

Formal semantics of temporal e

Draft version – please do not distribute!

January 30, 2004

1 Abstract syntax

This section gives an abstract syntax of temporal expressions.

1.1 Atoms

Temporal e is defined with respect to a non-empty set of events \mathcal{E} , and a non-empty set of propositions \mathcal{P} . \mathcal{E} has a distinguished element **any**; \mathcal{P} has a distinguished element **cycle**. The set of events together with the set of propositions form the set of *atoms*: $\mathcal{A} = \mathcal{E} \cup \mathcal{P}$. We will use e and q to denote representative elements of \mathcal{E} , and p to denote a representative element of \mathcal{P} , and a to denote a representative element of \mathcal{A} .

- $e, q, \mathbf{any} \in \mathcal{E}$
- $p, \mathbf{cycle} \in \mathcal{P}$
- $a \in \mathcal{A}$

1.2 Temporal expressions

Let t, t_1 and t_2 be temporal expressions, then expressions listed in the first column of Table 1 also temporal expressions.

2 Semantics

The semantics of temporal e is defined with respect to finite words over an alphabet $\Sigma = 2^{\mathcal{A}}$. We denote a letter from Σ by ℓ , and an empty, or finite word from Σ by u, v , or w (possibly with subscripts.) We denote the length of a word w by $|w|$. An empty word $w = \varepsilon$ has length 0, a finite word $w = (\ell_0 \ell_1 \ell_2 \dots \ell_n)$ has length $n + 1$. We use i, j , and k to denote non-negative integers. We denote the i^{th} letter of w by w^{i-1} (since counting of letters starts at zero). We denote $w^{i..}$ the suffix of w starting at w^i . That is, for every $i < |w|$, $w^{i..} = w^i w^{i+1} .. w^{|w|-1}$. We denote by $w^{i..j}$ the finite sequence of letters starting from w^i and ending in w^j . That is, for $i \leq j < |w|$, $w^{i..j} = w^i w^{i+1} .. w^j$ and for $j < i$, $w^{i..j} = \varepsilon$.

Temporal expression	Name
a	atom
(t)	parenthesis
t_1 and t_2	conjunction
t_1 or t_2	disjunction
$t_1 ; t_2$	sequence
$[0]$	empty
$[1..]t$	true-match repetition
$t @ e$	sample
fm t	first-match
fail t	fail

Table 1: Abstract syntax of temporal expressions

2.1 Semantics of atoms

The semantics of atoms is assumed to be given by a relation $\models \subseteq \Sigma \times \mathcal{A}$, relating letters in Σ with atoms: $(\ell, a) \in \models$ iff $a \in \ell$; we use the equivalent notation $\ell \models a$, and say that ℓ satisfies a .

2.2 Semantics of temporal expressions in sampled-normal form

Temporal expressions are defined over finite words from the alphabet Σ . The notation $w \models t$, where w is a finite word and t is a temporal expression means that w *models tightly* t . The semantics of temporal expressions in sampled-normal form are defined as follows. Let p be a proposition, e, q denote events, and t, t_1 , and t_2 denote temporal expressions *in sampled-normal form*. Then:

$$w \models \mathbf{cycle @ any} \iff |w| = 1 \text{ and } w^0 \models \mathbf{any} \quad (1)$$

$$w \models \mathbf{cycle @ e} \iff w^{|w|-1} \models e \text{ and } \forall i < |w| - 1 : \neg(w^i \models e) \quad (2)$$

$$w \models p @ e \iff w^{|w|-1} \models p \text{ and } w \models \mathbf{cycle @ e} \quad (3)$$

$$w \models e @ q \iff \exists j < |w| : w^j \models e \text{ and } w \models \mathbf{cycle @ e} \quad (4)$$

$$w \models (t) \iff w = w \models t \quad (5)$$

$$w \models t_1 \text{ and } t_2 \iff (w \models t_1) \text{ and } (w \models t_2) \quad (6)$$

$$w \models t_1 \text{ or } t_2 \iff (w \models t_1) \text{ or } (w \models t_2) \quad (7)$$

$$w \models t_1 ; t_2 \iff \exists u, v \text{ s.t. } w = uv \text{ and } (u \models t_1) \text{ and } (v \models t_2) \quad (8)$$

$$w \models [0] \iff w = \varepsilon \quad (9)$$

$$w \models [1..]t \iff \text{either } w \models t \text{ or } \exists u, v \text{ s.t. } w = uv \text{ and } u \models t \text{ and } v \models [1..]t \quad (10)$$

$$w \models t @ e \iff (w = \varepsilon \text{ and } \varepsilon \models t) \text{ or } (\exists u, \ell, v \text{ s.t. } u\ell v = w \text{ and } u\ell \models t \text{ and } \ell v \models \mathbf{cycle @ e}) \quad (11)$$

$$w \models t @ e \iff (w = \varepsilon \text{ and } \varepsilon \models t) \text{ or } (\exists u, \ell, v \text{ s.t. } u\ell v = w \text{ and } u\ell \models t \text{ and } \ell v \models \mathbf{cycle @ e}) \quad (12)$$

$$w \models \mathbf{fm } t \iff w \models t \text{ and } \forall u, v \text{ uv} = w, |v| > 0 : \neg(u \models t) \quad (13)$$

$$w \models \mathbf{fail } t \iff (\forall u : \neg(wu \models t)) \text{ and } (\forall u, v \text{ uv} = w, |v| > 0 : \neg(u \models \mathbf{fail } t)) \quad (14)$$

2.3 Sampled-normal form

Every top-level temporal expression has a sampling event. If the top-level expression is of the form $t @ e$, then e is its sampling event. Otherwise if that top-level temporal expression is embedded in a TCM, it inherits the sampling event from that TCM. In a last case, the top-level temporal expression appears under an event, expect or assume struct member; in that case its sampling event is **any**.

The first step to uncover the semantics of a top-level temporal expression is to put it in *sampled-normal form*. Formally this is accomplished by a function \mathcal{S} that takes as first argument a temporal expression and as second argument an event, and produces a temporal expression in sampled-normal form as a result.

For a top-level temporal expression t whose sampling event is q , $\mathcal{S}(t, q)$ gives the sampled-normal form of t .

\mathcal{S} is defined recursively as follows:

$$\mathcal{S}(a, q) = a @ q \quad (15)$$

$$\mathcal{S}((t), q) = (\mathcal{S}(t, q)) \quad (16)$$

$$\mathcal{S}(t_1 \mathbf{and} t_2, q) = \mathcal{S}(t_1, q) \mathbf{and} \mathcal{S}(t_2, q) \quad (17)$$

$$\mathcal{S}(t_1 \mathbf{or} t_2, q) = \mathcal{S}(t_1, q) \mathbf{or} \mathcal{S}(t_2, q) \quad (18)$$

$$\mathcal{S}(t_1 ; t_2, q) = \mathcal{S}(t_1, q) ; \mathcal{S}(t_2, q) \quad (19)$$

$$\mathcal{S}([0], q) = [0] \quad (20)$$

$$\mathcal{S}([1..]t, q) = [1..]\mathcal{S}(t, q) \quad (21)$$

$$\mathcal{S}(t @ e, q) = \mathcal{S}(t, e) @ q \quad (22)$$

$$\mathcal{S}(\mathbf{fm} t, q) = \mathbf{fm} \mathcal{S}(t, q) \quad (23)$$

$$\mathcal{S}(\mathbf{fail} t, q) = \mathbf{fail} \mathcal{S}(t, q) \quad (24)$$

2.4 Semantics of temporal struct members

2.4.1 Event declaration

Consider the following struct member declaration:

$$\mathbf{event} \ e \ \mathbf{is} \ t @ q$$

Let w be the trace, then this construct defines an event e as follows:

$$w^i \models e \iff \exists j \leq i \text{ s.t. } (\underline{\text{either}} (j = 0) \ \underline{\text{or}} \ w^{j-1} \models q) \ \underline{\text{and}} \ w^{j..i} \models t @ q$$

[In other words, e holds at point i if either the prefix of the trace ending with i matches $t @ q$, or there is a earlier point j so that q holds at $j - 1$ and the part of the trace starting with j and ending in i matches $t @ q$.]

2.4.2 Expect declaration

Consider the following struct member declaration:

$$\mathbf{expect} \ P \ \mathbf{is} \ t @ q$$

This is equivalent to the definition of an event:

$$\mathbf{event} \ P \ \mathbf{is} \ \mathbf{fail} \ t @ q$$

Any occurrence of this event will be reported as a violation of property P .

3 Derived operators and syntactic sugar

To make it easier to map the concrete syntax onto the abstract syntax, we define the following syntactic sugar for the abstract syntax:

$$[..]t = ([0]) \text{ or } [1..]t \quad (25)$$

$$[..n]t = ([0]) \text{ or } [1]t \text{ or } \dots \text{ or } [n]t \quad (26)$$

$$[n..]t = [n-1]t ; [1..]t \quad (27)$$

$$[m..n]t = [m]t ; [..(n-m)]t \quad (28)$$

$$[0]t = [0] \quad (29)$$

$$[n]t = \underbrace{t; t; \dots t}_{n \text{ terms}} \quad (30)$$

Table 2 summarizes the mapping of the concrete syntax to the abstract syntax.

Concrete Syntax	Abstract Syntax	Name
cycle	cycle	<i>True</i>
@any	any	<i>Any</i>
@e	<i>e</i>	<i>Event</i>
true(exp)	<i>p</i>	<i>Proposition</i>
@e @q	<i>e @ q</i>	<i>Sample</i>
true(expr) @q	<i>p @ q</i>	<i>Sample 2</i>
<i>t @q</i>	<i>t @ q</i>	<i>Sample 3</i>
<i>t₁ or t₂</i>	<i>t₁ or t₂</i>	<i>Disjunction</i>
<i>t₁ and t₂</i>	<i>t₁ and t₂</i>	<i>Conjunction</i>
{t₁ ; t₂}	<i>t₁ ; t₂</i>	<i>Sequence</i>
fail t	fail t	<i>Failure</i>
<i>t₁ => t₂</i>	(fail t₁) or (t₁ ; t₂)	<i>Yield</i>
[n] * t	[n]t	<i>Repeat</i>
~[...] * t	[..]t	<i>True Match</i>
~[..n] * t	[..n]t	<i>True Match</i>
~[n..] * t	[n..]t	<i>True Match</i>
~[m..n] * t	[m..n]t	<i>True Match</i>
{[...] * t₁ ; t₂}	fm ([..]t₁ ; t₂)	<i>First Match</i>
{[..m] * t₁ ; t₂}	fm ([..m]t₁ ; t₂)	<i>First Match</i>
{[m..] * t₁ ; t₂}	fm ([m..]t₁ ; t₂)	<i>First Match</i>
{[m..n] * t₁ ; t₂}	fm ([m..n]t₁ ; t₂)	<i>First Match</i>

Table 2: Summary of temporal e operators

4 To be done

- detach

- rise, fall, change
- delay
- not