Accellera SystemVerilog Assertions
Design Working Group - Working Proposal (Rev0.6.5)
10/29/02

Section 11 Assertions

This is a working proposal for SystemVerilog Assertions. There are still a number of issues to be settled, but this is an attempt to document what has been agreed to by the DWG. There are a number of outsdanding issues still to be decided, including some of the syntax. In some cases, a proposed syntax is shown to illustrate the semantic concepts on which agreement has been reached.

11.1 Introduction (informative)

An assertion is a statement that a property must be true. There are two kinds of assertions: concurrent or declarative assertions, which state that the property must be always be true, e.g. throughout a simulation, and procedural assertions which are incorporated in procedural code and apply only for a limited time or under limited conditions. There are various applications of assertions. They can be included in the design, to document the assumptions made by the designer and to facilitate “white box” testing, or they can be outside the design, either in a testbench to check the response of the design to the stimulus, or to control a tool such as a stimulus generator or a model checker.

Declarative assertions can be coded as modules in a library, but this limits the complexity of the property that can be expressed easily. It is more difficult to code procedural assertions as a library of tasks in Verilog, because events cannot be arguments, each assertion may need static data, and tasks block. SystemVerilog allows both declarative and procedural assertions to be coded directly, or to be encapsulated as templates allowing assertions to be specified once and reused.

11.1.1 Discussion of sampled assertions vs. immediate assertions

One of the goals of SystemVerilog assertions is to provide a common semantic meaning for assertions so that they may be used to drive various design and verification tools. Many tools, such as formal verification tools, evaluate circuit descriptions using a cycle-based semantic which typically relies on a clock signal or signals to drive the evaluation of the circuit. Any timing or event behavior between clock edges is abstracted away. While this approach generally simplifies the evaluation of a circuit description, there are a number of scenarios under which this cycle-based evaluation provides different behavior from the standard event-based evaluation of SystemVerilog described elsewhere in this document.

In an effort to minimize these discrepancies and provide a common assertion semantic for both event- and cycle-based tools, this chapter describes an evaluation semantic for assertions that relies on *sampled* values of signals relative to clock edges. While it is not possible to eliminate completely the evaluation differences between event- and cycle-based tools, this sampled semantic for assertions provides the greatest area of overlap between the two, and also allows for semantic equivalence between declarative and procedural assertions.

11.2 Immediate assertions

The immediate assert statement is a test of an expression performed when the statement is executed in the procedural code. The expression is treated as a condition like in an if statement.

[identifier **:**] **assert (** expression **)** [pass_statement] [**else** fail_statement]

NOTE: We may need to change the assert keyword to property. I thought that “property” could denote sampled semantics and “assert” could just be immediate.

Copyright 2002 Accellera. All rights reserved.

The pass statement is executed if the assertion succeeds, i.e. the expression evaluates to true. As with the if statement, if the expression evaluates to 'X', 'Z' or '0', then the assertion fails. The pass statement may, for example, record the number of successes for a coverage log, but may be omitted altogether. If the pass statement is omitted, then no action is taken if the assert expression is true. The fail statement is executed if the assertion fails (i.e. the expression does not evaluate to a known, non-zero value) and can be omitted. The optional assertion label (identifier and colon) creates a notional named block around the assertion statement (or any other SystemVerilog statement) and can be displayed using the `%m` format code.

```
assert_foo : assert (foo) $display("%m passed"); else $display("%m failed");
```

Since the assertion is a statement that something must be true, the failure of an assertion shall have a severity associated with it. By default, the severity of an assertion failure is "error". Other severity levels may be specified by including one of the following severity system tasks in the fail statement.

— **`$fatal`** is a run-time Fatal, which terminates the simulation with an error code. The first argument passed to $fatal shall be consistent with the argument to $finish.

— **`$error`** is a Run-time Error.

— **`$warning`** is a Run-time Warning, which can be suppressed in a tool-specific manner.

— **`$info`** indicates that the assertion failure carries no specific severity.

The syntax for these system tasks is shown in section 16.4.

All of these severity system tasks shall print a tool-specific message indicating the severity of the failure, and specific information about the specific failure, which shall include the following information:

— The file name and line number of the assertion statement,

— The hierarchical name of the assertion, if it is labeled, or the scope of the assertion if it is not labeled.

For simulation tools, these tasks shall also include the simulation run-time at which the severity system task is called.

Each system task can also include additional user-specified information using the same format as the Verilog **`$display`**.

If more than one of these system tasks is included in the **`else`** clause, then each shall be executed as specified.

If an assertion fails and no **`else`** clause is specified, the tool shall, by default, call **`$error`**, unless a tool-specific command-line option is enabled to suppress the failure.

If the severity system task is executed at a time other than when the assertion fails, the actual failure time of the assertion can be recorded and displayed programmatically. For example:

```
time t;

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

If the assertion fails at time 10, the error message will be printed at time 15, but the user-defined string printed will be "assert failed at time 10".

The display of messages of warning and info types can be controlled by a tool-specific command-line option.

Copyright 2002 Accellera. All rights reserved.

Since the fail statement, like the pass statement, is any legal SystemVerilog procedural statement, it can also be used to signal a failure to another part of the testbench.

```
assert (myfunc(a,b)) count1 = count + 1; else ->event1;
assert (y == 0); else flag = 1;
```

The assert statement serves as guidance to non-simulation tools that the condition should be true. The second statement above is equivalent to:

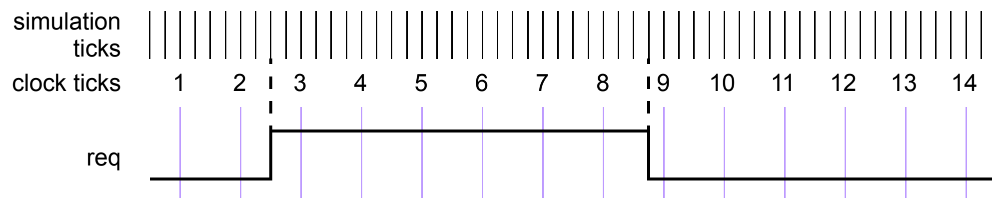
```
if ( y!=0) begin flag = 1; end
```

11.3 Sampled Semantics for Assertions

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

A clock tick is an atomic moment in time and implies that there is no duration of time in a clock tick. It is also given that a clock may tick only once at any simulation time. The value of a variable in an expression at a clock tick is sampled at the end of the simulation timestep (i.e. at read-only synchronization time, as defined by the PLI) immediately before the clock tick. In an assertion, the sampled value is the only valid value of a variable at a clock tick. <hotlink>Figure 11-1 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 11-1—Sampling a Variable on Simulation Ticks



For accessing the value of a SystemVerilog variable at a simulation tick, 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 sampled value of a signal with respect to its clock is the value of the variable at the end (i.e. read-only sync) of the simulation tick immediately before the clock event occurs.**

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 expressions (such as rose and fell) or boolean sub-expressions that are required to determine a match with respect to a sequence expression.

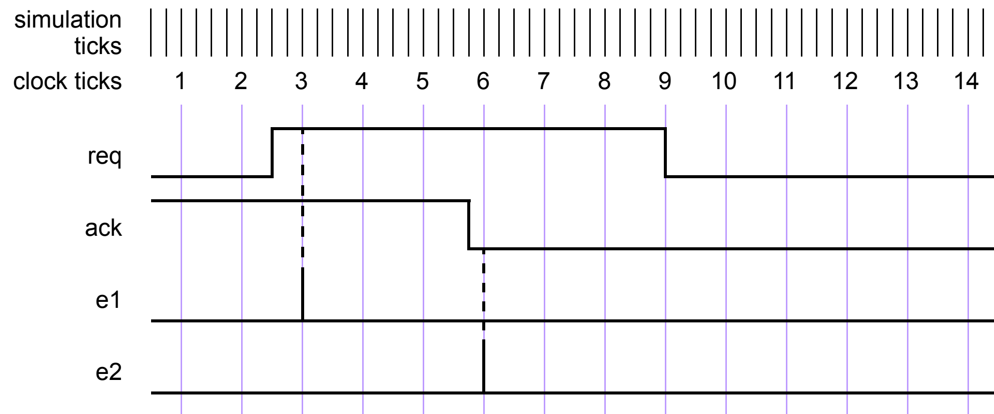
An edge expression at a clock tick changes the value of an expression from the value of that expression at the previous clock tick. Like boolean expressions, an edge expression evaluates to true if the event occurs, and to false if the event does not occur.

For example, when a signal changes its value from low to high (a rising edge), it is considered a rose event. <hotlink>Figure 11-2 illustrates two examples of edge events:

- edge expression `e1` is defined as `(rose req)`
- edge expression `e2` is defined as `(fell ack)`

Copyright 2002 Accellera. All rights reserved.

Figure 11-2—Edge Expressions



The clock used for sampling the events is different than the simulation ticks. Assume, for now, that this clock is defined in this language elsewhere. At clock tick 3, edge `e1` occurs because the value of `req` at clock tick 2 was low and at clock tick 3, the value is high. Similarly, edge `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 <hotlink>Figure 11-2, without an arrow on the top or the bottom of the bar indicates an occurrence of an edge expression.

In order to ensure proper behavior of the system and conform as closely as possible to truly cycle-based semantics, the sample event or clock signal used to control evaluation of sequences must be glitch-free and may only transition once at any simulation time. In addition, any combinational logic driving expressions that are evaluated on the sample event must stabilize before the sample event occurs. If the sample event is qualified using the "iff" event qualifier, then the qualifier expression must also stabilize before the sample event occurs.

The clock expression that controls evaluation of a sequence may be more complex than just a single signal name. An expression such as (clk && gate) could be used to represent a gated clock. Other more complex expressions are possible. However, such clock expressions must be evaluated with zero delay.

11.4 Declaring Sample Events

To declare sample events on which to sample the signals in a property block, the standard event declaration syntax is extended:

event_declaration ::= **event** list_of_event_identifiers [**=(**<event_expression>**)**]**;**

NOTE: The event assignment is required for the event to be used in a property, but the standard Verilog syntax makes it optional.

Events declared with an explicit event expression, as in

```
event myclk = (posedge clk1);
```

may not be explicitly triggered from elsewhere in the code:

```
always @(posedge clk2)
   if(foo)
      -> myclk; // illegal
```

11.5 Sequences

A sequence is a list of SystemVerilog boolean expressions in a linear order of increasing time. These boolean

Copyright 2002 Accellera. All rights reserved.

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.To determine a match of a sequence, the boolean expressions are evaluated at each successive sample point to satisfy the sequence. If all expressions are true, then a match of the sequence occurs.

A sequence expression describes one or more sequences by using *regular expressions* that concisely specify a range of possibilities of and repetitions of sequences. These sequential regular expressions can actually describe a set of one or more sequences that satisfy the sequential expression.

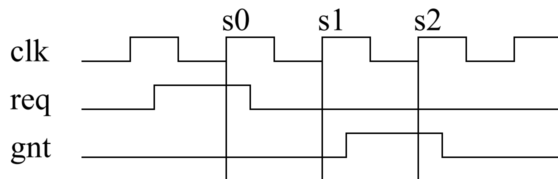
A sequential regular expression is a semicolon-delimited list of boolean expressions, each of which is evaluated on successive sample events. Thus, the sequence

```
req;gnt;!req
```

specifies that **req** be true on the current sample, **gnt** will be true on the first subsequent sample and **req** will be false on the next sample after that. In other words, the ‘;’ operator specifies a one-sample delay between expressions. To specify more than a one-sample delay, the number of samples to delay is prepended to the next expression in the sequence, as in

```
req;[2]gnt
req;;gnt // alternate notation: note double semicolon
```

This specifies that req will be true on the current sample, and **gnt** will be true on the second subsequence sample, as shown in figure 1



By extension, it follows that the following two sequential expressions are equivalent to each other:

```
a;b
a;[1]b
```

Therefore,

```
req;;gnt
```

is also equivalent to

```
req;[1]1;gnt
```

Since the *true* value is implicit, this is also equivalent to

```
req;[1];gnt
```

In other words, the bracketed value can be used between semicolons to indicate “skip the specified number of samples.” To specify that ‘b’ will be true on the Nth sample after ‘a’, the following two sequences are equivalent:

```
a;[N]b      // check b on the Nth sample
a;[N-1];b   // skip N-1 samples, then sample b on the next sample
```

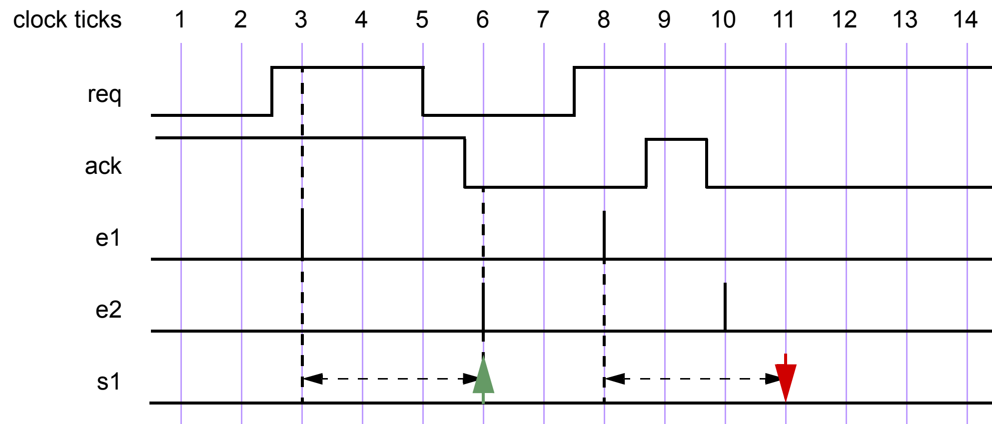
11.5.1 Matching a Sequence

For example, consider the sequence of boolean expressions, `s1,` in <hotlink>Figure 11-22. `s1` is defined as:

```
e1 ;[3] e2
```


Copyright 2002 Accellera. All rights reserved.

Figure 11-3—Matching a Sequence



The above example says that e2 is expected to occur at the third clock tick after the occurrence of e1. <hotlink>Figure 11-22 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. Expression `e2` occurs at clock tick 6, so the sequence 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.

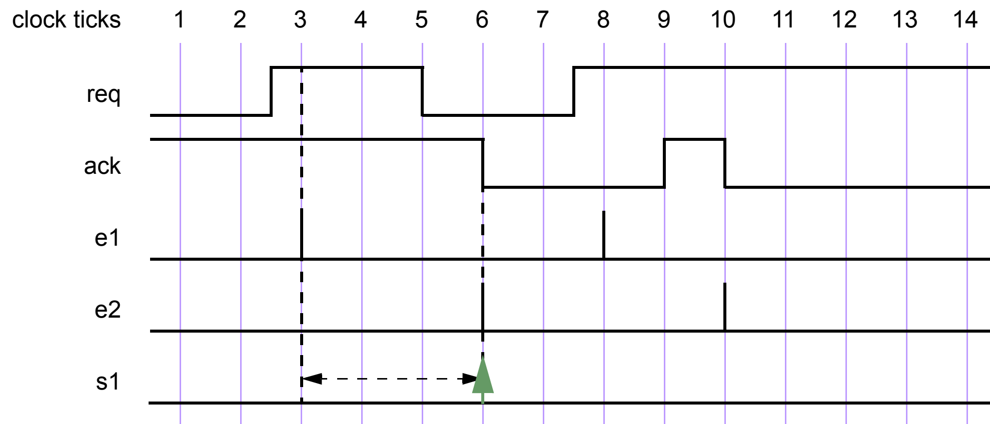
11.5.2 Start and End Time of A Sequence

Each sequence has a start time and an end time. As seen from the examples in <hotlink>Figure 11-22 on page 11-65 and <hotlink>Figure 11-23 on page 11-66, 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 <hotlink>Figure 11-24. 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.

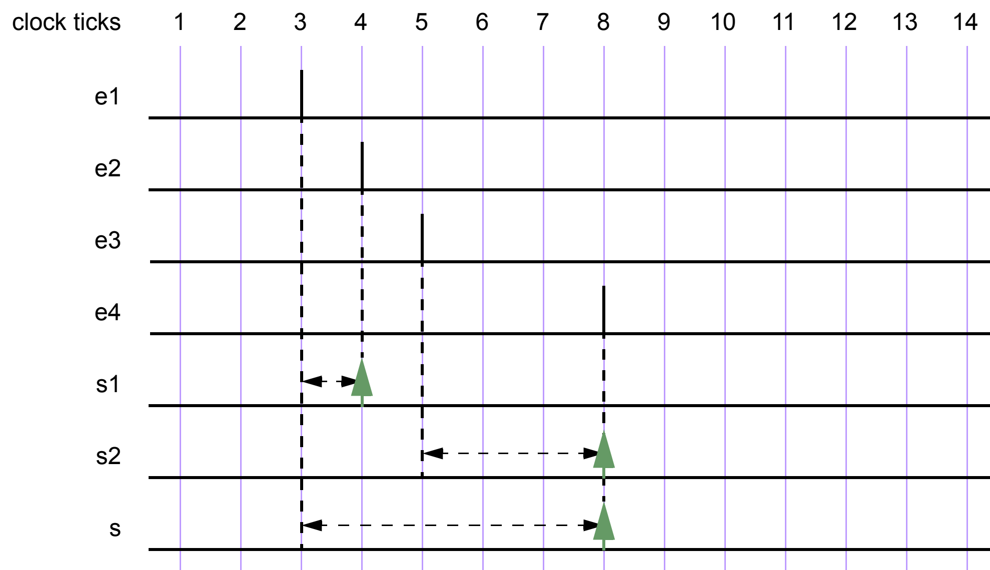
Copyright 2002 Accellera. All rights reserved.

Figure 11-4—Start and End Times of a Sequence



A sequence can consist of sub-sequences, again dispersed in time. The same rules apply to sub-sequences regarding the start time and end time. Now, assume a series of expressions (`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 ; e2)` and s2 is `(e3 ;[3] e4)`, and s is defined as `(s1 ; s2)`, and shown in <hotlink>Figure 11-25 on page 11-67. The time clause `;` 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 11-5—Start and End Times of Sub-sequences



The sequence expression is:

```
(e1 ; e2) ; (e3 ;[3] e4)
```

Let us examine the evaluation attempt at clock tick 3 in <hotlink>Figure 11-25.

- The attempt starting at clock tick 3 succeeds for sub-sequence s1 at clock tick 4.

Copyright 2002 Accellera. All rights reserved.

- Next, 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 expression e4 occurs, resulting in the end time for sub-sequence s2 as clock tick 8.

11.5.3 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.

To specify a range of possible delays between subsequent samples, the delay specifier is modified to use the standard range syntax, as in

```
a;[1:3]b;c
```

This specifies that a will be true on the current sample, followed by b on the first, second, or third subsequent sample, and c will be true on the sample following b. Thus, any of the following sequences will match this sequential expression:

```
a;b;c
a;1;b;c
a;1;1;b;c
```

In the range syntax,

```
a;[min:max]b
c;[min:max];d
```

both min and max must be a constant expression or a literal, min must be greater than or equal to zero, and max must be greater than or equal to min.

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)
```

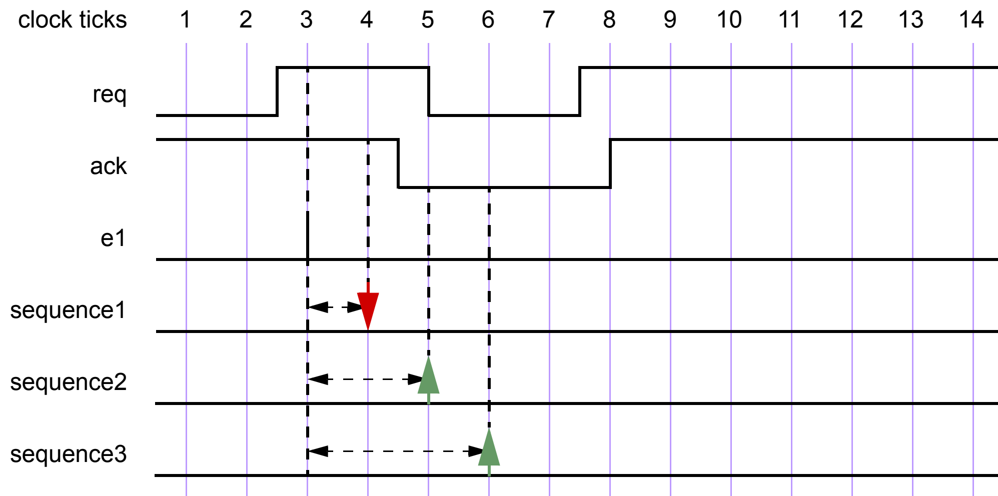
Event e1 is defined as (rose req).

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 <hotlink>Figure 11-23. The three sequences are:

```
e1 ; (ack==0)
e1 ;[2] (ack==0)
e1 ;[3] (ack==0)
```

Copyright 2002 Accellera. All rights reserved.

Figure 11-6—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.

Note: When a sequence expression is used in a *property* directive that is to be verified on the design, the first match establishes the success of the property. In the example in <hotlink>Figure 11-23 a property verifying
`e1 ;[1:3] (ack==0)`
would declare success at clock tick 5.

11.5.4 Repetition in Sequences

To specify the repetition of a boolean expression within a sequence, the boolean may simply be repeated, as:

```
a;b;b;b;c
```

or the number of repetitions may be specified with a trailing “*[N]”, as:

```
a;b*[3];c
```

If it can be repeated a minimum or maximum number of times, this can be expressed with a trailing *[min:max]. These repetition counts must also be literals or constant expressions.

```
(a; b)*[5]     // a;b;a;b;a;b;a;b;a;b
(a*[0:3];b;c)  // equivalent to(b;c) or (a;b;c) or (a;a;b;c) or (a;a;a;b;c).
```

This means that a sequence `a;ab;a;b;c;` will pass. The expression sequence is not equivalent to `((a && !b)* [0:3];b;c)`, which would fail the same sequence.

The rules for specifying repeat counts are summarized as:

— Each form of repeat count specifies a minimum and maximum number of occurrences

— expr*[n:m], where n is the minimum, m is the maximum

— expr*[n] is the same as expr*[n:n]

— [n] is the same as 1*[n:n]

Copyright 2002 Accellera. All rights reserved.

— The sum of the minimum repeat counts for all terms in a sequence must be greater than 0

— The sequence as a whole cannot be empty

— The last term in a sequence shall not have a min:max range of repetition. If it does, it shall be an error.

In addition, the keyword **inf** is introduced to specify a potentially infinite maximum number of repetitions. So,

```
a;b*[1:inf];c
```

means ‘a’ is true on the current sample, then ‘b’ will be true on every subsequent sample until ‘c’ is true. On the sample in which ‘c’ is true, ‘b’ does not have to be true.

The “*[N]” notation indicates consectutive repetition of an expression. It is also possible to specify non-consecutive repetition of a *boolean* expression with

```
a;b*=[min:max];c
```

This is equivalent to

```
a;((!b*[0:inf];b))*[min:max];!b*[0:inf]);c // NOTE: I think this is fixed.
```

This non-consecutive repetition operator allows sequences to specify that a boolean expression does not occur:

```
b*=[0] // b repeated 0 times non-cosecutively means that b does not occur
```

Therefore, both of the following sequences match the sequence a;c

```
a;b*=[0];c
a;b*[0];c
```

Adding the range specification to this allows the construction of useful sequences containing a boolean expression that is true for at most N samples:

```
a;b*=[0:N];c // a followed by at most N occurences of b, followed by c
```

The delay and repetion syntax for sequences is summarized in the following table:

Table 11-1Sequence Concatenation and Repetition Summary

Expression	Meaning	Comment
a;b	‘a’ followed by ‘b’ on the next clock tick	
a;[N]b	‘a’ followed by ‘b’ on the Nth clock tick	
a;[N];b	‘a’ followed by N consecutive samples of “true”, followed by ‘b’ on the N+1st clock tick	a;[1];b is equivalent to a;;b
a;[N:M]b	‘a’ followed by ‘b’ on any of the Nth to Mth clock ticks	
a;b*[N];c	‘a’ followed by N consecutive samples of ‘b’, followed immediately by ‘c’	
a;b*[N:M];c	‘a’ followed by at least N and at most M consecutive samples of ‘b’, followed immediately by ‘c’.	
a;b*=[N];c	‘a’ followed by N non-consecutive samples of ‘b’, followed by ‘c’.	‘b’ occurs N times between ‘a’ and ‘c’. ‘c’ may or may not immediately follow the Nth occurence of ‘b’.
a;b*=[N:M];c	‘a’ followed by at least N and and most M non-consecutive samples of ‘b’, followed by ‘c’.	

Copyright 2002 Accellera. All rights reserved.

Table 11-1Sequence Concatenation and Repetition Summary

Expression	Meaning	Comment
a;b*=[0];c	'a' followed by zero occurences of 'b', followed by 'c'.	'b' does not occur between 'a' and 'c'.
a;b*[1:inf];c	'a' followed by an infinite number of samples of 'b', followed by 'c'.	Only fails in simulation if 'b' does not occur after 'a'.

Note that the ';' delay operator has higher precedence than the repetition operators, so

```
a;b*[2]
```

matches a;b;b, whereas

```
(a;b)*[2]
```

matches a;b;a;b. Also note that the repetition operators associate left-to-right, so

```
a;b*[2]*=[3];c
```

means that the sequence "b;b" is repeated 3 times between 'a' and 'c'. This is equivalent to

```
a;(b;b)*=[3];c
```

11.6 Declaring Sequences

Sequences can be reused by declaring them as objects of type seq:

seq_decl ::= **seq** @(<event_expression> | <event_identifier>) *name* [(<list_of_port_connections>)] =
(<sequential_expression>) [{,<sequential_expression>}];

The declaration can optionally include arguments that allow the same sequence to be instantiated multiple times with different argument values.

Note that, as with bools, variables referenced within a seq that are not formal arguments to the sequence are resolved hierarchically from the scope in which the seq is instantiated.

```
event clkev = (posedge clk);
seq @clkev s1 = (a;b;c), s2 = (d;e;f);
seq @(nededge clk) s3 = (g;h;i);
```

In this example, sequences s1 and s2 are sampled on each successive posedge clk. The sequence s3 is sampled on negedge clk.

11.7 Sequence Operations

11.7.1 AND operation

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

sequence_and ::= *sequence_expr* **and** *sequence_expr*

The two operands of ***and*** are sequence expressions. The requirement for the success of the ***and*** 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 last.

For the expression:

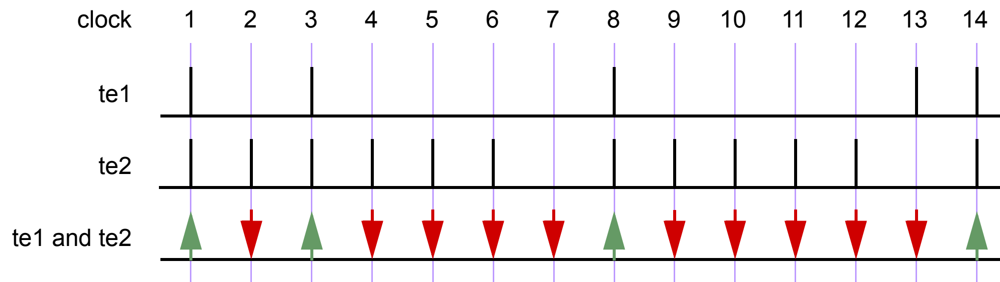
```
te1 and te2
```


Copyright 2002 Accellera. All rights reserved.

If `te1` and `te2` are sampled booleans (not sequences), the expression succeeds if `te1` and `te2` are both evaluated to be true.

An example is illustrated in <hotlink>Figure 11-7 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, the **and** operation fails because either `te1` or `te2` is false.

Figure 11-7—ANDing (and) Two Sequences



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

```
te1 and te2
```

— Succeeds if `te1` and `te2` succeed.

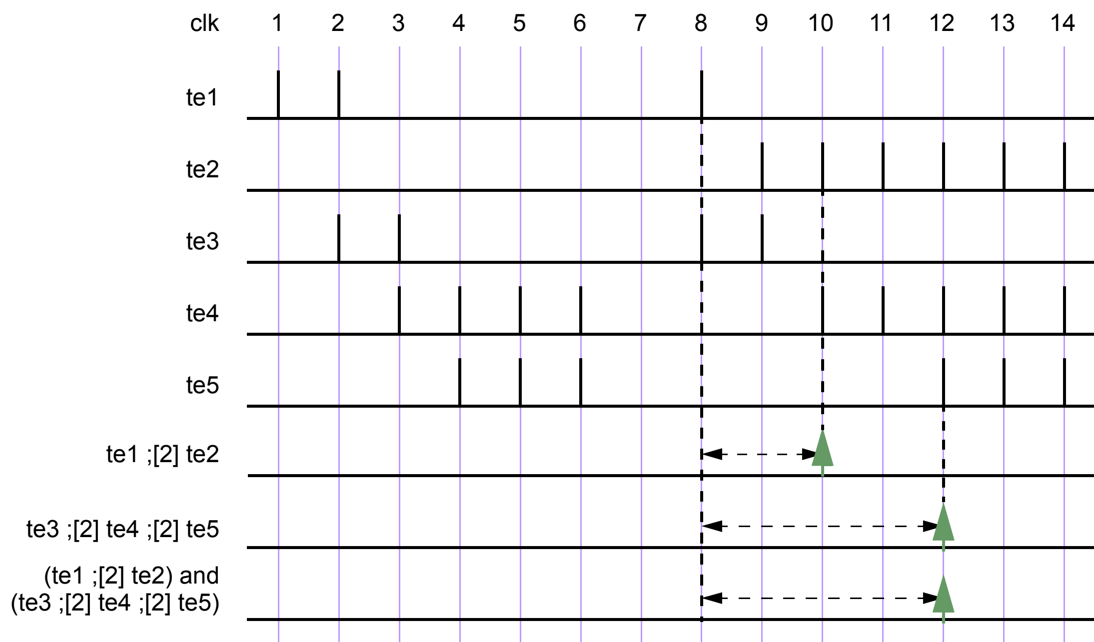
— The end time is the end time of either `te1` or `te2`, whichever terminates last.

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

An example is illustrated in <hotlink>Figure 11-8. Consider the following expression with operator ***and*** where the two operands are sequences.

```
(te1 ;[2] te2) and (te3 ;[2] te4 ;[2] te5)
```

Figure 11-8—ANDing (and) 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. <hotlink>Figure 11-8 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:

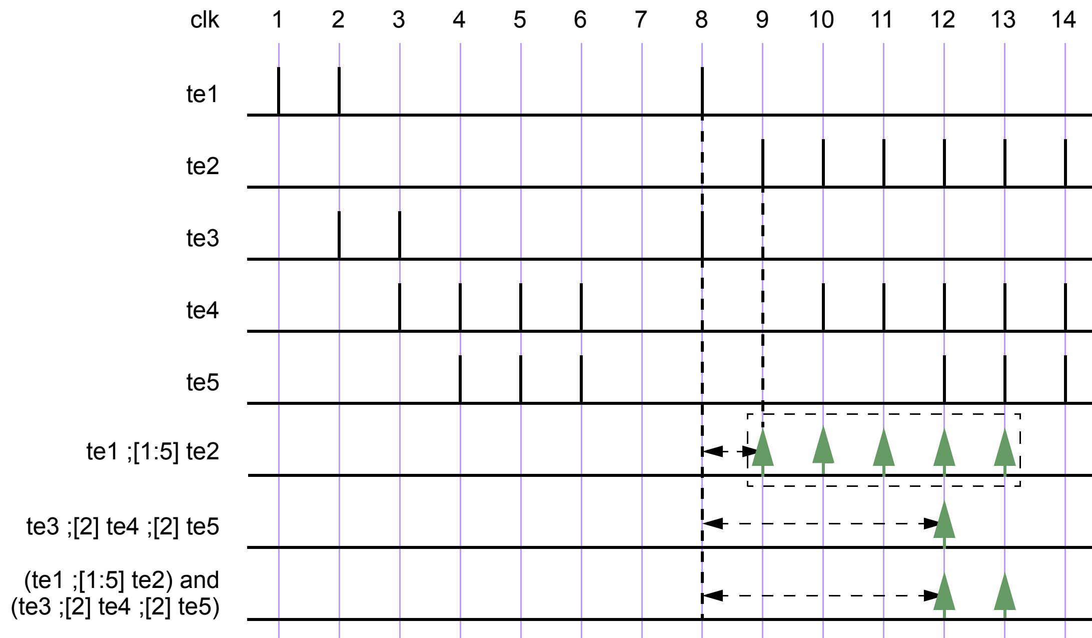
```
(te1 ;[1:5] te2) and (te3 ;[2] te4 ;[2] te5)
```

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.
- <hotlink>Figure 11-9 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 ***and*** 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. <hotlink>Figure 11-9 shows the two unique successes at clock ticks 12 and 13.

Figure 11-9—ANDing (and) Two Sequences Including a Time Range



Copyright 2002 Accellera. All rights reserved.

11.7.2 Intersection (AND with length restriction)

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

sequence_intersect ::= *sequence_expr* **intersect** *sequence_expr*

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

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

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

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

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

11.7.3 OR operation

The operator ***or*** is used when at least one of the two operand sequences is expected to match.

sequence_or ::= *sequence_expr* **or** *sequence_expr*

The two operands of ***or*** are sequence expressions.

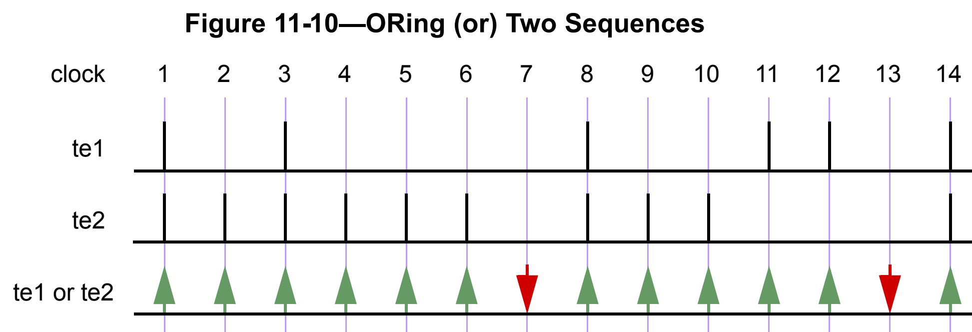
Let us consider these operand expressions as values, events and sequences separately to illustrate the details of ***or*** operations. For the expression

```
te1 or te2
```

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

<hotlink>Figure 11-10 illustrates ***or*** operation using `te1` and `te2` as simple values. The expression does not match at clock ticks 7 and 13 because `te1` and `te2` are both false at those times. At all other times, the expression matches, as at least one of the two operands is true.

Figure 11-10—ORing (or) Two Sequences



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

```
te1 or te2
```

matches if at least one of the two operand sequences `te1` and `te2` match. To evaluate this expression, first,

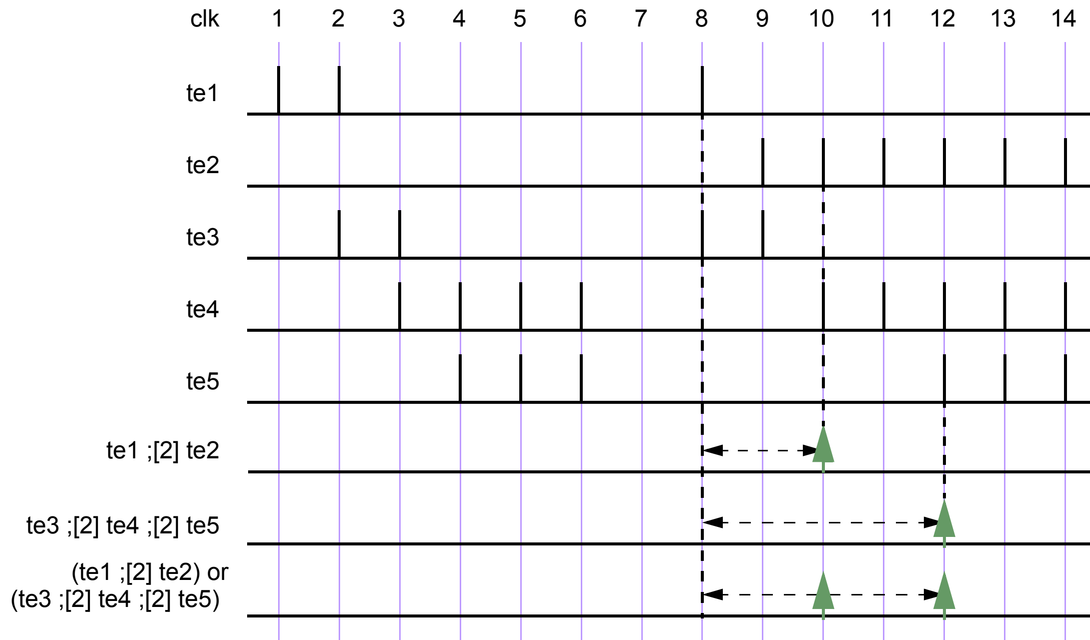
Copyright 2002 Accellera. All rights reserved.

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 <hotlink>Figure 11-11. Consider an expression with ***or*** operator where the two operands are sequences.

```
    (te1 ;[2] te2) or (te3 ;[2] te4 ;[2] te5)
```

Figure 11-11—ORing (or) 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 <hotlink>Figure 11-11, 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.

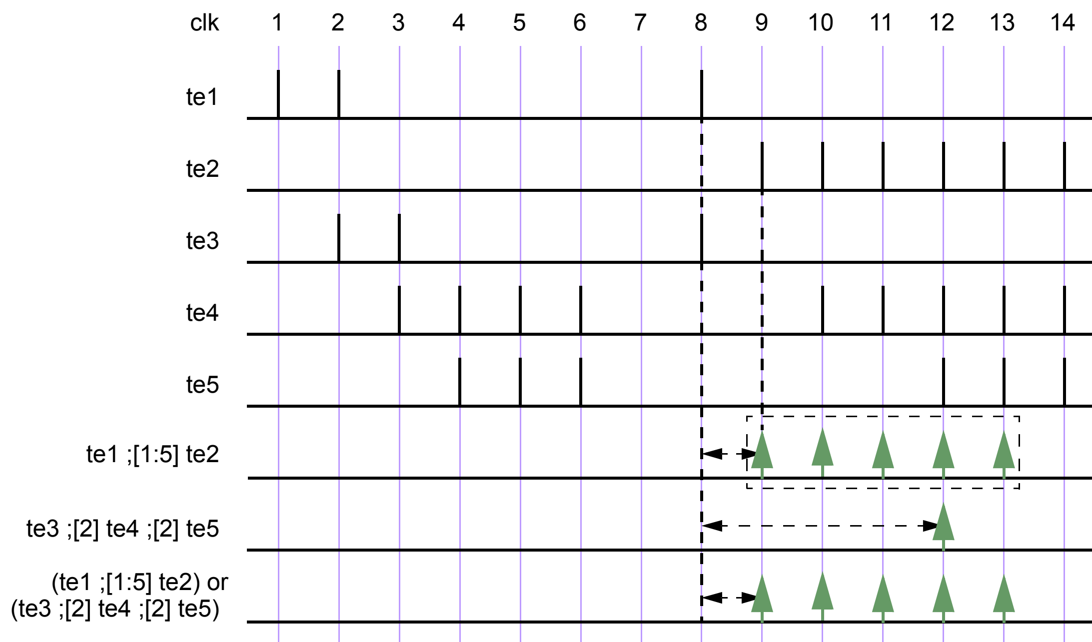
Consider an example where an operand sequence is associated with time range specification, such as:

```
    (te1 ;[1:5] te2) or (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 <hotlink>Figure 11-12, 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.

Copyright 2002 Accellera. All rights reserved.

Figure 11-12—ORing (or) Two Sequences Including a Time Range



11.7.4 first_match operation

Use the ***first_match*** operator when you are interested in only the first match from a sequence expression that can possibly result in multiple matches. This allows you to discard all subsequent matches from consideration. In particular, when the sequence expression is a sub-expression of a larger expression, then applying the ***first_match*** operator has significant effect on the evaluation of the embedding expression.

sequence_first_match ::= **first_match(** *sequence_expr* **)**

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

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

```
seq t1 = ( te1 ;[2:5] te2 );
seq ts1 = ( first_match(te1 ;[2:5] te2) );
```

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

```
te1 ;[2] te2, te1 ;[3] te2, te1 ;[4] te2, te1 ;[5] te2
```

However, sequence ts1 can result in a match for only one of the above four sequences. Whichever of the above four sequences matches first becomes the result of sequence ts1.

Recall also that properties declare a success on the first match of the underlying sequence expression.

The **first_match** operator can be used with the non-consecutive repetition operator as follows:

```
a;first_match(b*=[N]);c; // c occurs immediately after the Nth b
```

Copyright 2002 Accellera. All rights reserved.

11.7.5 Conditional sequences

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) sequence_expr
```

This clause is used to precondition monitoring of a sequence expression. The condition `boolean_cond` must be satisfied in order to monitor `sequence_expr`. If the condition `boolean_cond` fails then `sequence_expr` is skipped for monitoring. `boolean_cond` is a logical expression that results in true or false, and `sequence_expr` is a sequence expression that can result in one or match sequence matches. *If the **boolean_cond** evaluates to true, then the first element of the sequence_expr is evaluated on the same clock tick.*

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

```
matched clocked_sequence_expr.
```

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

```
if (boolean_cond) sequence_expr1 else sequence_expr2
```

This clause is used to select a sequence expression between two alternatives. If `boolean_cond` evaluates 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-else*** depends on the match of the sequence expression, `sequence_expr1` or `sequence_expr2`, whichever gets selected for monitoring.

Clauses ***if*** and ***if-else*** can be nested to contain another conditional sequence within it, such as:

```
if (!reset)
   if (data_phase) ([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)`.

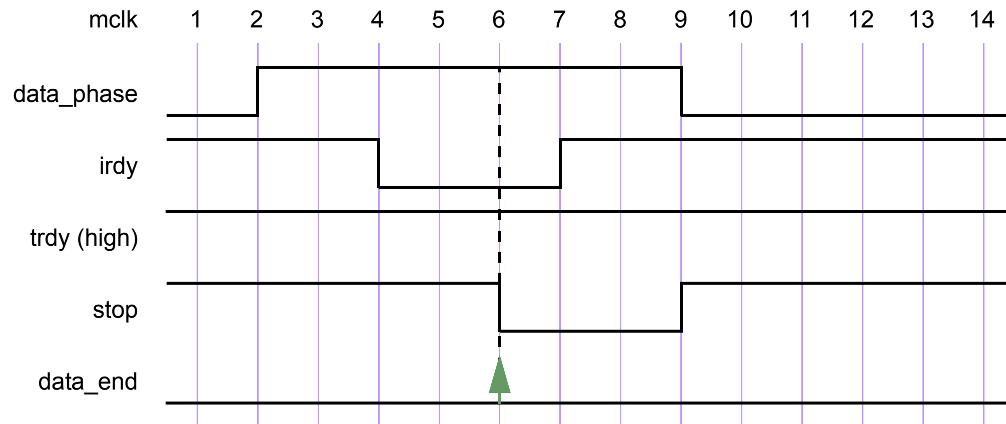
The semantics of ***if-else*** and ***if*** specifications is next illustrated by examples. Consider a bus operation for data transfer from a master to a target device. When the bus enters a data transfer phase, multiple data phases can occur to transfer a block of data. During the data transfer phase, 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:

```
seq @(posedge mclk) data_end = ( if (data_phase) ((irdy==0) && (fell trdy ||
fell stop)) );
```

Each time a data phase completes, a match for `data_end` is recognized. The attempt at clock tick 6 is illustrated in <hotlink>Figure 11-13. 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.

Copyright 2002 Accellera. All rights reserved.

Figure 11-13—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.

```
seq @(posedge mclk) data_end = ( data_phase &&   ((irdy==0) && (fell trdy ||
fell stop)) ),
data_end_rule1 =( if (ended data_end1) ([1:2] rose frame ; rose irdy) );
```

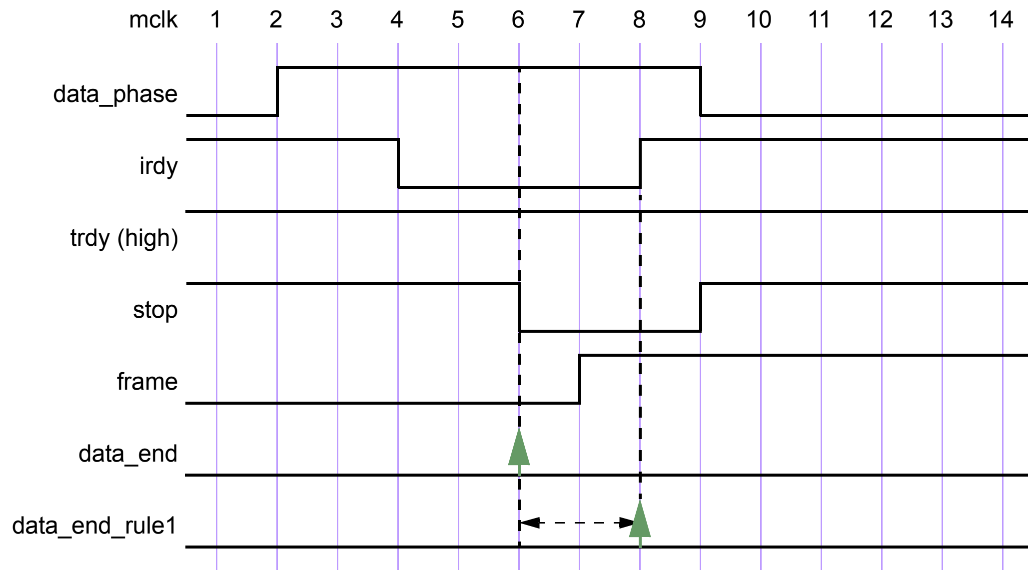
`seq data_end_rule1` first evaluates `data_end` at every clock tick to test if its value is true. If the value is false, then that particular attempt to check the assertion is considered a success. Otherwise, the sequence expression following the ***if*** clause is monitored. The sequence expression

```
[1:2] rose frame ; rose 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 <hotlink>Figure 11-14. Sequence `data_end` is acknowledged at clock tick 6. Next, `frame` toggles high at clock tick 7. Since this falls within the timing constraint imposed by `[1:2]`, it satisfies the sequence and continues to monitor further. At clock tick 8, `irdy` is evaluated. Signal `irdy` transitions to high at clock tick 8, satisfying the sequence specification completely for the attempt that began at clock tick 6.

Copyright 2002 Accellera. All rights reserved.

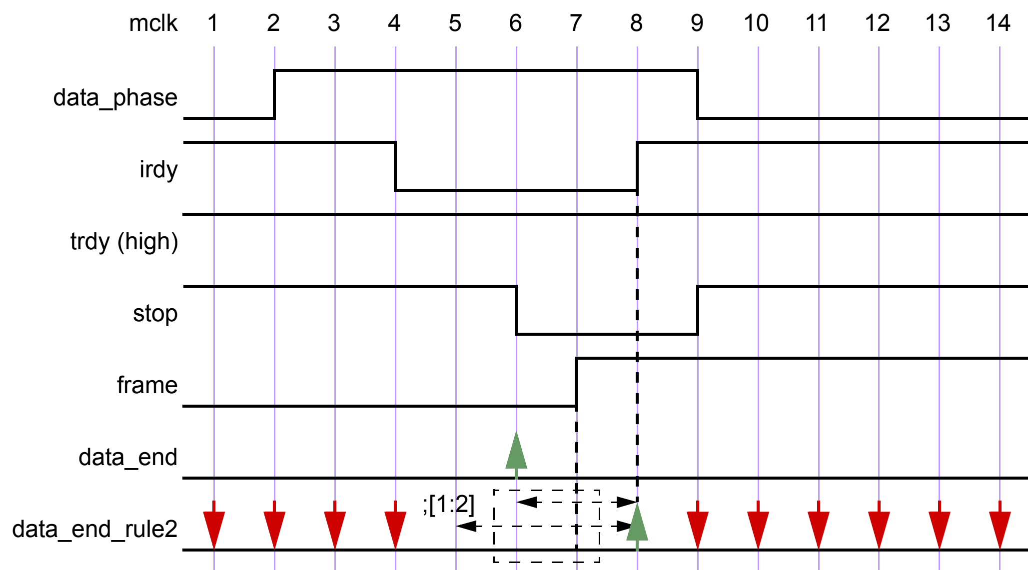
Figure 11-14—Nested Conditional Sequences



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 <hotlink>Figure 11-15.

```
seq @(posedge mclk) data_end_rule2 = ( [1:2] rose frame ; rose irdy );
```

Figure 11-15—Results without the Condition



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

Copyright 2002 Accellera. All rights reserved.

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 `rose frame` does not occur again. That also means that there is no match at 5, 6 and 7.

As one can see from <hotlink>Figure 11-15, removing the precondition of checking event `data_end` from the assertion causes failures that are not relevant to the verification objective. It becomes important from the validation standpoint to determine these preconditions and use them in the assertion to filter out inappropriate or extraneous situations.

11.7.6 Conditions over sequences

NOTE: Concept agreed, but syntax still open.

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. Also frequently, occurrence of certain events is prohibited while processing a transaction. Such situations can be expressed directly using the following construct:

```
istrue boolean_expr within sequence_expr
```

`boolean_expr` is an expression which must evaluate true at every clock tick while monitoring `sequence_expr`. If a sequence for `sequence_expr` starts at time t1 and ends at time t2, then `boolean_expr` must hold true from time t1 to t2. If either the sequence expression does not match or the boolean expression becomes false while the sequence is being evaluated, the composite sequence does not match and a property stated over this composite sequence would declare a failure.

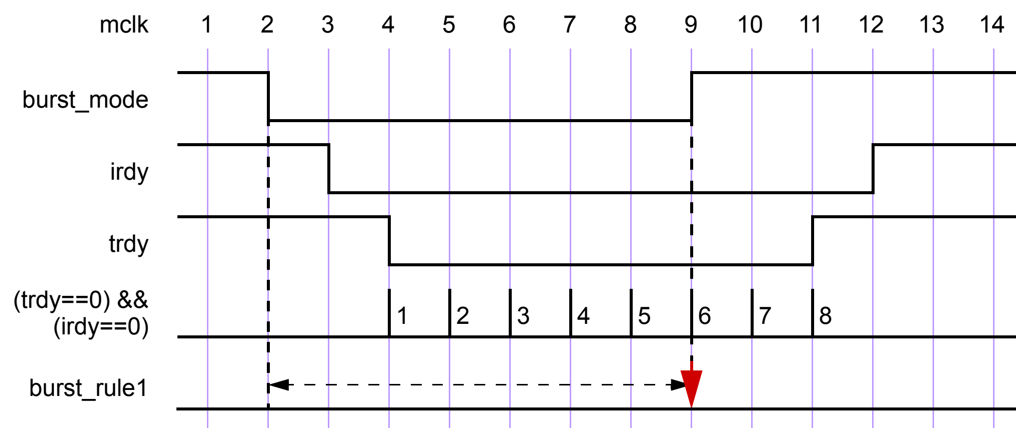
The *istrue* construct is an abbreviation for writing

```
(boolean_expr) *[0:inf] intersect sequence_expr
```

Consider the example illustrated in <hotlink>Figure 11-16. If an additional constraint were placed on the expression as shown below, then the checker `burst_rule` would fail at clock tick 9.

```
seq @(posedge mclk) burst1 = ( if (fell burst_mode)
            istrue (!burst_mode) within  ([2] ((trdy==0)&&(irdy==0)) * [7]) );
burst_rule1: property(burst1);
```

Figure 11-16—Match with istrue-within Restriction Fails



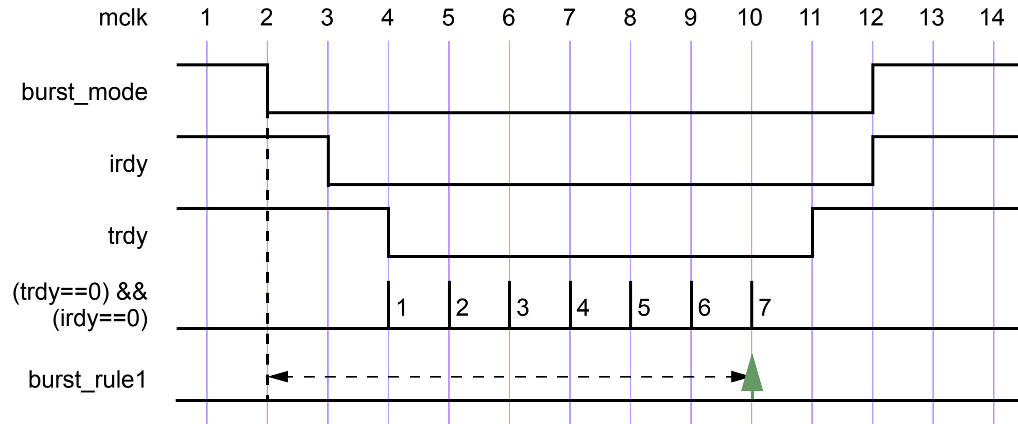
In the above expression, the value of signal `burst_mode` is required to be low during the sequence (from clock tick 2 to 11), and is checked at every clock tick during that period. At clock ticks from 2 to 8, signal `burst_mode` remains low and matches the expression at those clock ticks. At clock tick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

Copyright 2002 Accellera. All rights reserved.

as shown in <hotlink>Figure 11-17.

Figure 11-17—Match with istrue-within Restriction Succeeds

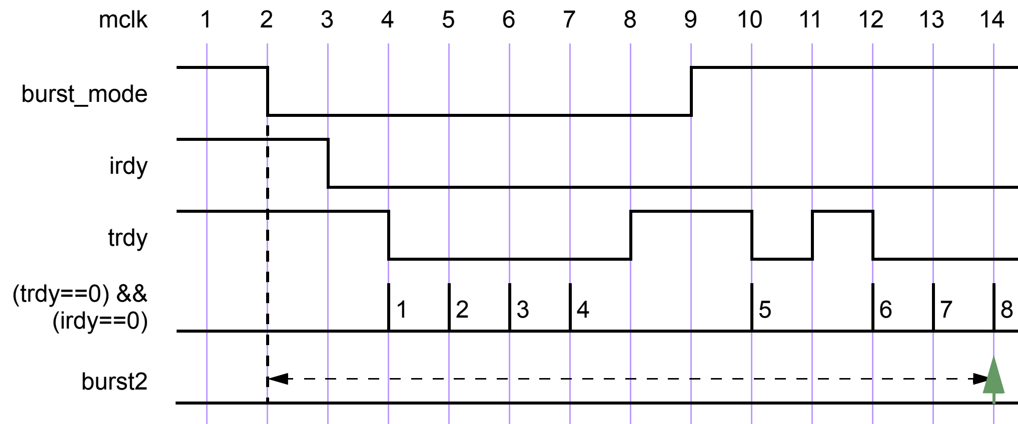


Let us consider a modified version of the example in <hotlink>Figure 11-17 as shown below.

```
seq @(posedge mclk) burst2 = ( if (fell burst_mode)
                                  ([0:4] (trdy==0) && (irdy==0)) * [8] );
```

The sequence `burst2` has been relaxed to require each repetition of the sequence to occur between 1 and 4 clock ticks after the preceding occurrence of the sequence. This is illustrated in <hotlink>Figure 11-18 on page 11-62.

Figure 11-18—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.

11.7.7 Sequence with length specification

NOTE: Concept agreed, but syntax still open.

Another type of restiction over sequences is the total length of the sequence in terms of the clock ticks. For example, a transaction must complete within a given period of time, no matter what variation of commands are issued in the transaction to be processed.

Copyright 2002 Accellera. All rights reserved.

```
length int within sequence_expr
length int_interval within 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 end with a match 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 must end with a match anytime within a time period determined by the minimum and maximum number of clock ticks.

The *length* construct is an abbreviation for writing

```
1*[int] intersect sequence_expr
```

or

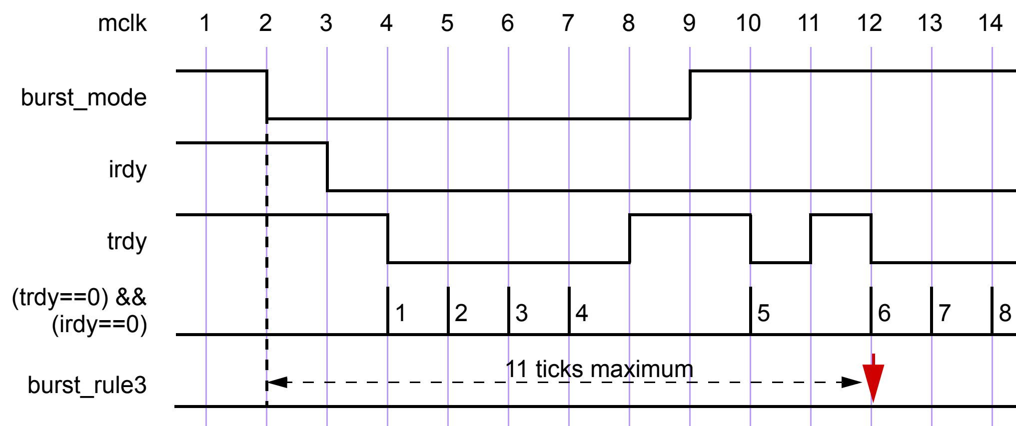
```
1*[int_interval] intersect sequence_expr
```

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 12.

```
seq @(posedge mclk) burst3 = ( if (fell burst_mode)
              length [9:11] within (;[1:4] ((trdy==0)&&(irdy==0))*[8]) );
burst_rule3: property(burst3);
```

In the above expression, the total length of the entire repeated sequence must not be less than 9 clock ticks and not greater than 11 clock ticks. This restriction is expressed by `length [9:11]` in the expression. From <hotlink>Figure 11-19 on page 11-63, 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 12.

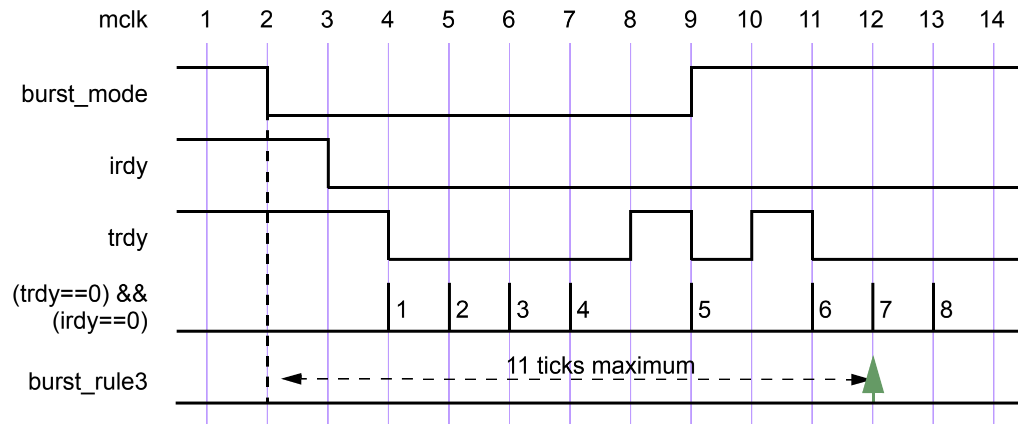
Figure 11-19—Match with length-within Restriction Fails



The failure is corrected by reducing the delay for the fifth repetition from 2 clock ticks to 1 clock tick. This is shown in <hotlink>Figure 11-20.

Copyright 2002 Accellera. All rights reserved.

Figure 11-20—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.

```
seq @(posedge mclk) burst4 = ( if (fell burst_mode)
      (istrue(!burst_mode), length [9:11]) within
      (;[1:4] ((trdy==0)&&(irdy==0))*[8]) );
burst_rule4: property(burst4);
```

Now the two constraints are:

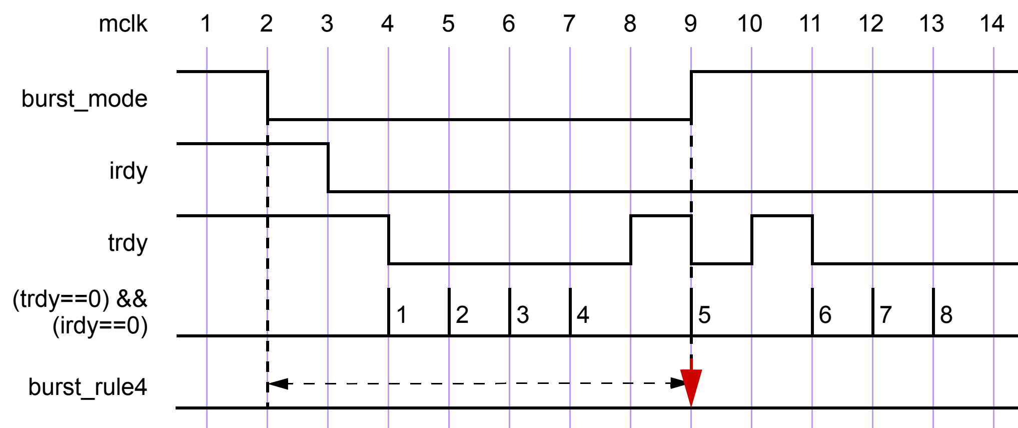
`istrue(!burst_mode)` to ensure that signal `burst_mode` remains low

`length [9:11]` to ensure that the sequence takes at least 9 clock ticks and completes in at most 11 clock ticks

Both constrains 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 <hotlink>Figure 11-21 on page 11-64 because signal `burst_mode` becomes high at clock tick 9.

Figure 11-21—Match with Two Restrictions Fails

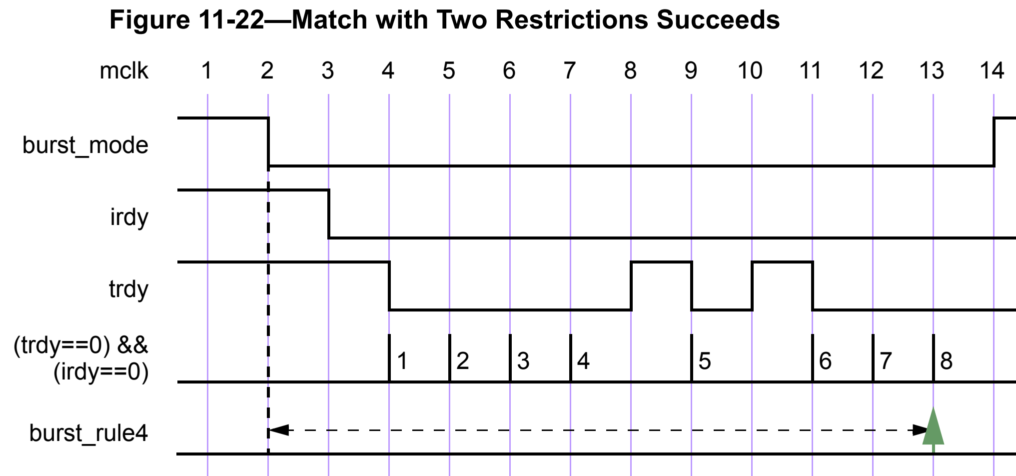


The failure is corrected by maintaining signal `burst_mode` to low value throughout the sequence and by

Copyright 2002 Accellera. All rights reserved.

allowing the lenghth to be in the range of 9 to 12 as shown in <hotlink>Figure 11-22 on page 11-65. 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 11-22—Match with Two Restrictions Succeeds



11.7.8 Sequence occurence within another sequence

The containment of a sequence expression within another sequence is expressed as

sequence_within ::= sequence_*expr1* **within** *sequence_expr2*

The sequence `sequence_expr1` must occur entirely within the sequence *`sequence_expr2`*.

That is `sequence_expr1` must satisfy the following:

- The start point of `sequence_expr1` must be between the start point and the end point of *`sequence_expr2`*
- The end point of `sequence_expr1` must be between the start point and the end point of *`sequence_expr2`*

11.8 Declaring Booleans

Because sequences are composed of boolean expressions, it is useful to allow boolean expressions to be declared as objects of type **bool**.

bool_decl ::= **bool** <identifier>[(<arg>{,<arg>})] = <boolean_expression>;

The boolean object can then be declared as:

```
bool b1(a,b) = a && b && c;
```

and used in a sequence as:

```
(b1(foo,bar);c;d)
(b1(.a(f1),.b(b1));c;d)
```

Note that, in the boolean expression b1, the formal arguments ‘a’ and ‘b’ are replaced by the corresponding actual arguments when the bool is instantiated. Any variables referenced within the bool that are not formal arguments get resolved via standard rules from the scope in which the bool is instantiated.

Copyright 2002 Accellera. All rights reserved.

11.9 Instantiating Properties

NOTE: Concept agreed, but syntax still open.

11.9.1 Declarative Assertions

The property block can be used within a module or interface directly.

```
module top(input bit clk);
reg a,b,c;
property @(posedge clk) (if(a) then (b;c));
...
endmodule
```

In this declarative context, the default *prop_directive* is **globally**, which means that on every occurence of the sample expression, the property is evaluated.

11.9.2 Procedural Assertions

The property block can be instantiated directly in a procedural block as in:

```
always @(posedge clk) begin
   <statements>;
   property @(posedge clk) (a;b;c);
   <statements>;
   end
```

When a property is instantiated in a procedural block, the sample event may be specified explicitly, or it may be inherited from the event expression that governs the procedural block in which it is placed. The above example is equivalent to:

```
always @(posedge clk) begin
   <statements>;
   property (a;b;c);
   <statements>;
   end
```

When a property is instantiated in a procedural block, the default *prop_directive* is **once**, which means that for each time the property statement is executed, the property will be checked starting at that time. It is illegal to specify the prop_directive **globally** for a procedural assertion.

In determining the execution of a procedural assertion, it is necessary to distinguish between the *execution of the assertion statement* and the *evaluation of the assertion*. As with any procedural statement, an assertion statement is executed when the flow of control reaches the statement.

If the execution of the assertion statement is caused by a triggering of the sampling event, and the assertion statement is executed at the same time as the sampling event, then the assertion is evaluated immediately. As described in section 11.3, the boolean or sequential expression in the assertion is evaluated using the sampled value from the previous simulation time. *Any other expressions that affect the control flow of the block in which the assertion statement appears are also evaluated using the sampled value.*

Consider:

```
always @(posedge clk)
  case(st)
  IDLE:
    if (cond)
      P1: property (a;b;c);
...
```

In this example, the sampled values of st and cond are used to determine whether the property statement actually gets executed. This allows the procedural property statement to infer the triggering condition from its placement within the block, but to be functionally equivalent to the declarative property.

Copyright 2002 Accellera. All rights reserved.

```
P2: property @(posedge clk) (if ((st==IDLE)&&cond) then (a;b;c)):
```

which uses the sampled values of st and cond in its evaluation.

If the execution of the assertion statement is caused by an event other than the sampling event, then the assertion is scheduled to be evaluated on the next sample event of the assertion/property. Again, *the sampled values are also used for any expressions that affect the control flow of the block in which the assertion statement appears*.

Consider:

```
always @(st, a,...)
  case(st)
  IDLE:
    if (cond)
      P3: property @(posedge clk)(a;b;c); // needs explicit clock
...
```

When the property statement is executed, the assertion is scheduled to be evaluated at the next posedge clk event. Since the block may be executed multiple times in the same simulation time, there are at least two ways to consider the evaluation of the assertion.

The first is to treat each pass through the block as potentially setting a flag to enable the scheduling of the assertion evaluation. If the property statement is executed, the flag is set to enable the evaluation. If the property statement is not executed, then the flag is reset to disable the evaluation. When the sampling event occurs, the flag is checked to determine whether to evaluate the assertion or not.

The second way is to schedule the assertion evaluation on the next occurence of the sampling event. If the evaluation is already scheduled, no new evaluations are scheduled. When the sampling event occurs, the sampled values of all the control-flow expressions in the block are re-evaluated, and if they would cause the property statement to be executed, then the evaluation proceeds.

Note that either of these interpretations causes the same execution semantics for the procedural property P3 as for the declarative property P2. These semantics also only apply to the evaluation of the first element in the property expression. If the expression is sequential, then the evaluation of the subsequent elements of the sequential expression proceeds on subsequent sample events, regardless of the value of the control expressions at that time.

To help ensure consistency between standard event-based simulation tools and sample-based formal verification (or other) tools, care should be taken to ensure that all control signals that contribute to the execution of the property statement should be synchronous to the sample event(s) that control the assertion evaluation.

Copyright 2002 Accellera. All rights reserved.