OpenVera Language Reference Manual: Assertions

G-2012.09
September 2012

Comments?
E-mail your comments about this manual to:
vcs_support@synopsys.com.

Copyright Notice and Proprietary Information

Right to Copy Documentation

The license agreement with Synopsys permits licensee to make copies of the documentation for its internal use only. Each copy shall include all copyrights, trademarks, service marks, and proprietary rights notices, if any. Licensee must assign sequential numbers to all copies. These copies shall contain the following legend on the cover page:

> "This document is duplicated with the permission of Synopsys, Inc., for the exclusive use of _____ and its employees. This is copy number _____."

Destination Control Statement

All technical data contained in this publication is subject to the export control laws of the United States of America. Disclosure to nationals of other countries contrary to United States law is prohibited. It is the reader's responsibility to determine the applicable regulations and to comply with them.

Disclaimer

SYNOPSYS, INC., AND ITS LICENSORS MAKE NO WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, WITH REGARD TO THIS MATERIAL, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE.

Registered Trademarks (®)

Synopsys, AEON, AMPS, Astro, Behavior Extracting Synthesis Technology, Cadabra, CATS, Certify, CHIPit, CoMET, Confirma, CODE V, Design Compiler, DesignWare, EMBED-IT!, Formality, Galaxy Custom Designer, Global Synthesis, HAPS, HapsTrak, HDL Analyst, HSIM, HSPICE, Identify, Leda, LightTools, MAST, METeor, ModelTools, NanoSim, NOVeA, OpenVera, ORA, PathMill, Physical Compiler, PrimeTime, SCOPE, Simply Better Results, SiVL, SNUG, SolvNet, Sonic Focus, STAR Memory System, Syndicated, Synplicity, the Synplicity logo, Synplify, Synplify Pro, Synthesis Constraints Optimization Environment, TetraMAX, UMRBus, VCS, Vera, and YIELDirector are registered trademarks of Synopsys, Inc.

Trademarks (™)

AFGen, Apollo, ARC, ASAP, Astro-Rail, Astro-Xtalk, Aurora, AvanWaves, BEST, Columbia, Columbia-CE, Cosmos, CosmosLE, CosmosScope, CRITIC, CustomExplorer, CustomSim, DC Expert, DC Professional, DC Ultra, Design Analyzer, Design Vision, DesignerHDL, DesignPower, DFTMAX, Direct Silicon Access, Discovery, Eclypse, Encore, EPIC, Galaxy, HANEX, HDL Compiler, Hercules, Hierarchical Optimization Technology, High-performance ASIC Prototyping System, HSIM^{plus}, i-Virtual Stepper, IICE, in-Sync, iN-Tandem, Intelli, Jupiter, Jupiter-DP, JupiterXT, JupiterXT-ASIC, Liberty, Libra-Passport, Library Compiler, Macro-PLUS, Magellan, Mars, Mars-Rail, Mars-Xtalk, Milkyway, ModelSource, Module Compiler, MultiPoint, ORAengineering, Physical Analyst, Planet, Planet-PL, Polaris, Power Compiler, Raphael, RippledMixer, Saturn, Scirocco, Scirocco-i, SiWare, Star-RCXT, Star-SimXT, StarRC, System Compiler, System Designer, Taurus, TotalRecall, TSUPREM-4, VCSi, VHDL Compiler, VMC, and Worksheet Buffer are trademarks of Synopsys, Inc.

Service Marks (^{sm})

MAP-in, SVP Café, and TAP-in are service marks of Synopsys, Inc.

SystemC is a trademark of the Open SystemC Initiative and is used under license.
ARM and AMBA are registered trademarks of ARM Limited.
Saber is a registered trademark of SabreMark Limited Partnership and is used under license.
All other product or company names may be trademarks of their respective owners.

Contents

1

Getting Started

This chapter introduces OpenVera Assertions.

In this chapter:

Introducing OpenVera Assertions

A big part of verifying digital electronic designs is testing that signals have the right values, in the right order, and at the right times. An interrupt signal must be followed by an acknowledgement within a

certain range of clock ticks. Outputs must go to specified values one clock cycle after the reset signal goes low. Only one of several states should be active at the same time.

With random value testing, you also want to know how many times various actions have occurred. In a PCI burst test, how many times did an abnormal termination occur? How many times were each of several states active? Was every type of control packet received while the buffer was full?

OpenVera Assertions (OVA) provides a clear, easy way to describe sequences of events and to test for their occurrence. With clear definitions and less code, testbench design is faster and easier. And you can be confident that you are testing the right sequences in the right way.

OVA is a declarative method that is much more concise and easier to read than the procedural descriptions provided by hardware description languages such as Verilog. With OpenVera Assertions:

- Descriptions can range from the most simple to the most complex logical and conditional combinations.
- Sequences can specify precise timing or a range of times.
- Descriptions can be associated with specified modules and module instances.
- Descriptions can be grouped as a library for repeated use.

How Sequences Are Tested

Testing starts with a *temporal assertion file*, which contains the descriptions of the sequences and instructions for how they should be tested. OpenVera Assertions is designed to resemble Verilog with similar data types, operators, and lexical conventions.

A typical temporal assertion file consists mostly of *temporal expressions*, which are the descriptions of the event sequences. *Events* are values or changes in value of any Verilog regs, integers, or nets. (Events declared in Verilog cannot be part of an expression.) Temporal expressions can be combined to form longer or more complex expressions. The language supports not only linear sequences but logical and conditional combinations.

The basic instruction for testing is a *temporal assertion*. Assertions specify an expression or combination of expressions to be tested. Assertions come in two forms: *check*, which succeeds when the simulation matches the expression, and *forbid*, which succeeds when the simulation does not match.

The temporal expressions and assertions must also be associated with a clock that specifies when the assertions are to be tested. Different expressions can be associated with different clocks. A clock can be defined as posedge, negedge, or any edge of a signal; or based on a temporal expression. Also, asynchronous events can use the simulation time as a clock.

A functional set of such expressions and assertions is called a *checker*. A checker can be associated with all instances of a specified module or limited to a specific instance.

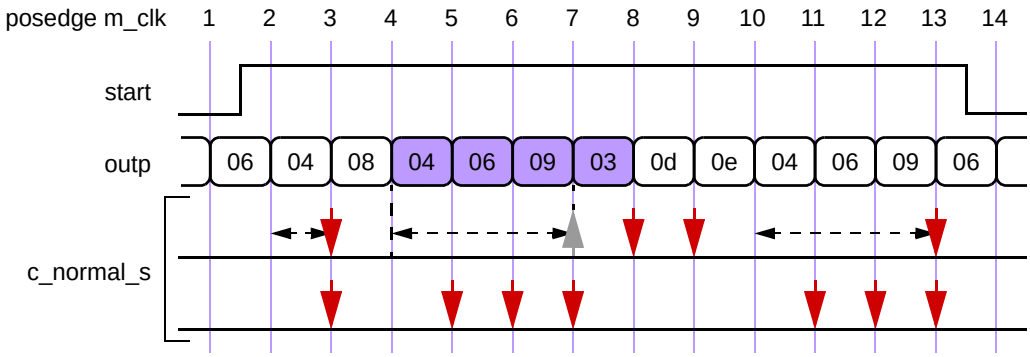
Example 1-1 shows an example temporal assertion file. It tests for a simple sequence of values (4, 6, 9, 3) on the device's outp bus. To limit reporting to relevant parts of the simulation, the test also checks for the start signal to be asserted.

Example 1-1 Temporal Assertion File, cnt.ova

```
/* Define a unit with expressions and assertions */
unit 4step
  #(parameter integer s0 = 0)// Define parameters
  (logic en, logic clk,    // Define ports
  logic [7:0] result);

  // Define a clock to synchronize attempts:
  clock posedge (clk)
  {
    // Define expressions:
    event t_0 : (result == s0);
    event t_1 : (result == 6);
    event t_2 : (result == 9);
    event t_3 : (result == 3);

    event t_normal_s:
      // Define a precondition to limit reporting:
      if (en) then
        (t_0 #1 t_1 #1 t_2 #1 t_3);
  }

  // Define an assertion:
  assert c_normal_s : check(t_normal_s,
    "Missed a step.");

endunit

/* Bind the unit to one or more instances in the design.
*/
// bind module cnt : // All instances of cnt or
bind instances cnt_top.dut : // one instance.
  4step start_4   // Name the unit instance.
  #(4)            // Specify parameters.
  (!reset, m_clk, outp); // Specify ports.
```

When the temporal assertion file is compiled and run with a simulator, the assertions are continuously tested for the duration of the simulation. New attempts to match each assertion to the simulation's values are started with every cycle of the assertion's associated clock. Each attempt continues until it either fails to match or succeeds in matching the complete expression. See Figure 1-1. The up arrow at clock tick 7 indicates a match that started at tick 4. The down arrows are failures. The failure or success of each attempt is logged to a file that can be reviewed later. Optionally, the logging of results can be limited to attempts that passed a precondition set by an if-then statement.

Figure 1-1 Assertion Attempts for cnt.ova



About This Document

The *OpenVera Language Reference Manual: Assertions* document is one volume of the two volume set that documents the OpenVera language. The other volume is *OpenVera Language Reference Manual: Testbench.*

2

Defining Simple Expressions, Assertions, and Coverage Expression Statements

This chapter describes the language for expressing simple 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.

In this chapter:

- “Packaging Assert and Cover Statements for Use”
- “Name Resolution”

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:

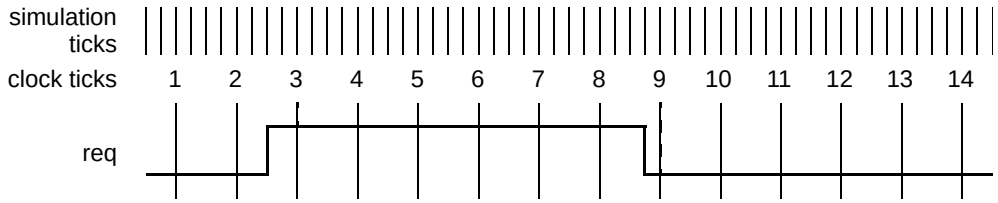
In this manual variable in an expression refers to these types of design objects and not the sense of the term variable in the Verilog-1364-2001 standard.

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. Simulation time is based on the minimum time unit defined for the simulation (such as with the Verilog `timescale directive).

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 2-1 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 2-1 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. All timing diagrams in this document reflect the values of expressions sampled by their associated 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.

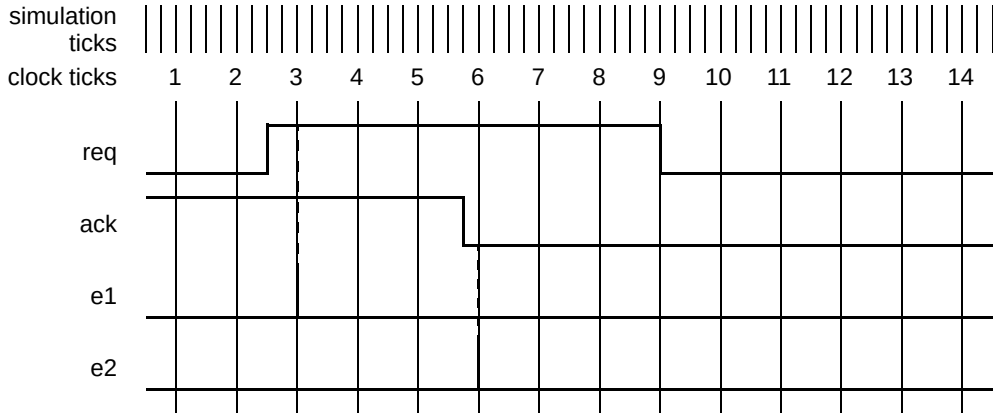
An edge event at a clock tick changes the value of an expression from the value of that expression at the previous clock tick. Like boolean expressions, an edge event evaluates to true if the event occurs, and to false if the event does not occur.

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

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

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

Figure 2-2 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 2-2, 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

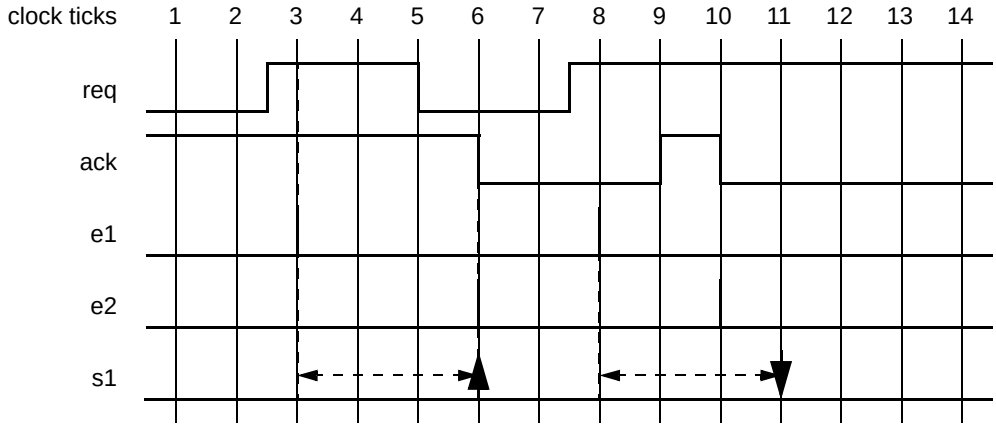
edge event is evaluated, resulting in a true/false value. A boolean expression is evaluated in the same way as a Verilog expression. To determine a match of a sequence, checkpoints are evaluated at appropriate times to satisfy the expression. If all the checkpoints are satisfied, then a match of a sequence 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 2-3. `s1` is defined as:

```
e1 #3 e2
```

Figure 2-3 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 2-3 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 ◄- - - - ► with a left and a right arrow to indicate that an evaluation is in progress during that time period.

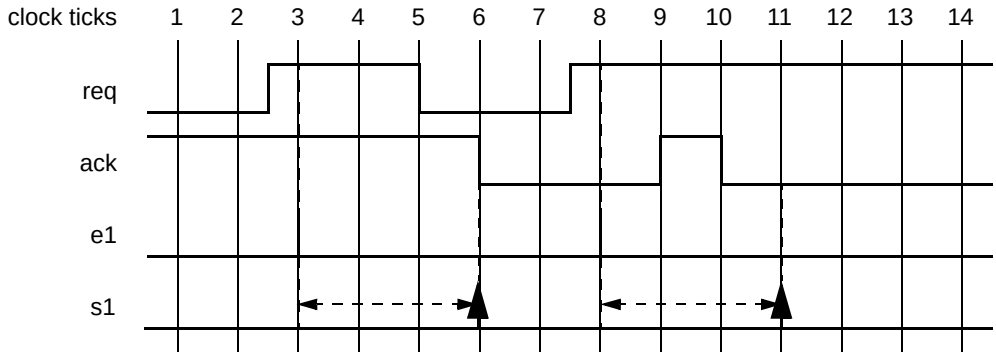
The values of signals shown in diagrams in this manual are the derived sampled values of those signals with respect to their clock, and not the actual simulation values at the corresponding simulation time.

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 2-4 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 2-4 Matching a Sequence with a Variable



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
```

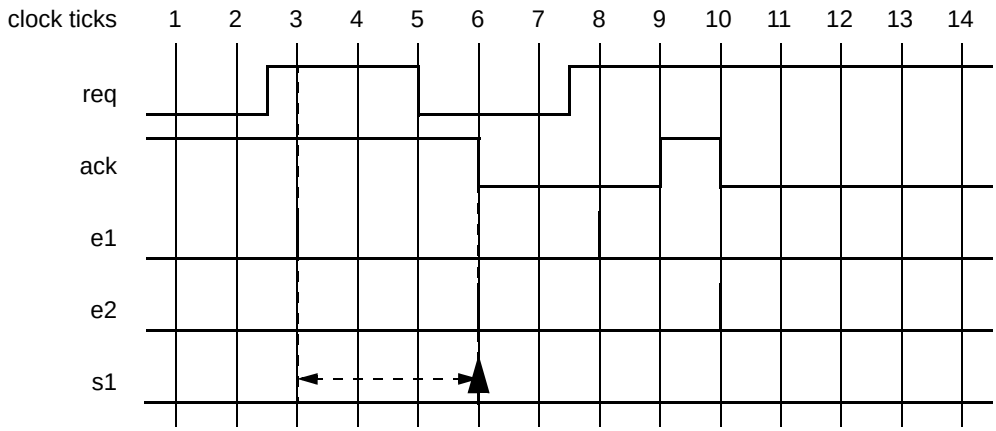
Then, `ack` is converted to its logical value using the above rule and the expression is true if `ack` is true at the proper clock tick.

Start and End Time of a Sequence

Each sequence has a start time and an end time. As seen from the examples in Figure 2-7 on page 2-12 and Figure 2-4 on page 2-8, while monitoring sequences the reference time (current time) is advanced according to the clock ticks between the checkpoints.

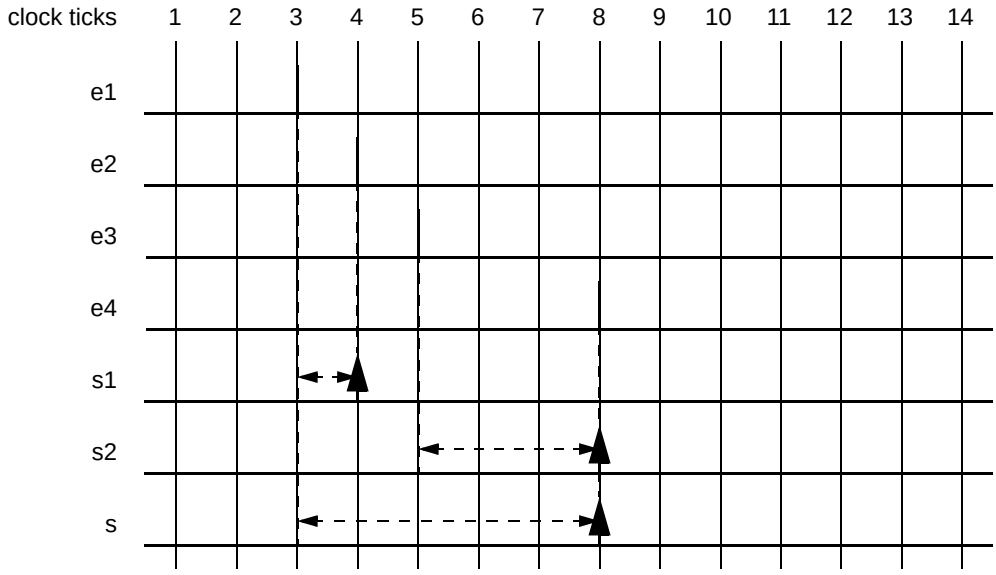
The start time for a sequence match is the time from which a new evaluation attempt of the sequence expression begins. The end time is the time at which a success or a failure for the sequence is detected. Let us examine the start and end times of the evaluation attempt at clock tick 3 for the example illustrated in Figure 2-5. 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 2-5 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 (`e1`, `e2`, `e3` and `e4`) at the corresponding clock ticks (3, 4, 5 and 8). Consider a sequence s consisting of two sub-sequences s1 and s2, where s1 is `(e1 #1 e2)` and s2 is `(e3 #3 e4)`, and s is defined as `(s1 #1 s2)`, and shown in Figure 2-6 on page 2-10. 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 2-6 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 2-6.

- 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.

Thus, sub-sequences comply with the associative law:

s1 s2 s3 = (s1 s2) s3 = s1 (s2 s3)

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:

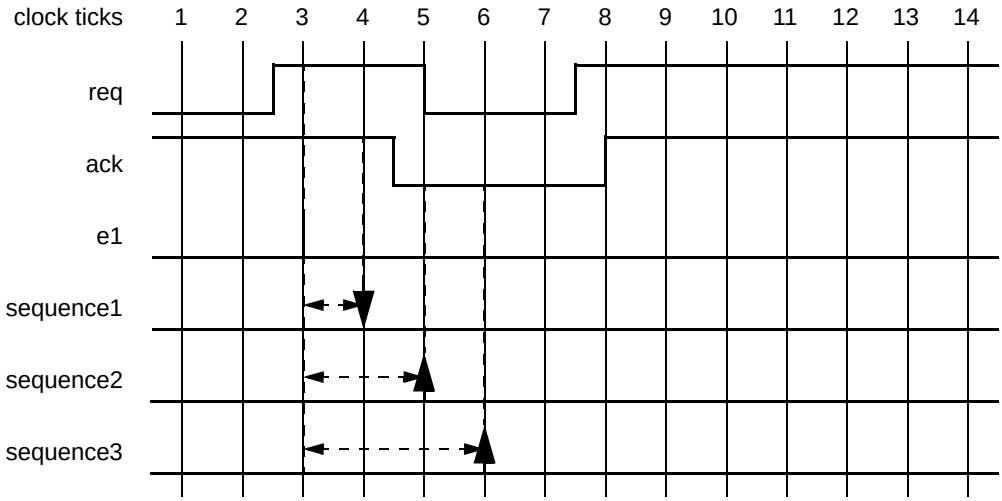
```
e1 #[1..3] (ack==0)
```

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 2-7. The three sequences are:

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

Figure 2-7 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.
- If this multiple sequence is the whole assertion then sequence3 is irrelevant because sequence2 matched successfully. The end time of the multiple sequence is 5.

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
```

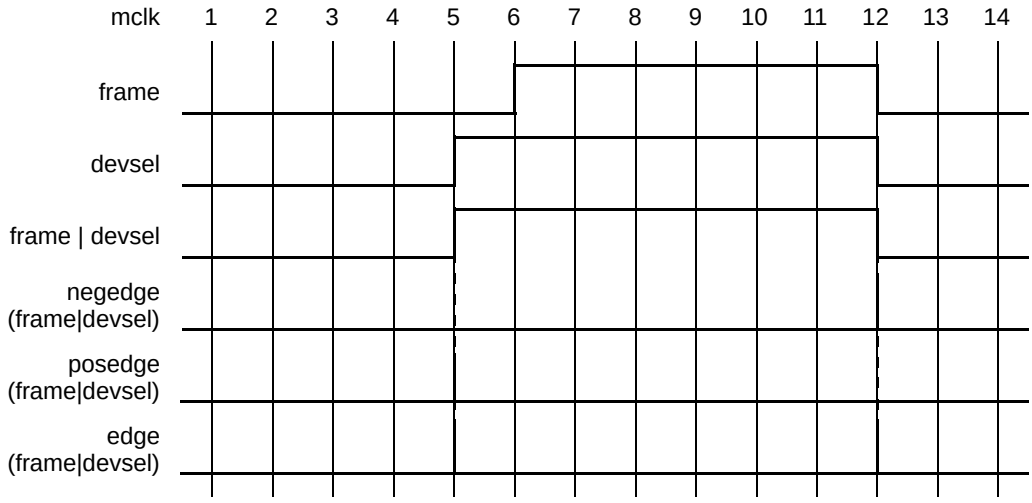
One important use of events is being able to describe change in values of variables. In practical situations, most activities in systems are initiated based on detecting a change in value. is an expression that results in a single or multi-bit vector. Three clauses are provided to specify change in values:

- posedge is used to express positive edge and generates an event upon 0 to1 transition on the value of the expression *bit_vector_expr*.
- negedge 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 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 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 2-8.

Figure 2-8 Edge Events



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

```
matched ended
```

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 *event_name* refers to a sequence expression specified by the ***event*** clause. Use ***ended*** to indicate that the operator and the event are in the same clock domain. Use ***matched*** when the operator and the event are in different clock domains. The ***event***, ***ended***, and ***matched*** clauses are explained in “Defining Expressions” on page 23.

Clocks

The syntax for specifying a clock is as follows:

```
clock
```

```
{
}
```

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

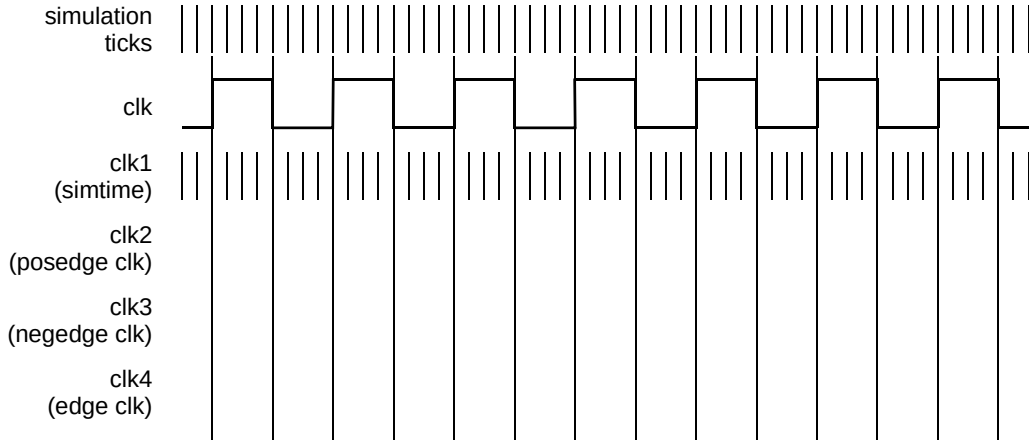
A clock tick occurs whenever the edge event described by occurs during simulation.

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 down the simulation.

Figure 2-9 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 2-9 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`.

Specifying Time Shift Relationships

The syntax for time shift relationships is as follows:

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

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 2-1 shows variations of time specifications that can be used.

Table 2-1 Time Specification Syntax

Table 2-2

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 or equal to t2. A special case is provided for passing an open-ended interval to templates: The interval #[t1..t2] with t1 > 0 and t2 = 0 is interpreted as #[t1..].
#[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 or equal to `t1`. There is one exception to this: When `t1> 0` and `t2 == 0`, then it is interpreted as the interval `#[t1..]`. This was done to accommodate specifying open-ended intervals using parameters of units (templates).

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:

```
[]
```

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

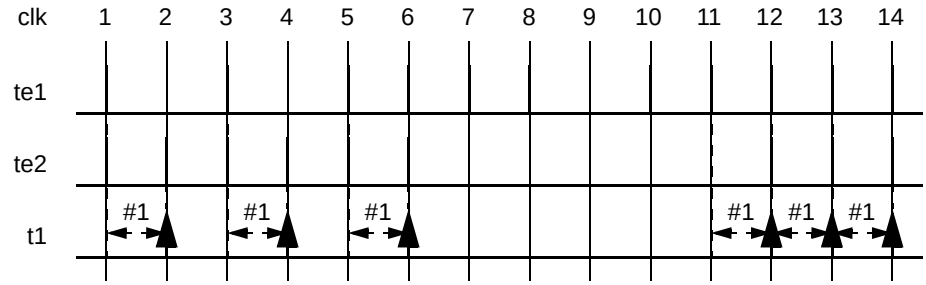
Clock ticks between two sub-expressions is specified using:

All variations of can be used between the two sequence expressions. Note that the time specified by can take on only a non-negative 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 2-10, 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 2-10 Time Specification `event t1: te1 #1 te2;`



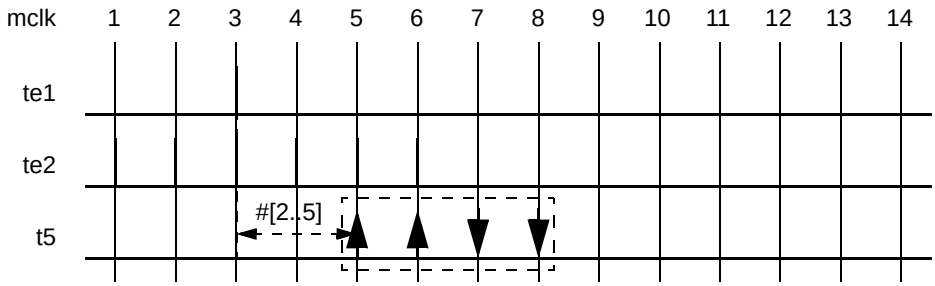
When an activity is allowed to end within a range of time (time window), then the `time_shift` clause with a minimum and maximum time specification is used to express the time window. When a range of time is specified, multiple matches can occur. At each clock tick within the time window, a new evaluation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

Consider the following example:

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

Here, `te2` can be true anywhere within a time window starting at 2 clock ticks after `te1` becomes true, and ending at 5 clock ticks after `te1` becomes true. Figure 2-11 illustrates this example for the evaluation attempt at clock tick 3.

Figure 2-11 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 `te2` is true at those clock ticks. At clock ticks 7 and 8, `te2` does not match.

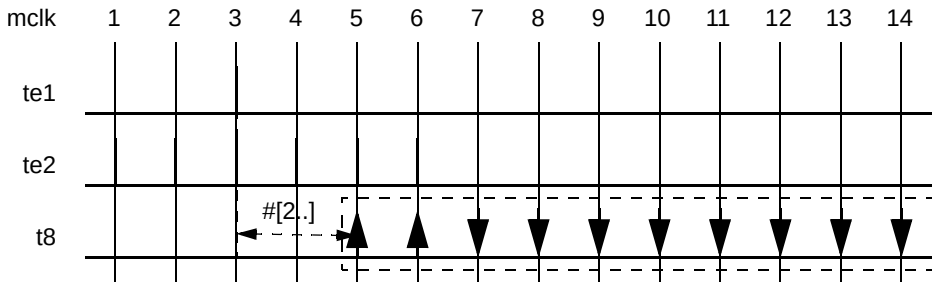
Consider another example:

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

In the expression `t8`, the maximum time is the end of simulation for `te2` to evaluate to true. `te1` must first evaluate to true, followed by `te2` some time after 1 clock tick, but before the end of simulation.

The time window for `t8` is shown in Figure 2-12 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 2-12 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:

```
 ;
```

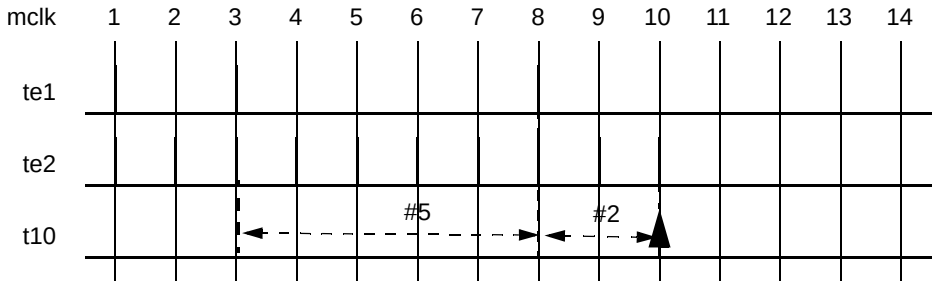
All variations of `time_shift` can be used. Time can be of 0 or positive value.

```
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 2-13 with the t10 event evaluation starting at clock tick 3. Note that te1 at clock ticks 1 and 3 do not lead to matches because less then four ticks precede them.

Figure 2-13 Time Shift as an Unary Operator



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

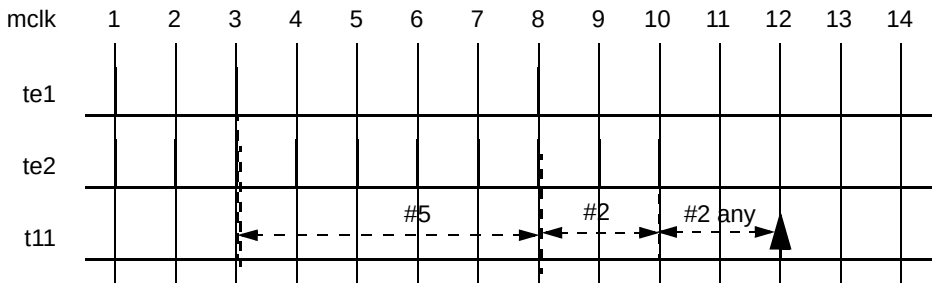
```
any ;
```

any denotes an empty expression that always evaluates to true (see "Specifying an Unconditional True" on page 28). 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 2-14 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 2-14 any at the End of a Sequence



Defining Expressions

The syntax for defining expressions follows:

```
bool  [()] : ;
```

or

```
bool  [[]] : ;
```

Note that the bitvector expression can contain an instance of another bool.

```
event  [()] :  ;
```

or

```
event  [[]] :  ;
```

Note that *sequence_expr* can contain an instance of another event.

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 {
     event 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 {
     event 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): (!ad[0])&&ad[1]&&!ad[2]&&ad[3]);
clock posedge sysclk {
     event 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 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. A one-dimensional array can be created by including an index. 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. An event with parameters inherits its clock from the declaration clock domain. If this is the simulation time, the instance obtains the clock of the instance clock domain.

An indexed instance of an event cannot refer to another instance of the same object. If an event refers to another indexed object, they must both use the same index value and the other object must already be defined (that is, no mutual cross-references).

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 ended e1 #1 branch_back;
  }
  ```

  In this example sequence expression e1 must end successfully one clock tick after inst. If the keyword ended was not 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 (ended t1) then
    ended t2 #1 ended 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 ended clause is used for building sub-expressions.

The matched keyword enables you to specify sequence expressions at different clocks and use their results in another sequence expression under yet another clock.

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 posses no timing relationship.

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

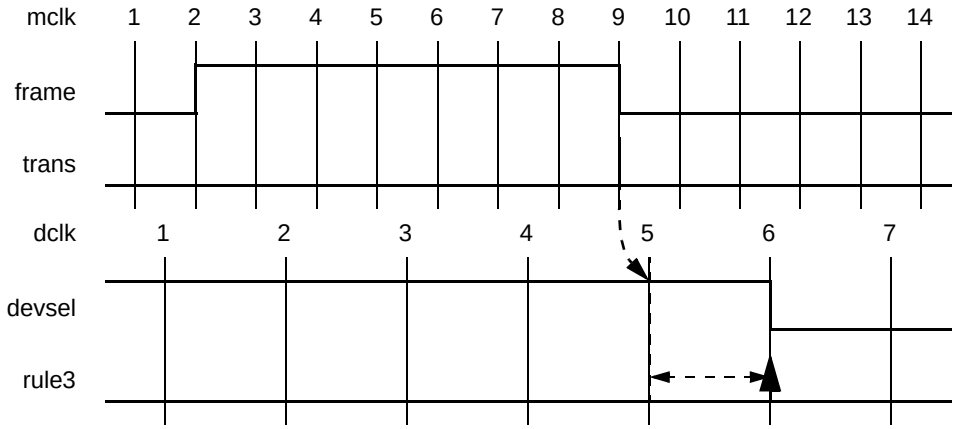
The semantics of the ***if then*** clause is discussed in “Specifying Conditional Sequence Matching” on page 10. A brief description of its usage follows.

```
event   : if   then
```

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 2-15, at clock tick 9 for clock `mclk`, `negedge frame` occurs. Figure 2-15 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 2-15 if-then Example

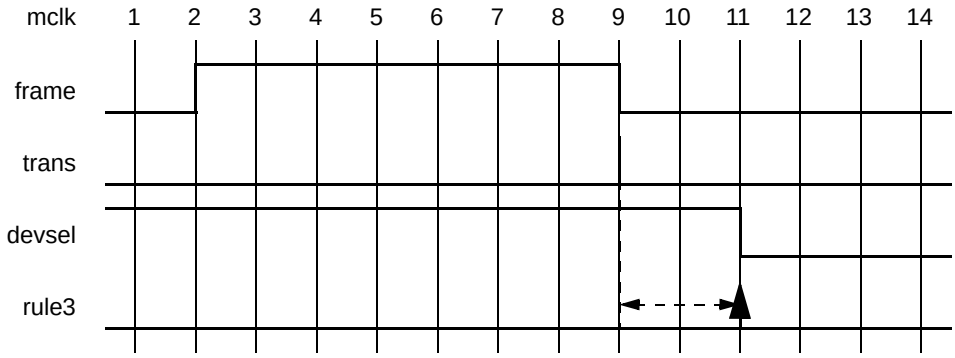


Note that the sequence event is only generated when its associated sequence expression succeeds. At all other times, the sequence event does not occur. Furthermore, the sequence event gets latched, in the sense that it can be tested for its occurrence, until the next clock tick of the sequence expression 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 2-15, 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 when the keyword ended is used. This is illustrated in Figure 2-16. 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 (ended trans)
    then #[1..3] negedge devsel;
}
```

Figure 2-16 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 unclocked event is instantiated in an unclocked definition, the definition remains unclocked. In the example, event t9 is unclocked as both events t1 and t8 are unclocked.
5. After resolution of clocks with respect to the sub-expression, if the event remains unclocked, it assumes the simulation time as clock.

Using the `assert` and `cover` Directives

This section describes the use of the assert and cover directives for:

- Specifying temporal assertions
- Specifying event coverage expressions

Specifying Temporal Assertions

The syntax for specifying a temporal assertion is as follows:

assert [] [`[]`] : check | forbid
(| [, [, *severity* [,
category]]]) [*action_block*];

Where

`action_block ::= [else display_statement]`
`display_statement` is similar to the standard $display system task:
`$display(formatting_string, list_of args);`
The 'formatting_string' parameter to the $display() call is optional. If this string is not provided, the arguments in the 'list_of_args' are printed using the default Verilog $display formats for the various number and expression types.

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.

An ***assert*** directive can be declared with an identifier *name* to name the assertion. While reporting, the name serves to identify the results for that specific assertion. If no name is given, the assertion name is a sequence number based on when the assertion is compiled.

An array of assertions can be created by including an index. An indexed instance of an assertion cannot refer to another instance of the same object.

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.

Declare the assertion with either a sequence expression *sequence_expr* or an to specify the relationship for monitoring. In this case, is an identifier of an event that specifies the sequence expression. The sampled values of the expressions are printed.

No explicit clock can be associated with the ***assert*** directive. If an event name is used as the argument and if there is a clock specified for the event, that clock is used; if no clock is specified then the simulation clock is used. If the argument is a sequence expression, then the simulation clock is used as the clock to evaluate the expressions. If the sequence expression includes the name of an event that has a clock, the assertion, as a whole, still uses the simulation clock. The event is evaluated independently with its own clock and the result included in the assertion evaluation at the corresponding simulation tick.

With check and forbid, there is also an optional failure message. The parameter is a quoted text string. The string is included with OVA's reporting when the expression in a check fails to match or when the expression in a forbid succeeds (and so, the forbid fails).

Two optional arguments are intended to check and forbid specifiers: *severity* and *category*. The *severity* argument is an unsigned number in the 0-255 range for the assertion severity level. The *category* argument is an unsigned number in the 1-(2^{24}-1)range for the user-defined category. Value 0 is reserved as "unspecified" indicator. Both field values can be specified as normal OVA parameters, as well as by constants (numbers). The default for the *severity* and *category* arguments is 0.

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

The action_block specifies the display_statement to display upon failure of the assertion. That is, the expression does not evaluate to a known, non-zero value. The display_statement is similar to the standard $display system task. The action block is executed immediately after the evaluation of the assert expression.

When both message and `$display` are specified then both are output on assertion failure. There is no concatenation of $`$display` output and msg.

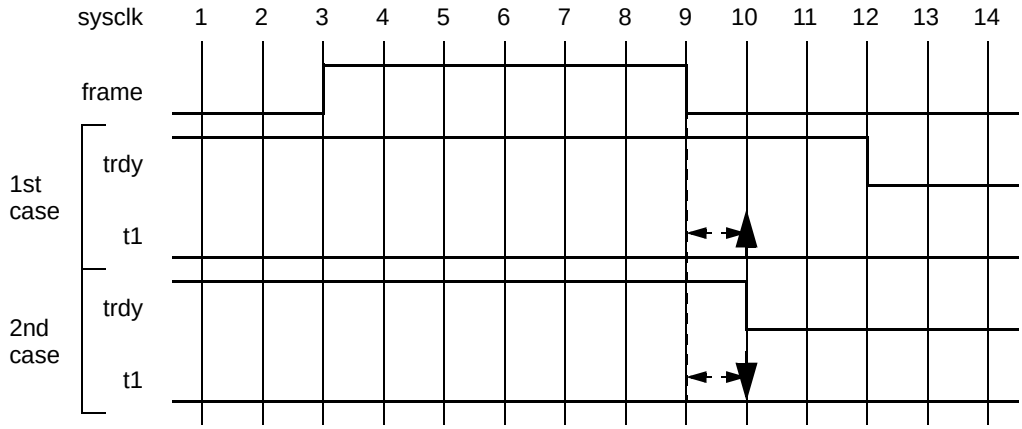
For example, in a bus read transaction, assume that there is a turn-around time of one clock cycle. After the signal named `frame` toggles low, the signal named `trdy` must not get toggled low in the next clock tick. An assertion can be written for this as follows:

```
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 2-17, 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 2-17 assert Example



Specifying Event Coverage Expressions

Event coverage statements using the cover directive record all successful matches of the coverage expression. The coverage statements behave in the same way as assertions and can be specified within assertions or standalone.

The syntax for specifying a coverage statement is as follows:

```
cover [] [[]]:(|
[, [, severity [, category]]])
            [$display (formatting_string, list_of_args) ];
```

Event coverage expressions are specified as ***cover*** directives. Event expressions *event_expr* specify event criteria for match success. Event expressions differ from assert statements in that the event expression can be sequential. Cover works on non-temporal events as well, where the number of first matches will actually be the same as the number of all matches.

When the event expression results in a match, the cover always increments a counter named total_matches. Multiple matches per attempt may be generated and reported. With compile time option -ova_enable_diag, if the match is the first success of the attempt, then the the cover directive also increment a second counter first_matches. The sampled values of the expressions are printed.

For exampe, if you have an OVA event:

```
event e: a #[1..3 b;
```

and then cover c:(e);

while the trace contains

```
a b b b
```

then the attempt starting with the a being 1 will match the event e 3 times, on all 3 b's.

The first match occurs on a b, the second one on `a _ b` and the 3rd one on `a_ _ b`.

Cover gives the count of all matches and also the first matches.

In this case we have total matches 3 and first match - 1. "

An array of cover expressions can be created by including an index. An indexed instance of an assertion cannot refer to another instance of the same object.

Two optional arguments are *severity* and *category*. The *severity* argument is an unsigned number in the 0-255 range for the cover severity level. The *`category`* argument is an unsigned number in the 1-(2^{24}-1)range for the user-defined category. Value 0 is reserved

as "unspecified" indicator. Both field values can be specified as normal OVA parameters, as well as by constants (numbers). The default for the *severity* and *category* arguments is 0.

The optional `$display` used the standard system task to show a custom message when the event expression results in a match. The 'formatting_string' parameter to the $display() call is optional. If this string is not provided, the arguments in the 'list_of_args' are printed using the default Verilog $display formats for the various number and expression types.

Packaging Assert and Cover Statements for Use

Before assertions and cover statements can be connected to a design, the statements, along with all associated clock and bool definitions, must be grouped in a package called a unit. Think of a unit as a library for assertions and cover statements. The unit can then be bound with the design one or more times as needed.

Note:

Units cannot be instantiated into designs or into other units. You must bind units to the design.

Input to the units are defined as parameters and ports that provide strong data typing.

Declaring Units

The syntax for specifying a unit is:

```
unit
[#(; ...; )]
```

```
(, ..., );
endunit
```

Note that the Verilog-like order is also accepted:

```
 [#(, ..., )]
[]
[(, ..., )];
```

The unit's name must be unique among all OVA sources being processed, including the names of template definitions. That is, no unit or template can share the same name. (Templates are described in "Grouping Assertions as a Macro" on page 40.)

Each is a list of parameter names and default values, all of the same data type. The parameters can be used within the unit any place constants can be used. Every parameter must have a default value, which must be an OVA or HDL compile-time constant, depending on how it is used in the unit. The syntax for a is:

```
parameter [integer | real | string]
 = , ...,  =
```

The data type defaults to integer.

Certain parameters must be *OVA compile-time constants*. That is, the values of the parameters must be determined when the OVA code is processed, which is *before* the HDL code. The rest of the parameters can be determined during HDL or OVA compile-time. Parameter values cannot be determined during runtime. Any attempt to determine a parameter value at the wrong time results in an error.

OVA compile-time constants must be used to control for-loops and to define temporal expressions. That is, the *`int`* (integer) in the syntax of any expression operator, such as `#`, must be determined before the HDL code is processed.

HDL compile-time constants can be used (as can OVA compile-time constants) to specify the width of signals and OVA variables, elements of arrays (such as `matched e_array[]`), assertion messages, and values used in boolean expressions (such as `sigA ==`).

The design signals link to the assertions through the ports of the unit. Ports can be of logic, integer, or real types. The syntax for a declaration is the following, depending on the type:

```
logic [[:]]  [[:]...]
```

The logic type is for “reg” signals and can take the values of 0, 1, Z, or X. If a logic port is more than one bit wide, it must include a bit range. A port can be a multi-dimensional array by including a set of ranges for each dimension after the port name. The bit and dimension ranges are specified with constant expressions, which can include parameters from the unit’s parameter lists.

The following is an example of declaring a unit:

```
unit states
  #(parameter integer no_ns = 1, width = 1, st = 0;
    parameter string msg = "Wrong state.")

  // Ports:
  ( logic clk,
    logic [width-1:0] exp, // "width" bits wide
    logic [width-1:0] ns [0:no_ns-1]);
                          // array of "no_ns" elements

  assert state_chk: check(ova_e_state, msg);
  clock posedge clk {
    event ova_e_state: exp == ns[st];
  }
endunit
```

The assertion, state_chk, compares a bitvector, exp, with a specified element, st, of an array, ns. The width of the bitvector depends on the “width” parameter. The size of the array depends on the no_ns parameter. If the assertion fails, the report message includes the value of the msg parameter.

The unit declaration only shows the default values of the parameters. The parameter and signal names can be changed when the unit is bound to the design.

Binding Units to the Design

Units can only be bound to a design. They cannot be instantiated into designs or units. Unit instances can be bound to either all instances of a design module or only specific instances. The syntax to bind units is:

```
bind module  : ;
bind instances , ...,  :
;
```

Use the ***bind module*** construct to connect a unit to all instances of a module. Use the ***bind instances*** construct to connect a unit to specific module instances. Identify the instances with full hierarchical names. The instances in a list do not have to be of the same module.

For example, the synatx for the states unit defined in the previous section is:

```
#(3, 8, 2, "Not state 2")
(clk, state_var, {{0}, {1}, {2}});
```

Or it can be:

```
#(.st(2), .msg("Not state 2"), .no_ns(3), .width(8))
(.exp(state_var), .ns({{0}, {1}, {2}}), .clk(clk));
```

The parameters can use constants, constant expressions, and expressions using Verilog parameters. However, expressions using Verilog parameters are not OVA compile-time constants and so are limited in when they can be used. They cannot be used when they effect the temporal aspects of the unit:

`#` time shift

`*` repetition factor

`for` loop bounds

The ports can use constants, local signal names, and full hierarchical names. Bit ranges must use constant selectors. Verilog expressions can be bound to a port. If the signal is wider than the port is defined for, the signal's left-most bits are truncated. If the signal is narrower than the port, the signal is padded with zeros for an unsigned logic port and with copies of the left-most bit for a signed logic port. If a port is not specified, the unit instance looks for a local signal of the same name. If none is found, the port uses an X value.

For example, the previously defined states unit can be bound to all instances of a design module, mpu, with:

```
bind module mpu : states mpu_states
  #(3, 8, 2, "Not state 2")
  (clk, state_var, {{0}, {1}, {2}});
```

The states unit can be bound to just two instances of the mpu module with:

```
bind instances tb.mpu1, tb.mpu2 : states mpu_states
  #(3, 8, 2, "Not state 2")
  (clk, state_var, {{0}, {1}, {2}});
```

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:

- A design object name is also a sequence expression language keyword such as **in**.
- 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'in;
}
```

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

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`.

3

Constructing Complex Sequences

This chapter describes the language for combining expressions into complex sequences. This chapter also describes adding variables to expressions and combining expressions and assertions into reusable libraries.

In this chapter:

- “For Loops”
- “Building Expressions Iteratively”

Specifying Composite Sequences

Sequences can be combined with AND and OR functions. The syntax for specifying composite sequences is as follows:

```
sequence_expr && sequence_expr
sequence_expr || 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.

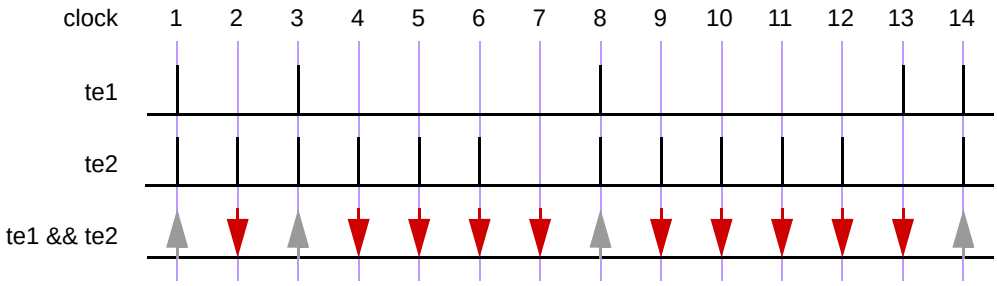
For the expression:

```
te1 && te2
```

If `te1` and `te2` are events containing just booleans, the expression succeeds if `te1` and `te2` are both evaluated to be true at the same time.

An example is illustrated in Figure 3-1 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 3-1 ANDing (&&) Two Events



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

```
te1 && te2
```

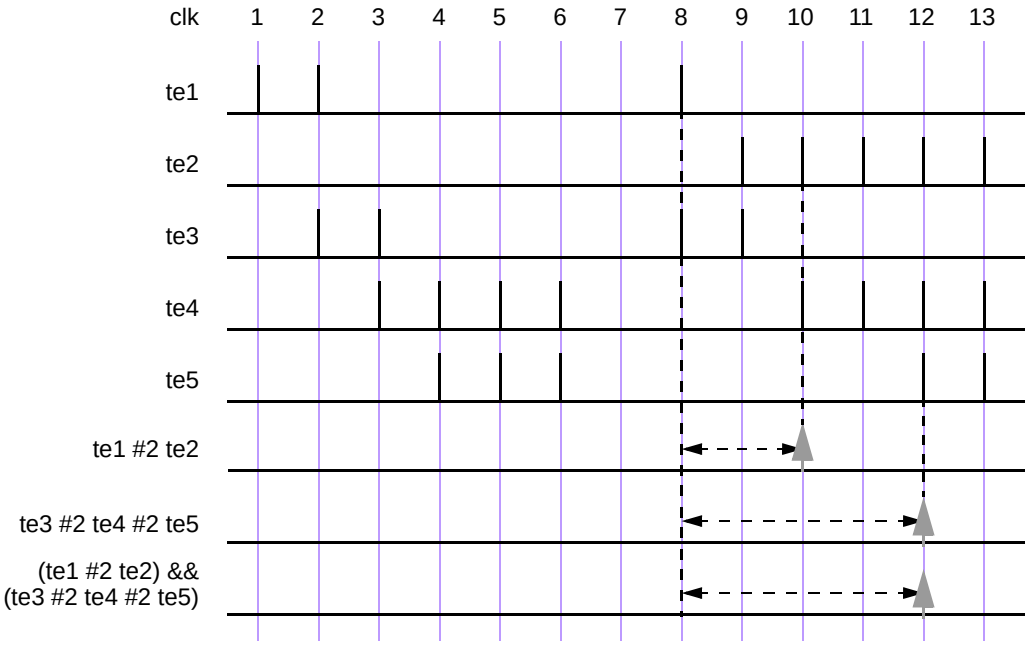
- Succeeds if `te1` and `te2` succeed.
- The end time is the end time of either `te1` or `te2`, whichever terminates last.

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

An example is illustrated in Figure 3-2. Consider the following expression with operator ***&&*** where the two operands are sequences.

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

Figure 3-2 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 3-2 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:

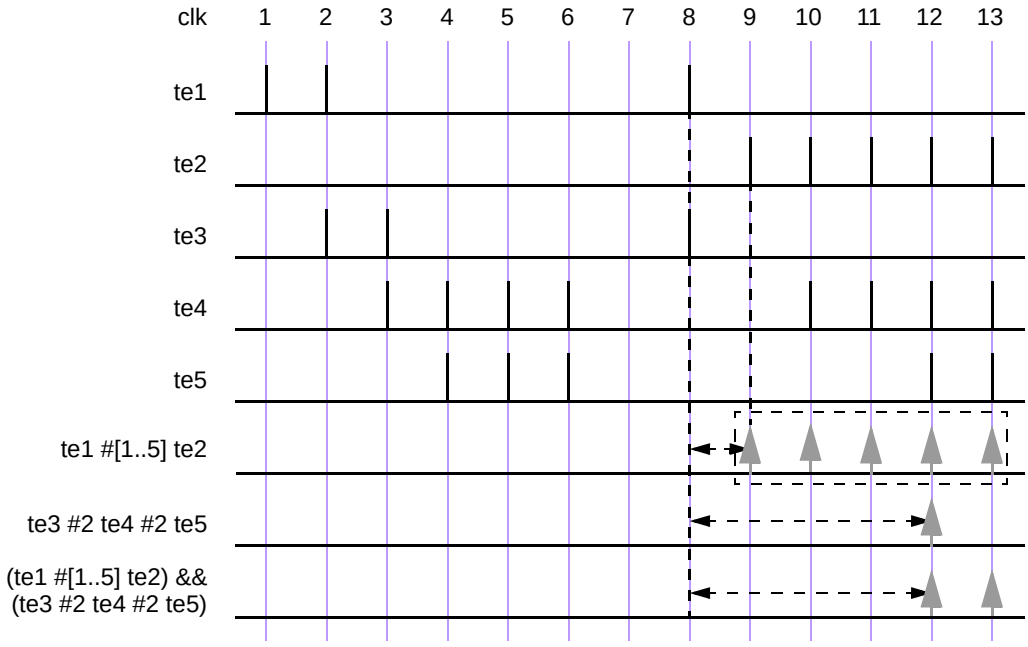
```
(te1 #[1..5] te2) && (te3 #2 te4 #2 te5)
```

The first operand sequence consists of an expression with a time range from 1 to 5 and implies that when `te1` evaluates to true, `te2` 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 3-3 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 3-3 shows the two unique successes at clock ticks 12 and 13.

Figure 3-3 ANDing (&&) Two Sequences Including a Time Range



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.

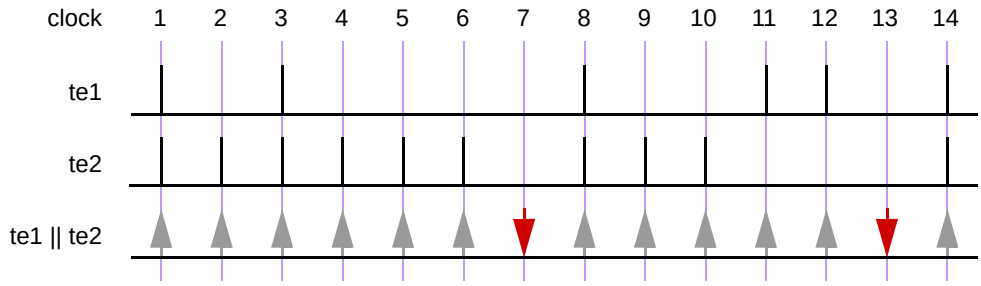
Let us consider these operand expressions as values, events and sequences separately to illustrate the details of **||** operations. For the expression:

```
te1 || te2
```

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

Figure 3-4 illustrates `||` operation using `te1` and `te2` as booleans. 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 3-4 ORing (||) Two Events



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

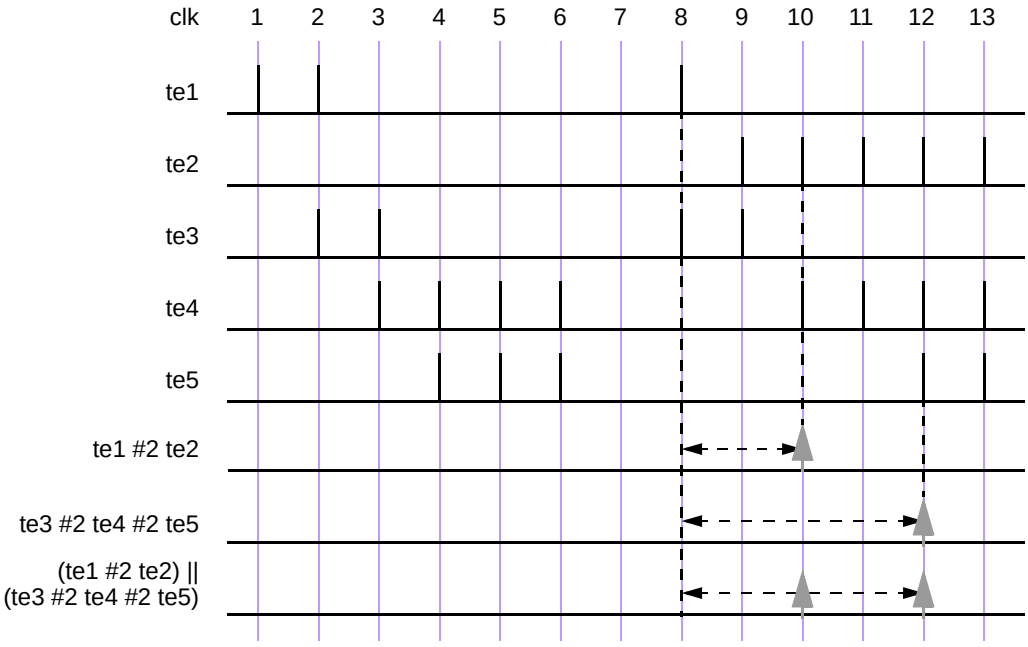
```
te1 || te2
```

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 3-5. Consider an expression with **||** operator where the two operands are sequences.

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

Figure 3-5 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 3-5, 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.

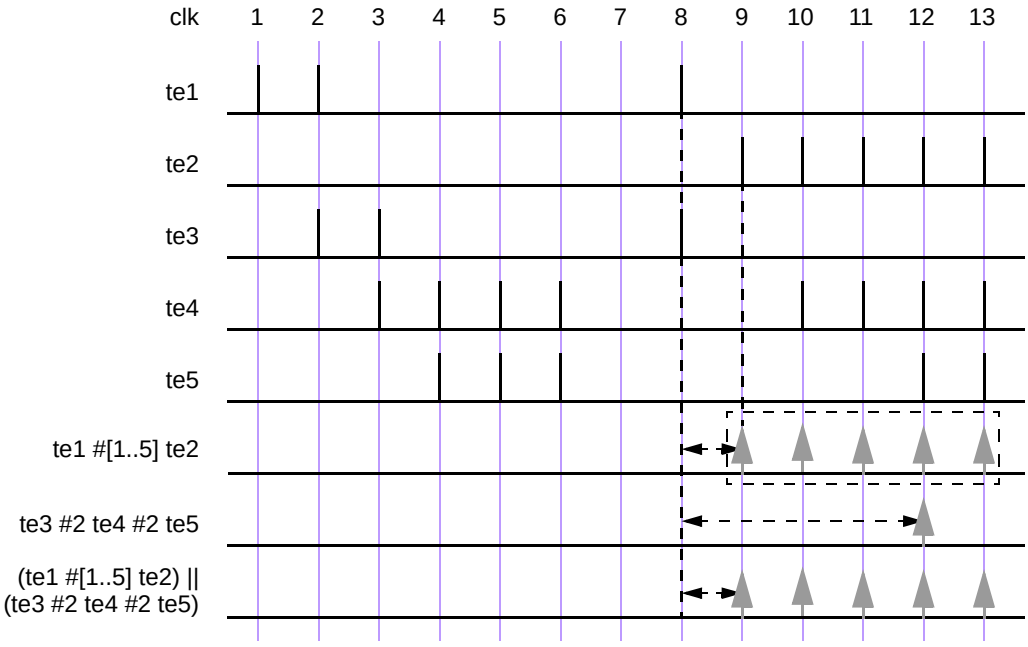
Consider an example where an operand sequence is associated with time range specification, such as:

```
(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 3-6, 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 3-6 ORing (||) Two Sequences Including a Time Range



AND and OR Operator Precedence

The following examples explain the operator precedence when `&&` or `||` is used:

```
a #1 b && c #1 d;
```

is equivalent to:

```
a #1 (b && c) #1 d;
```

and

```
#1 a && #2 b;
```

is equivalent to:

```
#1 (a && (#2 b));
```

This is because `&&` binds more strongly with booleans than sequences.

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 many sequence matches.

Please note that `boolean_cond` cannot be a sequence expression but it can be `ended` *`clocked_sequence_expr`* or `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` 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;
```

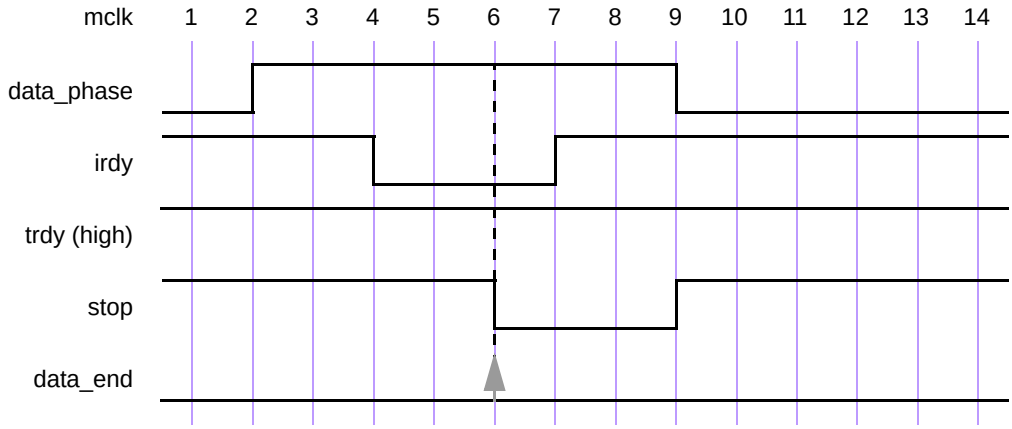
The semantics of ***if then else*** specification is explained by examples here. Consider a bus operation for data transfer from a master to a target device. When the bus enters a data transfer phase, multiple data phases can occur to transfer a block of data. During the data transfer phase, a data phase 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 : (data_phase) && (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 3-7. 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 3-7 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 {
```

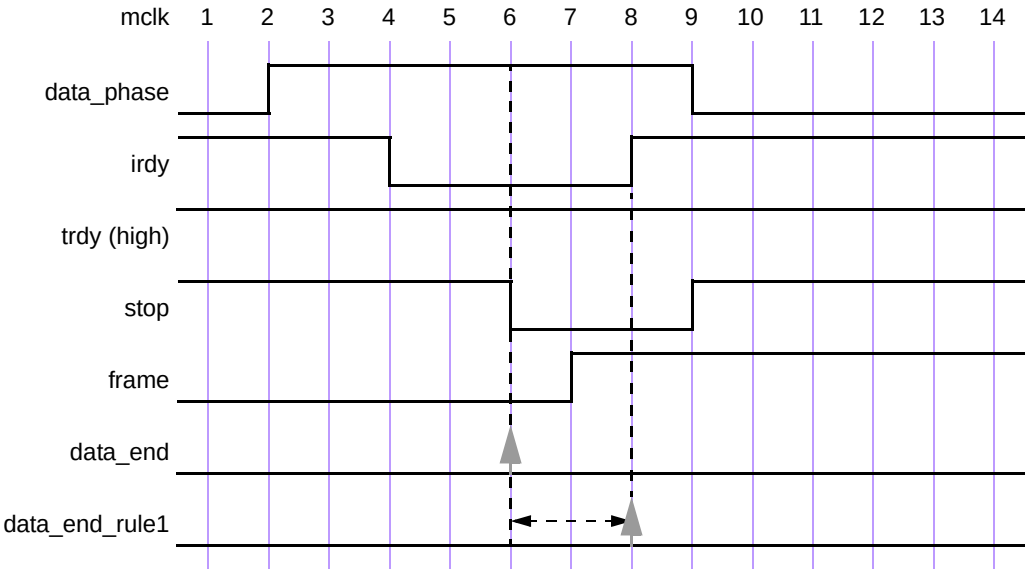
```
     event data_end_rule1:
          if (ended data_end) then
               #[1..2] posedge frame #1 posedge irdy;
}
```

`event data_end_rule1` 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 3-8. Event `data_end` is acknowledged at clock tick 6. Next, `frame` toggles high at clock tick 7. Since this occurs 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.

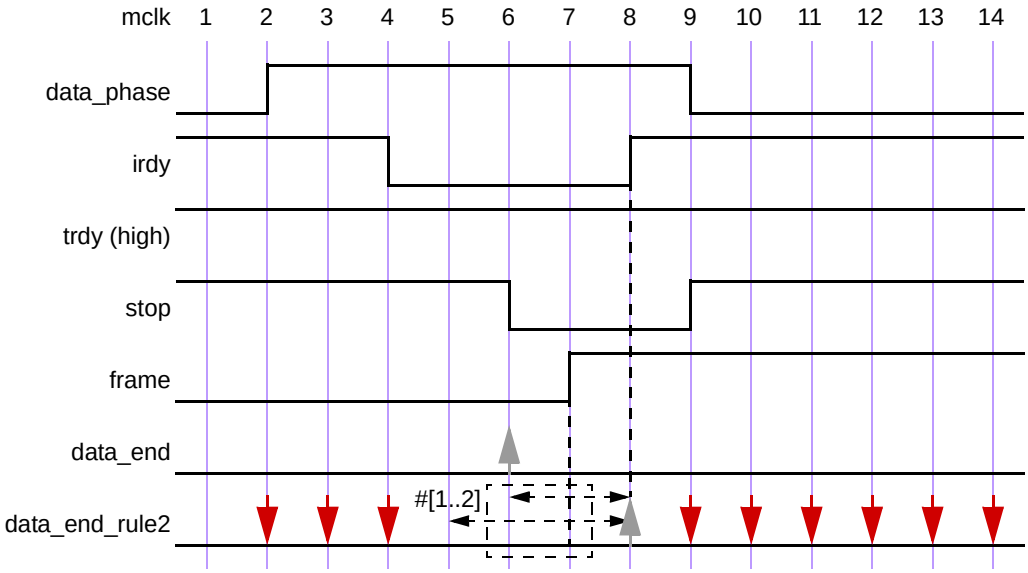
Figure 3-8 Nested Conditional Sequences



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 3-9.

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

Figure 3-9 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 for evaluations starting 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 3-9, 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:

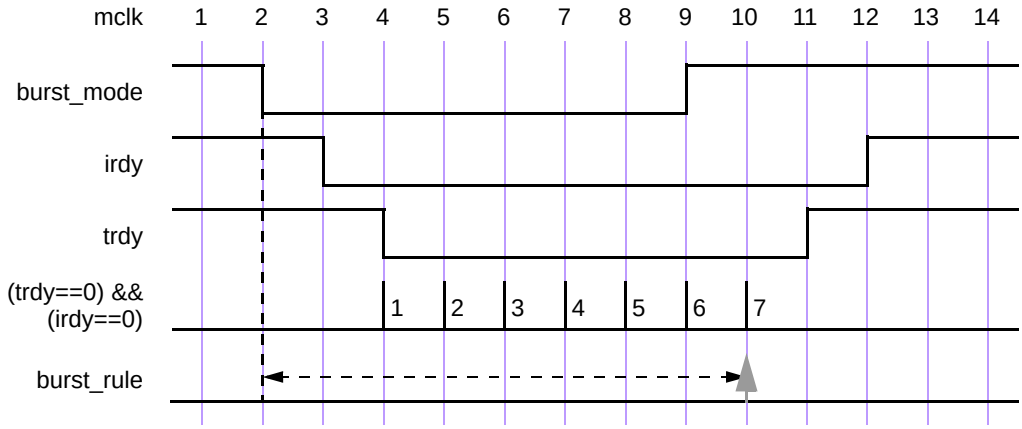
  ```
  (ev1 #1 ev2) #1 (ev1 #1 ev2) #1 (ev1 #1 ev2)
  ```

- 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 3-10. 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 asserted low at the same time.

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

Figure 3-10 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 3-10, the falling edge of `burst_mode` occurs at clock tick 2 and data transfer begin 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:

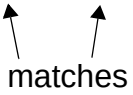
```
(#4 (ev1 #1 ev2) #5 (ev1 #1 ev2) #5 (ev1 #1 ev2)) ||
(#4 (ev1 #1 ev2) #5 (ev1 #1 ev2) #5 (ev1 #1 ev2) #5 (ev1 #1 ev2))||
(#4 (ev1 #1 ev2) #5 (ev1 #1 ev2) #5 (ev1 #1 ev2) #5 (ev1 #1 ev2) #5 (ev1 #1 ev2))
```

The following example demonstrates the case of `*[0]` repetition, particularly in the context of surrounding time shift operators:

```
event e: a #1 b*[0..] #1 c.
```

What sequences will satisfy it?

```
a  10001000
b  00000100
c  01000010
```



To model this, "If a, then either c is true at the next cycle, or b must be true any number of times, and the last be true at the same time as c."

```
if a then #1 (c || b*[1..] #0 c);
```

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:

- istrue *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:

```
istrue 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
```

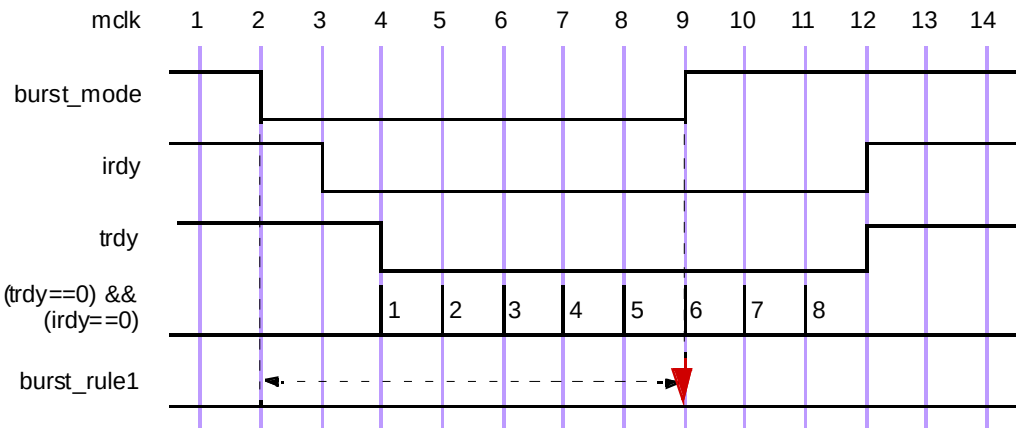
int or int_interval specifies the length of the sequence_expression. The length is measured as the total number of clock ticks during the sequence. All variations of the int_interval specification are allowed. If a single number is specified for int_interval, then it represents fixed length. In other 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 3-11. 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]);
}
```

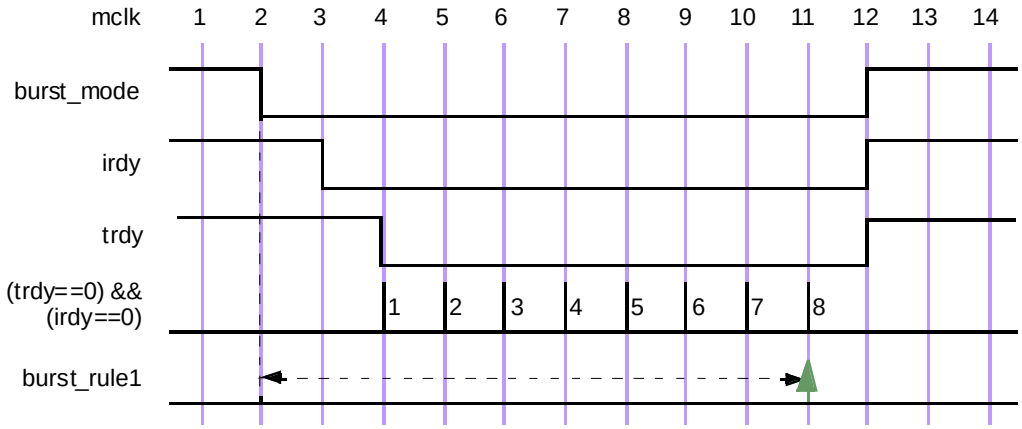
Figure 3-11 Match with istrue-in Restriction Fails



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 3-12.

Figure 3-12 Match with istrue-in Restriction Succeeds

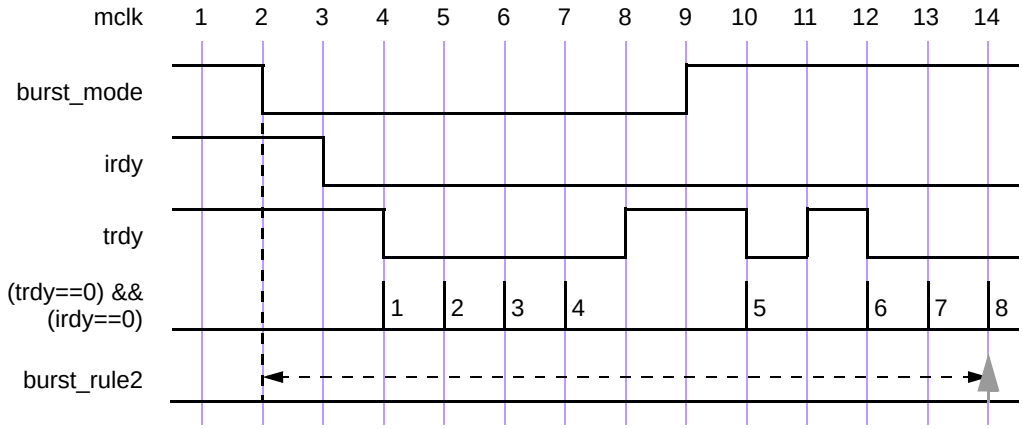


Let us consider a modified version of the example in Figure 3-12 as shown below.

```
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 5 clock ticks after the preceding occurrence of the sequence. This is illustrated in Figure 3-13.

Figure 3-13 Match without Restriction Succeeds



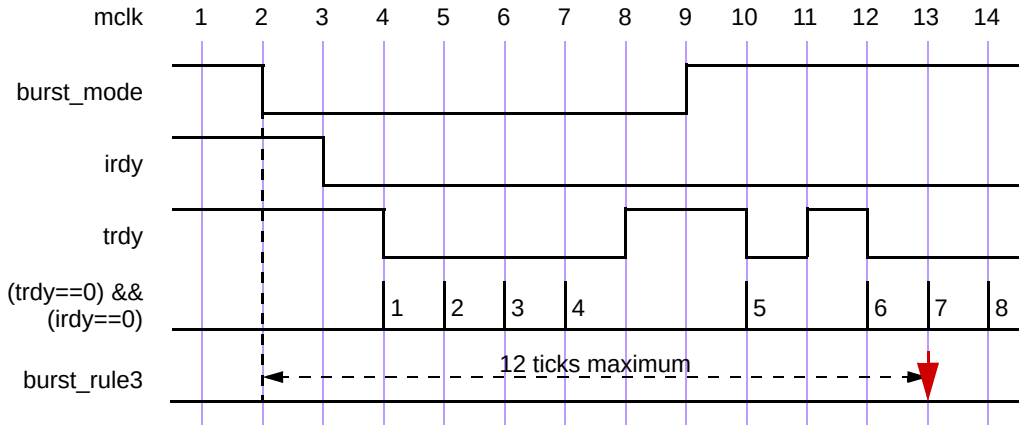
Two additional clock ticks delay the fifth repetition and one additional clock tick delays the sixth repetition as signal trdy becomes high to suspend the next data phase for two clock ticks and one clock tick respectively. The expression matches at clock tick 14.

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..12]
          in (#[0..4] (trdy == 0) && (irdy == 0)) * [8];
}
```

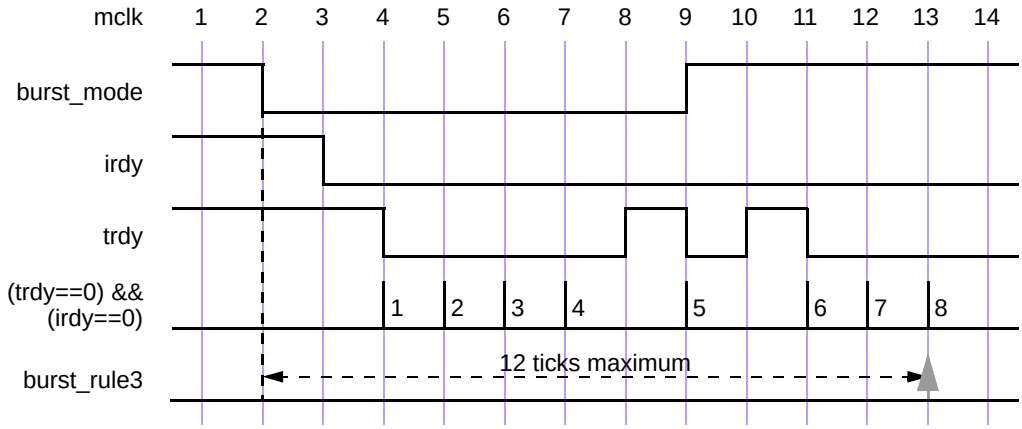
In the above expression, the total length of the entire repeated sequence must not be less than 9 clock ticks and not greater than 12 clock ticks. This restriction is expressed by length [9..12] in the expression. From Figure 3-14, the corresponding time to complete all repetitions is 13 clock ticks which exceeds the maximum allowed length, so the expression fails to match at clock tick 13.

Figure 3-14 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 3-15.

Figure 3-15 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
      istrue(!burst_mode), length [9..12]
        in (#[0..4] (trdy == 0) && (irdy == 0)) * [8];
}
```

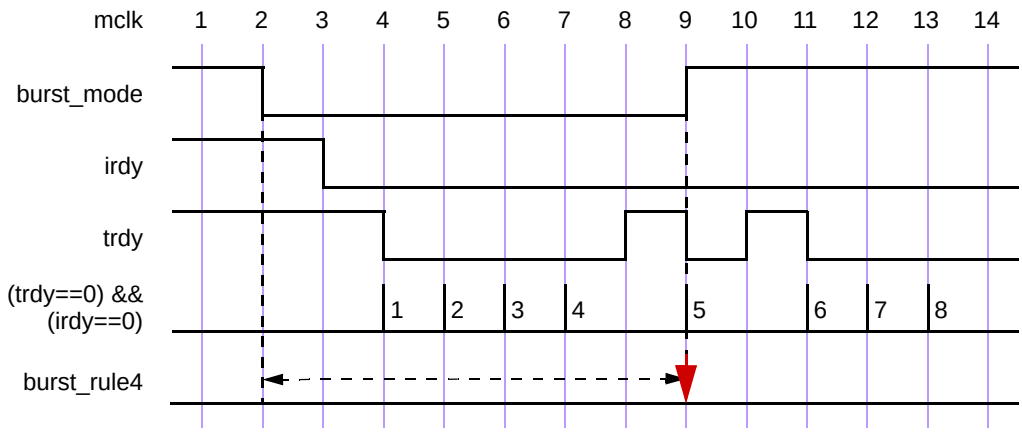
Now the two constraints are:

- `istrue(!burst_mode)` to ensure that signal `burst_mode` remains low
- `length [9..12]` to ensure that the sequence takes at least 9 clock ticks and completes in at most 12 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 3-16 because signal `burst_mode` becomes high at clock tick 9.

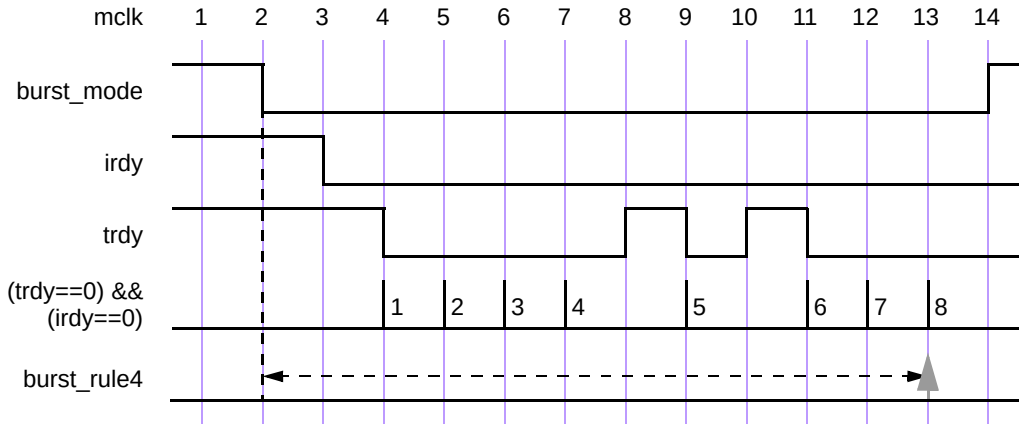
Figure 3-16 Match with Two Restrictions Fails



The failure is corrected by maintaining signal `burst_mode` to low value throughout the sequence as shown in Figure 3-17. 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 3-17 Match with Two Restrictions Succeeds



Specifying an Unconditional True

The syntax for specifying an unconditionally true boolean 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:

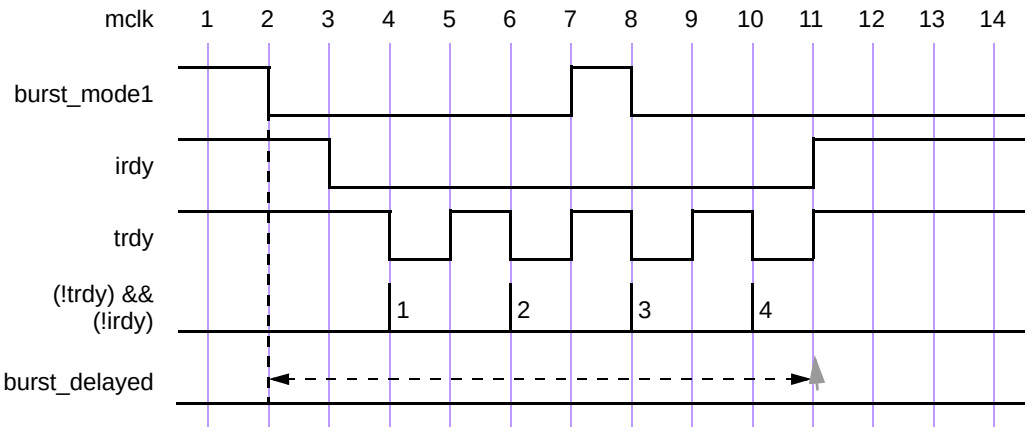
```
te1 #t1 any
```

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 3-18 shows such an example with the assertion:

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

Figure 3-18 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 3-18, the burst mode starts at clock tick 2 when signal `burst_mode1` becomes low.

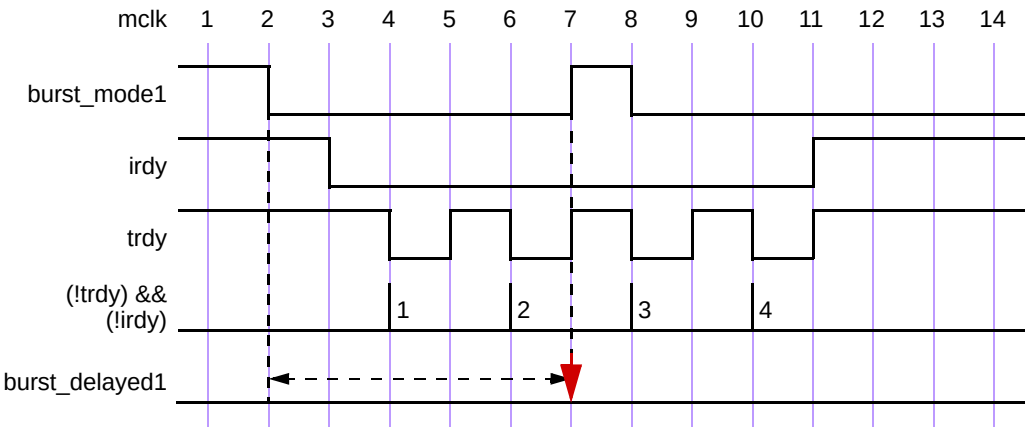
First repetition is satisfied 2 clock ticks. Between each repetition 2 clock ticks are expected, and satisfied accordingly at clock ticks 4, 6, 8 and 10. The expression matches at clock tick 11.

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 ***istrue 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 3-19 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 3-19 Using any with an istrue-in Restriction



Manipulating and Checking Data

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.

For one-dimensional arrays, the initial value must be a constant bit vector expression. The assignment can be made as follows:

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

For two-dimensional arrays, the initial value can be either a constant bit vector expression or a constant aggregate expression such as {{2}, {3}}.

```
var [[int:int]] name [int:int] = initial_const_value or
initial_const_array_aggregate;
```

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.

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 = initial_const_value;
```

The `init` statement is not a replacement for the declaration. If the declaration of the variable does not have the initial value assignment (optional), then the initial value assignment can be in the init statement. Synopsys recommends using the `var` statement instead if the `init` statement.

If the OVA checkers are to be used with tools that combine simulation and formal techniques, it is recommended that all OVA variables be initialized to some known value rather than leaving it to the default unknown value (x).

The *`var_name_ref`* references a variable name with optional bit-select, part-select or word-select with the same rules as Verilog. Prior to the beginning of simulation, the simulator first performs initialization of all design objects according to the design description, followed by the initialization of these assertion variables by executing the ***init*** statements.

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 should be placed under a clock. If not, the statement uses the simulation clock. 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.

There can be only one such assignment to a variable.

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. A write of one cycle duration begins the command. The number of words is specified by signal `w_size` which can vary from 1 to 31. Each word written is specified by signal `w_start` and the end of block write is indicated by signal `w_end`.

```
clock posedge sysclk {
  var[4:0] n = 0;
    n <= (write) ? w_size : n - w_start;
  event prop_w: if (write) then
    (istrue(!w_end) in #1 (n > 0)*[1..])
      #1 w_end && (n == 0);
}
assert block_write_rule: check(prop_w);
```

You can also have two-dimnensional arrays. The following example shows how to initilize a 2-D array using `{{elem0}, {elem1}, ..., {elemn_minus_1}}`:

```
var [15:0] masks [0:5] = {{16'b0}, {16'b1}, {16'h0003},
{16'h000f}, {16'h00ff}, {16'hffff}};
```

The following example shows a simple queue implemented using a 2D variable.

```
var [4:0] v_tail_ptr = 0;
var [4:0] v_head_ptr = 0;
var [4:0] v_q_size; = 0;
var  [15:0] v_q  [0:31] = {32{16:60}};
v_tail_ptr    <= reset ? 0 :
              b_enq  ? (v_tail_ptr + 5'b1)  :  v_tail_ptr;
v_head_ptr <= reset ? 0 ;
            b_deq  ? (v_head_ptr +  5'b1)  :  v_ head_ptr;
v_q[v_tail_ptr]    <= !reset &&  b_enq  ? enq_data :
v_q[v_tail_ptr];
v_q_size <= reset ? 0 :
        b_enq ? ( b_deq ? v_q_size  : ( v_q_size + 5'b1) ) :
                ( b_deq ?  ( v_q_size -  5'b1) : v_q_size  );
```

Note that there would still be assertion of `v_q_size` to make sure it does not overflow or underflow.

Using the `logic` Declaration

The logic declaration replaces the var expression.

There are three kinds of assignments to logic variables:

- A *continuous-like assignment* outside a clock domain. This cannot have an initial value in the declaration.
- *A non-blocking assignment.* The assignment must be inside a clock domain. An initial value can be assigned in the declaration (which is again outside any clock domain). For example,
- A *continuous-like assignment* inside a clock domain. The declaration must not have an initial value specification

See the following sections for details of the logic assignments

The syntax for the logic declaration is:

logic [[range]] name [[range]] = constant_expression;

Where `name` must be unique in the unit specification.

Multiple assignments to disjoint parts of the same variable are now allowed. All these assignments must be of the same kind and in the same clock domain (if applicable).

- Verilog_expression is 4-valued, 2-valued interpretation according to Verilog rules in 2-valued contexts
- Multiple disjoint assignments to bit/part selects of vectors and disjoint members of an array are now supported.
  - If two or more assignments to the same element are detected decidedly to the same element, issue an error. For example, the following results in an error:

```
logic [0:2] a;
a <= 1;
a <= 2;
```

An error also results from the following:

- a[2] <= 1'b0;
  a[2] <= 1'b1;

  When it cannot be determined that the assignments could collide, OVA proceeds without any warning.

  - If two or more assignments occur to the same element of a variable, then the order in which the assignments take place is undetermined. For example:

```
logic [2:0] a[0:9];

a[i][j] <= 1'b0;
a[k][l] <= 1'b1;
```

  - is accepted, but if `i==k` and `j==l,` then the value assigned to the selected bit could be either 0 or 1.

- Must NOT contain `past ended e, matched e` and `posedge" negedge, edge`
- May refer to OVA variables, ports, and bit or part selectsthere of, and to bool instances
- May refer to other "logic" objects, including bit and part select

logic can be declared anywhere. It is the assignment, not the declaration, that determines its meaning.

Continuous-like `logic` Assignment Outside a Clock Domain

logic can be used outside a clock domain as a continuous assignment:

```
logic v;
assign v = expr;
```

This assignment behaves like a continuous assignment in Verilog and *must not have an initial value specification*.

The assignment is specified using `assign` as follows:

```
assign name [[bit or part select]] [[ index]] =
Verilog_expression;
```

An example of an assignment outside a clock domain and assigned using `assign` follows.

```
logic [0:3] b;
     assign b = x & y[3:0]; // x, y are ports or OVA variables
     clock posedge clk { ... }
```

Non-blocking `logic` Assignment Inside a Clock Domain

A non-blocking assignment must be inside a clock domain. An initial value can be assigned in the declaration (which is again outside any clock domain). For example:

```
logic v = 0;
clock posedge clk {
   v <= expr;
```

Where `expr` can contain clock-dependent operators like in (2).

NOTE: The use of var will be still be accepted.

The assignment inside a clock domain is a non-blocking assignment:

```
name [[bit or part select]] [[index]] <=
bit_vector_expression;
```

Continuous-like `logic` Assignment Inside a Clock Domain

A blocking-like assignment. This assignment behaves like a continuous assignment except that expr may contain references to ended, matched, posedge, edge, negedge, past. It cannot have an initial value assignment in the declaration. For example

```
logic v;
clock posedge clk {
   v = expr;
```

- name [[bit or part select]] [[index]] = bit_vector_expression;
- bit_vector_expression is 4-valued, 2-valued interpretation according to Verilog rules in 2-valued contexts
- The RHS expression can contain clock-dependent operators.

- Continuous-like assignment, the behavior is as if assigned using a continuous assignment after any ended, matched and other subexpression have been evaluated, i.e., respecting the evaluation order implied by the dependencies. Similar to the use of “ended”, the variable can only be read in the same clock domain as assigned. It is an error to cross clock domains.
- Multiple disjoint assignments to bit/part selects of vectors and disjoint members of an array are now supported. These assignments are treated in a fashion similar to that described for continuous assignments in terms of what are valid assignments and what are error conditions.
- When assignment and use should be in different clock domains, the non-blocking assignment (similar to the use of “matched”) must be used.
- Can refer to clock dependent operators

The following example also behaves as a continuous assignment, except that the clock-dependent operators are updated on the active clock edge as usual.

```
//b, x, y and z are ports
     logic b1 = 0;
     clock posedge clk {
     event e: x #1 y;
     b1 = (ended e) && z;
     event e1: b1 #1 (b[0:1] == 2’b0);
     }
```

If "b1 <= (matched e) & y;" were used then the value of `b1` seen by `e1` would be delayed by one clock cycle.

OVA Built-in Functions

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(`*`name`* `[,` *`number_of_ticks`*`])`

The identifier name refers to the signal name or to an OVA variable name. 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 signal name that was present number_of_ticks prior to the time of evaluation of ***past***.

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`*`)`

Example:

```
template one_hot(en=1, clk, state, strict=0,
                 msg = "assertion triggered") : {
clock clk {
   event ova_e_one_hot :
   if (en) then
    ((count(state) == 1) || (!strict && (count(state) == 0)));
        }
assert ova_c_one_hot : check( ova_e_one_hot, msg );
}
```

Grouping Assertions as a Macro

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

The syntax for macro groupings is as follows:

```
template name [(formal_param1, ..., formal_paramN)] :
{
  template_body
}
With formal_param  being:
name [= boolean_expr | sequence_expr]
```

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 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 [instance_name] [(actual_param1, ..., actual_paramN)];
```

If a single instance of the template is created the instance name is optional and in that case the internal objects can be referred to from the instantiating context directly by their names without the *instance_name* prefix.

The actual parameters can be given as an ordered list or as a named list. In an ordered list, the parameters are listed in the same order as in the template definition. In a named list, the parameters can be in any order because they are paired with the formal parameters with the following syntax:

```
.formal_param(actual_param)
```

For example, the ova_hold template defined above can be instantiated with:

```
ova_hold ordered(counter, 2, 5, posedge clk);
```

Or it can be instantiated with:

```
ova_hold named(.exp(counter), .min(2), .max(5),
     .clk(posedge clk));
```

Ordered lists are easier if the list is short. Named lists are clearer if the list is long. Named lists might also be easier if there are many optional parameters.

As template instances are expanded, the names of expression definitions and variables declared in the template body are constructed by attaching the template instance name and an underscore character in front of the definition name. 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);
}

unit test #(...) (...);
     range t1();
     range t2();
     clock posedge clk2 {
          event term_chk: if (t1_c1) then p_low #1 p_end;
     }
     assert a_term_chk: check(term_chk);
endunit
```

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

```
unit test #(...) (...);
     clock posedge clk2 {
```

```
          bool t1_c1: enable;
          event t1_crange_en: if (t1_c1) then
               (minval <= expr);
     }
     assert t1_range_chk: check(t1_crange_en);

     clock posedge clk2 {
          bool t2_c1: enable;
          event t2_crange_en: if (t2_c1) then
               (minval <= expr);
     }
     assert t2_range_chk: check(t2_crange_en);

     clock posedge clk2 {
          event term_chk: if (t1_c1) then p_low #1 p_end;
     }
     assert a_term_chk: check(term_chk);
endunit
```

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 in another template of a unit, the unresolved names are bound to the objects within the unit (including ports).

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];
     }
}
unit test #(...) (...);
     ova_hold hold_instance(s, 5, , posedge clk);
```

```
endunit
```

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 <= ended e1 ? data_in : data_store;
          event e1: pre;
          event e2: post;
          event consistent: if (ended 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);
```

For Loops

The syntax for a for loop is as follows:

```
for (name = expr; term_expr; name = incr_expr)
```

```
{
  for_loop_body
}
```

With *term_expr* being:

```
name op1 expr
| expr op1 name
```

The *op1* operator can be: ==, !=, >, >=, <, or <=.

With *incr_expr* being:

```
name op2 expr
```

The *op2* operator can be: + or -.

name is a local variable limited to the for loop. It does not need to be declared outside of the for loop. Within the loop, *name* can also be used as an array index and to specify vector parts.

The value of *name* is initially set to *expr*. If *name* satisfies *term_expr*, the *for_loop_body* is executed. After each execution of the *for_loop_body*, *name* is modified according to the *incr_expr*. The *for_loop_body* is repeated until *name* does not satisfy the *term_expr*.

Since the for-loop is processed during compile-time, all expressions must be constant types with known values during the OVA compile.

The body of the for loop may contain:

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

- unit instances
- template instances
- nested for loops
- bindings

Following is an example of a for loop:

```
for (i = 0; i < no_chnl; i = i + 1) {
  assert resp_cycles[i] : check(ova_e_resp_cycles[i]);
}
```

This loop declares an array of assertions that check response cycles on each of a set of channels. The number of channels is represented by the no_chnl variable, and the for loop generates assertions for each element ova_e_resp_cycles[i] of the array of events.

Unit/template instances can be instantiated using an array of names and parameterized unit/template arguments within a for loop, similar to the currently supported definitions of arrays of events and bools. The template instance name may reference the for loop index, and the actual parameters to the unit/template instances may also reference the for loop index, and use the index values in compile-time evaluatable expressions.

The following is an example of template instantiations within a for loop

```
template my_template_bit (clk, en, inp, outp) : {
   clock posedge clk {
      event e1: if (en) then outp == inp;
      event e_inp_coverage: inp == 1;
   }
   assert c1: check(e1);
}
```

```
unit my_unit (logic clk,
              logic en,
              logic [15:0] inp,
              logic [15:0] outp);

for (i=0; i<16; i=i+1) {
   my_template_bit bitchk[i] (clk, en, inp[i], outp[i]);
}

endunit
```

A top level for loop used to specify an array of unit bindings will also be supported with this enhancement. Nested loops may contain bindings and template instances.

The following is an example of a top level for loop around unit bindings.

```
for (i=1; i <= 16; i=i+1) {
  bind module test: ova_valid_id  my_valid_id[i]
    #(2,3,i,2,i+1,i-1,0,"triggered")
    (1'b1,clk,i*4,issued_id,ret_sig,ret_id,1'b0,2'b00);
}
```

Building Expressions Iteratively

The syntax for building expressions iteratively is as follows:

expr **[[***index***]]** **:** *sequence_expr* **;**

The index is optional. The *index* can be multi-dimensional. The *sequence_expr* can include another *expr*[*index*] that has already been processed.

For example:

```
expr[0] : st[0] == 1;
```

```
for (i = 1; i < max; i = i + 1) {
  expr[i] : expr[i-1] #1 st[i] == 1;
}
clock posedge clk {
  event e : if (st[0] == 1 && en) then expr[max-1];
}
assert : check (e);
```

A

Summary of OpenVera Assertions Features

This appendix provides a quick reference to the syntax of OpenVera Assertions.

In this appendix:

Keywords for Sequence Expressions

Table A-1 summarizes the keywords used to express sequences.

Table A-1 Keywords for Sequence Expressions

Keyword	Description	Examples

#t1	Delays forwards	`#2 r1`
		r1 must be true at the second clock tick after the beginning of expression evaluation.
		`r1 #4 r2`
		r2 must be true at the fourth clock tick after r1 is true.

#[t1..t2]	Expressions with time range specification	`r1 #[3..5] r2` r2 must be true between 3 and 5 clock ticks after r1 is true.

->>	Shorthand for a range of 1 to the simulation's end	`r1 ->> r2`
		r2 must be true some clock ticks after r1 is true. It is the same as:
		`r1 #[1..] r2`

&&	Logical "and" of two expressions. Both expressions must match.	`seqA && seqB` `seqA` must match and `seqB` must match.
||	Logical "or" of two expressions. At least one of the two expressions must match.	`seqA || seqB` At least `seqA` or `seqB` must match. The expression is false if neither of them matches.
if then	Conditional matching of an expression with if-then clause. The condition must be a boolean expression.	`if (e1) then r1` If e1 is true then r1 must be true. Evaluation of e1 and r1 begin at the same time.
if then else	Conditional matching of an expression with if-then-else clause. The condition must be a boolean expression.	`if (e1) then r1 else r2` If e1 is true then r1 is true, otherwise r2 is true. Evaluation of e1, r1 and r2 begin at the same time.
*	An expression must be true for a range of specified number of times	`r1 * [2..4]` r1 must occur any number of times between 2 and 4.

Keyword	Description	Examples
istrue in	Ensure that a boolean expression holds true in a sequence expression	`istrue expr1 in r1` Ensure that expr1 is true all the time while sequence r1 is monitored.
length in	Limit the length of the time period for matches of an expression	`length [2..4] in r1` Ensure that r1 completes within 2 to 4 ticks (2 and 4 inclusive).

Additional Keywords For Edge Expressions

Table A-2 summarizes the keywords used to specify edge expressions.

Table A-2 Keywords for Events

Keyword	Description	Example
posedge	A change in the value of an expression from false to true.	`r1 ->> (posedge r2) ->> r3` r2 must transition from 0 to 1 between r1 and r3.
negedge	A change in the value of an expression from true to false.	`r1 ->> (negedge r2) ->> r3` r2 must transition from 1 to 0 between r1 and r3.
edge	A change in the value of an expression.	`r1 ->> (edge r2) ->> r3` r2 must transition from 1 to 0 or 0 to 1 between r1 and r3. edge is the same as either posedge or negedge.
ended	True if the associated sequence expression (in the same clock domain) matched the trace.l	`p ->> (ended seqA) ->> r2` `seqA` must match between p and r2.
matched	True if the associated event (in a different clock domain) matched the trace.	`p ->> (matched seqA) ->> r2` `seqA` must match between p and r2.

For clock edges, the posedge, negedge, and edge (as “posedge or negedge”) keywords follow Verilog semantics. However, for signals sampled by a clock edge (that is, inside a clock domain specification), edge detection is done by comparing the current and preceding samples of the signal to determine if the signal rose between the two clock ticks (posedge) or fell (negedge) or changed (posedge or negedge).

Edge detection operates on a single bit. In the case of a bitvector, it checks only the least significant bit of the vector. To detect change on a full bitvector, use the past operator. For example, as a boolean:

```
bool b_change(s): past(s) != s;
```

Reserved Template and Unit Names

All names used by templates and units must be unique. Some tools ship with a predefined set of checkers. For example, Synopsys tools ship with the OVA Checker Library (for more information, see the *OpenVera Assertions Checker Library Reference Manual*). Check the documentation for your tool to see the names of these checkers and avoid these names when using the tool.

Syntax of Constructs

The syntax of constructs is described using Backus-Naur Form (BNF).

The following notation is used to specify the syntax:

- ***Bold Italic*** font for terminals

- plain text for non-terminals
- `[ ]` for empty or present
- { } for empty, or any number of times
- Bold **`[ ]`** are language operators
- < > for semantic description of non-terminals

OVA data types are:

Integer Atomic types

- integer

Integer Vector types

- logic

Non-Integer types

- string

Additionally, the integer type modifiers:

- signed
- unsigned

Arrays of vectored types can also be constructed, using normal Verilog syntax. This includes the ability to define multiple dimensional vectors and memories.

```
typed-name::=                          integer arrayed-ident
                                     |  real arrayed-ident
                                     |  vector-name
vector-name::=                             [ signed ] logic [ range ]
arrayed-ident
range::=                                   [ msb_constant_expr :
lsb_constant_expr ]
```

```
arrayed-ident::=                                        identifier
                                                      |     identifier dimension {
dimension }
dimension::=                                          [ dim_constant_expr :
dim_constant_expr ]
```

All the types apply to boolean expressions with the semantics defined in the Verilog 2001 LRM

```
file ::= {declaration}
declaration ::=
        unit_spec
        | bind_spec
        | comment
        | template name [( formal_parameter_list )]: { {template_body} }
unit_spec ::=                                 unit identifier [ params ] ( port-
declarations );
                                                      entity
                                              endunit
params ::= #( parameter_decl { , parameter_decl } )
parameter_decl ::= parameter [ integer | real | string ]

list_of_param_assignments ;
list_of_param_assignments ::=
                                                      param_assignment { ,
param_assignment }
param_assignment ::= param_identifier = constant_expr
port-declarations ::= port-declaration { ,  port-declarations }
port-declaration ::= typed-name
                                                                    |     output [ range ]
identifier
bind_spec::= module_binding | instance_binding
module_binding ::= bind module module_name : unit_instance ;
instance_binding ::= bind instances instance_list :

unit_instance ;
instance_list ::= instance  { , module_instance_list }
```

instance ::= fully_qualified_instance_name

```
unit_instance ::=
        unit_name [ unit_instance_name ]
                                                      [ parameter-bindings ] [ ( port-
bindings ) ]
|       unit_name [ parameter-bindings ]
                                                      [ unit_instance_name ] [ ( port-
bindings ) ]
parameter-bindings ::= #( list-of-param-bindings )
port-bindings ::= port-binding { , port-bindings }
port-binding ::= by_name_binding | plain-binding
list-of-param-bindings ::= param-binding { , param-binding }
```

```
param-binding ::= by_name_binding | plain-binding
by_name_binding ::= .formal-ident ( plain-binding )
plain-binding ::=                                       constant
                                                  |     hdl_signal_name
                                                  |     hdl_signal_name vector-
range
                                                  |     <empty>
template_body ::=
        clock edge_expr { {non_scoped_decl} }
        | directive
        | non_scoped_decl
entity ::=
        clocked_entity
        | unclocked_entity
clocked_entity ::=
        clock edge_expr { {unclocked_entity} }
        | directive
        | <clocked template_instance> [name] [( argument_list )]
        | <clocked unit_instance>
unclocked_entity ::=
                             non_scoped_decl
                         | <unclocked template_instance> [name] [( argument_list
)]
                         | <unclocked unit_instance>
non_scoped_decl ::=
        expr_defn
        | expr_instance
        | var_spec
        | var_assign
        | comment
        | expr_constructor
expr_constructor ::= name [index] : sequence_expr ;
index ::= [ expr ] [index]
argument_list ::=
        formula_expr {, formula_expr}
comment ::= <any Verilog style comment and any nested c language style comments>
var_spec ::=
        var [[int:int]] name [ range ];
var_assign ::=
        init var_name_ref = const_expression;
        | var_name_ref <= bit_vector_expr;
const_expr ::= <Verilog compile-time constant bitvector or
                                                        v2k constant array
expression>
var_name_ref ::= <Verilog conventions for referencing bit, part, and word select>
directive ::=
        assert name [ index ]:
                             builtin_formula (sequence_expr [, [message]
                                                                        [,
[severity] [, category]]]);
message ::= <constant string enclosed between “”>
```

```
severity ::= int
category ::= int
builtin_formula ::=
        check
        |forbid
expr_defn ::=
        event name [(parameter_list)]: sequence_expr;
        | bool name [(parameter_list)]: boolean_expr;
        | event name [index]: sequence_expr;
        | bool name [index]: boolean_expr;
parameter_list ::= name {, name}
formal_parameter_list::=
        formal_parameter {, formal_parameter}
formal_parameter ::=
        name
        | name = boolean_expr
        | name = sequence_expr
template_instance ::=
        name [ name ] [(actual_parameter_list)];
actual_parameter_list ::=
        actual_parameter {, actual_parameter }
actual_parameter ::=
        /* empty */
        | name
        | boolean_expr
        | sequence_expr
formula_expr ::=
        sequence_expr
        | <name of any expr_defn>
        | ( formula_expr )
sequence_expr :: =
        boolean_expr
        | <name of event from expr_defn> [(actual_parameter_list)]
        | ( sequence_expr )
        | sequence_expr && sequence_expr
        | sequence_expr || sequence_expr
        | [sequence_expr] time_shift sequence_expr
        | if boolean_expr then sequence_expr [else sequence_expr]
        | sequence_expr * [ non_neg_int ]
        | sequence_expr * int_interval
        | cond_spec_list in sequence_expr
        | any
cond_spec_list ::=
        cond_spec  { , cond_spec }
cond_spec ::=
        istrue boolean_expr
        | length [ non_neg_int ]
        | length int_interval
```

```
time_shift ::=
        # non_neg_int
        | # int_interval
        | ->>
int_interval ::=
        [ non_neg_int .. non_neg_int]
        | [ non_neg_int .. ]
boolean_expr ::= <casting of bit_vector_expr to width 1>
bit_vector_expr ::=
        <name of bool from expr_defn>[(actual_parameter_list)]
        | ( bit_vector_expr )
        | count ( bit_vector_expr )
        | past ( bit_vector_expr [, non_neg_int ] )
        | any
        | edge_expr
        | <Verilog boolean expression, including variables>
edge_expr ::=
        ( edge_expr )
        | posedge bit_vector_expr
        | negedge bit_vector_expr
        | edge bit_vector_expr
        | matched <clocked sequence_expr>
int ::= <integer compile time expression>
non_neg_int ::= <non-negative integer compile time expression>
for_loop ::= for ( name = expr ; term_expr ; name = incr_expr )
                                                        { for_loop_body }
term_expr ::= name op1 expr
                                    | expr op1 name
incr_expr ::= name op2 expr
op1 ::=
                        ==
                    | !=
                    | >
                    | >=
                    | <
                    | <=
op2 ::=
                        +
                    | -
for_loop_body ::=
                        /* empty */
                    | for_loop_body for_loop_stmt
for_loop_statement ::=
                        template_instance
                    | expr_constructor
                    | var_spec
                    | var_assign
                    | directive
                    | for_loop
```

There are items in this BNF that are referred to as clocked or unclocked. Following is the meaning of these items:

- `clocked template_instance` refers to a template that is allowed, although not required, to contain an explicit clock declaration.
- `clocked unit_instance` refers to a unit that is allowed, although not required, to contain an explicit clock declaration.
- `unclocked template_instance` may not contain an explicit clock declaration in the template specification.
- `unclocked unit_instance` may not contain an explicit clock declaration in the unit specification.
- `clocked_entity` is allowed, although not required, to contain an explicit clock declaration in the entity specification.
- `unclocked_entity` may not contain an explicit clock declaration in the entity specification.
- clocked sequence expressions are resolved to a clock prior to their usage in the construction of the enclosing clause.

Verilog Compiler Directives

This section describes Verilog compiler directives that are supported in this language. Like Verilog, all compiler directives are preceded by the (`) character. The scope of any compiler directive extends from the point where it is specified, across all files processed, to the point where another compiler directive supersedes it or the end of processing.

The syntactic description for the compiler directives in BNF notation is presented below. For details, please refer to the IEEE standard 1364-1995.

```
compiler_directives::=
     include_directive
     | conditional_directive
     | macro_directive
     | undefine_directive
include_directive::=
     `include  "filename"
conditional_directive::=
     `ifdef text_macro_name
     first_group_of_lines
     [ `else
          second_group_of_lines
     ]
     `endif
macro_directive::=
     `define text_macro_name  macro_text
text_macro_name::=
     text_macro_identifier [(list_of_formal_arguments)]
list_of_formal_arguments::=
     formal_argument_identifier
          {,formal_argument_identifier}
text_macro_usage::=
     `text_macro_identifier[(list_of_actual_arguments)]
list_of_actual_arguments::+
     actual_argument {,actual_argument}
actual_argument::=
     verilog_expression
undefine_directive::=
     `undef text_macro_name
```

Table A-3 briefly describes the purpose of the compiler directives and specifies any differences from the Verilog language definition of them.

Table A-3 Verilog Compiler Directives

Compiler Directive	Description	Difference from Verilog
`` `ifdef `` , `` `else `` and `` `endif ``	These are conditional compilation directives, which include optional lines of a specification during processing of the input file.	None
`` `define `` and `` `undef ``	The `` `define `` compiler directive provides a text macro substitution feature, which defines a text macro with parameters for substitution. A previously defined macro can be undefined using the `` `undef `` compiler directive.	None
`` `include ``	This compiler directive provides a file inclusion feature, where the contents of another file with specification can be inserted in a file during processing.	None

Lexical Conventions

This section describes the lexical conventions used in this language. The lexical conventions are similar to the Verilog HDL lexical conventions. The differences from the Verilog conventions are identified in Table A-4. Please refer to the IEEE standard 1364-1995 for the details of the Verilog conventions.

Table A-4 Lexical Conventions

Lexical Convention	Difference from Verilog
Lexical tokens	None
White Space	None
Comments	None
Operators	None
Numbers	Real constants are not allowed
Identifiers, keywords and system names	System calls are not allowed

Lexical Convention	Difference from Verilog
Escaped identifiers	None
Compiler directives	None
`v'` escaped identifiers	To resolve conflicts between Verilog objects and event names

Data Types

Verilog and VHDL provide sets of data types to represent storage and manipulation. This language accesses the data types to evaluate expressions. Table A-5 lists the supported Verilog data types and any deviations from Verilog. Table A-6 lists the supported VHDL data types and any deviations from VHDL.

Table A-5 Verilog Data Types

Data Type	Description	Difference from Verilog
Value set	0, 1, x, z	None
Nets, regs and integers	Data types declared are `wire` or `reg`. These are accessed in expressions.	None
Vector specification	Range specification for multibit net or `reg`	None
Wire strengths	Small, medium and large	Not supported
Net types	`wire`, `wand`, `wor`, `tri`, `triand`, `trior`, `tri0`, `tri1`, `trireg`, `supply0`, `supply1`	None
Memories	Array of regs	Not supported
The register types `integer`, `real`, `time` and `realtime`	Different data types used for behavioral modeling	Not supported
Parameters	Constants	None
Event declarations		Not supported

Table A-6 VHDL Data Types

Data Type	Difference from VHDL
Name space	Supports scope. Scopes containing generated blocks are not supported. Does not support entity or architecture.
Boolean	None
bit	None
bit_vector	None
std_logic and std_logic_vector	Nine-state value is mapped to four-state Verilog value.
SIGNED and UNSIGNED	Both are treated as unsigned.
Integer	Not supported
Enumerate	Not supported
1D array	Array range must always be specified and must use OVA syntax. For example, to refer to X(0 to 7), use X[0:7].
2D array	Not supported
Record	Not supported

Support for multidimensional arrays from the design and other V2k enhancements in boolean expressions, compliant with IEEE Standard 1364-2001 and VCS implementation, includes:

- Indexed part selects by variables using [base_expr +: width_expr] and [base_expr -: width_expr]
- Signed variables in less-than/greater-than comparisons and signed constants
- Multidimensional array support
  - Selecting a subarray, word, part, and bit
  - Index can be an MDA element
- Exponentiation operation **
- Extending ‘bz to full-size word in assignments to free variables and in comparisons

- Arithmetic shift operation (<<<, >>>)

Compatibility with Verilog Logical Expression

A logical expression, as intended in this document, is a Verilog logical expression that results in true or false as defined by the semantics of the Verilog language.

Table A-7 lists the supported operands for expressions, and any deviations from the Verilog language usage.

Table A-7 Supported Verilog Operands

Operand	Description	Deviation from Verilog
Constant number	integers, strings, and real	Real constants are not supported
Net	All wire types	None
Register	Variable declared as reg	None
Part-select and bit-select for registers and nets	Part-select selects a range of bits, bit-select selects a bit	None
Integer, time, real and realtime	Data storage types	Not supported
Memory word	Array of register	None
Function and task	A call to a user-defined or system function or task	Not supported

All operators from Verilog are supported as shown in Table A-8.

Table A-8 Verilog Operators

Operator	Description	Operator	Description
{ }, {{ }}	Concatenation, replication	|	Bit-wise inclusive or
+ - * / **	Arithmetic	^	Bit-wise exclusive or

Operator	Description	Operator	Description
%	Modulus	^~ or ~^	Bit-wise equivalence
> >= < <=	Relational	&	Reduction and
!	Logical negation	|	Reduction or
&&	Logical and	~|	Reduction nor
||	Logical or	^	Reduction exclusive or
==	Logical equality	~^ or ^~	Reduction xnor
!=	Logical inequality	<<	Left shift
===	Case equality	>>	Right shift
!==	Case inequality	<<<	Arithmetic left shift
~	Bit-wise negation	>>>	Arithmetic right shift
&	Bit-wise and	?:	Conditional

Table A-9 shows the precedence of binary operators and the conditional operator (?:). Operators shown on the same row have the same precedence. Rows are arranged in order of decreasing precedence for the operators. All operators associate left to right with the exception of the conditional operator, which associates right to left.

Table A-9 Precedence of Binary Operators

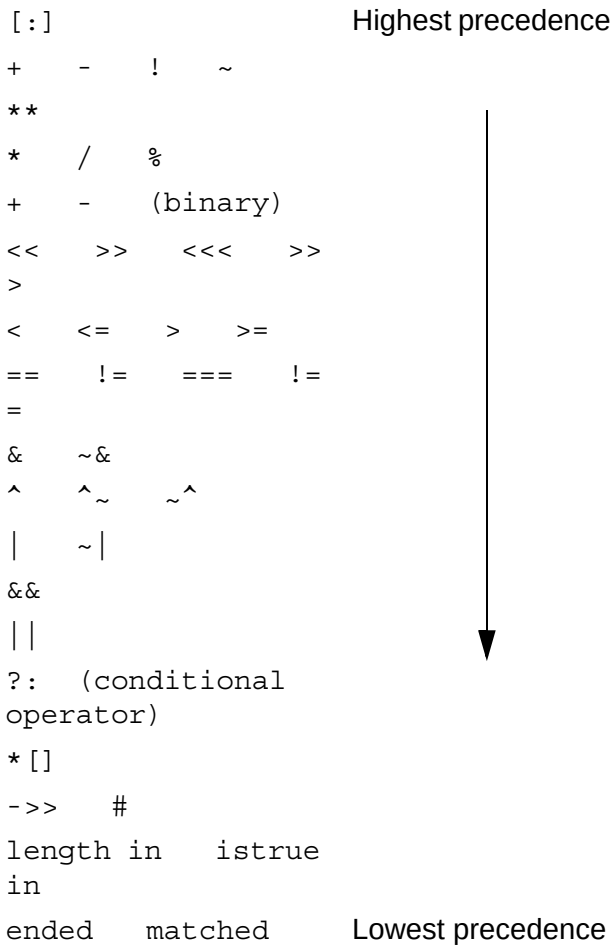


Table A-10 summarizes other semantic rules regarding the usage of operators and expression evaluation.

Table A-10 Other Semantic Rules for Operators

Description	Difference from Verilog
Using integer constants in expression	None
Expression evaluation order	None
Arithmetic operators	None
Arithmetic expressions with registers and integers	None
Relational, equality and logical operators	None

Description	Difference from Verilog
Bit-wise, reduction, shift and conditional operators	None
Concatenation	None
Bit-select and part-select evaluation	None
Event or	Not supported
String operations	Not supported
Delay expressions	Not supported
Rules for expression bit lengths	None

B

Formal Analysis Subset

OpenVera Assertions (OVA) are targeted for application in both dynamic simulation and formal property verification. The following are noted exceptions that define the formal analysis subset of OVA.

Simtime Clock

All events and OVA variables be clocked by an edge of a signal in the design. Simulation-time clocking is not supported. The following is an example of an unsupported unclocked specification:

```
unit unclocked (logic a, logic b);
     event e1: a #2 b;
     assert a1: check (e1);
endunit
```

e1 is an unclocked event, a1 is an unclocked assertion

Boolean Operators Not Supported in Formal Tools

Some boolean operators are not supported in formal tools, such as case equality, and comparisons to X and Z.

Example: the `!==` operator is not supported in formal tools.

```
unit case_equality (logic clk, logic [3:0] bus1,
                    logic [3:0] bus2);
     assert a1: check (e1);
     clock posedge clk {
          event e1: bus2 !== bus1;
          }
endunit
```

Posedge and Negedge Semantics

Formal tools do not recognize transitions to or from an X or Z state when evaluating `posedge` and `negedge` operators. This applies to both clock and sampled signal transitions.

Multidimensional Array References

Multidimensional array references may not be supported in formal tools, in particular, if the whole array needs to be passed through ports to the generated RTL modules. For example:

```
unit MDA_check1 (logic clk, logic enable, logic [7:0]
                  A[0..9][0..9], logic [3:0] index1,
                   logic [3:0] index2);
         clock posedge clk {
              event e1: enable && A[index1][index2] == 0;
         }
         assert a1: forbid(e1);
```

```
endunit
```

Index

Symbols

A

B

C