

9. Temporal expressions 1

Temporal expressions provide a declarative way to describe temporal behavior. The *e* language provides a set of temporal operators and keywords that can be used to construct temporal expressions. A *temporal expression* (TE) is a combination of events and temporal operators that describes behavior. A TE expresses temporal relationships between events, values of fields, variables, or other items during a test. 5

Temporal expressions are used to define the occurrence of events, specify sequences of events as checkers, and suspend execution of a thread until the given sequence of events occurs. Temporal expressions are only legal in the following constructs: 10

- **wait** and **sync** actions in time-consuming methods (TCMs)
- **event** definitions and **expect** or **assume** struct members. 15

9.1 Overview

Temporal expressions are built from a set of temporal atoms (see 9.1.2) and a set of operators (see 9.2). 20

Temporal expressions are interpreted over finite paths, which are successions of states. A *state* is a complete valuation of all atoms, i.e., for each atom, a given state tells us whether that atom holds or does not hold. A *path* is an abstraction of the execution of an *e* program. The notion of an abstract state does away with the detail between the occurrence of `tick_start` and a subsequent `tick_end` when simulating the *e* program (see 8.4.2). This detail is not needed to define the semantics of temporal expressions. 25

The meaning of a TE is given in terms of tight satisfaction (“holds tightly”), which is a relationship between paths and temporal expressions. A TE *holds tightly* on a path **iff** (if and only if) that complete path exhibits the behavior expressed by the TE. 30

- a) The first step to determine the meaning of a TE is to determine its sampling event (see 9.1.3) and to transform it to its sampled normal form (see 9.1.4).
- b) The second step is to apply the definitions of the operators (see 9.2).
- c) New events can be defined in terms of temporal expressions using the **event** construct (see 8.3.1). It is not necessary to consider the an atom’s origin when defining the semantics of temporal expressions. 35
- d) To determine the success or failure of a TE, which is relevant for describing the meaning of constructs that use temporal expressions, see 9.3.

9.1.1 Terminology

This section defines some of the terms used in this clause. 40

holds — a term used to talk about the meaning of atoms. If the atom is an event, the atom holds if the event occurs at the *state*. If the atom is a proposition `true (exp)`, the *proposition* holds if *exp* evaluates at the *state* to `TRUE` if it is a Boolean expression or to non-zero if *exp* is an numeric expression. 45

holds tightly — a term used to talk about the meaning of temporal expressions. A TE is interpreted over finite paths. Informally, a TE holds tightly on a path **iff** the path exhibits the behavior described by the TE.

occurs, occurrence — a term used to talk about the meaning of events. For each event, a *state* identifies whether the event is present or not. 50

path — a succession of *states* of the *e* program. For a path *p*, the notation $p = s_0 s_1 \dots s_n$ can be used to indicate *p* has *n*+1 states. The first state of *p* is *s*₀; the last state of *p* is *s*_{*n*}. 55

prefix — a prefix of a path is a *sub-path* of that consists of all *states* of the original *path* up to a certain point. The empty path is also a prefix of the original path.

proposition — a *temporal atom* of the form `true (exp)`.

sampling event — an atom associated with a temporal expression which determines when the temporal expression is evaluated. Every temporal expression has a sampling event.

sampled-normal form — the result of propagating the *sampling event* of a TE. In this form, every atom under the TE is explicitly sampled.

state — a valuation of all atoms. For each event, the state identifies whether the event occurs. For each *proposition*, the state identifies whether the proposition holds.

sub-path — a contiguous part of a *path*. The sub-paths of a path $p = s_0 s_1 .. s_n$ are the paths $sp = s_i s_{i+1} .. s_j$, where $0 \leq i \leq j \leq n$.

success — a term used to talk about the meaning of *top-level temporal expressions*. The temporal expression succeeds at a point of a *path* if there is a *sub-path* ending in the given point such that the temporal expression *holds tightly* over the *sub-path*.

temporal atom — an event, a proposition `true (exp)`, or cycle.

tight satisfaction — see: *holds tightly*.

top-level temporal expression — a temporal expression that not nested underneath another TE, i.e., it appears directly under a **wait** or **sync** action, or an **event**, **expect**, or **assume struct** member.

9.1.2 Temporal atoms

Temporal expressions are built from a set of temporal atoms. These atoms consist of the following:

- events, including predefined events such as the event `sys.any`
- propositions `true (exp)`
- the atom `cycle`

Atoms are interpreted at states of the *e* program. Each and every state identifies which atoms hold and which do not hold.

Events are said to “occur”, propositions “hold”. The proposition `true (exp)` holds at a state if *exp* evaluates to TRUE if it is a Boolean expression or to non-zero if *exp* is an numeric expression. The atom `cycle` holds at any state.

9.1.3 Sampling event

A key step in determining the meaning of a temporal expression is to identify its sampling event. The sampling event for a TE is one of the following, in decreasing order of precedence.

- a) The sampling event specified with the binary @ operator.
- b) The sampling event inherited from the parent temporal expression.
- c) The sampling event of the TCM if the TE appears under a **wait** or **sync** action of that TCM.
- d) `sys.any`, if none of the above applies.

9.1.4 Sampling propagation and sampled normal form

1

The first step in uncovering the meaning of a top-level temporal expression is to propagate its sampling event so every temporal atom that appears under the given TE is explicitly sampled. This transformation is called *sampling propagation* and the result is the *sampled normal form* of the given TE.

5

The rules for transforming a TE to its sampled normal form are as follows.

- a) Given a (top-level) temporal expression t whose sampling event is q , let S be a function that returns the sampled normal $S(t, q)$.
- b) Let b be an expression; e and q be events; $t, t1$, and $t2$ be temporal expressions; exp be a numeric expression; and $range$ be either $exp..exp$, $exp..$, $..exp$, or $..$.

10

$$S(\text{true}(b), q) = \text{true}(b) @q$$

15

$$S(\text{cycle}, q) = \text{cycle} @q$$

$$S(@e, q) = @e @q$$

20

$$S((t), q) = (S(t, q))$$

$$S(t1 \text{ and } t2, q) = S(t1, q) \text{ and } S(t2, q)$$

25

$$S(t1 \text{ or } t2, q) = S(t1, q) \text{ or } S(t2, q)$$

$$S(\{t1 ; t2\}, q) = \{S(t1, q) ; S(t2, q)\}$$

$$S([exp]*t, q) = [exp]*S(t, q)$$

30

$$S(\sim[range]*t, q) = \sim[range]*S(t, q)$$

$$S(\{[range]*t; t2\} q) = \{[range]*S(t1, q); S(t2; q)\}$$

$$S(t @ e, q) = S(t, e) @q$$

35

$$S(\text{fail } t, q) = \text{fail } (S(t, q))$$

$$S(t1 \Rightarrow t2, q) = S(t1, q) \Rightarrow S(t2, q)$$

40

$$S(\text{rise}(exp), q) = \text{rise}(exp) @q$$

$$S(\text{fall}(exp), q) = \text{fall}(exp) @q$$

$$S(\text{change}(exp), q) = \text{change}(exp) @q$$

45

$$S(\text{not } t, q) = \text{not } (S(t, q))$$

$$S(\text{detach } (t), q) = \text{detach } (S(t, q))$$

$$S(t \text{ exec action}, q) = S(t, q) \text{ exec action}$$

50

Example

Consider the temporal expression:

55

$\{\text{@a}; \sim[\dots]^* \text{cycle}; \text{@b}\} \text{@q}$

The sampling event of this TE is specified explicitly as q .
The temporal atoms a , cycle , and b are not explicitly sampled.

To obtain the sampled-normal form, q is propagated into the sub-expressions:

$\{\text{@a } \text{@q}; \sim[\dots]^* \text{cycle } \text{@q} ; \text{@b } \text{@q}\}$

9.2 Temporal operators and constructs

This section gives the semantics of the operators for building temporal expressions. These definitions only apply after a top-level TE has first been interpreted to determine its sampling event and sampled normal form (see 9.1.4). Each meaning is given in terms of tight satisfaction (see 9.1.1).

To illustrate tight satisfaction, consider the temporal expression $\text{cycle } \text{@q}$, which holds tightly on any path where event q occurs in the last state of that path and in no other state. It is clear that tight satisfaction considers the path as a whole and makes requirements on every state of the given path. This is different from the notion success of a TE (see 9.3).

9.2.1 Precedence of temporal operators

Table 1 shows the precedence of temporal operators, listed from highest precedence to lowest.

Table 1—Precedence of temporal operators

Operator name	Operator example
<i>named event</i>	@event-name
exec	TE <i>exec</i> <i>action-block</i>
repeat	[] * TE
fail not	fail TE not TE
and	TE1 and TE2
or	TE1 or TE2
sequence	{TE1 ; TE2}
yield	TE1 => TE2
<i>sample event</i>	TE @ <i>event-name</i>

9.2.2 cycle

Purpose	Specify an occurrence of a sampling event
Category	Temporal expression
Syntax	cycle
Informal semantics	<code>cycle @e</code> holds tightly on a path iff <code>e</code> occurs at the last state of that path and at no other state of that path.

This represents one cycle of some sampling event. With no explicit sampling event specified, this represents one cycle of the sampling event from the context (i.e., the sampling event from the overall temporal expression or the sampling event for the TCM that contains the temporal expression). When a sampling event is specified, as in `cycle@sampling-event`, this is equivalent to `@sampling-event@sampling-event`.

See also: 8.3.1 and 8.4.

Syntax example:

```
wait cycle;
```

9.2.3 true(exp)

Purpose	Boolean temporal expression
Category	Temporal expression
Syntax	true (<i>bool</i> : exp)
Parameters	<i>bool</i> A Boolean expression.
Informal semantics	<code>true(exp) @e</code> holds tightly on a given path iff <ol style="list-style-type: none"> <code>true(exp)</code> holds at the last state of the path and <code>cycle @e</code> holds tightly on the path.

This uses a Boolean expression as a temporal expression. Each occurrence of the sampling event causes an evaluation of the Boolean expression. The Boolean expression is evaluated only at the sampling point.

The temporal expression succeeds each time the expression evaluates to TRUE. The expression `exp` is evaluated after `pclk`. Changes in `exp` after `true(exp) @pclk` has been evaluated are ignored.

See also: 3.1.1.

Syntax example:

```
event rst is true(reset == 1) @clk;
```

9.2.4 @ unary event operator

Purpose	Use an event as a temporal expression	
Category	Temporal expression	
Syntax	$@[struct-exp.]event-type$	
Parameters	<i>struct-exp.</i> <i>event-type</i>	The name of an event. This can be a predefined event or a user-defined event, and can include the name of the <i>struct</i> instance where the event is defined.
Informal semantics	$@e @q$ holds tightly on a given path iff a) event <i>e</i> occurs at a state of the path and b) <code>cycle @q</code> holds tightly on that path.	

An event can be used as the simplest form of a temporal expression. The temporal expression $@event-type$ succeeds every time the event occurs. Success of the expression is simultaneous with the occurrence of the event.

The *struct-exp* is an expression that evaluates to the struct instance containing the event instance. If no *struct* expression is specified, the default is the current struct instance. If a *struct* expression is included in the event name, the value of the *struct* expression shall not change throughout the evaluation of the temporal expression.

See also: 8.3.1 and 8.4.

Syntax example:

```
wait @rst;
```

9.2.5 @ sampling operator (binary @)

Purpose	Specify a sampling event for a temporal expression	
Category	Temporal expression	
Syntax	<i>temporal-expression @event-name</i>	
Parameters	<i>temporal-expression</i>	A temporal expression.
	<i>event-name</i>	The sampling event.
Informal semantics	$t @e$ holds tightly on a given path iff there exist paths <i>p1</i> and <i>p2</i> and state <i>s</i> so that a) the concatenation of <i>p1</i> , <i>s</i> , and <i>p2</i> is the original path; b) <i>t</i> holds tightly on <i>p1 s</i> ; c) <code>cycle @e</code> holds tightly on <i>s p2</i> .	

This is used to specify the sampling event for a temporal expression. The specified sampling event overrides the default sampling event. The sampling event applies to all subexpressions of the temporal expression. It can be overridden for a subexpression by attaching a different sampling event to the subexpression. A sampled temporal expression succeeds when its sampling event occurs with or after the success of the temporal expression.

See also: 8.3.1 and 8.4.

Syntax example:

```
wait cycle @sys.reset;
```

9.2.6 and

Purpose	Temporal expression and operator
Category	Temporal expression
Syntax	<i>temporal-expression and temporal-expression</i>
Parameters	<i>temporal-expression</i> A temporal expression.
Informal semantics	$t1$ and $t2$ holds tightly on a path iff both $t1$ and $t2$ hold tightly on that path.

The temporal **and** succeeds when both temporal expressions start evaluating in the same sampling period and succeed in the same sampling period.

Syntax example:

```
(@TE1 and @TE2)@clk
```

9.2.7 or

Purpose	Temporal expression or operator
Category	Temporal expression
Syntax	<i>temporal-expression or temporal-expression</i>
Parameters	<i>temporal-expression</i> A temporal expression.
Informal semantics	$t1$ or $t2$ holds tightly on a path iff either $t1$ or $t2$ holds tightly on that path (or they both hold tightly on that path).

The **or** temporal expression succeeds when either temporal expression succeeds.

An **or** operator creates a parallel evaluation for each of its subexpressions. It can create multiple successes for a single temporal expression evaluation.

Syntax example:

```
(@TE1 or @TE2)@clk
```

9.2.8 { exp ; exp }

Purpose	Temporal expression sequence operator
Category	Temporal expression
Syntax	{ <i>temporal-expression</i> ; <i>temporal-expression</i> ; ...}
Parameters	<i>temporal-expression</i> A temporal expression.
Informal semantics	{ t1 ; t2 } holds tightly on a given path iff there exist paths <i>p1</i> and <i>p2</i> so that a) <i>p1</i> concatenated with <i>p2</i> gives the original path; b) t1 holds tightly on <i>p1</i> ; c) t2 holds tightly on <i>p2</i> .

The semicolon (;) sequence operator evaluates a series of temporal expressions over successive occurrences of a specified sampling event. Each temporal expression following a semicolon (;) starts evaluating in the sampling period following the one where the preceding temporal expression succeeded. The sequence succeeds whenever its final expression succeeds.

Curly braces ({}) are used in the scope of a temporal expression define a sequence; do not use them in any other way here.

Example

Figure 1 shows the results of evaluating the temporal sequence:

```
{@ev_a; @ev_b; @ev_c} @qclk;
```

over the series of *ev_a*, *ev_b*, and *ev_c* events shown at the top of Figure 1. The sequence evaluation starts whenever event *ev_a* occurs.

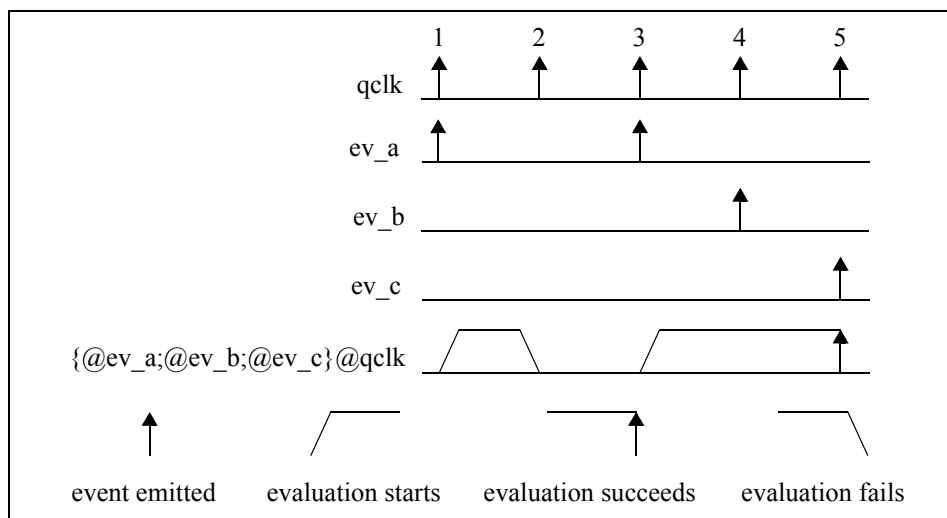


Figure 1—Example evaluations of a temporal sequence

Syntax example:

```
{@ev_d; @ev_e} @ev_f
```

9.2.9 [exp]

Purpose	Fixed repetition operator	
Category	Temporal expression	
Syntax	[<i>exp</i>] [<i>* temporal-expression</i>]	
Parameters	<i>exp</i>	A 32-bit, non-negative integer expression, which specifies the number of times to repeat the evaluation of the temporal expression. It cannot contain any functions.
	<i>temporal-expression</i>	A temporal expression. If <i>* temporal-expression</i> is omitted, <i>* cycle</i> is automatically used in its place.
Informal semantics	[<i>exp</i>] * <i>t</i> is equivalent to { <i>t</i> ; <i>t</i> ; .. ; <i>t</i> }, where the sequence consists of as many terms as the value of <i>exp</i> . A special case is [0] * <i>t</i> , which holds tightly only on the empty path.	

Repetition of a temporal expression is frequently used to describe cyclic or periodic temporal behavior. The [*exp*] fixed repeat operator specifies a fixed number of occurrences of the same temporal expression. If the numeric expression evaluates to zero (0), the temporal expression succeeds immediately.

Syntax example:

```
wait [2]*cycle;
```

9.2.10 ~[exp..exp]

Purpose	True match variable repeat operator
Category	Expression
Syntax	~[[<i>from-exp</i>].. <i>to-exp</i>] [* <i>temporal-expression</i>]
Parameters	<i>from-exp</i> An optional non-negative, 32-bit numeric expression that specifies the minimum number of repetitions of the temporal expression. If the <i>from-exp</i> is missing, zero (0) is used.
	<i>to-exp</i> An optional non-negative, 32-bit numeric expression that specifies the maximum number of repetitions of the temporal expression. If the <i>to-exp</i> is missing, infinity is used.
	<i>temporal-expression</i> A temporal expression that is to be repeated some number of times within the <i>from-exp</i> .. <i>to-exp</i> range. If * <i>temporal-expression</i> is omitted, * <code>cycle</code> is automatically used in its place.
Informal semantics	The following definitions apply. a) ~[range] is equivalent to ~[range]* <code>cycle</code> b) ~[<i>exp1</i> .. <i>exp2</i>] * <i>t</i> is equivalent to { [<i>exp1</i>]* <i>t</i> ; ~[.. <i>exp2</i> - <i>exp1</i>] * <i>t</i> } c) ~[<i>exp1</i> ..] * <i>t</i> is equivalent to { [<i>exp1</i>]* <i>t</i> ; ~[..] * <i>t</i> } d) ~[.. <i>exp2</i>] * <i>t</i> is equivalent to [0]* <i>t</i> or [1]* <i>t</i> or .. [<i>exp2</i>]* <i>t</i> e) ~[..] * <i>t</i> is equivalent to [0]* <i>t</i> or [1]* <i>t</i> or ..

The true match repeat operator can be used to specify a variable number of consecutive successes of a temporal expression. True match variable repeat succeeds every time the subexpression succeeds. This expression creates a number of parallel repeat evaluations within the range.

True match repeat also enables specification of behavior over infinite sequences by repeating an infinite number of occurrences of a temporal expression. The expression ~[..] *TE is equivalent to:

[0] or [1]*TE or [2]*TE...

This construct is mainly useful for maintaining information about past events. *See also:* 9.2.9.

Syntax example:

~[2..4]*@pclk

9.2.11 [*exp..exp*]

Purpose	First match variable repeat operator
Category	Expression
Syntax	{ ... ; [[<i>from-exp</i>].[<i>to-exp</i>]] [* <i>repeat-expression</i>]; <i>match-expression</i> ; ... }
Parameters	<i>from-exp</i> An optional non-negative, 32-bit numeric expression that specifies the minimum number of repetitions of the <i>repeat-expression</i> . If the <i>from-exp</i> is missing, zero (0) is used.
	<i>to-exp</i> An optional non-negative, 32-bit numeric expression that specifies the maximum number of repetitions of the <i>repeat-expression</i> . If the <i>to-exp</i> is missing, infinity is used.
	<i>repeat-expression</i> A temporal expression that is to be repeated some number of times within the <i>from-exp</i> .. <i>to-exp</i> range. If * <i>repeat-expression</i> is omitted, * <i>cycle</i> is automatically used in its place.
	<i>match-expression</i> The temporal expression to match.
Informal semantics	The following definitions apply. a) { [<i>exp</i> .. <i>exp</i>] * <i>t1</i> ; <i>t2</i> } is equivalent to FirstMatch { ~[<i>exp</i> .. <i>exp</i>] * <i>t1</i> ; <i>t2</i> } b) FirstMatch <i>t</i> holds tightly on a given path iff <i>t</i> holds tightly on the path <i>p</i> and <i>p</i> has no prefix on which <i>t</i> holds tightly.

The first match repeat operator is only valid in a temporal sequence {TE; TE; TE} (see 9.2.8); [it is not a temporal expression operator](#). The first match repeat expression succeeds on the first success of the *match-expression* between the lower and upper bounds specified for the *repeat-expression*.

First match repeat also enables specification of behavior over infinite sequences by allowing an infinite number of repetitions of the *repeat-expression* to occur before the *match-expression* succeeds. Where @ev_a is an event occurrence, { [. .] *TE1; @ev_a } is equivalent to:

{@ev_a} or {[1]*TE1; @ev_a} or {[2]*TE1; @ev_a} or {[3]*TE1; @ev_a}...

Syntax example:

{[2..4]*@pclk;@reset}

9.2.12 fail

Purpose	Temporal expression failure operator
Category	Temporal expression
Syntax	fail <i>temporal-expression</i>
Parameters	<i>temporal-expression</i> A temporal expression.
Informal semantics	fail t holds tightly on a given path iff a) that path cannot be extended so that t holds tightly on the extended path and b) the given path does not have a prefix on which fail t also holds tightly.

A **fail** succeeds whenever the temporal expression fails. If the temporal expression has multiple interpretations (e.g., **fail** (TE1 or TE2)), the expression succeeds **iff** all the interpretations fail.

The expression **fail** TE succeeds at the point where all possibilities to satisfy TE have been exhausted. Any TE can fail at most once per sampling event.

The **not** operator (see 9.2.16) differs from the **fail** operator, as illustrated in Figure 2.

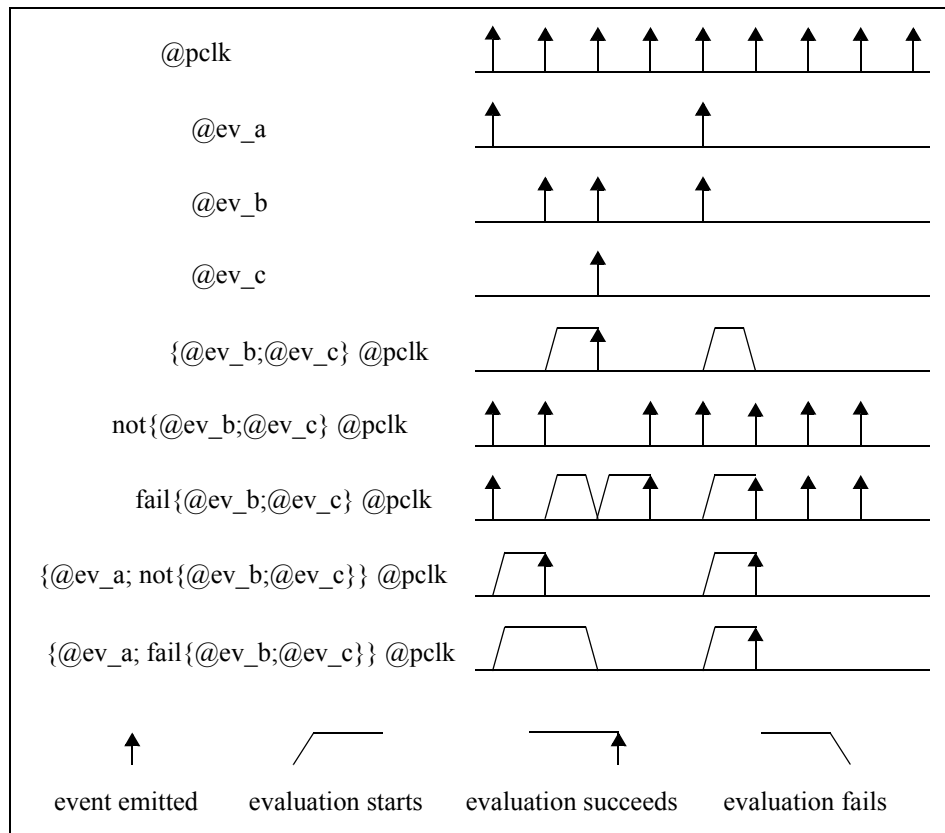


Figure 2—Comparison of temporal not and fail operators

Syntax example:

```
fail{@ev_b; @ev_c}
```

9.2.13 =>

Purpose	Temporal yield operator	
Category	Temporal expression	
Syntax	<i>temporal-expression1 => temporal-expression2</i>	
Parameters	<i>temporal-expression1</i>	The first temporal expression. The second temporal expression is expected to succeed if this expression succeeds.
	<i>temporal-expression2</i>	The second temporal expression. If the first temporal expression succeeds, this expression is also expected to succeed.
Informal semantics	The temporal expression t1 => t2 is equivalent to (fail t1 or {t1; t2})	

The yield operator is used to assert that success of one temporal expression depends on the success of another temporal expression. The yield expression succeeds without evaluating the second expression if the first expression fails. If the first expression succeeds, then the second expression needs to also succeed in sequence.

The sampling event from the context applies to both sides of the yield operator expression. The entire expression is essentially a single temporal expression, so that (TE1 => TE2)@sampling_event is effectively (TE)@sampling_event, where TE is the temporal expression made up of TE1 => TE2.

NOTE—Yield is typically used in conjunction with the **expect** struct member (see 10.2) to express temporal rules.

Syntax example:

```
@A => {[1..2]*@clk; @B}
```

9.2.14 eventually

Purpose	Temporal expression success check
Category	Temporal expression
Syntax	eventually <i>temporal-expression</i>
Parameters	<i>temporal-expression</i> A temporal expression.
Informal semantics	The temporal expression eventually t @q is equivalent to t @q or fail @quit @q

This is used to indicate the temporal expression is expected to succeed at some unspecified time. Typically, **eventually** is used in an **expect struct** member to specify a temporal expression is expected to succeed sometime before the **quit** event occurs for the struct. *See also:* 23.1.2.5.

Syntax example:

```
{@ev_d; eventually @ev_e}
```

9.2.15 detach

Purpose	Detach a temporal expression
Category	Temporal expression
Syntax	detach (<i>temporal-expression</i>)
Parameters	<i>temporal-expression</i> A temporal expression to be independently evaluated.
Informal semantics	The temporal expression detach(t1) is equivalent to @unnamed_event_1 where is unnamed_event_1 defined as event unnamed_event_1 is t1;

A detached temporal expression is evaluated independently of the expression in which it is used. It starts evaluation when the main expression does. Whenever the detached TE succeeds it emits an “implicit” event which is only recognized by the main TE. The detached temporal expression inherits the sampling event from the main temporal expression.

Example

In the following example, both S1 and S2 start with @Q. However, the S1 temporal expression expects E to follow Q, while the S2 temporal expression expects E to precede Q by one cycle. The **detach()** construct causes the temporal expressions to be evaluated separately. As a result, the S3 temporal expression is equivalent to the S2 expression, as shown in Figure 3.

```

struct s {
  event pclk is @sys.pclk;
  event Q;
  event E;
  event T is {@E; [2]} @pclk;
  event S1 is {@Q; {@E; [2]}} @pclk;
  event S2 is {@Q; @T} @pclk;
  event S3 is {@Q; detach({@E; [2]})} @pclk;
};

```

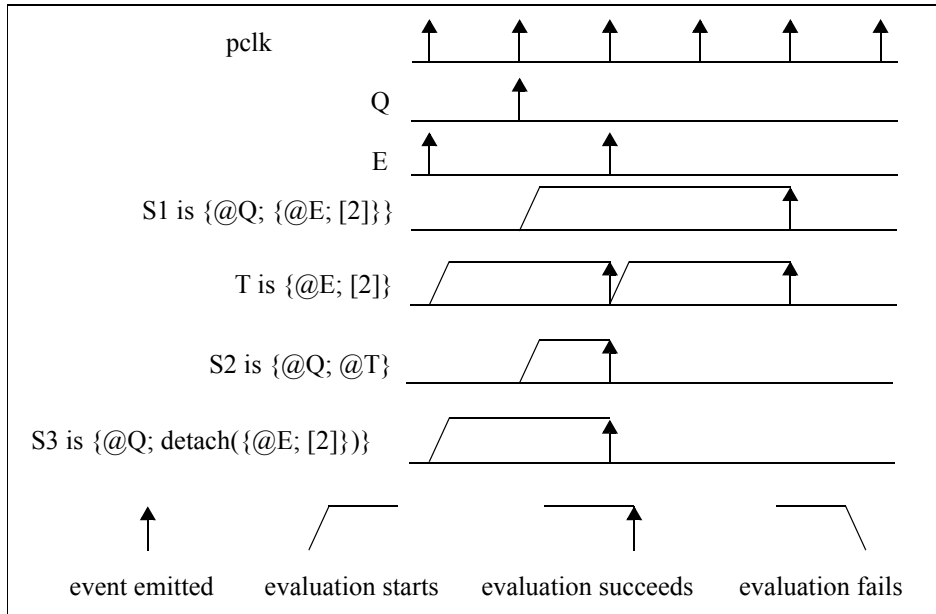


Figure 3—Examples illustrating detached temporal expressions

Syntax example:

```

@trans.end => detach({@trans.start; ~[2..5]})@pclk

```

9.2.16 not

Purpose	Temporal expression inversion operator
Category	Temporal expression
Syntax	not <i>temporal-expression</i>
Parameters	<i>temporal-expression</i> A temporal expression.
Informal semantics	The temporal expression <code>not t</code> is equivalent to <code>detach(fail t)</code>

The **not** temporal expression succeeds if the evaluation of the subexpression does not succeed during the sampling period. Thus, **not** TE succeeds on every occurrence of the sampling event if TE does not succeed.

NOTE—If an event is explicitly *emitted* (see 8.3.2), a race condition can arise between the event occurrence and the **not** of the event when used in some temporal expressions.

Syntax example:

```
not {@ev_b;@ev_c}
```

9.2.17 change(exp), fall(exp), rise(exp)

Purpose	Transition or edge temporal expression	1
Category	Temporal expression	5
Syntax	change fall rise (<i>scalar</i> : exp) [<i>@event-type</i>] change fall rise ('HDL-pathname') @sim	10
Parameters	<i>scalar</i> A Boolean expression or an integer expression.	
	<i>event-type</i> The sampling event for the expression.	
	'HDL-pathname' An HDL object enclosed in single quotes (' ').	15
	@sim A special annotation used to detect changes in HDL signals.	
Informal semantics	<p>a) The following definitions apply for the first syntax type: change fall rise(<i>scalar</i>: exp) [<i>@event-type</i>]</p> <ol style="list-style-type: none"> 1) change(exp) @q is equivalent to <code>true(prev(exp,q) != exp)@q</code> 2) fall(exp) @q is equivalent to <code>true(prev(exp,q) > exp)@q</code> 3) rise(exp) @q is equivalent to <code>true(prev(exp,q) < exp)@q</code> 4) <code>prev(exp,q)</code> is a function that gives the value of <code>exp</code> at the previous point for which <code>q</code> holds. If no such point exists, it gives the value of <code>exp</code> at the first point of the path. <code>prev()</code> is not part of the <i>e</i> syntax, it is only an aid for defining the meaning of <code>rise(exp) @q</code>. <p>b) The following definitions apply for the second syntax type: change fall rise('HDL-pathname') @sim</p> <ul style="list-style-type: none"> — The temporal expression rise('HDL-pathname') @sim serves two main purposes. <ol style="list-style-type: none"> 1) It causes the simulator to pass control back to the <i>e</i> program at the end of a simulator delta cycle in which <i>HDL-pathname</i> changes (see 8.4.2). 2) It is a temporal expression. If rise('HDL-pathname') @sim appears as a top-level TE, it is already in sampled normal form. — rise('HDL-pathname') @sim holds tightly on a path iff <ol style="list-style-type: none"> 1) the last point of the path corresponds to a tick in which the signal <i>HDL-pathname</i> changes; 2) the signal changes at no other point of the path; 3) the signal's new value is larger than its old value. — fall and change perform in a similar fashion. 	20 25 30 35 40 45

This detects a change in the sampled value of an expression. The expression is evaluated at each occurrence of the sampling event and compared to the value it had at the previous sampling point. Only the values at sampling points are detected. The value of the expression between sampling points is invisible to the temporal expression.

Table 2 describes the behavior of each of the three temporal expressions (**change**, **fall**, and **rise**). When applied to HDL variables, the expressions examine the value after each bit is translated from the HDL four-value or nine-value logic representation to *e* two-value logic representation.

Table 2—Edge condition options

Edge condition	Meaning
rise (<i>exp</i>)	Triggered when the expression changes from FALSE to TRUE. If it is an integer expression, the rise () temporal expression succeeds upon any change from x to $y > x$. Signals wider than one bit are allowed. Integers larger than 32 bits are not allowed.
fall (<i>exp</i>)	Triggered when the expression changes from TRUE to FALSE. If it is an integer expression, the fall () temporal expression succeeds upon any change from x to $y < x$. Signals wider than one bit are allowed. Integers larger than 32 bits are not allowed.
change (<i>exp</i>)	Triggered when the value of the expression changes. The change () temporal expression succeeds upon any change of the expression. Signals wider than one bit are allowed. Integers larger than 32 bits are not allowed.

Table 3 describes the default translation of HDL nine-value logic representation to *e* values. The @x and @z HDL value modifiers can be used to override the default translation (see 25.3).

Table 3—Transition of HDL values

HDL values	<i>e</i> value
0, X, U, W, L, -	0
1, Z, H	1

A special notation, @sim, can be used in place of a sampling event for the **rise**, **fall**, or **change** of HDL objects. If @sim is used, the HDL object is watched by the simulator. The @sim notation does not signify an event, but is used only to cause a callback any time there is a change in the value of the HDL object to which it is attached.

When a sampling event other than @sim is used, changes to the HDL object are detected only if they are visible at the sampling rate of the sampling event (see Figure 4).

NOTE—An *e* program ignores glitches (see 8.4.2) that occur in a single simulation time slot. Only the first occurrence of a particular monitored event in a single simulation time slot is recognized by the *e* program.

Syntax example:

```
event hv_c is change('top.hold_var')@sim;
```

Example

Figure 4 shows evaluations of the **rise**, **fall**, and **change** expressions for the HDL signal *v*, with the sampling events @sim and @qclk. The qclk event is an arbitrary event that is emitted at the indicated points. The *v* signal rises and then falls between the second and third occurrences of event qclk. Since the signal's value is the same at the third qclk event as it was at the second qclk event, the **change('V')@qclk** expression does not succeed at the third qclk event.

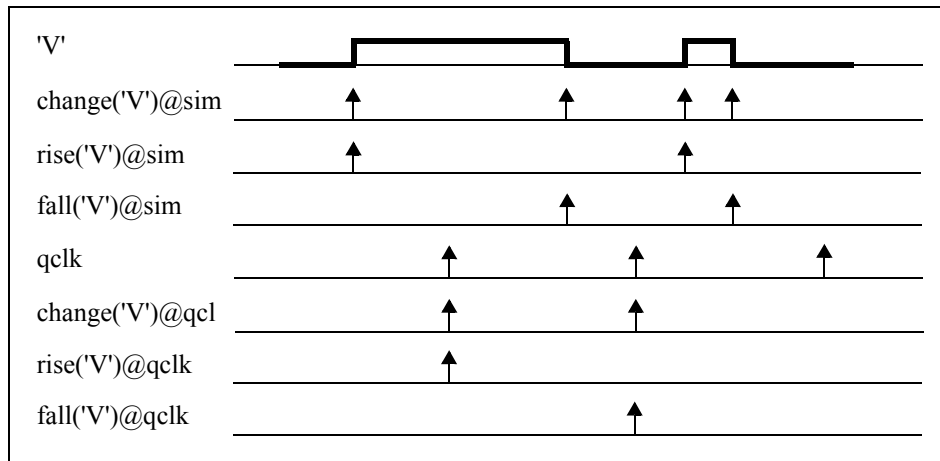


Figure 4—Effects of the sampling rate on detecting HDL object changes

9.2.18 delay

Purpose	Specify a simulation time delay
Category	Temporal expression
Syntax	delay (<i>int</i> : <i>exp</i>)
Parameters	<i>int</i> An integer expression or time expression no larger than 64 bits. The number specifies the amount of simulation time to delay. The time units are in the timescale used in the HDL simulator.
Informal semantics	<code>delay (exp)</code> holds tightly on a path whose elapsed simulation time is <i>exp</i> .

This succeeds after a specified simulation time delay elapses. A callback occurs after the specified time. A delay of zero (0) succeeds immediately. *See also*: 11.1.2.

Attaching a sampling event to **delay** has no effect. The **delay** ignores the sampling event and succeeds as soon as the delay period elapses. Also, this expression can only be used if the e program is being run with an attached HDL simulator.

Syntax example:

```
wait delay(3);
```

9.2.19 exec

Purpose	Attach an action block to a temporal expression	
Category	Temporal expression side effect	
Syntax	<i>temporal-expression</i> exec <i>action</i> ; ...	
Parameters	<i>temporal-expression</i>	The temporal expression that invokes the action block.
	<i>action</i> ; ...	A series of actions to perform when the expression succeeds.

This invokes an action block when a temporal expression succeeds. The action block cannot contain any time-consuming actions. The actions are executed immediately upon the success of the expression, but not more than once per tick. To support extensibility, use method calls in the **exec** action block rather than calling the actions directly.

The usage of **exec** is similar to the **on** struct member, except:

- any temporal expression can be used as the condition for **exec**, while only an event can be used as the condition for **on**;
- **exec** needs to be used in the context of a temporal expression in a TCM or an event definition, while **on** can only be a struct member.

An **exec** action cannot be attached to a first match variable repeat expression. However, it can be attached to the repeat expression of a first match variable repeat expression. An **exec** action also cannot be attached to an implicit repeat expression, it needs to be attached to an explicit repeat expression.

See also: 2.2.3 and 10.1.

Syntax example:

```
wait @watcher_on exec {print watcher_status_1};
```

9.3 Success and failure of a temporal expression

The meaning of temporal expressions is defined in terms of tight satisfaction (“holds tightly”), as seen in 9.2. For top-level temporal expressions, it is useful to introduce the notions of “success” and “failure”, which are derived from tight satisfaction. The notion of *success* is used for describing the meaning of **event** struct members and **wait** and **sync** actions. The notion of *failure* is used to describe the meaning of **assume** and **expect** struct members.

9.3.1 Success of a temporal expression

A TE *succeeds* at a point of a path if it holds tightly on a sub-path ending in the given point. More precisely:

A TE τ , whose sampling event is q , succeeds at the i^{th} state of a path $p = s_0 s_1 \dots s_i \dots s_n$, $0 \leq i \leq n$, **iff** either τ holds tightly on p or else there is a j , $0 \leq j \leq i$ such that event q holds at s_{j-1} and τ holds tightly on $s_j \dots s_i$.

The notion of success of a TE can also be used without specifying a path and a state. Then [it is presumed the path corresponds](#) to the execution of a given *e* program and the state [corresponds to](#) the current state of the execution. 1

9.3.2 Failure of a temporal expression 5

A TE τ , whose sampling event is q , *fails* at the i^{th} state of a path, **iff** `fail τ` succeeds at the i^{th} state of that path. Again, if no path and state are specified, [it is presumed the path corresponds](#) to the execution of a given *e* program and the state corresponds to the current state of the execution. 10

9.3.3 Start of an evaluation of a temporal expression

For a given path $p = s_0 s_1 .. s_i .. s_n$, a state s_i $0 \leq i \leq n$ of that path, and a temporal expression τ , whose sampling event is q , the definition of success of τ on p at s_i considers various sub-paths of p and tests for *tight satisfaction* of τ on these sub-paths. The sub-paths considered are those sub-paths that start with a state just after a state in which q occurs and end in s_i , and also the sub-path that starts with the first state of p and ends in s_i . 15

The evaluation of τ starts at the first state of p and also at any states following a state where the sampling event occurs. The TE can succeed only in states in which the sampling event occurs, except for temporal expressions that hold tightly on the empty path, e.g., `[0]*cycle @q`. 20

The paths over which a top level TE is interpreted during the execution of an *e* program depend on the construct in which the TE appears. 25

- For **event**, **expect**, and **assume** *struct* members, evaluation of the TE starts as soon as the parent struct is generated and ends when the parent struct is quit.
- For **sync** actions, the TE is evaluated as soon as the **sync** action is reached.
- For **wait** actions, the evaluation of the TE starts in the tick after reaching the **wait** action.
- For both **sync** and **wait** actions, evaluation ends if the TE succeeds or the parent TCM is terminated. 30

35

40

45

50

55