sequence_expr intersect sequence_expr
sequence_expr || sequence_expr
first_match sequence_expr
inv sequence_expr

```

Logically ANDing Sequences

The binary operator **&&** is used when both operand expressions are expected to succeed, but the end times of the operand expressions may be different.

sequence_expr && sequence_expr

The two operands of **&&** are sequence expressions. The requirement for the success of the **&&** operation is that both the operand expressions must succeed. When one of the operand expressions succeeds, it waits for the other to succeed. The end time of the composite expression is the end time of the operand expression that completes the last.

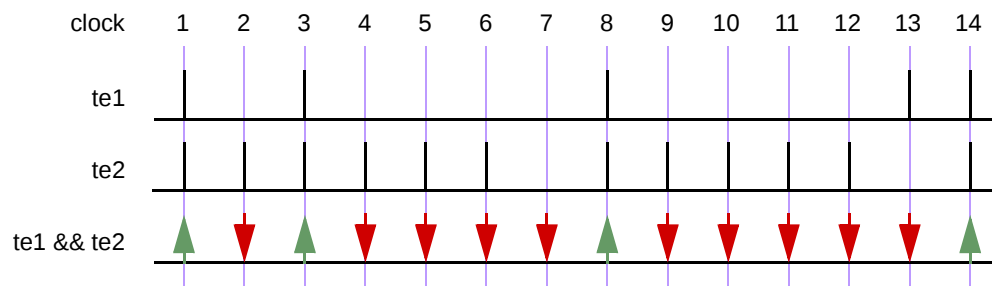
For the expression:

te1 && te2

If *te1* and *te2* are events, the expression succeeds if *te1* and *te2* are both evaluated to be true.

An example is illustrated in [Figure 1-18](#) to show the results for attempt at every clock tick. The expression matches at clock tick 1, 3 and 8 because both *te1* and *te2* are simultaneously true. At all other clock ticks, operation **&&** fails because either *te1* or *te2* is false.

Figure 1-18 ANDing (&&) Two Events



When `te1` and `te2` are sequences, then the expression:

`te1 && te2`

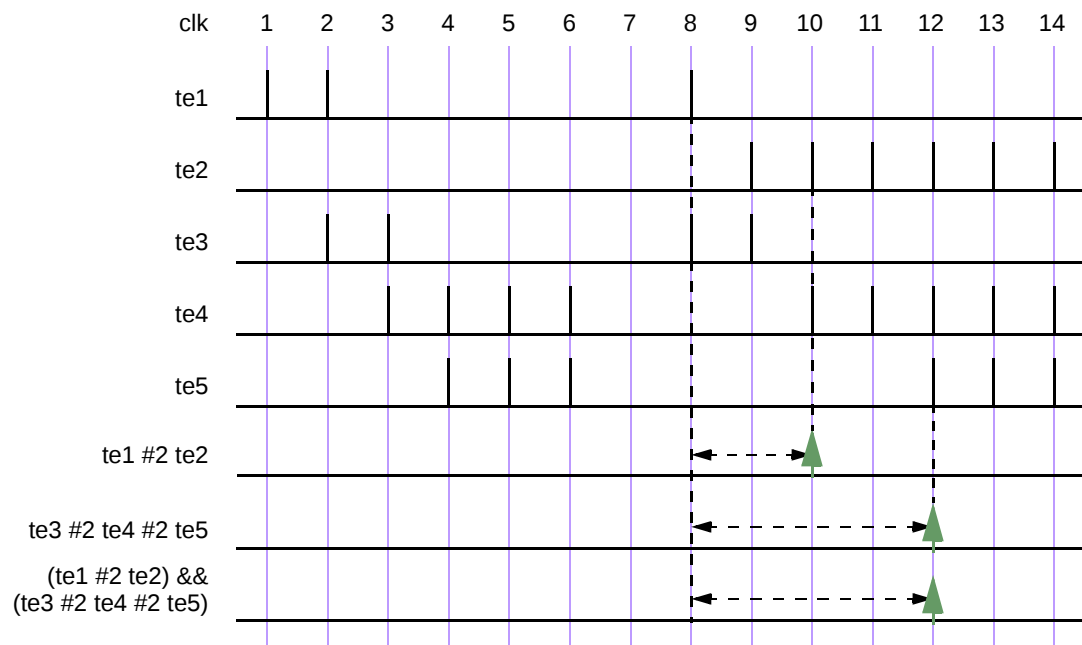
- Succeeds if `te1` and `te2` succeed.
- The end time is the end time of either `te1` or `te2`, whichever terminates last.

First, let us consider the case when both operands are single sequence evaluations.

An example is illustrated in [Figure 1-19](#). Consider the following expression with operator `&&` where the two operands are sequences.

`(te1 #2 te2) && (te3 #2 te4 #2 te5)`

Figure 1-19 ANDing (&&) Two Sequences



Here, the two operand sequences are (te1 #2 te2) and (te3 #2 te4 #2 te5). The first operand sequence requires that first te1 evaluates to true followed by te2 two clock ticks later. The second sequence requires that first te3 evaluates to true followed by te4 two clock ticks later, followed by te5 two clock ticks later.

Figure 1-19 shows the evaluation attempt at clock tick 8.

This attempt results in a match since both operand sequences match. The end times of matches for the individual sequences are clock ticks 10 and 12. The end time for the entire expression is the last of the two end times, so a match is recognized for the expression at clock tick 12.

Now, consider an example where an operand sequence is associated with a range of time specification, such as:

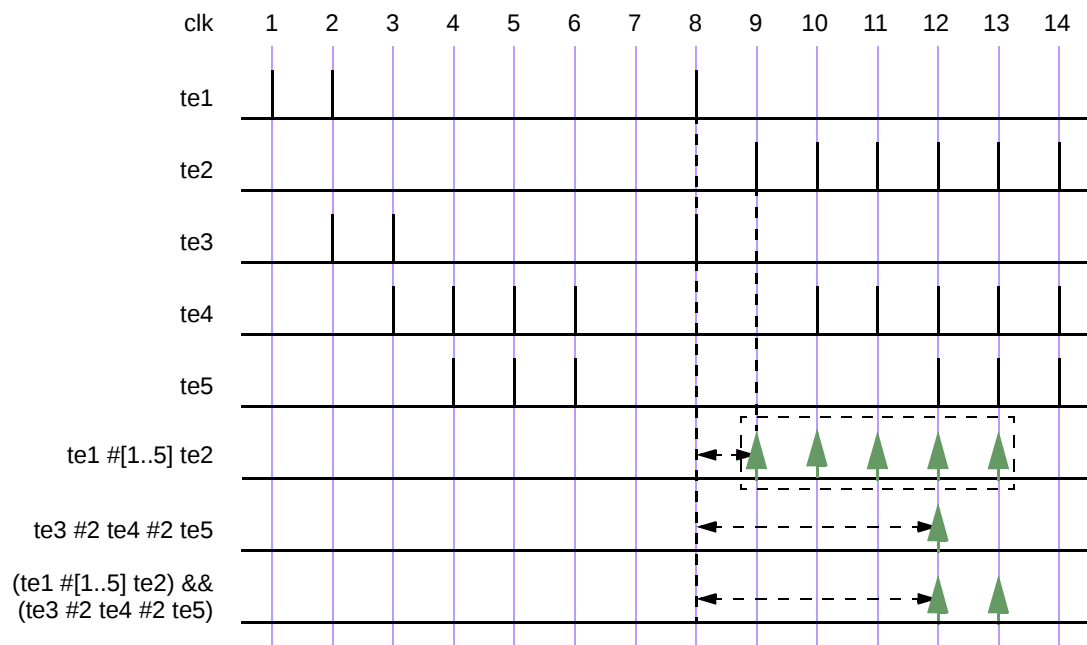
```
(te1 #[1..5] te2) && (te3 #2 te4 #2 te5)
```

The first operand sequence consists of an expression with a time range from 1 to 5 and implies that when t_{e1} evaluates to true, t_{e2} must follow 1, 2, 3, 4, or 5 clock ticks later. The second operand sequence is the same as in the previous example. To consider all possibilities of a match, following steps are taken:

- The first operand sequence starts five sequences of evaluation.
- The second operand sequence has only one possibility of match, so only one sequence is started.
- [Figure 1-20](#) shows the attempt to examine at clock tick 8 when both operand sequences start and succeed. All five sequences for the first operand sequence match, as shown in a time window, at clock ticks 9, 10, 11, 12 and 13 respectively. The second operand sequence matches at clock tick 12.
- To compute the result for the composite expression, each successful sequence from the first operand sequence is matched against the second operand sequence according to the rules of the **&&** operation to determine the end time for each match.

The result of this computation is five successes, four of them ending at clock ticks 12, and the fifth ends at clock tick 13. [Figure 1-20](#) shows the two unique successes at clock ticks 12 and 13.

Figure 1-20 ANDing (&&) Two Sequences Including a Time Range



Intersecting Sequences

The binary operator ***intersect*** is used when both operand expressions are expected to succeed, and the end times of the operand expressions must be the same.

```
sequence_expr intersect sequence_expr
```

The two operands of ***intersect*** are sequence expressions. The requirements for the success of the ***intersect*** operation are:

- Both the operand expressions must succeed.
- The length of the two operand sequences must be the same.

The additional requirement on the length of the sequences is the basic difference between ***&&*** and ***intersect***.

When there are multiple matches for each operand sequence expression, the results are computed as follows.

- A match from the first operand is paired with a match from the second operand with the same length (end time).
- If no such pair is found, the result of ***intersect*** is no match.
- If such pairs are found, then the result consists of matched sequences, one for each pair. The end time of each match is determined by the pair.

Logically ORing Sequences

The operator `||` is used when at least one of the two operand sequences is expected to succeed.

sequence_expr `||` *sequence_expr*

The two operands of `||` are sequence expressions.

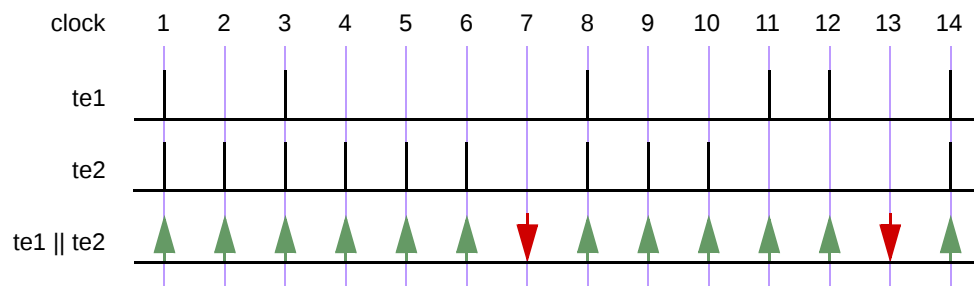
Let us consider these operand expressions as values, events and sequences separately to illustrate the details of `||` operations. For the expression:

`te1 || te2`

when the operand expressions `te1` and `te2` are events or values, the expression succeeds whenever at least one of two operands `te1` and `te2` is evaluated to true.

[Figure 1-21](#) illustrates `||` operation using `te1` and `te2` as simple values. The expression fails at clock ticks 7 and 13 because `te1` and `te2` are both false at those times. At all other times, the expression succeeds, as at least one of the two operands is true.

Figure 1-21 ORing (||) Two Events



When `te1` and `te2` are sequences, then the expression:

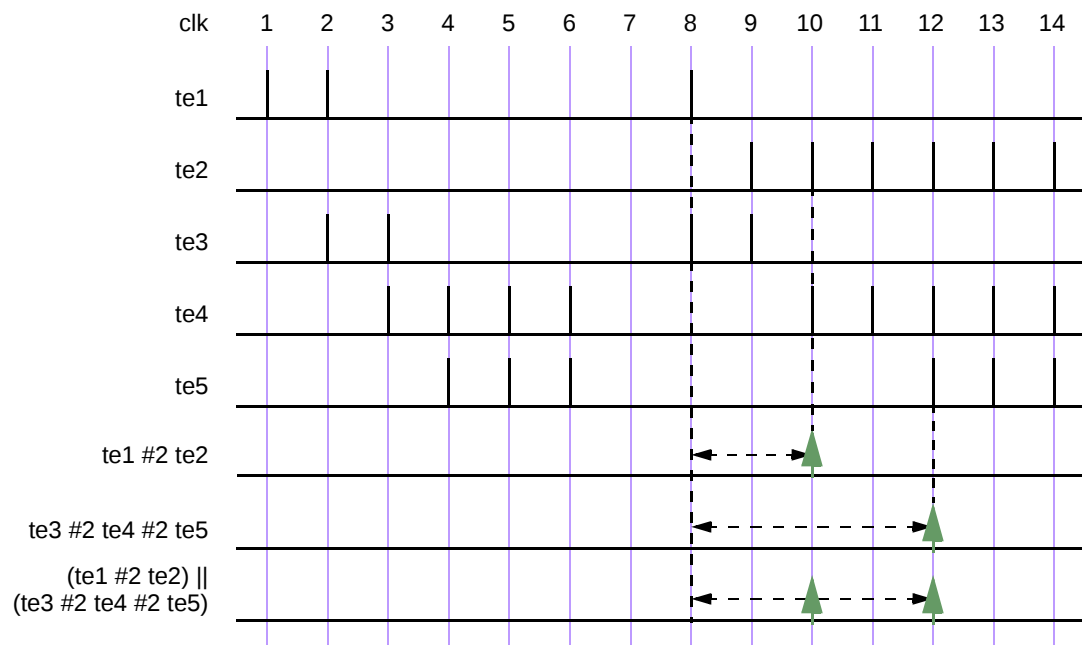
`te1 || te2`

Succeeds if at least one of the two operand sequences `te1` and `te2` succeed. To evaluate this expression, first, the successfully matched sequences of each operand are calculated and assigned to a group. Then, the union of the two groups is computed. The result of the union provides the result of the expression. The end time of a match is the end time of any sequence that matched.

An example is illustrated in [Figure 1-22](#). Consider an expression with `||` operator where the two operands are sequences.

`(te1 #2 te2) || (te3 #2 te4 #2 te5)`

Figure 1-22 ORing (||) Two Sequences



Here, the two operand sequences are: (te1 #2 te2) and (te3 #2 te4 #2 te5). The first sequence requires that te1 first evaluates to true, followed by te2 two clock ticks later. The second sequence requires that te3 evaluates to true, followed by te4 two clock ticks later, followed by te5 two clock ticks later. In [Figure 1-22](#), the evaluation attempt for clock tick 8 is shown. The first sequence matches at clock tick 10 and the second sequence matches at clock tick 12. So, two matches for the expression are recognized.

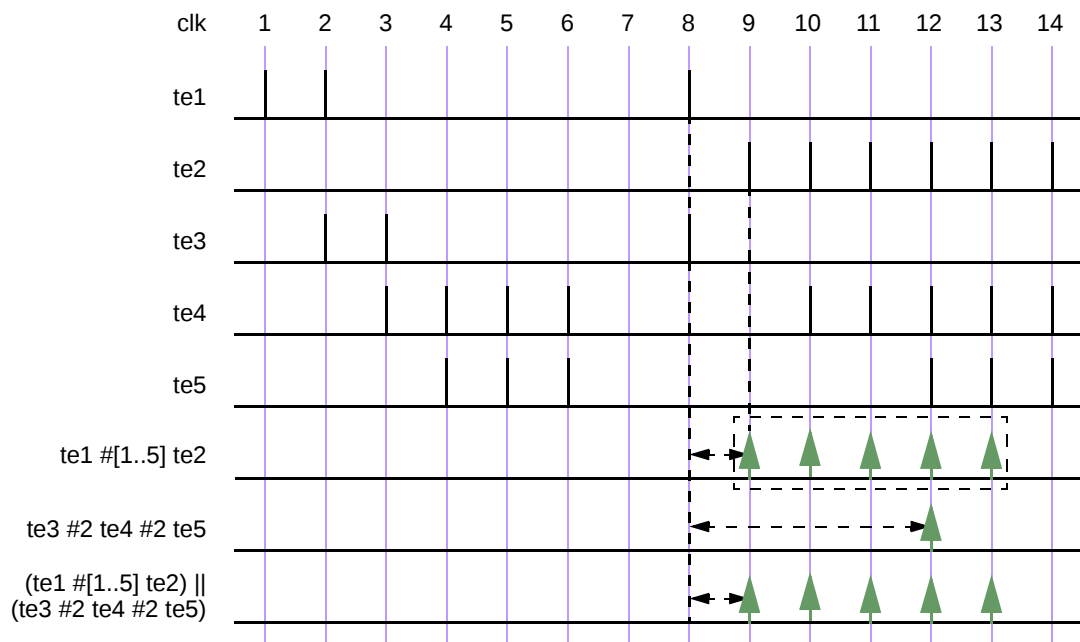
Consider an example where an operand sequence is associated with time range specification, such as:

```
(te1 #[1..5] te2) || (te3 #2 te4 #2 te5)
```

The first operand sequence consists of an expression with a time range from 1 to 5 and specifies that when te1 evaluates to true, te2 must be true 1, 2, 3, 4 or 5 clock ticks later. The sequences from the second operand require that first te3 must be true followed by te4

being true two clock ticks later, followed by $te5$ being true two clock ticks later. At any clock tick if an operand sequence succeeds, then the composite expressions succeeds. As shown in [Figure 1-23](#), for the attempt at clock tick 8, the first operand sequence matches at clock ticks 9, 10, 11, 12, and 13, while the second operand matches at clock ticks 12. The match of the composite expression is computed as a union of the matches of the two operand sequences, which results in matches at clock ticks 9, 10, 11, 12, and 13.

Figure 1-23 ORing ($\parallel$) Two Sequences Including a Time Range



Selecting First Match

Use the ***first_match*** operator when you are interested in only the first match from a sequence expression that can possibly result in multiple matches. This allows you to discard all subsequent matches from consideration. In particular, when the sequence expression is a

sub-expression of a larger expression, then applying the ***first_match*** operator has significant effect on the evaluation of the embedding expression.

```
first_match sequence_expr
```

The operand expression can be a sequence expression. `sequence_expr` is evaluated to determine the match for the (***first_match*** `sequence_expr`) expression. The composite expression matches if `sequence_expr` results in at least one match of a sequence, and fails to match if none of the sequences from the expression result in a match. Following the first successful match, the ***first_match*** operator stops matching subsequent sequences for `sequence_expr`. If there are multiple matches at the same time as the first detected match, then all those matches are considered as the result of the expression.

Consider an example with a variable delay specification as shown below.

```
event t1: te1 #[2..5] te2;  
event ts1: first_match(te1 #[2..5] te2);
```

Event `t1` can result in matches for up to four following sequences:

```
te1 #2 te2, te1 #3 te2, te1 #4 te2, te1 #5 te2
```

However, event `ts1` can result in only one of the above four sequences. Whichever of the above four sequences matches first becomes the result of event `ts1`.

Inverting Sequences

Use the ***inv*** operator when you are interested in inverting a sequence match to a no match. This allows you to detect a failure of an individual sequence and use it as an a precondition for checking. Please note that the ***inv*** operator does not compute the true complement of a sequence expression.

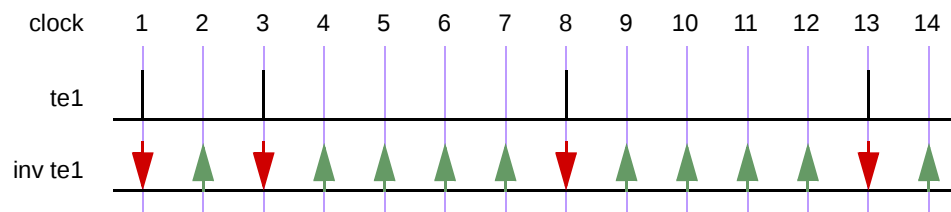
```
inv sequence_expr
```

The operand expression can be a sequence expression. `sequence_expr` is matched to determine the match for the (***inv*** `sequence_expr`) expression. The composite expression matches if `sequence_expr` results in at least one no match of a sequence, and fails to match if all the sequences from the expression result in a match. The ***inv*** operator simply inverts the match of its operand. This is an important point to note that the ***inv*** operator is applied to every single match occurring for its operand, and not just to the overall success/failure of an assertion. Let us first consider the case when the operand expression is a signal.

```
inv tel
```

[Figure 1-24](#) illustrates the operation of ***inv*** operator for all attempts of this expression. Since `tel` is a signal, its value is examined at every clock tick. If the value of `tel` is false, then the expression succeeds, otherwise the expression fails.

Figure 1-24 Inverting (inv) an Event



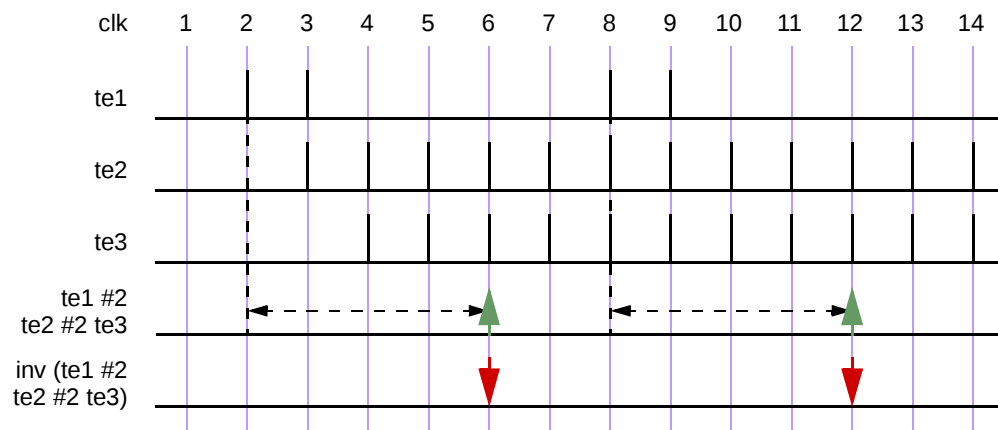
`te1` is false at clock ticks 2, 4, 5, 6, 7, 9, 10, 11, 12, and 14. Accordingly, the expression “`inv te1`” matches at those times. Conversely, `te1` is true at clock ticks 1, 3, 8, and 13, so the expression fails to match at those clock ticks.

Now consider an expression which is a sequence,

```
inv (te1 #2 te2 #2 te3)
```

The operand expression `(te1 #2 te2 #2 te3)` is a sequence. The above example is illustrated in [Figure 1-25](#) for the attempts at clock tick 2 and 8.

Figure 1-25 Inverting (inv) a Sequence



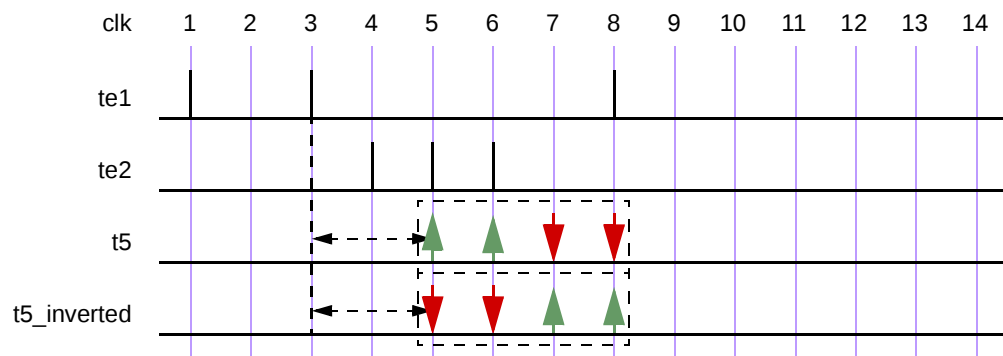
The sequence (te1 #2 te2 #2 te3) matches at clock ticks 6 and 12. The results of the inverted sequence are computed by inverting the match for the sequence. The inverted sequence `inv( te1 #3 te2 #3 te3 )` fails to match at clock ticks 6 and 12.

Consider an example with a variable delay specification as shown below.

```
event t5: te1 #[2..5] te2;
event t5_inverted: inv( te1 #[2..5] te2 );
```

The results of `t5` are shown in [Figure 1-26](#). `t5_inverted` is computed by inverting the match of `t5`.

Figure 1-26 Inverting (inv) a Sequence Including a Time Range



Specifying Conditional Sequence Matching

The syntax for conditional sequence matching is as follows:

```
if boolean_expr then sequence_expr
  [else sequence_expr]
```

These constructs allow a user to monitor sequences based on satisfying some criteria. Most common uses are to attach a precondition to a sequence, and to select a sequence between two alternatives, where the selection is made based on the success of a condition.

Two kinds of clauses are provided:

```
if boolean_cond then sequence_expr
```

This clause is used to precondition monitoring of a sequence expression. (The functionality provided here is the same as obtained by an implication operator in some temporal languages). The condition `boolean_cond` must be satisfied in order to monitor `sequence_expr`. If the condition `boolean_cond` fails then `sequence_expr` is skipped for monitoring. `boolean_cond` is a

logical expression that results in true or false, and `sequence_expr` is a sequence expression that can result in one or match sequence matches.

Please note that `boolean_cond` cannot be a sequence expression but it can be `matched clocked_sequence_expr`.

If the condition is evaluated to true, then the evaluation of `sequence_expr` is conducted. The sequence matches of `sequence_expr` matches become the matches of the clause ***if then***.

```
if boolean_cond then sequence_expr1 else sequence_expr2
```

This clause is used to select a sequence expression between two alternatives. If `boolean_cond` results is true, then `sequence_expr1` is monitored. If `boolean_cond` is false, then `sequence_expr2` is selected for monitoring. The expression `boolean_cond` is logical and must result in true or false. `sequence_expr1` and `sequence_expr2` can be sequence expressions. The match of clause ***if then else*** depends on the match of the sequence expression, `sequence_expr1` or `sequence_expr2`, whichever gets selected for monitoring.

Clause ***if*** can be nested to contain another ***if*** within it, such as:

```
if (!reset) then
    if (data_phase) then #[0..7] data_end;
```

When `(!reset)` is true, then the second ***if*** condition `data_phase` is tested. If `data_phase` evaluates to true, then the evaluation continues for the expression `#[0..7] data_end`.

Since the **else** of if-then-else is optional, the binding of the **else** can be confusing in the case of a nested **if** specification. The ambiguity of binding an **else** to its **if** is resolved by associating the **else** with the closest previous **if** that lacks an **else**. An example below illustrates the binding of a nested if with an else part.

```
if (!reset) then
    if (data_phase) then #[0..7] data_end
    else #[0..7] addr_phase;
if (!reset) then transfer_cmd
else if (data_phase)
    then #[0..7] data_end
    else #[0..7] addr_phase;
```

The bold font highlights the association of the **if** with its **then** and its **else**. If such association is not intended, then using parenthesis can enforce a binding, such as shown in the example below.

```
if (!reset) then
    (if (data_phase) then #[0..7] data_end)
    else #[0..7] addr_phase;
if (!reset) then transfer_cmd
else if (data_phase) then
    (if (burst_mode) then #[0..7] data_end)
    else #[0..7] addr_phase;
```

The semantics of **if then else** specification is explained by examples here. Consider a bus operation for data transfer from a master to a target device. When the bus enters a data transfer phase, multiple data phases can occur to transfer a block of data. During the data transfer phase, a data phase completes on any rising clock edge on which `irdy` is asserted and either `trdy` or `stop` is asserted. Note that an asserted signal here implies a value of low. The end of a data phase can be expressed as:

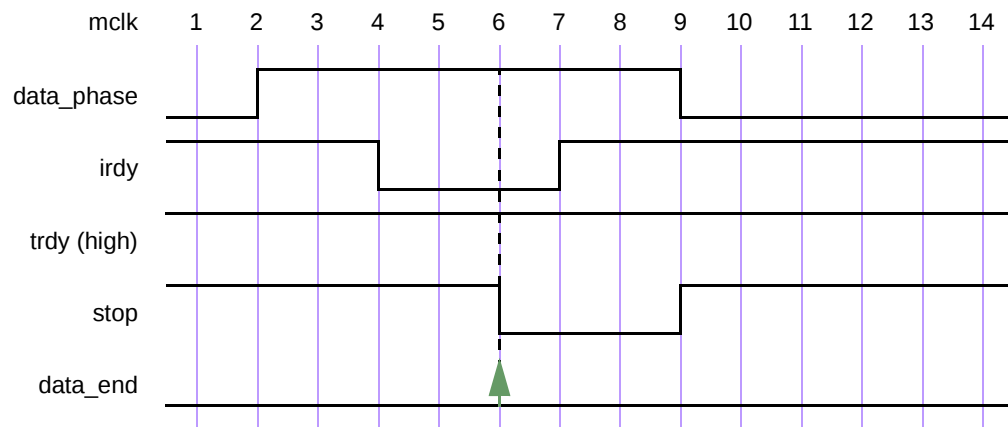
```

clock posedge mclk {
    event data_end :
        if (data_phase) then
            ((irdy==0)&&(negedge trdy||negedge stop));
}

```

Each time a data phase completes, a match for `data_end` is recognized. The attempt at clock tick 6 is illustrated in [Figure 1-27](#). The values shown for the signals are the sampled values with respect to the clock. At clock tick 6 `data_end` is matched because `stop` gets asserted while `irdy` is asserted.

Figure 1-27 Conditional Sequence Matching



`data_end` can be used to ensure that `frame` is de-asserted within 2 clock ticks after `data_end` occurs. Further, it is also required that `irdy` gets de-asserted one clock tick after `frame` gets de-asserted.

A sequence expression is written to express this condition as shown below.

```

clock posedge mclk {
    event data_end_rule1:
        if (matched data_end) then
            #[1..2] posedge frame #1 posedge irdy;
}

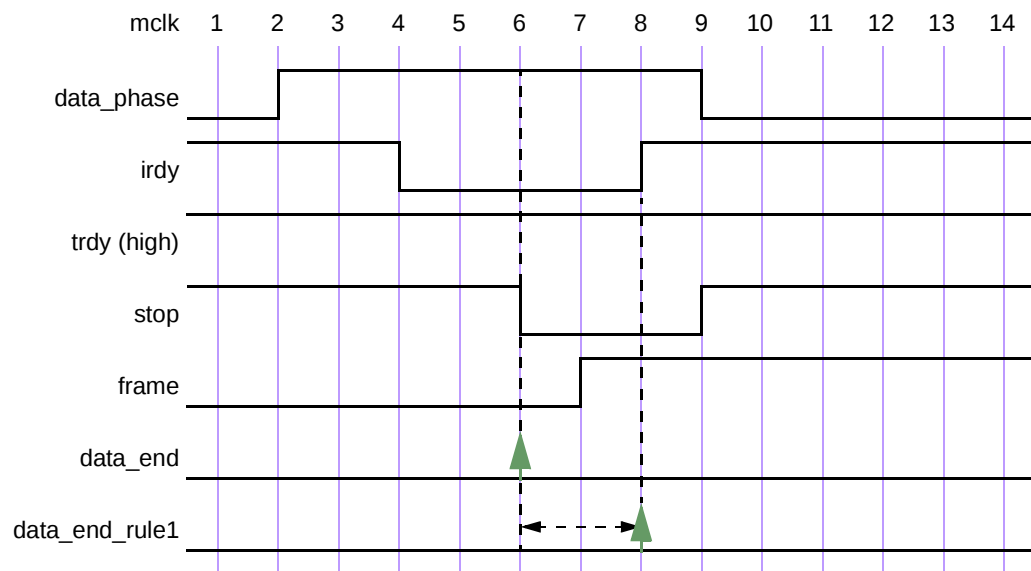
```

event `data_end_seq` first evaluates `data_end` at every clock tick to test if its value is true. If the value is tested to be false, then that particular attempt to check the assertion is considered a success. Otherwise, the sequence expression associated with the ***then*** clause is monitored. The sequence expression:

```
#[1..2] posedge frame #1 posedge irdy
```

Specifies looking for the rising edge of `frame` within two clock ticks in the future. After `frame` toggles high, `irdy` must also toggle high after one clock tick. This is illustrated in [Figure 1-28](#). Event `data_end` is acknowledged at clock tick 6. Next, `frame` toggles high at clock tick 7. Since this falls within the timing constraint imposed by `#[1..2]`, it satisfies the sequence and continues to monitor further. At clock tick 8, `irdy` is evaluated according to `#1` specification. Signal `irdy` transitions to high at clock tick 8, satisfying the sequence specification completely for the attempt that began at clock tick 6.

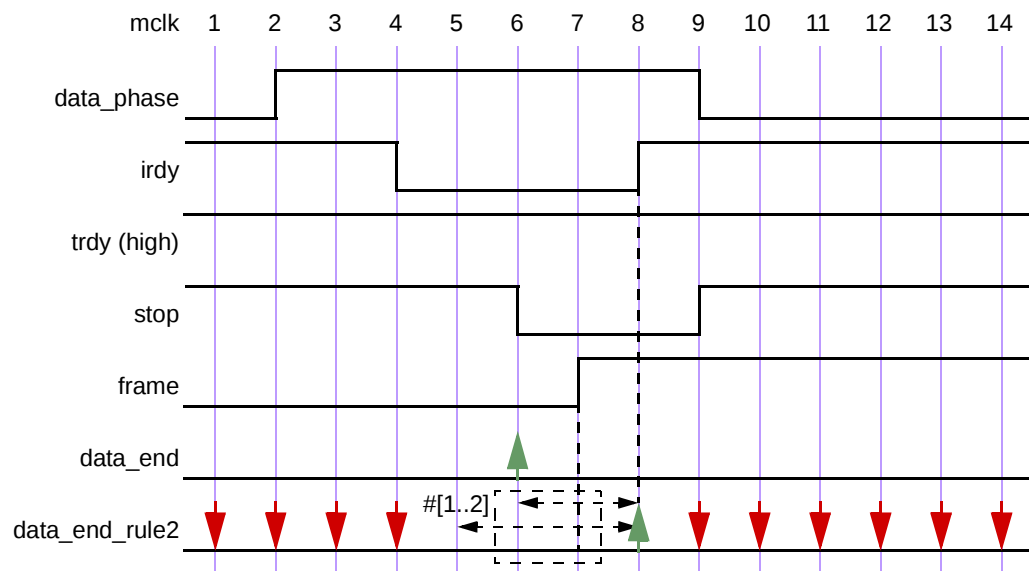
Figure 1-28 Nested Conditional Sequences



Generally, assertions are associated with preconditions so that the checking is performed only under certain specified conditions. As seen from the previous example, the **if** clause provides this capability to specify preconditions with sequences that must be satisfied before continuing to match those sequences. Let us modify the above example to see the effect on the results of the assertion by removing the precondition for the sequence. This is shown below and illustrated in [Figure 1-29](#).

```
clock posedge mclk {
    event data_end_rule2:
        #[1..2] posedge frame #1 posedge irdy;
}
```

Figure 1-29 Results without the Condition



The sequence is evaluated at every clock tick. For the evaluation at clock tick 1, the rising edge of signal `frame` does not occur at clock tick 1 or 2, so the evaluation fails and the result for the sequence is a failed match at clock tick 1. Similarly, there is a failure at clock ticks 2, 3, and 4. For attempts starting at clock ticks 5 and 6, the rising edge of signal `frame` at clock tick 7 allows checking further. At clock tick 8, the sequences complete according to the specification, resulting in a match for attempts starting at 5 and 6. All later attempts to match the sequence fail because `posedge frame` does not occur again.

As one can see from [Figure 1-29](#), removing the precondition of checking event `data_end` from the assertion causes failures that are not relevant for consideration. It becomes important from the validation standpoint to determine these preconditions and use them in the assertion to filter out inappropriate or extraneous situations.

Matching Repetition of Sequences

The BNF for matching repetitions of sequences is as follows:

```
sequence_expr * [int] | [int .. int] | [int .. ]
```

There are situations when a sequence expression is monitored repeated times in succession. In such cases, monitoring is performed for a specified number of times, and each time a success is expected to result from evaluating the sequence expression. In other words, repetition is same as concatenation of the same sequence expression for the specified number of times. Repetition is expressed with a repetition parameter to specify the number of times an expression needs to be monitored. This parameter can be a number or a range of values.

```
sequence_expr * [int]
```

The `int` operand must be a positive integer constant. `sequence_expr` can be any sequence expression. The above expression is semantically equivalent to the following expression:

```
sequence_expr #1 sequence_expr #1 sequence_expr ... for  
int number of times
```

- `sequence_expr` is repeated `int` times, where `int` is a positive integer. For example:

```
(ev1 #1 ev2) * [3]
```

says “sequence (ev1 #1 ev2) must occur three times in a row”. It is equivalent to writing:

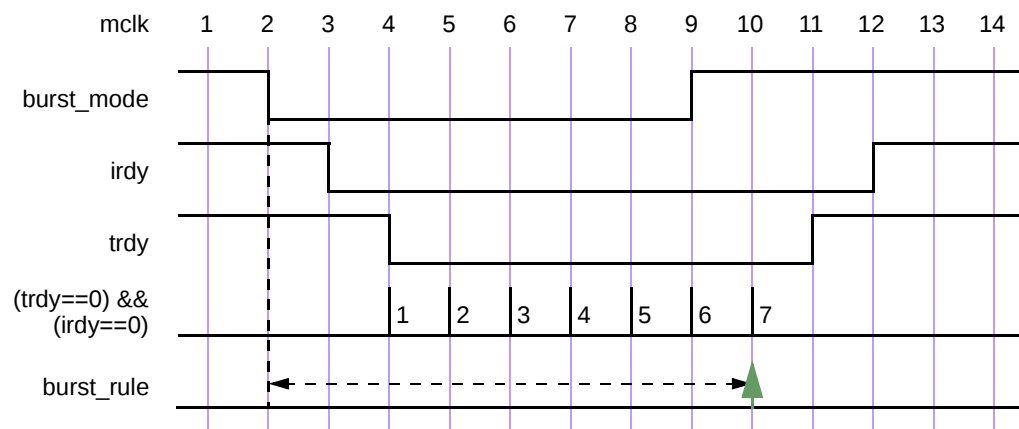
```
(ev1 #1 ev2) #1 (ev1 #1 ev2) #1 (ev1 #1 ev2)
```

- Note that the default number of clock ticks between repetitions is 1.

An example, where a sequence of events is repeated, is shown in [Figure 1-30](#). A bus read transaction in burst mode is expected to read data in eight data phases. Each data phase follows the next, where a data phase is said to occur when signals `irdy` and `trdy` are de-asserted at the same time. In the last data phase signal `frame` is de-asserted to indicate the end of transaction.

```
assert burst_rule: check(burst);
clock posedge mclk {
  event burst:
    if (negedge burst_mode) then
      #2((trdy==0) && (irdy==0)) * [7];
}
```

Figure 1-30 Matching Repetition of a Sequence



The assertion `burst_rule` says “when a falling edge of `burst_mode` is detected, two clock ticks later, data transfer begins (`trdy` and `irdy` both de-asserted) and continues for 7 times”. As can be seen from [Figure 1-30](#), the falling edge of `burst_mode` occurs at clock tick 2 and data transfer begins at clock tick 4. The assertion becomes successful at clock tick 10.

`sequence_expr * [int .. int] | [int .. ]`

The interval can be specified as `[n1..n2]` or `[n1..]`

The interval specifies the restrictions on the number of times a sequence expression can be repeated. `n1` and `n2` specify the minimum and the maximum respectively. If the repetition is required forever, i.e., until the end of simulation, then `n2` must not be specified.

Consider an example,

`(ev1 #1 ev2) * [3..5]`

Says `(ev1 #1 ev2)` must occur for at least 3 times and no more than 5 times. In this case there is a lower limit of 3 and an upper limit of 5 on the number of times that the sequence is expected to repeat. It is equivalent to writing:

```
((ev1 #1 ev2) #1 (ev1 #1 ev2) #1 (ev1 #1 ev2)) ||
((ev1 #1 ev2) #1 (ev1 #1 ev2) #1 (ev1 #1 ev2) #1 (ev1 #1 ev2)) ||
((ev1 #1 ev2) #1 (ev1 #1 ev2) #1 (ev1 #1 ev2) #1 (ev1 #1 ev2)) #1 (ev1 #1 ev2)
```

Again, repetitions occur back to back in terms of clock ticks. The delay between the repetitions can be adjusted to the requirement by adding addition explicit delay, such as:

`(#4 (ev1 #1 ev2)) * [3..5]`

which translates to:

```
(#4 (ev1 #1 ev2) #5 (ev1 #1 ev2) #5 (ev1 #1 ev2)) ||
(#4 (ev1 #1 ev2) #5 (ev1 #1 ev2) #5 (ev1 #1 ev2) #5 (ev1 #1 ev2)) ||
(#4 (ev1 #1 ev2) #5 (ev1 #1 ev2) #5 (ev1 #1 ev2) #5 (ev1 #1 ev2) #5 (ev1 #1 ev2))
```

Another requirement that is commonly encountered is a time range specification, which imposes indeterminate amount of time between each repetition. In such cases, each repetition of a sequence is expected to occur “sometime later” after the occurrence of a preceding sequence. This could be expressed as

```
(#[0..] (ev1 #1 ev2)) * [3..5]
```

which translates to:

```
(#[0..] (ev1 #1 ev2) #[1..] (ev1 #1 ev2) #[1..] (ev1 #1 ev2)) ||
(#[0..] (ev1 #1 ev2) #[1..] (ev1 #1 ev2) #[1..] (ev1 #1 ev2) #[1..] (ev1 #1 ev2)) ||
(#[0..] (ev1 #1 ev2) #[1..] (ev1 #1 ev2) #[1..] (ev1 #1 ev2) #[1..] (ev1 #1 ev2))
#[1..] (ev1 #1 ev2)
```

Specifying Conditions Over Sequences

The syntax for specifying conditions applied to a sequence expression is as follows:

```
cond_spec1, ..., cond_specN in sequence_expr
```

With *cond_spec* being either of:

- *istru boolean_expr*
- *length int | [int .. int] | [int ..]*

Sequences of events often occur under the assumptions of some conditions for correct behavior. A logical condition must hold true, for instance, while processing a transaction. Or, a transaction must complete within a given period of time, no matter what variation of commands are issued in the transaction to be processed. Also

frequently, occurrence of certain events is prohibited while processing a transaction. These situations can be expressed directly using the following two constructs:

```
istru boolean_expr in sequence_expr
```

boolean_expr is an expression which must result to true at every clock tick during the monitoring of *sequence_expr*, where *sequence_expr* is a sequence expression. If a sequence for the *sequence_expr* starts at time *t1* and ends at time *t2*, then *boolean_expr* must hold true from time *t1* to *t2*.

```
length int in sequence_expr  
length int_interval in sequence_expr
```

int or *int_interval* specifies the length of the *sequence_expression*. The length is measured as the total number of clock ticks during the sequence. All variations of the *int_interval* specification are allowed. If a single number is specified for *int_interval*, then it represents fixed length. In other words, the sequence expression must terminate at a specific clock tick that is determined by the *int_interval* number. If *int_interval* specifies a range of numbers, then the *sequence_expression* may terminate anytime within a time period determined by the minimum and maximum numbers.

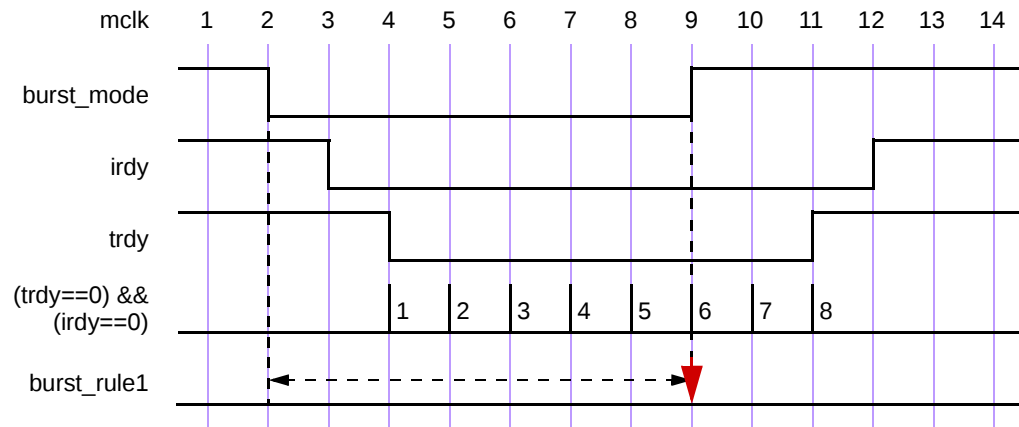
Consider the example illustrated in [Figure 1-31](#). If an additional constraint were placed on the expression as shown below, then the checker *burst_rule* would fail at clock tick 9.

```

assert burst_rule1: check(burst1);
clock posedge mclk {
    event burst1:
        if (negedge burst_mode)
            then istrue (!burst_mode)
                in (#2 ((trdy==0)&&(irdy==0)) * [8]);
}

```

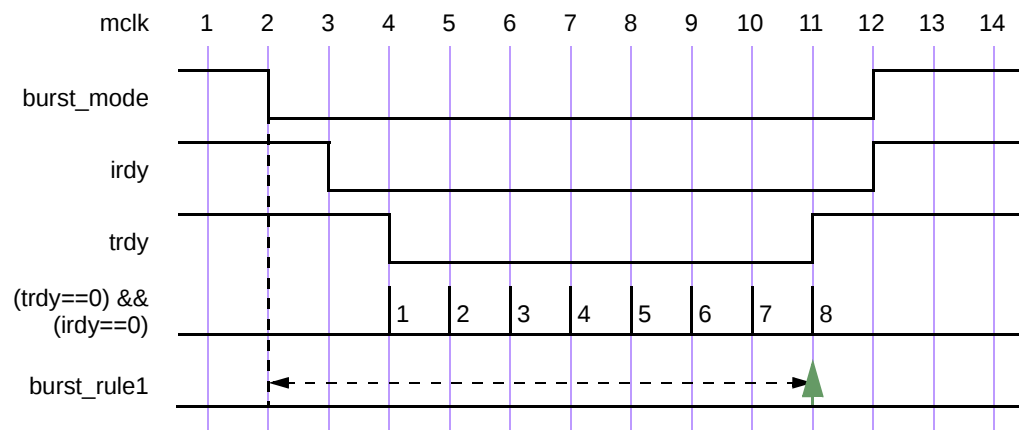
Figure 1-31 Match with istrue-in Restriction Fails



In the above expression, the value of signal `burst_mode` is required to be low during the sequence (from clock tick 2 to 11), and is checked at every clock tick during that period. At clock ticks from 2 to 8, signal `burst_mode` remains low and matches the expression at those clock ticks. At clock tick 9, signal `burst_mode` becomes high, thereby failing to match the expression for `burst_rule1`.

If signal `burst_mode` were to be maintained low until clock tick 11, the expression would result in a match as shown in [Figure 1-32](#).

Figure 1-32 Match with *istrue-in* Restriction Succeeds

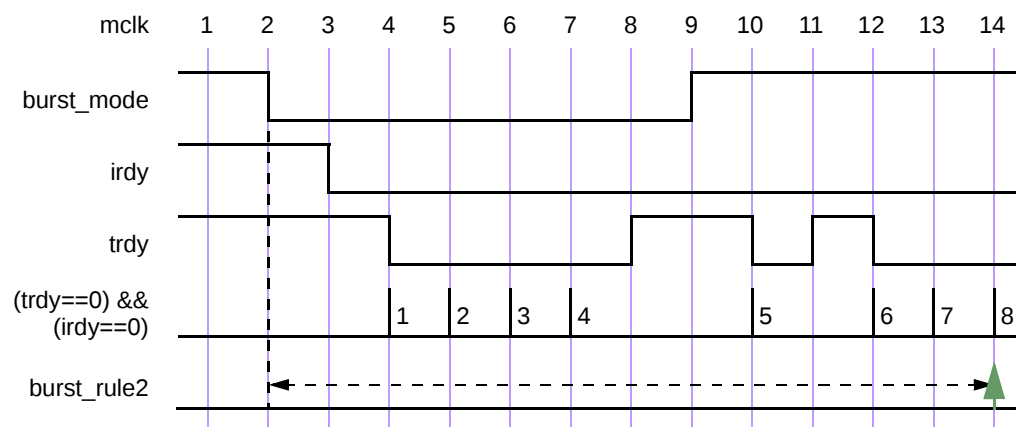


Let us consider a modified version of the example in [Figure 1-32](#) as shown below.

```
assert burst_rule2: check(burst2);
clock posedge mclk {
  event burst2:
    if (negedge burst_mode)
      then (#[0..4] (trdy==0) && (irdy==0)) * [8];
}
```

The assertion `burst_rule2` has been relaxed to require each repetition of the sequence to occur between 1 and 4 clock ticks after the preceding occurrence of the sequence. This is illustrated in [Figure 1-33 on page 1-72](#).

Figure 1-33 Match without Restriction Succeeds



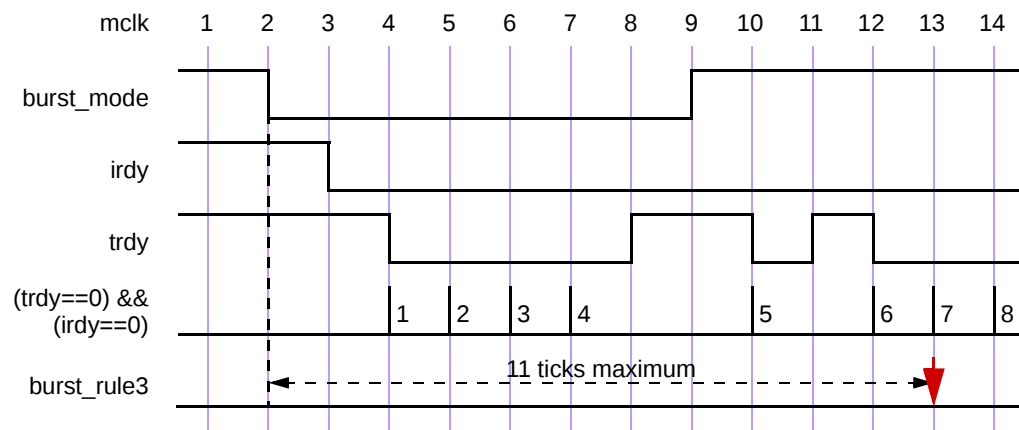
Two additional clock ticks delay the fifth repetition and one additional clock tick delays the sixth repetition as signal trdy becomes high to suspend the next data phase for two clock ticks and one clock tick respectively. The expression matches at clock tick 14.

If an additional constraint were placed on the expression as shown below, then the expression for checker burst_rule3 would not match at clock tick 13.

```
assert burst_rule3: check(burst3);
clock posedge mclk {
  event burst3:
    if (negedge burst_mode) then
      length [9..11]
      in ([1..4] ((trdy==0)&&(irdy==0))*[8]);
}
```

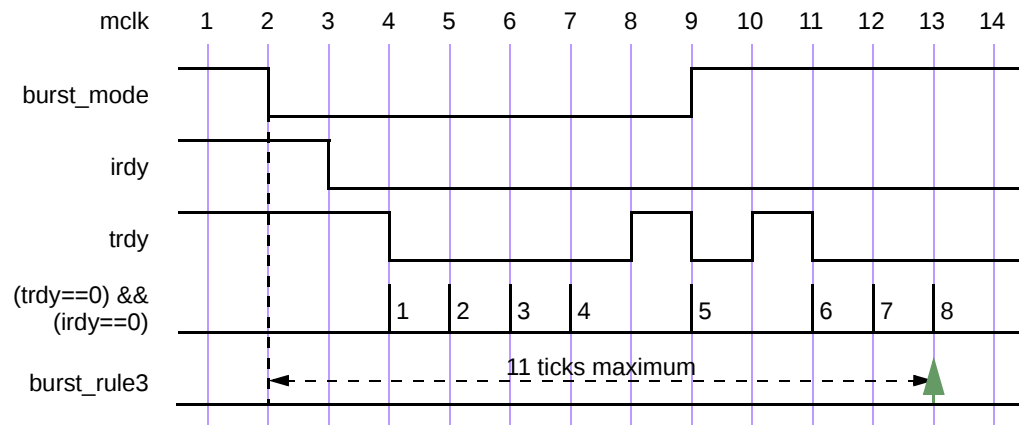
In the above expression, the total length of the entire repeated sequence must not be less than 9 clock ticks and not greater than 11 clock ticks. This restriction is expressed by `length [9..11]` in the expression. From [Figure 1-34 on page 1-73](#), the corresponding time to complete all repetitions is 12 clock ticks which exceeds the maximum allowed length, so the expression fails to match at clock tick 13.

Figure 1-34 Match with length-in Restriction Fails



The failure is corrected by reducing the delay for the fifth repetition from 2 clock ticks to 1 clock tick. This is shown in [Figure 1-35](#).

Figure 1-35 Match with length-in Restriction Succeeds



To express the constraints of a condition and a time period on the same sequence, the two constraint clauses are specified separated with a comma as shown below.

```

assert burst_rule4: check(burst4);
clock posedge mclk {
    event burst4:
        if (negedge burst_mode) then
            (isttrue(!burst_mode), length [9..11])
            in ([1..4] ((trdy==0)&&(irdy==0))*[8]);
}

```

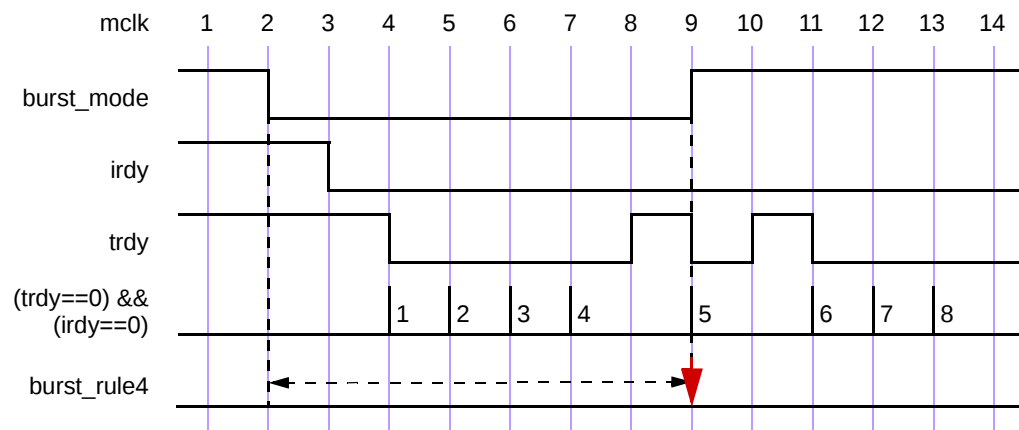
Now the two constraints are:

- `isttrue(!burst_mode)` to ensure that signal `burst_mode` remains low
- `length [9..11]` to ensure that the sequence takes at least 9 clock ticks and completes in at most 11 clock ticks

Both constraints must hold for the assertion to succeed, i.e, signal `burst_mode` must remain low throughout the allowed period for the sequence.

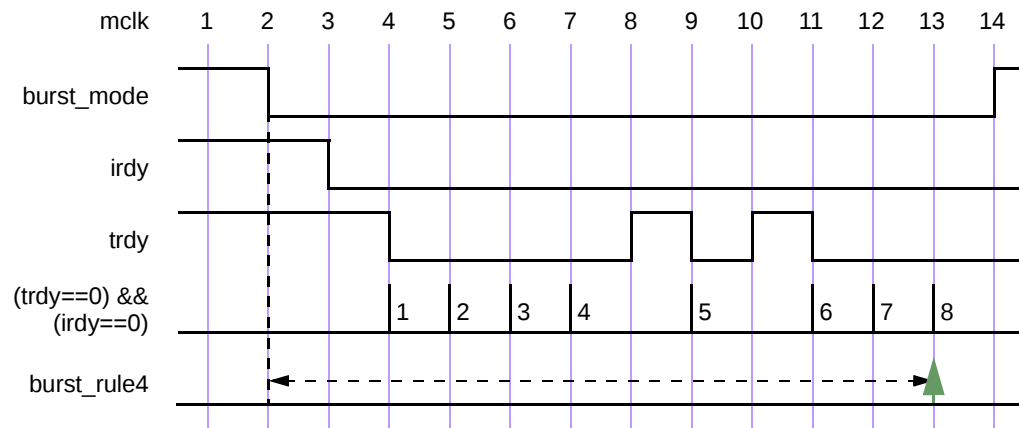
The expression for `burst_rule4` fails to match in [Figure 1-36 on page 1-75](#) because signal `burst_mode` becomes high at clock tick 9.

Figure 1-36 Match with Two Restrictions Fails



The failure is corrected by maintaining signal `burst_mode` to low value throughout the sequence as shown in [Figure 1-37 on page 1-76](#). The expression matches at clock tick 13 as it satisfies both constraints on the sequence: the total time period for the sequence is 12 clock ticks that is within the time period requirement, and signal `burst_mode` is held low throughout these 12 clock ticks.

Figure 1-37 Match with Two Restrictions Succeeds



Specifying Unconditional Number of Clock Ticks

The syntax for specifying an unconditional number of clock ticks is as follows:

`any`

The `any` specification returns true every time it is evaluated in an expression. `any` is most often used in an expression to extend a sequence by appending an unconditional delay, such as:

`te1 #t1 any`

In the above specification, sequence expression `te1` is extended by time `t1+1`. The entire expression completes on the `t1` clock tick after `te1` completes.

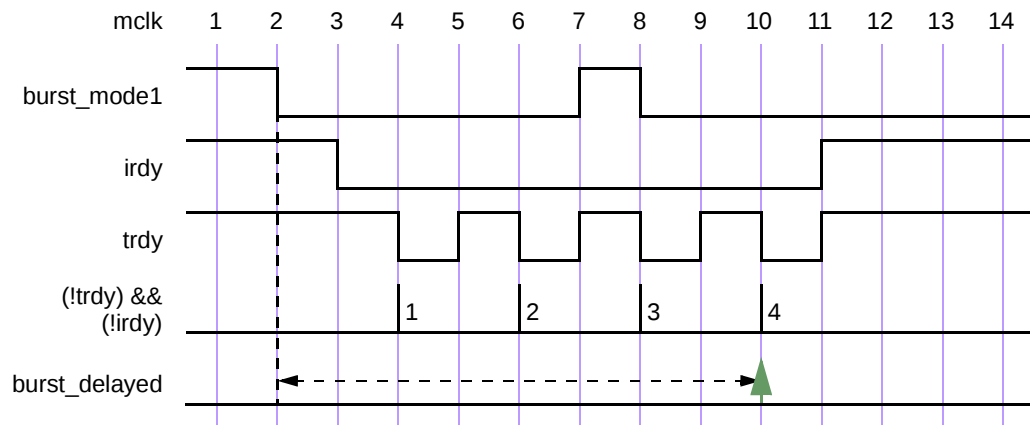
Let us consider an example of a burst mode transaction, where there are back to back repeated operations. [Figure 1-38](#) shows such an example with the assertion:

```

assert burst_delayed: check(burst_d);
clock posedge mclk {
    event burst_d:
        if (negedge burst_model) then
            #2 (((!trdy && !irdy) #1 any) * [4] );
}

```

Figure 1-38 Using any



In the above burst mode, a sequence representing the operation is repeated four times. The repeated sequence is:

```

((!trdy && !irdy) #1 any) #1 ((!trdy && !irdy) #1 any)
#1((!trdy && !irdy) #1 any) #1 ((!trdy && !irdy) #1 any)

```

No expectation is placed on any signal at the clock tick where any is monitored. By using any at the tail of each repetition, the operation is extended by an additional clock tick. In [Figure 1-38](#), the burst mode starts at clock tick 2 when signal burst_model becomes low. First repetition is satisfied 2 clock ticks. Between each repetition 2 clock ticks are expected, and satisfied accordingly at clock ticks 4, 6, 8, and 10. The expression matches at clock tick 10.

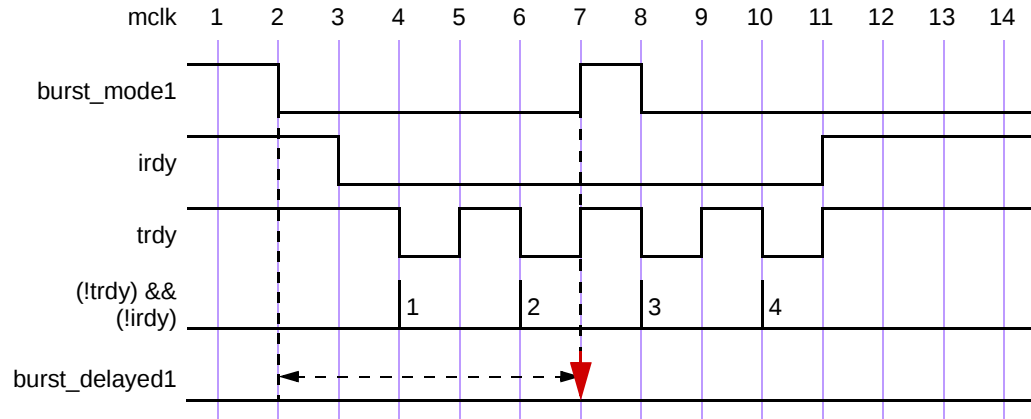
As seen from the previous example, **any** evaluates to true in an expression. However, if there was a constraint placed on a sequence to hold a condition true using the **istru** in clause, then that condition

must evaluate to true for **any** clause also. For example, the previous example now is modified to maintain signal `burst_mode1` to low during the entire sequence.

```
assert burst_delayed1: check(burst_d1);
clock posedge mclk {
  event burst_d1:
    if (negedge burst_mode1) then
      istrue (!burst_mode1)
      in (#2 (((!trdy && !irdy) #1 any) * [4]));
}
```

Figure 1-39 illustrates that the expression fails because signal `burst_mode1` becomes high at the time **any** is evaluated at clock tick 7. **istrue** requires that the condition `(!burst_mode1)` be maintained true at every clock tick during the time sequence `(#2 (((!trdy && !irdy) #1 any) * [4]))` is evaluated.

Figure 1-39 Using any with an istrue-in Restriction



Manipulating and Checking Data

The syntax for the constructs used in manipulating and storing data is as follows:

```
var [[int:int]] name [[int:int]];
init var_name_ref = bit_vector_expr;
var_name_ref <= bit_vector_expr;
assign var_name_ref = bit_vector_expr;
```

This section describes how to incorporate specification of properties that require either temporary storage or accumulation and manipulation of specific values to be associated at different times during a sequence. These features greatly simplify data checking along with the temporal relationships between design objects. Like in Verilog, a variable is declared and assigned with the capability of examining the assigned values at any point during a sequence.

A one- or two-dimensional variable is declared as follows:

```
var [[int:int]] name [[int:int]];
```

The syntax follows the syntactic rules of Verilog for the declaration of registers and memories. Also, the rules of scoping and qualification for the name are identical to any Verilog object name. Effectively, the declared variable can be accessed in the expressions in the same way as if it was declared in the corresponding Verilog module. If neither dimension is specified, the width is assumed to be one bit.

There is, however, one deviation from Verilog. The variables can be declared either for a module or for a scope. In the case of a variable declaration under a module, a separate copy of the variable is associated with each instance of the module. For the case of a variable declaration under a scope, the variable gets bound to only that instance and is not created for any other instance of its module.

By default, the value of this variable is initialized at the beginning of simulation to unknown. An initialization statement can be used to override the unknown value with the value of an expression as below:

```
init var_name_ref = bit_vector_expr;
```

The `var_name_ref` references a variable name with optional bit-select, part-select or word-select with the same rules as Verilog. ***init*** statement is executed only once at the beginning of simulation. All design variable values are considered unknown (value x) when evaluating the bit vector expression.

The value of the variable can be updated at every clock tick by using a non-blocking assignment statement.

```
var_name_ref <= bit_vector_expr;
```

This statement must be placed under the clock to which its execution is based upon. The clock tick triggers the assignment in the order as follows:

1. Evaluate ***event*** expressions.
2. Evaluate right-hand side of the assignment (`bit_vector_expr`).
3. Update the value of the variable on the left-hand side.

When the clock tick occurs, the statement is executed by evaluating the expression `bit_vector_expr` and placing the result in the variable. The updated value is available for the next clock tick. There can be only one assignment per variable.

Even though the assignment takes place under a specific clock, its value can be used in any event expression of another clock, just like a design variable. The name of the variable also follows the rules as if it was declared for the corresponding instance in Verilog.

Below is an example of using variables. The problem describes validating the number of words written for a block write command. The number of words is specified by signal `w_size` which can vary from 1 to 32. Each word written is specified by signal `w_start` and the end of block write is indicated by signal `w_end`. Signal `write` begins the command.

```
clock posedge sysclk {
    var[4:0] n;
    init n = 0;
    n <= (posedge write) ? w_size : n - matched(detect);
    event occur(e): (e || (!e * [1..] #1 e));
    event detect: posedge w_start;
    event prop_w: if (posedge write) then
        (istruce(!w_end) in #1 occur(n==0)) #1 w_end;
}
assert block_write_rule: check(prop_w);
```

Another form of assignment statement, known as continuous assignment in Verilog, is provided to model the assignments similar to Verilog.

```
assign var_name_ref = bit_vector_expr;
```

The ***assign*** statement allows asynchronous updates to the variable on the left hand side of the assignment. In contrast to the non-blocking assignment, this statement must not be specified under any clock. It assumes the simulation tick as the clock, and updates the value of the variable any time the bit vector expression on the right hand side changes its value. The rules of the ***assign*** statement follow the rules for continuous assignment in Verilog.

In addition to accessing values of signals at the time of evaluation of a boolean expression, the past values can be accessed with the ***past*** function.

```
past(bit_vector_expr [, number_of_ticks])
```

The argument `number_of_ticks` specifies the number of clock ticks in the past. If `number_of_ticks` is not specified, then it defaults to 1. ***past*** returns the value of the expression `bit_vector_expr` that was present `number_of_ticks` prior to the time of evaluation of ***past***.

For accessing values of signals one clock tick ahead, ***future*** can be used as:

```
future(bit_vector_expr)
```

future returns the value of `bit_vector_expr` that would be present in the next clock tick.

Another useful function provided for the boolean expression is ***count***, to count the number of 1s in a bit vector expression.

```
count(bit_vector_expr)
```

Grouping Assertions as a Library

The syntax for library groupings is as follows:

```
template name [(formal_param1, ..., formal_paramN)] :  
{  
    template_body  
}
```

With `formal_param` being:

```
name [= boolean_expr | sequence_expr]
```

This section describes how to group statements to construct a library of assertions and expressions. Such a group is called **template** which is given a name and can be instantiated with parameters. When instantiated with parameters, the parameters provide the binding to the actual design objects or other definitions specified elsewhere in the description. A **template** has the following syntax:

```
template name [(formal_param1, ..., formal_paramN)] :  
{  
    template_body  
}
```

A formal parameter is used to replace a name in the template body. A formal parameter can be an identifier representing one of the following:

- a **bool** name
- an **event** name
- an integer constant
- a bit vector
- a boolean or sequence expression

The default values for a formal parameter can be specified by using an equal sign with the left-hand side of the equal sign as the formal parameter name and right-hand side as the default value. For example,

```
template ova_hold(exp, min = 0, max = 15, clk):{  
    clock posedge clk {  
        event ova_e_hold:  
            (past(exp)==exp) * [min..max];  
    }  
}
```

The body of the template may contain:

- ***assert*** statements
- clocked or unclocked expression definitions ***event*** and ***bool***
- clocked or unclocked variable declarations and assignments to variables

A ***template*** is instantiated with the following syntax:

```
name [ins_name] [(actual_param1, ..., actual_paramN)];
```

The template instance name is optional. When the name is not specified, the name is the global sequence number of the instance in the form *tseq_number*. For example, the first template instance compiled would be assigned the name *t1*.

As template instances are expanded, the names of expression definitions and variables declared in the template body are constructed by appending the definition name with the template instance name and an underscore character. Such an expansion of a name uniquely identifies its definition. The following example illustrates the name expansion of definitions.

```
template range():{
    clock posedge clk2 {
        bool c1: enable;
        event crange_en: if (c1) then (minval <= expr);
    }
    assert range_chk: check(crange_en);
}
scope test {
    range t1();
    range t2();
    assert term_chk: if (t1_c1) then p_low #1 p_end;
}
```

The definitions `c1`, `crange_en`, and `range_chk` are expanded as shown below.

```
scope test {
  clock posedge clk2 {
    bool t1_c1: enable;
    event t1_crange_en: if (c1) then
      (minval <= expr);
  }
  assert t1_range_chk: check(crange_en);
  clock posedge clk2 {
    bool t2_c1: enable;
    event t2_crange_en: if (c1) then
      (minval <= expr);
  }
  assert t2_range_chk: check(crange_en);
  assert term_chk: if (t1_c1) then p_low #1 p_end;
}
```

Using this naming scheme, an expression defined within a template can be referenced outside the template as shown above in the definition of `assert term_chk`.

The actual parameters may not resolve all signals specified within the template. When the template is instantiated, the parameters and the unresolved signals get bound to the design objects.

If a formal parameter is specified with a default value in the template definition, then the corresponding actual parameter may be optionally omitted. In the example below, the formal parameter `max` is not supplied when the template is instantiated.

```

template ova_hold(exp, min = 0, max = 15, clk):{
    clock posedge clk {
        event ova_e_hold:
            (past(exp) == exp) * [min..max];
    }
}
scope test {
    ova_hold hold_instance(s, 5, , posedge clk);
}

```

If the default parameter value is not declared in the template definition, omission of the corresponding actual parameter value in the template instantiation will result in an error.

An example of a template is presented below to check data consistency during a transaction. Data `data_in` is latched at the occurrence of event `pre` in variable `data_store`. The latched data is checked when the last event `post` occurs against `data_out`. It is expected that the `data_out` at the time of the last event of the transaction must be equal to the data at the beginning of the transaction.

```

template data_check(pre, data_in, size, post,
    data_out, clk_expr): {
    clock clk_expr {
        var[size-1:0] data_store;
        data_store <= matched e1 ? data_in : data_store;
        event e1: pre;
        event e2: post;
        event consistent: if (matched e1) then
            #1 e2 #0 (data_out == data_store);
    }
    assert data_consistency: check(consistent);
}

bool rising: posedge sysclk;
event trans_begin: pck_init #1 pck * [8];
event trans_end: pck_trfr * [32] #1 pck_term;
data_check(trans_begin, id_in, 8, trans_end, id_out,
    rising);

```

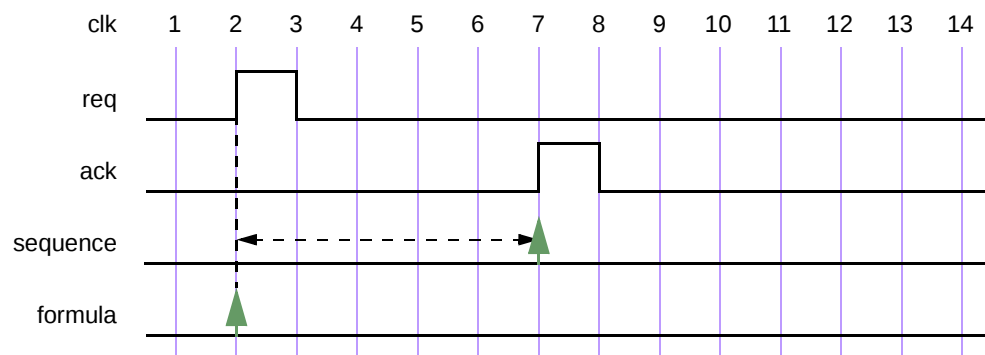
What Is a Formula

A formula introduces a powerful notion of specifying a behavior that results in true or false at a point in time. A formula can be a:

- boolean expression
- sequence expression
- combination of other formulas

Embedded in a formula, there can be references to values in the future. For instance a transaction that is described as “if there is a request, it must be followed by an acknowledge in four cycles” is a formula that results in true or false at any point in time where the formula is applied. Unlike a sequence which results in a starting point and an end point in time, a formula only has a single point in time. In this case, the formula would be either true or false at the time of request, while the sequence has a starting point at the time of request and an end point at the time of acknowledge. This is illustrated in the figure below.

Figure 1-40 Formula vs. Sequence Start and End Times

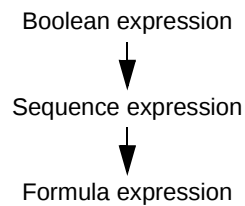


The usefulness of a formula becomes evident when formulas are combined to determine a complex behavior. To illustrate a simple combination of formulas, the behavior of the transaction described in

the figure is extended as, “if the transaction fails then the retry must succeed in six cycles”. Notice that there are two temporal behaviors in this statement: one, that states the failure of transaction in four cycles, and two, the retry in 6 cycles. The first behavior implies the second behavior.

Semantically, there is a hierarchy of expressions as shown in the figure below.

Figure 1-41 Hierarchy of Expressions



- A boolean expression is the simplest form of expression.
- A boolean expression is a sequence expression, but not vice versa. A sequence expression is built on top of boolean expressions.
- A sequence expression is a formula expression, but not vice versa. A formula expression is built on top of sequence expressions.

A key difference between a formula and a sequence expression is that the formula can address events in infinite time, while the sequence expression can only be applied to a finite simulation trace of values. For example, “acknowledge must eventually occur”. This difference is important for formal verification tools that statically check all possibilities of occurrence of events.

Defining Formulas

The syntax for defining expressions follows:

```
formula name [(param1, ..., paramN)] : formula_expr ;
```

A ***formula*** clause declares a formula expression with an identifier to name the expression. Optionally, a list of parameters, separated by commas, can be declared for the expression, in which case, the parameters supplied at the time of instantiation replace the corresponding variables.

Writing Simple Formula Expressions

A formula can be composed of one or more formulas using familiar boolean operators such as && and ||. This section describes each operator allowed to combine formula expressions.

The syntax for specifying composite sequences is as follows:

```
formula_expr1 && formula_expr2  
formula_expr1 || formula_expr2  
!formula_expr  
formula_expr1 implies formula_expr2  
formula_expr1 iff formula_expr2
```

Each operand is a formula, and can be composed of any number of arbitrarily complex formula expressions. The evaluation of a formula expression results in true or false. That is why the semantic interpretation of these operators in the context of operands that are formula expressions is the same as when the context is boolean expression operands.

The binary operator **&&** is used when both operand expressions are expected to be true. The result of such an expression is true when both operands are true at the point in time where the expression is evaluated. Otherwise, the result is false.

```
formula f1:
    (in_val implies l_out) && (in_t1 implies l1_out);
```

In the above example, formula f1 will be true if operand formula expressions (in_val implies l_out) and (in_t1 implies l1_out) are true.

The binary operator **||** is used when at least one of the operand expressions is expected to be true. The result of such an expression is true when either one operand or both operands are true. If both operands are false, then the result is false.

```
formula f2:
    (l2_read implies l1_miss) || (l1_hit iff data_out);
```

In the above example, formula f2 will be true if either operand formula expression (l2_read implies l1_miss) or (l1_hit iff data_out) is true. The result is also true if both operands are true.

The unary operator **!** negates the result of its operand formula expression. If the operand formula expression is true, then the result is false. Otherwise, the result is true.

```
formula f3: !f2;
```

Here, formula f3 is true if formula f2 is false, and vice versa.

The binary operator ***implies*** is the implication operator. If the left hand operand formula_expr1 is true, then the right hand operand formula_expr2 must also be true. If formula_expr1 is false, then the result is true.

```
formula f4: l2_read implies l1_miss;
```

In the above example, if `l2_read` is true, then `l1_miss` must also be true. Please note that formula expressions, `l2_read` and `l1_miss`, may themselves be previously defined as other complex formulas.

The binary operator ***iff*** is the equivalence operator. If the left hand operand `formula_expr1` is true, then the right hand operand `formula_expr2` must also be true. If `formula_expr1` is false, then `formula_expr2` must also be false.

```
formula f4: f1 iff f2;
```

Formula `f4` is true when one of the following conditions holds. Otherwise, `f4` results in false.

- `f1` is true and `f2` is true.
- `f1` is false and `f2` is false.

Specifying Temporality Using Formulas

A formula can refer to a value in the future. This reference can be made for a specific time, a range of time, or indefinite time. The constructs used to express such temporality are taken from Linear Temporal Logic (LTL). LTL is a well-defined mathematical formalism for reasoning about dynamically changing quantities. Within the LTL temporal framework, there are four basic operators:

- *Always p*: formula `p` is always true.
- *Eventually p*: formula `p` will eventually be true.
- *Next p*: formula `p` will be true the next clock tick.

- *p Until q*: formula p will be true until formula q is true.

These operators have been extended with the capability of specifying a time range so that the operators are applied only during the specified range of time. Also, a “weaker” version of the *Next* and *Until* operators are provided. The OVA versions of these operators are described in this section.

Globally Operator

The syntax for ***globally*** operator is:

```
globally [ [int .. int] | [int .. ] ] formula_expr
```

Operator ***globally*** has the same meaning as the LTL operator *Always*. The *formula_expr* must be true at every clock tick. If an interval of time is specified, then *formula_expr* must be true within the range of specified time. Examples of time intervals are shown below (assume that the formula is evaluated at clock tick 0):

- `globally [5..10] formula_expr`

formula_expr must be true at every clock tick in the period starting at clock tick 5 and ending at clock tick 10.

- `globally [2..] formula_expr`

formula_expr must be true at clock tick 2 and every clock tick thereafter.

```
clock posedge mclk {
    event data_end :
        if (data_phase) then
            ((irdy==0) && (negedge trdy || negedge stop));
        formula enable_rule: globally data_end;
}
```

In this example, event `data_end` must be true at every clock tick.

eventually Operator

The syntax for the ***eventually*** operator is:

```
eventually [ [int .. int] | [int .. ] ] formula_expr
```

Operator ***eventually*** has the same meaning as the LTL operator *Eventually*. The *formula_expr* must be true now or sometime in the future. If an interval of time is specified, then *formula_expr* must be true at least once during the range of specified time. Examples of time intervals are shown below (assume that the formula is evaluated at clock tick 0):

- `eventually [5..10] formula_expr`

formula_expr must be true at least once in the period starting at clock tick 5 and ending at clock tick 10.

- `eventually [2..] formula_expr`

formula_expr must be true at least once at clock tick 2 or thereafter.

```
clock posedge sysclk {  
    formula trdy_rule:  
        (negedge frame) implies  
            eventually [2..] negedge trdy;  
}
```

In this example, if `(negedge frame)` is true, then `(negedge trdy)` must be true sometime after clock tick 1.

next Operator

The syntax for **next** and **wnext** operator is:

```
next [int] formula_expr
```

Operator **next** has the same meaning as the LTL operator *Next*. The *formula_expr* must be true at the next clock tick. If a specific time is specified, then *formula_expr* must be true that clock tick.

Operator **wnext** is the “weaker” version of **next**. By “weaker” version, it is meant that if the clock ticks do not occur sufficiently to move to the time where *formula_expr* is to be evaluated, the formula is considered true. Conversely, for operator **next**, the formula is considered false under that situation.

```
next [5] formula_expr
```

Assuming that the formula is evaluated at clock tick 0, *formula_expr* must be true at clock tick 5.

```
clock posedge mclk {  
    formula irdy_rule:  
        posedge frame implies next posedge irdy;  
}
```

In this example, if `posedge frame` is true, then `negedge irdy` must be true at the next clock tick.

```
clock posedge mclk {  
    formula irdy_rule:  
        posedge frame implies wnext posedge irdy;  
}
```

In the above example, **next** is replaced by **wnext**. In this case, if the clock `posedge mclk` does not occur, then formula `irdy_rule` is still considered true.

until Operator

The syntax for **until** and **wuntil** operator is:

```
formula_expr1 until [ [int .. int] | [int .. ] ]  
                formula_expr2
```

Operator **until** has the same meaning as the LTL operator *Until*. The *formula_expr1* must be true at every clock tick until the clock tick when *formula_expr2* is true. Please note that:

- *formula_expr2* is required to be true some time in the future.
- *formula_expr1* need not be true at the clock tick when *formula_expr2* is true.

Operator **wuntil** is the “weaker” version of **until**. For **wuntil**, *formula_expr2* is not required to become true in the future. Under this condition, *formula_expr1* must remain true for all clock ticks in the future.

A time range can be specified with these operators as shown below (assume that the formula is evaluated at clock tick 0).

```
formula_expr1 until [5..10] formula_expr2
```

- Operator until is applied between clock tick 5 and 10.
- If *formula_expr2* is true at clock tick 5, then the formula is true, regardless of the value of *formula_expr1*.

- If *formula_expr2* is false at clock tick 5, then it must become true at or before clock tick 10. *formula_expr1* must be true at least for all clock ticks before *formula_expr2* becomes true.

```
formula_expr1 wuntil [2..] formula_expr2
```

- Operator *wuntil* is applied starting at clock tick 2 and thereafter.
- If *formula_expr2* is true at clock tick 2, then the formula is true, regardless of the value of *formula_expr1*.
- If *formula_expr2* is false at clock tick 2 and becomes true at some time later, then *formula_expr1* must be true at least for all clock ticks before *formula_expr2* becomes true.
- If *formula_expr2* is false at clock tick 2 and remains false forever, then *formula_expr1* must be true for all clock ticks starting at clock tick 2.

```
clock posedge sysclk{
    formula stop_rule:
        stop_a implies stop_a until frame_a;
}
```

In the above example, when signal *stop_a* is asserted, it must remain asserted until signal *frame_a* is asserted.

Using Sequences as Conditions to Formulas

When a temporal property is to be expressed, in some cases, writing a sequence expression is easier than writing an equivalent formula. A combination of a sequence expression and a formula expression enhances readability as well as provide a concise way to express properties in the most natural way.

Two constructs are provided to allow using a sequence expression as a prefix to a formula. These constructs are ***followed_by*** and ***triggers***.

The syntax for the ***followed_by*** and ***triggers*** operators is:

```
sequence_expr followed_by formula_expr
sequence_expr triggers formula_expr
```

For both constructs, a sequence expression is evaluated, and based on its results, a formula is evaluated at the next clock tick associated with the formula. Please note that the clocks associated with the sequence expression and formula expression may be different.

For operator ***followed_by***:

- *sequence_expr* must result in at least one match. If there is no match, the formula is false.
- *formula_expr* is evaluated after every match.
formula_expr must result in true for at least one match. Otherwise, the formula is false.

For operator ***triggers***:

- *sequence_expr* need not result in a match. If there is no match, the formula is true.
- When *sequence_expr* results in one or more matches, *formula_expr* is evaluated for every match. *formula_expr* must result in true for all matches. Otherwise, the formula is false.

```
clock posedge sysclk{
    formula ack_rule:
        (req #3 gnt) followed_by (eventually ack);
}
```

In the above example, sequence expression (`req #3 gnt`) must result in a match. At the clock tick after signal `gnt` is true, formula expression (`eventually ack`) must be true.

```
clock posedge sysclk{
    formula proc_rule:
        (proc_start #[10..100] pkg_out) triggers
        (eventually proc_stop);
}
```

In this example, (`proc_start #[10..100] pkg_out`) is a sequence expression, in which signal `proc_start` initiates processing that can result in `pkg_out` being asserted multiple times between clock tick 10 and 100. For each asserted `pkg_out`, signal `proc_stop` must be true some time later.

Specifying Reset Conditions

As reset is a common hardware activity, two constructs are provided to directly express reset situations. The syntax of these constructs is:

```
accept boolean_expr in formula_expr
reject boolean_expr in formula_expr
```

The basic notion behind the reset conditions is that the formula should be checked only until the arrival of a reset signal. For both constructs, the value of *boolean_expr* is evaluated at every clock tick during the evaluation period of *formula_expr*. If *boolean_expr* evaluates to true, then the formula evaluation is terminated. For the case of `accept`, the formula is considered to be true, while for `reject` the formula is considered to be false.

An important point to note is that *boolean_expr* is evaluated at the most basic unit of time. (For simulation, the basic unit of time is the simulation tick), while the formula is evaluated at its associated clock. Thus, the reset detection is asynchronous with respect to the formula clock.

To synchronize reset detection with the clock ticks of the formula, the keyword **clock** must be ANDed with *boolean_expr*, as shown below.

```
accept (boolean_expr & clock) in formula_expr
reject (boolean_expr & clock) in formula_expr

clock mclk {
    formula burst4:
        (negedge burst_mode) implies transfer until t_end;
    formula burst4_r: reject (sys_reset) in burst4;
}
```

In the above example, formula *burst4* can take several clock ticks to complete. During this time, if *sys_reset* arrives then the formula *burst4_r* is false asynchronously as soon as *sys_reset* becomes true. To modify the behavior of formula *burst4_r* to being synchronous, the description is modified as shown below.

```
clock mclk {
    formula burst4:
        (negedge burst_mode) implies transfer until t_end;
    formula burst4_r:
        reject (sys_reset & clock) in burst4;
}
```

Here, the reset behavior due to signal *sys_reset* is synchronized with the clock *mclk*.

Directives for Formal Verification

For specifying a formula as an assertion, the ***assert*** directive is used. Please note that the syntax for the ***assert*** directive is same as for the sequence expressions as described in [“Specifying Temporal Assertions” on page 1-34](#).

```
assert formula_name;  
    | name: formula_expr;  
    | name: formula_name;
```

For example,

```
assert burst4_r;  
  
clock mclk {  
    formula burst4:  
        (negedge burst_mode) implies transfer until t_end;  
    formula burst4_r: reject (sys_reset) in burst4;  
}
```

When OVA is used for formal verification, additional directives are provided to support description of the environment of the design that is being verified. These directives are:

- ***assume*** to specify assumptions
- ***restrict*** to specify restrictions
- ***model*** to model the assumptions

The syntax for specifying these directives is similar to the ***assert*** directive as shown below.

```
assume formula_name;  
    | name: formula_expr;  
    | name: formula_name;
```

```

restrict formula_name;
    | name: formula_expr;
    | name: formula_name;

model formula_name;
    | name: formula_expr;
    | name: formula_name;

```

The ***assume*** directive is used when the formula expression is applied as a constraint for the environment to the design under test. It is expected that this assumption is formally verified with a proof.

The ***restrict*** directive is similar to the ***assume*** directive, except that the formula expression specified as the restriction is assumed without any obligation to prove that it holds for the environment. This directive is used primarily to deal with the capacity limitations of the formal verification methods.

To assist the formal verification procedure, formula expressions are often used as models to supplement the design models under verification. The ***model*** directive is used to specify such behaviors.

2

The OVA Engine API

This chapter describes the OVA Engine API. The API provides a convenient way for external modules to control the OVA Engine, providing flexibility and modularization to the OVA environment without the sacrifice of efficiency.

- [General Requirements](#)
- [The Use Model](#)
- [The API](#)
- [Notes](#)

General Requirements

External modules require means of two-way communication with the OVA Engine in both synchronous and asynchronous modes. Synchronous mode is required for interactive modules (GUI, debugger, scripting, etc.) while asynchronous mode is used by batch modules (report generators and such). The API is designed to minimize the latency of communication between the OVA Engine and its clients, retaining enough control over the OVA Engine operation. The API is easily implemented in diverse communication environments from C-level API to IPC.

The Use Model

The event-driven use-model is utilized by this API. The client registers itself with the engine and receives its unique ID to identify itself to the engine. Then the client subscribes to be notified of the events it is interested in receiving. The client may influence the engine's behavior by issuing commands.



Two basic types of events are defined to enable the communication:

- *Event* - This type of event is used by the engine to notify its clients about the changes in the engine environment state. See [“OvaEngEvent Group” on page 2-11](#) and [“OvaAssertEvent Group” on page 2-12](#) for more information.
- *Action (Command)* - This type of event is used by the clients to control the engine operation. See [“OvaEngAction Group” on page 2-12](#) and [“OvaAssertAction Group” on page 2-13](#) for more information.

In addition to event-driven communication, the engine provides iterator-based access to the assertions data. The client may iterate over assertions, assertion validation attempts, and request all relevant data.

The API

The OVA Engine Data Types/Constants

Ova_ClientID - Client identifier type that is ensured to be unique in the OVA Engine environment.

- **OVA_CLIENTID_NULL** - Null value equivalent for Ova_ClientID.

Ova_AssertID - Assertion identifier type that is ensured to be unique in the OVA Engine environment.

- **OVA_ASSERTID_NULL** - Null value equivalent for Ova_AssertID.

Ova_AssertAttemptID - Assertion attempt identifier type that is ensured to be unique in the OVA Engine environment.

- **OVA_ASSERTATTEMPTID_NULL** - Null value equivalent for Ova_AssertAttemptID.

Ova_AssertClockID - Assertion clock expression identifier type that is ensured to be unique in the OVA Engine environment.

- **OVA_ASSERTCLOCKID_NULL** - Null value equivalent for Ova_AssertClockID.

Ova_AssertName - Assertion name type.

Ova_String - The OVA string representation.

Ova_SrcFileBlock - Source file reference block type. Has the following fields:

- **fileName** - Name of the OVA source file.
- **startRow** - Definition start row in the OVA source file.
- **startColumn** - Definition start column in the OVA source file.
- **endRow** - Definition end row in the OVA source file.
- **endColumn** - Definition end column in the OVA source file.

Ova_EngEvent - The OVA Engine state change event type. See [“OvaEngEvent Group” on page 2-11.](#)

- **OVA_ENGEVENT_ALL** - Alias to the set of all event types in Ova_EngEvent group.

Ova_EngAction - The OVA Engine action event type. See [“OvaEngAction Group” on page 2-12.](#)

Ova_AssertEvent - The assertion state change event type. See [“OvaAssertEvent Group” on page 2-12.](#)

- **OVA_ASSETEVENT_ALL** - Alias to the set of all event types in Ova_AssertEventgroup.

Ova_AssertAction - The assertion action event type. See [“OvaAssertAction Group” on page 2-13.](#)

Ova_EngCallback - The OVA Engine event callback function type. See [“The OVA Engine Client Interface” on page 2-10.](#)

Ova_AssertCallback - The OVA Assertion event callback function type. See [“The OVA Engine Client Interface” on page 2-10.](#)

Ova_UserData - User data type. Used as optional argument for callback functions. See [“The OVA Engine Client Interface” on page 2-10](#).

- **OVA_USERDATA_NULL** - Null value of Ova_UserData.

Ova_Time - Character string.

Ova_Mode - The OVA Engine operation mode.

- **OVA_MODE_SYNC** - Blocking (synchronous) mode of operation.

Ova_Bool - Boolean type.

- **OVA_TRUE** - Boolean true.
- **OVA_FALSE** - Boolean false.

Ova_ExprType - Assertion expression type.

- **OVA_OVA_EXPR_TYPE** - Type of the assertion expression is “*ova*”.
- **OVA_CHECK_EXPR_TYPE** - Type of the assertion expression is “*check*”.
- **OVA_FORBID_EXPR_TYPE** - Type of the assertion expression is “*forbid*”.

Ova_AssertSyntaxInfo - Assertion syntax information type. Has two fields:

- **Ova_AssertName name** - Assertion name.
- **Ova_ExprType exprType** - Assertion expression type.
- **Ova_SrcFileBlock srcBlock** - OVA source file reference block.
- **Ova_AssertSyntaxInfoNull** - Null value of Ova_AssertSyntaxInfo.

Ova_AssertAttemptSyntaxInfo - Assertion syntax information type. Has one field:

- **Ova_Time timestamp** - The attempt start time.
- **Ova_AssertAttemptSyntaxInfoNull** - Null value of **Ova_AssertAttemptSyntaxInfo**.

Ova_AssertClockSyntaxInfo - Assertion syntax information type. Has one field:

- **clockType** - Textual representation of clock type.
- **Ova_AssertClockSyntaxInfoNull** - Null value of **Ova_AssertClockSyntaxInfo**.

Ova_ConfigSwitch - The OVA Engine configuration switch. These are binary (true/false) switches that can be passed as command line switches to the simulator.

- **Ova_ShowLineInfoConfSwitch** - Show line info in messages. Default: *OVA_FALSE*.
- **Ova_Quiet0ConfSwitch** - Do not print any messages at runtime. Default: *OVA_FALSE*.
- **Ova_PrintReportConfSwitch** - Print report at the end of simulation. Default: *OVA_TRUE*.

Ova_ConfigOption - The OVA Engine configuration options. These are options that can be passed as command line options to the simulator. Currently no configuration options are supported.

The OVA Engine Interface

The OVA Engine should implement the following interface and expose it to the outside world.

Ova_ClientID ovaRegisterClient() - The OVA engine constructs new, unique ID for client to identify itself on the following requests.

Ova_Bool ovaSetMode(Ova_ClientID clientID, Ova_Mode modeID) - Set interaction with *clientID* to the operating mode *modeID*.

Ova_Bool ovaAddEngListener(Ova_ClientID clientID, Ova_EngEvent eventID, Ova_Callback ref, Ova_UserData udRef) - Notify *clientID* when state change *eventID* happens by calling *ref*. If *eventID* equals *OVA_ENGEVENT_ALL*, the client is notified of all events of *OvaEngEvent* type.

Ova_Bool ovaAddAssertListener(Ova_ClientID clientID, Ova_AssertEvent eventID, Ova_AssertID assertID, Ova_Callback ref, Ova_UserData udRef) - Notify *clientID* when assertion *eventID* happens by calling *ref*. If *eventID* equals *OVA_ASSERTEVENT_ALL*, the client is notified of all events of *OvaAssertEvent* type.

Ova_Bool ovaDoAction(Ova_ClientID clientID, Ova_EngAction eventID, Ova_UserData udRef) - Execute action *eventID* command.

Ova_Bool ovaAssertDoAction(Ova_ClientID clientID, Ova_AssertAction eventID, Ova_AssertID assertionID, Ova_AssertAttemptID attemptID, Ova_UserData udRef) - Perform action *eventID* for assertion or assertion attempt *assertionID*.

Ova_AssertID ovaGetAssertByName(Ova_ClientID clientID, Ova_AssertName name) - Get ID of the assertion with the name *name*. If no matching assertion is found, *OVA_ASSERTID_NULL* value is returned.

Ova_AssertID ovaFirstAssert (Ova_ClientID clientID) - Get ID of first assertion. In other words, reset assertion iterator of *clientID*. If no assertions are loaded into the engine, OVA_ASSERTID_NULL value will be returned.

Ova_AssertID ovaNextAssert (Ova_ClientID clientID) - Get ID of next assertion. In other words, advance assertion iterator of *clientID*. If there are no more assertions left or this client has not called ovaFirstAssert before ovaNextAssert was called, OVA_ASSERTID_NULL value will be returned.

Ova_AssertAttemptID ovaFirstAssertAttempt (Ova_ClientID clientID, Ova_AssertID assertionID) - Get ID of first attempt of the assertion *assertionID*. In other words, reset assertion attempts iterator for *clientID* over attempts for assertion *assertionID*. If no assertion evaluation attempts were started prior to the point when ovaFirstAssertAttempt function was called, OVA_ASSERTATTEMPTID_NULL value is returned.

Ova_AssertAttemptID ovaNextAssertAttempt (Ova_ClientID clientID, Ova_AssertID assertionID) - Get ID of next assertion attempt of *assertionID*. In other words, advance iterator of assertion *assertionID* attempts iterator of *clientID*. If no more assertion evaluation attempts are left or this client has not called ovaFirstAssertAttempt before ovaNextAssertAttempt was called, OVA_ASSERTATTEMPTID_NULL value is returned.

Ova_AssertClockID ovaGetAssertClock(Ova_ClientID clientID, Ova_AssertID assertionID) - Get ID of clock expression for assertion *assertionID*.

Ova_Bool ovaHasSyntaxInfo(Ova_ClientID clientID) - Returns OVA_TRUE if syntax information is available. Should be called before *ovaGetSyntaxInfo(id)* is called.

Ova_AssertSyntaxInfo ovaGetAssertSyntaxInfo(Ova_ClientID clientID, Ova_AssertID id) - Get syntax information for the assertion *id*.

Ova_AssertAttemptSyntaxInfo ovaGetAssertAttemptSyntaxInfo (Ova_ClientID clientID, Ova_AssertID assertID, Ova_AssertAttemptID attemptID) - Get syntax information for the assertion attempt *attemptID*.

Ova_AssertClockSyntaxInfo ovaGetAssertClockSyntaxInfo(Ova_ClientID clientID, Ova_AssertID assertID, Ova_AssertClockID id) - Get syntax information for the assertion clock *id*.

Ova_Bool ovaSetConfigSwitch(Ova_ClientID clientID, Ova_ConfigSwitch confSwitch, Ova_Bool enable) - Enable/Disable the OVA Engine runtime switch.

Ova_Bool ovaSetConfigOption(Ova_ClientID clientID, Ova_ConfigOption confSwitch, Ova_UserData udRef) - Set value of the OVA Engine configuration option.

The OVA Engine Client Interface

The OVA Engine clients should implement and register a callback function for each event type that the client is interested in receiving notifications for. The callback function signatures are as follows:

void ovaProcessStateEvent (Ova_EngEvent eventID, Ova_Time time, Ova_UserData udRef) - Function called when the OVA Engine state change event *eventID* occurs.

void ovaProcessAssertEvent (Ova_AssertEvent eventID, Ova_Time time, Ova_AssertID assertion, Ova_AssertAttemptID attempt, Ova_UserData udRef) - Function called when the OVA Engine assertion event *eventID* occurs.

OvaEngEvent Group

The events generated by the OVA Engine when its state changes.

OvaInitializeBeginEngE - Emitted by the OVA Engine before initialization.

OvaInitializeEndEngE - Emitted by the OVA Engine upon completion of initialization.

OvaStartEngE - Emitted by the OVA Engine at simstart but before evaluation attempts start.

OvaResetBeginEngE - Emitted by the OVA Engine at the beginning of the reset sequence.

OvaResetEndEngE - Emitted by the OVA Engine at the end of the reset sequence.

OvaLoadBeginEngE - Emitted by the OVA Engine at the beginning of execution of the load command.

OvaLoadEndEngE - Emitted by the OVA Engine upon completion of the load command.

OvaFinishEngE - Emitted by the OVA Engine after *simend*.

OvaTerminateBeginEngE - Emitted by the OVA Engine when it is about to terminate and before any data is destroyed.

OvaTerminateEndEngE - Emitted by the OVA Engine as a last signal before it exits.

OvaErrorEngE - Unrecoverable error. The engine will terminate. This event is issued to all clients regardless of whether the client registered to receive this event or not.

OvaEngAction Group

The action events accepted by the OVA Engine as commands altering its behavior.

OvaResetEngA - Reset all data.

OvaFinishEngA - Finish. Result is the same as if *simend* was encountered.

OvaTerminateEngA - Clean up and terminate.

OvaAssertEvent Group

The events emitted by the OVA Engine when the state of a particular assertion changes.

OvaResetAssertE - Reset assertion: terminate all evaluation attempts that are in progress.

OvaNewAttemptStartAssertE - New evaluation attempt started.

OvaAttemptRemovedAssertE - Evaluation attempt removed, for example after this attempt failure or after assertion has been removed from the environment.

OvaAttemptFailureAssertE - Assertion match attempt failed.

OvaAttemptSuccessAssertE - Assertion match attempt succeeded.

OvaDisableNewAttemptsAssertE - Generation of new evaluation attempts of particular assertion disabled.

OvaEnableNewAttemptsAssertE - Generation of new evaluation attempts of particular assertion enabled.

OvaAssertAction Group

The action events accepted by the OVA Engine to alter behavior of the assertion evaluation attempts.

OvaResetAssertA - Reset assertion. The reset assertion command has the following effect: all ongoing evaluation attempts are discarded and all accumulated unreported history of evaluations are discarded.

OvaDisableNewAttemptsAssertA - Do not make new attempts on the assertion. All the attempts started before this command was received continue to evaluate.

OvaEnableNewAttemptsAssertA - Cancel the effect of **OvaAssertDisableNewAttempts**.

OvaAttemptKillAssertA - Kill an evaluation attempt.

Notes

To be able to access functions described by this API, client code must call *ovaRegisterClient()* before any other API call. Such a transfer of control from engine to the client is implementation specific and as such beyond the scope of this document. C level implementation will leave this initial stage for the client code care entirely.

No data consistency is guaranteed if any of the OvaEngEvent group engine events happens while a client is iterating over assertions or assertion attempts. It means that once the client is notified of any event of the OvaEngEvent type, the result of ovaNextAssert() and ovaNextAssertAttempt() calls is undefined and the client should reset its iterators by calling ovaFirstAssert() or ovaFirstAssertAttempt().