

# Semantics of Temporal *e*

Verisity Ltd.

8 Hamelacha St., Rosh-Ha'ain, 48091 Israel

December 17, 2003

---

This article summarises the syntax and mathematical semantics of the temporal fragment of the *e* language. This fragment of *e* is essentially a linear time temporal logic interpreted over sequences of program states. The following is not intended as an introduction to Specman's temporal language but, rather, to serve as the definition of the language described in the reference manuals, and as the starting point for future developments to the language.

---

## 1 Introduction

The *e* language has evolved to meet the functional verification requirements of the builders of modern digital electronic devices such as processor cores, DSPs, micro-controllers, memory interfaces, and so on. An *e verification program* is most often a model of the environment in which such devices are intended to operate. A verification program may include more or less detailed models of the hardware under test, and of the hardware/software devices with which it communicates. The *e* language therefore provides facilities for modelling both the *data* processed by such components, as well as the *communications* through which they interact.

The *e* language is well suited to modelling data: it is an object-oriented programming language that combines imperative features for capturing sequential and concurrent algorithms, declarative features for expressing constraints over data, and a flexible extension mechanism that supports rapid prototyping. These features combine in *e* with a powerful constraint-based random generation engine for highly effective, dynamic creation of stimuli for the device under test.

For modelling communication protocols, and the orchestration of concurrent activities within a model, *e* provides primitives for spawning, forking and joining concurrent threads of execution. Synchronisation between threads is achieved via *events* that are broadcast so that they are simultaneously visible throughout the program. Events in *e*

carry no data, but communication between threads can be achieved via shared variables protected, when necessary, by semaphores.

Events in an  $e$  verification program typically represent outputs from the hardware device under test; they may also represent interesting scenarios such as the rules specifying communications protocols. In order to declare such rules, either for implementation or verification purposes,  $e$  provides a number of *temporal operators* for combining events into complex temporal expressions. This sublanguage, called *temporal  $e$* , is a linear temporal logic reminiscent of Choppy Logic [4] and Regular Temporal Logic [2], that has a natural syntax for expressing complex sequences of events.

Since the temporal language is central to our understanding of concurrent computation in  $e$ , and since this language enables formal verification of  $e$  and HDL models, it is the purpose of this article to give temporal  $e$  a rigorous mathematical semantics. We begin in Section 2 with a brief introduction to the main constructs of the language (the intended semantics of the basic operators), and some illustrations of the contexts in which they are typically employed. The denotational (or *logical*) semantics of the language are gradually developed through Section 3, which also introduces the very minimal mathematical concepts required, and Section 4. The final Section 5 concludes with a summary and some discussion of the expressiveness and complexity issues raised by the formal semantics.

## 2 Introducing temporal $e$

To convey some of the intended semantics it is prudent to begin with an illustration of the contexts in which temporal expressions arise. The examples in Section 2.1 show temporal expressions used to define events, temporal properties, and actions in the body of processes implementing reactive behaviour. Abstract syntax is discussed in Section 2.2.

### 2.1 Temporal contexts

#### 2.1.1 Event definitions

The first of these temporal contexts is the *event definition*. For example the declaration

```
event ready is {@reset; [3]} @clock
```

states that the event `ready` will occur three ‘clock cycles’ after a `reset`. The `clock` event here is referred to as a *sampling event*, and the term in braces is a *sequence*. The first term of the sequence is just the occurrence of the event `reset` (sampled, in

this example, by the `clock` event); the second term, `[3]`, abbreviates a sequence of three clock events. To a close approximation the *evaluation* of this sequence begins with the occurrence of a `reset` event, and terminates successfully, or *succeeds*, on the fourth clock thereafter.

### 2.1.2 Expect declarations

The second context in which temporal expressions play a crucial role is in the declaration of temporal properties. These may be assertions about the behaviour of the program in which they reside, or assumptions about its environment. An example might be something like:

```
expect @start => {[3..5]; @stop or @restart} @clock
```

The *expect* introduces a global invariant. Initially we expect a `start` event sampled by `clock`; if a `start` event does not occur before the `clock` the temporal expression as a whole succeeds. Otherwise, within three to five further cycles we expect to see either a `stop` or a `restart` event. If more than five cycles elapse following a `start` event without a `stop` or `restart` event occurring—that is, if the temporal expression fails—then the expect declaration will cause the program to halt with an appropriate error message.

### 2.1.3 Wait actions

The *expect* and *event* declarations introduce temporal expressions that are continually reevaluated throughout the lifetime of the program structure in which they are declared. A *wait* action is not usually so long-lived. This construct is used as an action in regular sequential program code to pause the execution at this location until the embedded temporal expression succeeds. For example the `wait` in

```
...; wait {@ready; @start}; busy = 1; ...
```

pauses the execution of the program until a `ready` event is followed on the next cycle by a `start` event. Only when the temporal expression succeeds does the program resume—by setting the `busy` variable, and continuing with the rest of the thread that follows. The semantics of the *wait* action implies that if the embedded temporal expression never succeeds, then the program (thread) will never resume execution.

## 2.2 Abstract syntax

When developing the formal semantics for a programming language it is normally best to focus on the abstract syntax, rather than the concrete syntax, so as not to burden the

mathematical developments with concerns of a lexical nature such as operator precedence, token strings, and so on. Generally the abstract syntax corresponds closely to the syntax tree built the parser—but in any case it is presumed to be composed *only* of those program phrases that have independent semantic significance.

$$t ::= \varepsilon \mid a \mid any \mid \bigvee_{i \in I} t_i \mid \{t_1; t_2\} \mid \neg t \mid \mathbf{pre} \ t \mid t@q$$

Table 1: Abstract syntax of temporal  $e$

Anticipating for a moment that, in common with other linear temporal languages, expressions in temporal  $e$  will be interpreted over sequences of *states*, the following intuitions can be supplied regarding the operators defined in Table 1.

**Empty**  $\varepsilon$  – this construct represents the empty sequence which, although vacuous in itself, plays a pivotal role in the semantics of temporal  $e$ .

**Atom**  $a$  – an atom may be an *event* or a *proposition*, either determining a sequence of length one where that atom is present/true.

**Cycle**  $any$  – this distinguished event is used to denote the passage of time according to some external clock.

**Disjunction**  $t_1 \vee t_2$  – this represents the simultaneous possibilities of both  $t_1$  and  $t_2$ . The index set  $I$  may be finite or infinite.

**Sequence**  $\{t_1; t_2\}$  – this represents the possibilities of  $t_1$  followed (beginning at the next cycle) by  $t_2$ . *Sequence* is the main temporal operator of the language.

**Negation**  $\neg t$  – this complements the expression  $t$  so that a sequence of states satisfying  $t$  will not satisfy  $\neg t$ , and vice versa.

**Prefix**  $\mathbf{pre} \ t$  – if  $t$  specifies a sequence of states the *Prefix* operator specifies all prefixes of that sequence (including  $t$  and the empty sequence).

**Sample**  $t@q$  – the expression is clocked, or *sampled*, according to the named event  $q$ . To a first approximation this is a projection of the sequence defined by  $t$ .

Neither *Negation* nor *Prefix* are available in the concrete syntax of temporal  $e$ ; they are used in the sequel to define less powerful, but more useful, temporal operators such as *failure*, and a form of *until* (see Section 3.3).

### 3 Denotational semantics

The denotation of a temporal expression tells us what property that expression defines. It is simplest to explain the theory by first considering a negation-free subset of temporal  $e$ ; this language is then developed to include the central notion of a *First Match*, and the concept of *Failure*. We begin with some notational preliminaries.

#### 3.1 Preliminaries

##### 3.1.1 Domain of discourse

Expressions in temporal  $e$  are built out of sequences of atomic entities which may be composed in various ways. There are two types of atom:

- Let  $\mathcal{P}$  be a set of *proposition symbols*, and let  $p \in \mathcal{P}$  be a representative element.
- Let  $\mathcal{E}$  be a set of *event names*, and let  $e, q \in \mathcal{E}$  be representative elements.
- Let  $\mathcal{A} = \mathcal{P} + \mathcal{E}$  be the set of *atoms* (atomic propositions and events) and let  $a \in \mathcal{A}$  be a representative element of this set.

Propositions represent relations over program variables. A distinguished ‘event’ *any* is used to represent time determined by some external clock. Events carry no memory from one cycle to another, whereas the data underlying propositions are state retaining.

##### 3.1.2 Interpretations

Temporal expressions are interpreted over sequences of states.

- Let  $\Sigma = 2^{\mathcal{A}}$  and  $s \in \Sigma$  (or  $s : \mathcal{A} \rightarrow \{0, 1\}$ ) be a *state*. If  $a \in s$  the interpretation is that the atomic event is present (or the atomic proposition is true) in that state.
- Let  $\Sigma^*$  be the (infinite) set of *all finite sequences* over  $2^{\mathcal{A}}$ , and let  $\sigma \in \Sigma^*$  be a representative element.
- If  $\sigma$  is the sequence of states  $\langle s_1, s_2, \dots, s_n \rangle$  then  $|\sigma| = n \geq 1$  is the length of the sequence. The empty sequence is represented by  $\epsilon$ .
- If  $\sigma_1$  and  $\sigma_2$  are sequences then  $\sigma_1\sigma_2$  denotes the *concatenation* of these two sequences.  $|\sigma_1\sigma_2| = |\sigma_1| + |\sigma_2|$

Finally, the notation  $\sigma \simeq \sigma'$  recalls the notion of *matching* in regular expression languages:  $\langle s_1, s_2, \dots, s_n \rangle \simeq \langle s'_1, s'_2, \dots, s'_n \rangle$  whenever  $s_i \in s'_i$  for  $i = 1..n$ .

### 3.1.3 Models

Finally, suppose  $t$  is a temporal expression then the *denotation*, or meaning, of  $t$  is a set  $\|t\| \subseteq \Sigma^*$ . The sequence  $\sigma$  *satisfies*, or is a model of, the expression  $t$  if and only if  $\sigma$  is in the denotation—i.e.,  $\sigma \models t$  iff  $\sigma \in \|t\|$ . The set  $\|t\|$  is inductively defined according to the syntactic structure of the expression  $t$  as we shall see in the following section. Readers unhappy with the notion of a denotation can instead regard  $\|t\|$  as a notation for the *language* defined by  $t$ , often written  $L(t)$ .

## 3.2 The negation-free language

### 3.2.1 Abstract syntax

The negation-free subset of temporal  $e$  is defined here in order to introduce the main semantic notions without confusing matters by trying to simultaneously explain the more difficult constructs of the full temporal language. This sublanguage is only ‘negation-free’ in the sense that negation (or complement) applies to atoms, not terms. The syntax is succinctly captured in the inductive definition of Table 2. With the excep-

$$t ::= \varepsilon \mid \text{any} \mid a \mid \bar{a} \mid \bigvee_{i \in I} t_i \mid t_1 \wedge t_2 \mid \{t_1; t_2\}$$

Table 2: The negation-free subset of temporal  $e$

tions of  $\bar{a}$  which represents the complement of the atom  $a$ , and the conjunction  $t_1 \wedge t_2$ , these constructs were informally defined in Section 2.2.

### 3.2.2 Semantics

Given a negation-free temporal expression  $t$ , the meaning  $\|t\|$  is inductively defined in Table 3. From the definitions we see that  $\|\text{any}\|$  is the set of all sequences that are of length one; similarly,  $\|a\|$  (respectively,  $\|\bar{a}\|$ ) is the set of all sequences of length one in which the atom holds (respectively, does not hold) true at the first and only state. For the Boolean constructs note that *Disjunction* is just the union of the sets of sequences defined by the individual components; for *Conjunction* we take the intersection. As for *Sequence*, the term  $\{t_1; t_2\}$  denotes the set of sequences that can be divided somewhere along their length such that the first subsequence satisfies  $t_1$ , while the second subsequence satisfies  $t_2$ . Thus  $;$  is interpreted as sequence *concatenation*—in contrast to the *fusion* semantics of this operator in Interval Temporal Logic [1, 3].

<i>Empty</i>	$\ \varepsilon\ $	$\stackrel{\text{def}}{=} \{\}$
<i>Cycle</i>	$\ \text{any}\ $	$\stackrel{\text{def}}{=} \{\sigma :  \sigma  = 1\}$
<i>Atom</i>	$\ a\ $	$\stackrel{\text{def}}{=} \{\sigma : \sigma = \langle s \rangle, a \in s\}$
<i>Complement</i>	$\ \bar{a}\ $	$\stackrel{\text{def}}{=} \{\sigma : \sigma = \langle s \rangle, a \notin s\}$
<i>Disjunction</i>	$\ \bigvee_{i \in I} t_i\ $	$\stackrel{\text{def}}{=} \bigcup_{i \in I} \ t_i\ $
<i>Conjunction</i>	$\ t_1 \wedge t_2\ $	$\stackrel{\text{def}}{=} \ t_1\  \cap \ t_2\ $
<i>Sequence</i>	$\ \{t_1; t_2\}\ $	$\stackrel{\text{def}}{=} \{\sigma : \exists \sigma_1 \sigma_2. \sigma = \sigma_1 \sigma_2, \sigma_1 \in \ t_1\ , \sigma_2 \in \ t_2\ \}$

Table 3: Semantics of the negation-free language

**Example 3.1** For atomic  $x$ ,  $y$ , and  $z$ :

$$\|x \wedge \bar{y} \vee \{y; z\}\| = \{\sigma : \sigma = \langle s \rangle, x \in s, y \notin s\} \cup \{\sigma : \sigma = \langle s_1, s_2 \rangle, y \in s_1, z \in s_2\}$$

Satisfying sequences  $\sigma$  match  $\langle x\bar{y} \rangle | \langle y, z \rangle$  (using a regular expression like syntax). ♣

### 3.2.3 Fixed repeat

Having defined the kernel of the negation-free language of temporal expressions we can now derive a number of (more recognisable) temporal operators. *Fixed Repeat* is an  $n$ -fold repetition with a slight twist:

$$[n]t \stackrel{\text{def}}{=} \begin{cases} \{t; [n-1]t\} & \text{if } n > 0 \\ \varepsilon & \text{if } n = 0 \end{cases}$$

Note that since  $\|\varepsilon\| = \{\}$  this definition implies  $\{\}$  is the unique model for  $[0]t$ , for all  $t$ . As a second consequence we have  $\|\{t_1; [0]t_2; t_2\}\| = \|\{t_1; t_2\}\|$ , for all  $t$ ,  $t_1$ , and  $t_2$ .

### 3.2.4 True match variable repeat

This operator is rarely used in temporal  $e$  in practice. However, the operator is important for the definition of *First Match Repeat* which follows in Section 3.3, and which is frequently used in practice. *True Match Repeat* has four subcases:

$$\begin{aligned} [m..n]t &= \{[m]t; [..n-m]t\} & [m..]t &= \{[m]t; [..]t\} \\ [..n]t &\stackrel{\text{def}}{=} \bigvee_{0 \leq k \leq n} [k]t & [..]t &\stackrel{\text{def}}{=} \bigvee_{k \geq 0} [k]t \end{aligned}$$

These operators are well defined only if  $0 \leq m \leq n$ . Note that  $\in \|[..n]t\|$  and that

$$[..n]t = [0]t \vee [1]t \vee [2]t \vee \cdots \vee [n]t$$

while  $[..]t$  is an infinite disjunction of finite sequences. Intuitively a sequence satisfies  $[..n]t$  if and only if it is a concatenation of up to  $n$  sequences satisfying  $t$ .

**Example 3.2** Two instances of *True Match* are useful for expressing eventualities.

- The first is  $[\dots]any$  (usually abbreviated to  $\sim[\dots]$  in the concrete syntax) which specifies an arbitrary sequence of finite, but unbounded, length.
- The second is  $[\dots]\bar{a}$  which specifies that the atom  $a$  is continuously unsatisfied over a sequence of finite, but unbounded, length.

The positive form of the latter, viz  $[\dots]a$ , is also of interest: ‘globally  $a$ .’ ♣

Notice that  $\|[\dots]any\| = \Sigma^*$  and so we may reasonably think of this property as (temporal) *true*—the property all models satisfy.

### 3.2.5 Eventually

We use the above examples of *True Match Repeat* to define:

$$\begin{aligned} \blacklozenge a &\stackrel{\text{def}}{=} \{[\dots]any; a\} && \text{True match eventually } a \\ \blacklozenge a &\stackrel{\text{def}}{=} \{[\dots]\bar{a}; a\} && \text{First match eventually } a \end{aligned}$$

It is not difficult to see that  $\|\blacklozenge a\|$  includes all finite sequences of states  $\sigma, |\sigma| \geq 1$ , where the atom  $a$  is true in the final state; in contrast,  $\|\blacklozenge a\| \subset \|\blacklozenge a\|$  includes only those sequences where the atom *only* holds in the final state. In the negation-free language the  $\blacklozenge$  operator only applies to atoms or their complements. We shall see in Section 3.3 how both eventually operators generalise to arbitrary temporal expressions.

## 3.3 First Match, and Failure

The negation-free subset of temporal  $e$  is a useful sublanguage, but it is not quite sufficient to capture properties expressed using the *until* operator available in most linear temporal logics. Nor can we express the concept of the failure of a temporal expression. These concepts require the introduction of negation applied to arbitrary temporal expressions, not just the atoms. The syntax

$$t ::= \varepsilon \mid any \mid a \mid \bigvee_{i \in I} t_i \mid \{t_1; t_2\} \mid \neg t \mid \mathbf{pre} \ t$$

adds *Negation* and *Prefix* to the negation-free language (Table 1 on page 4). The semantics of these two operators are specified in Table 4.

While the meaning of  $\neg t$  is rather easy to explain the operation itself is far from straightforward to compute (when model checking, or when computing simulation runs); its use is therefore restricted in the concrete syntax. The **prefix** operator specifies the set of all sequences that can be extended to  $\|t\|$ . Notice that  $\|t\| \subseteq \|\mathbf{pre} \ t\|$ , and that  $\in \|\mathbf{pre} \ t\|$ .

These operators are used to define the **firstmatch** and **fail** temporal operators.

<i>Negation</i>	$\ \neg t\ $	$\stackrel{\text{def}}{=} \Sigma^* - \ t\ $
<i>Prefix</i>	$\ \mathbf{pre} t\ $	$\stackrel{\text{def}}{=} \{\sigma : \exists \sigma'.  \sigma'  \geq 0, \sigma\sigma' \in \ t\ \}$

Table 4: Semantics of *Negation* and *Prefix*, extending Table 3 definitions

### 3.3.1 Firstmatch

Earlier we defined  $\diamond a$  as  $\{[..]\bar{a}; a\}$  and called this operator *first match eventually*. The reason this is a ‘first match’ is that it specifies the shortest sequence that satisfies  $a$  eventually—i.e., no prefix of  $\diamond a$  also satisfies the expression. This notion generalises as follows:

$$\mathbf{fm} t \stackrel{\text{def}}{=} t \wedge \neg\{t; [1..]any\}$$

Note that  $\|[1..]any\| = \Sigma^+$ . Applying the semantics, and simplifying, this yields

$$\|\mathbf{fm} t\| = \|t\| - \{\sigma : \exists \sigma_1 \sigma_2. \sigma = \sigma_1 \sigma_2, \sigma_1 \in \|t\|, |\sigma_2| \geq 1\}$$

$\|\mathbf{fm} t\|$  includes all sequences  $\|t\|$  except those having a (strict) prefix in  $\|t\|$ .

### 3.3.2 First match variable repeat

First and foremost the **firstmatch** operator is used to define *First Match Repeat*, and the general version of first match eventually:

$$\begin{aligned} t_1 \mathcal{U}_m t_2 &\stackrel{\text{def}}{=} \mathbf{fm} \{[..m]t_1; t_2\} && \text{First match repeat } t_1 \text{ until } t_2 \text{ (for } m \geq 0) \\ t_1 \mathcal{U} t_2 &\stackrel{\text{def}}{=} \mathbf{fm} \{[..]t_1; t_2\} && \text{First match repeat } t_1 \text{ until } t_2 \text{ (unbounded)} \end{aligned}$$

The bounded first match repeat specifies a sequence where the first suffix that satisfies  $t_2$  follows the concatenation of at most  $m$  sequences satisfying  $t_1$ ; the unbounded first match repeat allows the leading expression to repeat indefinitely until  $t_2$  is matched.

In analogy to the eventually operators in Section 3.2.5 we can define ‘eventually  $t$ ’ for arbitrary temporal expressions:

$$\begin{aligned} \diamond t &\stackrel{\text{def}}{=} \mathbf{fm} \{[..]any; t\} && \text{First match eventually } t \\ \blacklozenge t &\stackrel{\text{def}}{=} \{[..]any; t\} && \text{True match eventually } t \end{aligned}$$

It is not difficult to check that  $\|\mathbf{fm} \{[..]any; a\}\| = \|\{[..]\bar{a}; a\}\|$ , for atom  $a$ , so that the definition above agrees with the characterisation of  $\diamond a$  given in Section 3.2.5.

**Example 3.3** Given atoms  $a$  and  $b$ , consider  $u \equiv a \mathcal{U}_2 b$  which, intuitively, specifies that  $b$  follows at most two  $a$ ’s. Unrolling the definitions  $\|\mathbf{fm} \{[..2]a; b\}\| = \|a \mathcal{U}_2 b\| = \mathbf{A} - \mathbf{B}$  where

$\mathbf{A} = \|\{[..2]a;b\}\|$  are the *true matches*, and

$\mathbf{B} = \{\sigma : \exists \sigma_1 \sigma_2. \sigma = \sigma_1 \sigma_2, \sigma_1 \in \mathbf{A}, |\sigma_2| \geq 1\}$  the true matches that are too long.

The set  $\mathbf{A}$  is the union of the disjoint sets  $\{\sigma \simeq \langle b \rangle\}$ ,  $\{\sigma \simeq \langle a, b \rangle\}$ , and  $\{\sigma \simeq \langle a, a, b \rangle\}$ , but clearly a sequence like  $\langle ab, b \rangle \in \{\sigma \simeq \langle a, b \rangle\}$  is not a *first match*. The set  $\mathbf{B}$  cuts out the sequences  $\{\sigma \simeq \langle b, 1 \rangle\}$ , and  $\{\sigma \simeq \langle b, 1, 1 \rangle\} \cup \{\sigma \simeq \langle a, b, 1 \rangle\}$ , which leaves

$$\|a \mathcal{U}_2 b\| = \{\sigma \simeq \langle b \rangle\} \cup \{\sigma \simeq \langle \bar{a}\bar{b}, b \rangle\} \cup \{\sigma \simeq \langle \bar{a}\bar{b}, \bar{a}\bar{b}, b \rangle\} \quad (\dagger)$$

as the satisfying sequences for this first match expression.  $\clubsuit$

### 3.3.3 Fail

Continuing with the example above, the set of sequences  $(\dagger)$  specifies precisely those sequences for which the evaluation of the temporal expression  $u \equiv a \mathcal{U}_2 b$  *succeeds*. Conversely, one can ask when the expression fails. The three sequences

$$(a) \ \langle \bar{a}\bar{b}, \bar{a}\bar{b} \rangle \quad (b) \ \langle \bar{a}\bar{b}, 1 \rangle \quad (c) \ \langle \bar{a}\bar{b}, \bar{a}\bar{b}, b, 1 \rangle$$

are members of the set  $\|\neg u\|$ , but only (a) is a *failure* according to the semantics of temporal  $e$ . This is because **fail**  $t$  specifies only the shortest sequences that do not satisfy  $t$ . Thus (b) is rejected since the prefix  $\langle \bar{a}\bar{b} \rangle$  is a failure of  $u$ , and (c) is rejected because it is too long—the prefix  $\langle \bar{a}\bar{b}, \bar{a}\bar{b}, b \rangle \in \|u\|$ .

The formal definition of **fail**  $t$  is cumbersome

$$\begin{aligned} \mathbf{fail} \ t &\stackrel{\text{def}}{=} \neg\{t; [1..]any\} \wedge \mathbf{fm} (\neg \mathbf{pre} \ t) \\ &= \underbrace{\neg\{t; [1..]any\}}_{\mathbf{A}} \wedge \underbrace{\neg \mathbf{pre} \ t}_{\mathbf{B}} \wedge \underbrace{\neg\{\neg \mathbf{pre} \ t; [1..]any\}}_{\mathbf{C}} \end{aligned}$$

but some intuition can be applied to the various components:

- A** no sequence that satisfies **fail**  $t$  can be an extension of a sequence that satisfies  $t$ , so  $\|\mathbf{fail} \ t\| \subseteq \|\neg\{t; [1..]any\}\|$ ;
- B** from the set  $\|\neg\{t; [1..]any\}\|$  we must remove all prefixes of  $t$  since these *might* satisfy  $t$  by definition;
- C** lastly, we include only the shortest sequences that deviate from being prefixes of  $t$ , hence we take the *first match* of  $\neg \mathbf{pre} \ t$  (i.e., the set  $\mathbf{B} \cap \mathbf{C}$ ).

The advantage of **fail**  $t$  over  $\neg t$  is that it is simpler to implement for the purpose of property checking (not that this is obvious from the above definition).

**Example 3.4** The set complements arising from application of the embedded negations in the definition of **fail** make computation of  $\|\mathbf{fail} t\|$  far from straightforward, even for relatively simple  $t$ . The intrepid can nevertheless verify that for atomic  $a, b$ :

- $\|\mathbf{fail} a\| = \{\sigma \simeq \langle \bar{a} \rangle\} = \{\sigma : \sigma = \langle s \rangle, a \notin s\}$
- $\|\mathbf{fail} \{a; b\}\| = \{\sigma \simeq \langle \bar{a} \rangle\} \cup \{\sigma \simeq \langle a, \bar{b} \rangle\}$
- $\|\mathbf{fail} a \mathcal{U}_2 b\| = \{\sigma \simeq \langle \bar{a}\bar{b} \rangle\} \cup \{\sigma \simeq \langle a\bar{b}, \bar{a}\bar{b} \rangle\} \cup \{\sigma \simeq \langle a\bar{b}, a\bar{b}, \bar{b} \rangle\}$
- $\|\mathbf{fail} \varepsilon\| = \|\mathbf{fail} [n]any\| = \|\mathbf{fail} [..]any\| = \|\mathbf{fail} [..m]any\| = \{\}$

Note that  $\mathbf{fail} a$  was denoted by  $\bar{a}$  in the negation-free language. If  $\|\mathbf{fail} t\| = \{\}$  for some  $t$ , as in the latter example above, this indicates that  $t$  cannot fail. Whereas earlier we noted that  $[..]any$  represented the temporal property *true*, the empty set of sequences  $\{\}$  represents *temporal false*—a property no model can satisfy. ♣

## 4 Sampling

In Section 3 we defined a rich collection of temporal operators, but omitted one of the more important concepts—sampling  $t@q$  for events  $q \in \mathcal{E} \cup \{any\}$  called *sampling events*. The *Sample* operator  $t@q$  has to be introduced as a primitive of the language, thus:

$$t ::= \varepsilon \mid any \mid a \mid \bigvee_{i \in I} t_i \mid \{t_1; t_2\} \mid \neg t \mid \mathbf{pre} t \mid t@q$$

The syntax  $t@q$  conceals considerable complexity for there are three cases to consider. Firstly the sampling of atomic expressions in Section 4.1 introduces a distinction between propositions on the one hand, and events on the other. Secondly, sampling non-atomic temporal expressions in Section 4.3 highlights the ‘sampling’ effect that the event  $q$  has on the expression  $t$ . This is explained in Section 4.2.

### 4.1 Atomic cases

Hitherto there has been nothing to distinguish events  $e \in \mathcal{E}$  from propositions  $p \in \mathcal{P}$ . A distinction only emerges in the presence of sampling events. Given a named event  $q \in \mathcal{E} \cup \{any\}$  we consider the case of (a) an event  $e$  sampled at the event  $q$ , and (b) a proposition  $p$  sampled at  $q$ . The difference in intended meaning between these two expressions,  $e@q$  and  $p@q$  respectively, is illustrated in Figure 1. In the case of  $e@q$ , the occurrence of the event  $e$  is *latched until* the sampling event occurs (unless the two events coincide). For  $p@q$  the Boolean expression  $p$  is *evaluated when* the sampling event occurs.

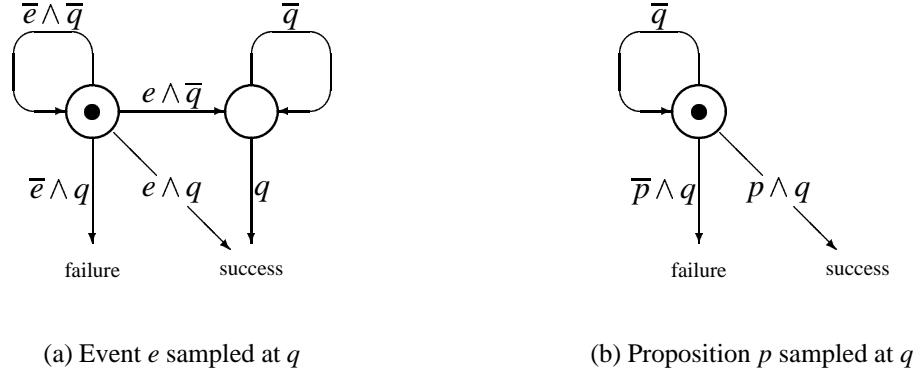


Figure 1: Sampling atomic events and propositions

It is straightforward to interpret Figure 1(a) in  $e$ :

$$e@q \text{ ‘equals’ } (\bar{e} \wedge \bar{q}) \mathcal{U} ((e \wedge q) \vee \{(e \wedge \bar{q}); \diamond q\})$$

All components of the figure are represented here. The repeated expression in this *First Match Repeat* is  $(\bar{e} \wedge \bar{q})$ . If  $q$  occurs before an  $e$  the temporal expression fails—this is the only way  $e@q$  can fail. If an  $e$  occurs simultaneously with  $q$  the expression succeeds immediately; otherwise if the  $e$  precedes the sampling event we’ll wait, perhaps indefinitely, for the next  $q$ . Note that the above interpretation for  $e@q$  is equivalent to  $\diamond((e \wedge q) \vee \{(e \wedge \bar{q}); \diamond q\})$ .

To interpret Figure 1(b) in a similar way is simpler because the sampling always coincides with the evaluation of  $p$ :

$$p@q \text{ ‘equals’ } \blacklozenge p \wedge \diamond q$$

The conjunct  $\diamond q$  succeeds on the first  $q$  event; the true match eventually  $p$  in the other conjunct succeeds on every cycle that  $p$  is true. So, if  $p$  is true on the first occurrence of  $q$  then  $p@q$  succeeds, and if  $p$  is false in that cycle then the expression  $p@q$  fails.

Now the interpretations listed above are not definitions of the meaning of  $a@q$ . The precise definitions are not syntactic, but semantic, as follows:

$$\|e@q\| \stackrel{\text{def}}{=} \|(\bar{e} \wedge \bar{q}) \mathcal{U} ((e \wedge q) \vee \{(e \wedge \bar{q}); \diamond q\})\| \quad (1)$$

$$\|p@q\| \stackrel{\text{def}}{=} \|\blacklozenge p \wedge \diamond q\| \quad (2)$$

The distinction may not appear to be very interesting, but it turns out that one cannot substitute the expression  $p@q$  (say) with the ‘replacement’  $\blacklozenge p \wedge \diamond q$  in all contexts—in particular, this substitution is not valid in the context of an outer application of the *Sample* operator. This topic is discussed further in Section 4.3.

**Example 4.1** It is useful to compute  $\|a@q\|$  for atomic events and propositions:

$$\begin{aligned}\|e@q\| &= \langle \bar{e}q \rangle^* \langle \langle eq \rangle | \langle e\bar{q} \rangle \langle \bar{q} \rangle^* \langle q \rangle \rangle \\ \|p@q\| &= \langle \bar{q} \rangle^* \langle pq \rangle\end{aligned}$$

(borrowing again from the syntax of regular expressions). In the case that  $q \equiv \text{any}$  notice that  $\|a@q\| = \|a\| = \{\sigma \simeq \langle a \rangle\}$ , which is consistent with the notion that *any* is the *fastest clock*. Lastly, if  $e \equiv \text{any}$  it is easy to check that  $\|e@q\| = \|\diamond q\|$ . ♣

## 4.2 Sampled normal form

The definitions (1) and (2) apply only to atoms. Applying ‘@ $q$ ’ to an arbitrary temporal expressions  $t$  means that sampling events (or clocks) may be nested. The syntax  $t@q$  is complicated by serving two purposes simultaneously:

- Firstly, the ‘@ $q$ ’ specifies that the success of  $t$  is latched, if necessary, until the sampling event occurs (this is a trailing  $\diamond q$ , in effect—see Section 4.3).
- Secondly, the ‘@ $q$ ’ supplies a *default* sampling event for all subexpressions of  $t$  that do not supply their own sampling event.

We deal with this latter point when defining the semantics of sampling by insisting that all subexpressions under a sampling event are explicitly sampled. This transformation (*normalisation*, or *clock propagation*) is carried out by the function  $\llbracket t \rrbracket_q$  in Table 5. In most cases the transformation  $\llbracket t \rrbracket_q$  just pushes the sampling event ‘through’ the operator; the interesting cases are *Atom* and *Sequence* (where the sampling event is retained), and *Sample* itself.

The intuition behind the clock propagation rule  $\llbracket \{t_1; t_2\} \rrbracket_q = \{\llbracket t_1 \rrbracket_q; \llbracket t_2 \rrbracket_q\} @q$  is that the effect the sampling operator has is to modify the behaviour of ; so that transitions over the sequence happen not on the default clock *any* but on the (slower) clock  $q$ . Then the clause for  $t@q$  specifies that the outer sampling event ceases to have immediate effect on the term and that the inner sampling event takes precedence. These two rules indicate that some care has to be exercised when applying clock propagation to the operators derived from *Sequence*, including **fm**  $t$  and **fail**  $t$ . The intuition is that the innermost sampling event is propagated, whilst outer sampling events ‘stick’ to the derived operator as in  $\llbracket \{t_1; t_2\} \rrbracket_q$  above.

$\llbracket \varepsilon \rrbracket_q$	=	$\varepsilon$
$\llbracket \text{any} \rrbracket_q$	=	$\text{any}@q$
$\llbracket a \rrbracket_q$	=	$a@q$
$\llbracket t_1 \vee t_2 \rrbracket_q$	=	$\llbracket t_1 \rrbracket_q \vee \llbracket t_2 \rrbracket_q$
$\llbracket \neg t \rrbracket_q$	=	$\neg \llbracket t \rrbracket_q$
$\llbracket \text{pre } t \rrbracket_q$	=	<b>pre</b> $\llbracket t \rrbracket_q$
$\llbracket \{t_1; t_2\} \rrbracket_q$	=	$\{\llbracket t_1 \rrbracket_q; \llbracket t_2 \rrbracket_q\} @q$
$\llbracket t @q' \rrbracket_q$	=	$\llbracket t \rrbracket_{q'} @q$

Table 5: Normalising  $t@q$

**Example 4.2** Further intuitions can be gained by computing a few examples. For distinct events  $a, b, x, y$  and  $z, q, q' \neq \text{any}$

- $\llbracket \{a; b\} @q \rrbracket_{any} = \{a @q; b @q\} @q$
- $\llbracket (\{a; b\} @q \vee x) @q' \rrbracket_{any} = (\{a @q; b @q\} @q @q') \vee (x @q')$
- $\llbracket (\{a; b\} @q \vee \{x; y @z\}) @q' \rrbracket_{any} = (\text{left as an exercise}).$

Note that  $\llbracket any \rrbracket_q = any @q$ , (first match eventually  $q$ ). ♣

We say that  $\hat{t} \stackrel{\text{def}}{=} \llbracket t \rrbracket_{any}$  is in *sampled normal form*. In each of the above examples that computed normal forms we can also verify that  $\llbracket \hat{t} \rrbracket_{any} = \hat{t}$ . In general one can prove

**Proposition 4.1**  $\llbracket \hat{t} \rrbracket_{any} = \hat{t}$  for all temporal expressions  $t$  (i.e.,  $\llbracket \cdot \rrbracket_{any}$  is idempotent).  $\square$

**Proposition 4.2** If  $t$  is a sampling-free temporal expression then  $\|t\| = \|\hat{t}\|$ .  $\square$

These results tell us that once an expression has been normalised it remains unchanged if it is renormalised, and that the semantics of  $t$  and  $\hat{t}$  agree if the expression is sampling free. Proposition 4.2 generalises easily to expressions  $t$  where sampling is applied (only) to atomic events or propositions that are subterms of  $t$ .

### 4.3 Sampling non-atomic expressions

The outstanding issue is to define the semantics of  $t @q$  for arbitrary  $t$ . Firstly, we require that  $t @q$  is in sampled normal form. Secondly, note that  $\|t\|$  partitions into the two sets of sequences  $\|t \wedge \blacklozenge q\|$  and  $\|t \wedge \blacklozenge \bar{q}\|$  (in other words, either the event  $q$  is present if, and when,  $t$  succeeds, or it is not present). With this intuition define

$$\|t @q\| \stackrel{\text{def}}{=} \|(t \wedge \blacklozenge q) \vee \{(t \wedge \blacklozenge \bar{q}); \blacklozenge q\}\| \quad (3)$$

The  $\blacklozenge q$  in the first disjunct ensures that if the sampling event coincides with the success of  $t$  then the expression  $t @q$  succeeds immediately; otherwise (because of the  $\blacklozenge \bar{q}$  in the second disjunct) the expression succeeds at the next  $q$ .

**Example 4.3** For the events  $a$ ,  $b$ , and  $c$ , (3) yields

$$\|a @b @c\| = \|(a @b \wedge \blacklozenge c)\| \cup \|(a @b \wedge \blacklozenge \bar{c}); \blacklozenge c\|$$

The left disjunct is the set of sequences matching  $\langle \bar{a} \bar{b} \rangle^* (\langle abc \rangle | \langle \bar{a} \bar{b} \rangle \langle \bar{b} \rangle^* \langle bc \rangle)$ ; the other is left as an easy exercise. For another example consider, for  $e \in \mathcal{E}$  and  $p \in \mathcal{P}$ :

$$\|(e \wedge p) @q\| = \|e @q\| \cap \|p @q\|$$

Via (1) and (2) this gives all sequences matching  $\langle \bar{e} \bar{q} \rangle^* (\langle epq \rangle | \langle e \bar{q} \rangle \langle \bar{q} \rangle^* \langle pq \rangle)$ . ♣

Finally let us consider a sampled sampled expression like  $t@q@q'$  for  $q \neq q'$  and ‘sequential’ term  $t$ . For the inner term, given the above definition, clearly  $(t \wedge \blacklozenge q) \vee \{(t \wedge \blacklozenge \bar{q}); \blacklozenge q\}$  is in sampled normal form (since  $t@q$  and hence  $t$  are in normal form). However, the expression

$$((t \wedge \blacklozenge q) \vee \{(t \wedge \blacklozenge \bar{q}); \blacklozenge q\}) @q'$$

is *not* in sampled normal form. Consequently the set of sequences defined by the above term differs, in general, from the set  $\|(t@q)@q'\|$ . A similar caveat applies here in the case that  $t$  is an atom for

$$\|p@q@q'\| \neq \|(\blacklozenge p \wedge \blacklozenge q)@q'\| = \|([\blacklozenge p \wedge \blacklozenge q]_{q'})@q'\|$$

and similarly if the atom is an event  $e$ .

## 5 Summary

The full collection of temporal constructs defined above includes the primitives of the language, viz

$$t ::= \varepsilon \mid \text{any} \mid a \mid \bigvee_{i \in I} t_i \mid \{t_1; t_2\} \mid \neg t \mid \mathbf{pre} t \mid t@q$$

and large collection of derived operators. Those defined via *Sequence* and *Disjunction*:

$$t ::= [n]t \mid [..m]t \mid [..]t \mid \blacklozenge t$$

and those defined through the *Negation* and *Prefix*:

$$t ::= t_1 \wedge t_2 \mid \mathbf{fm} t \mid t_1 \mathcal{U}_m t_2 \mid t_1 \mathcal{U} t_2 \mid \blacklozenge t \mid \mathbf{fail} t \mid \bar{a}$$

Of these operators neither  $\mathbf{fm} t$ ,  $\neg t$ , nor  $\mathbf{pre} t$  are (directly) accessible from the concrete syntax of temporal  $e$ , while disjunction is of course finitely presented in  $t_1 \vee t_2$  (infinite disjunction being allowed in the restricted setting of *True Match*).

Table 6 provides a summary of these operators, their definitions, and the concrete syntax as defined by Specman. This list is not complete. Firstly, the *True Match* and *First Match* variable repeat operators may have a lower bound other than 0: in either case the expansion  $[n..n+m]t = \{[n]t; [..m]t\}$  applies. Secondly, the ‘\*  $t$ ’ may be omitted from each of the repeat operators—if the expression is omitted ‘\* **cycle**’ is assumed. For the remaining omissions see below.

### 5.1 Transition predicates or unnamed events

In temporal  $e$  there are three transition predicates. If  $s$  is a Boolean or arithmetic expression (without loss of generality, a variable or *signal*) then  $\mathbf{rise}(s)$ ,  $\mathbf{fall}(s)$ , and

Concrete Syntax	Logical Syntax	Name	Definition
<b>cycle</b>	<i>any</i>	<i>Cycle</i>	Table 6
<b>@any</b>	<i>any</i>	<i>Cycle</i>	Table 3
<b>@e</b>	<i>e</i>	<i>Event</i>	Table 3
<b>true(<i>p</i>)</b>	<i>p</i>	<i>Prop</i>	Table 3
<b>@e @<i>q</i></b>	<i>e @q</i>	<i>Sample 1</i>	Section 4.1(1)
<b>true(<i>p</i>) @<i>q</i></b>	<i>p @q</i>	<i>Sample 2</i>	Section 4.1(2)
<b><i>t</i> @<i>q</i></b>	<i>t @q</i>	<i>Sample 3</i>	Section 4.3(3)
<b>[<i>n</i>] * <i>t</i></b>	[ <i>n</i> ] <i>t</i>	<i>Repeat</i>	Section 3.2.3
<b>fail <i>t</i></b>	<b>fail <i>t</i></b>	<i>Failure</i>	Section 3.3.3
<b>~[..<i>m</i>] * <i>t</i></b>	[.. <i>m</i> ] <i>t</i>	<i>True Match 1</i>	Section 3.2.4
<b>~[..<i>n</i>] * <i>t</i></b>	[.. <i>n</i> ] <i>t</i>	<i>True Match 2</i>	Section 3.2.4
<b><i>t</i><sub>1</sub> ⇒ <i>t</i><sub>2</sub></b>	( <b>fail <i>t</i><sub>1</sub></b> ) ∨ { <i>t</i> <sub>1</sub> ; <i>t</i> <sub>2</sub> }	<i>Yield</i>	Table 6
<b><i>t</i><sub>1</sub> <b>or</b> <i>t</i><sub>2</sub></b>	<i>t</i> <sub>1</sub> ∨ <i>t</i> <sub>2</sub>	<i>Disjunction</i>	Table 3
<b><i>t</i><sub>1</sub> <b>and</b> <i>t</i><sub>2</sub></b>	<i>t</i> <sub>1</sub> ∧ <i>t</i> <sub>2</sub>	<i>Conjunction</i>	Table 3
<b>{<i>t</i><sub>1</sub> ; <i>t</i><sub>2</sub>}</b>	{ <i>t</i> <sub>1</sub> ; <i>t</i> <sub>2</sub> }	<i>Sequence</i>	Table 3
<b>{[..<i>m</i>] * <i>t</i><sub>1</sub> ; <i>t</i><sub>2</sub>}</b>	<i>t</i> <sub>1</sub> $\mathcal{U}_m$ <i>t</i> <sub>2</sub>	<i>First Match 1</i>	Section 3.3.2
<b>{[..<i>n</i>] * <i>t</i><sub>1</sub> ; <i>t</i><sub>2</sub>}</b>	<i>t</i> <sub>1</sub> $\mathcal{U}$ <i>t</i> <sub>2</sub>	<i>First Match 2</i>	Section 3.3.2

Table 6: Summary of temporal  $e$  operators

**change**( $s$ ) are temporal expressions. The intended meaning is that, say, **rise**( $s$ ) @ $q$  is true if  $q$  holds (at the current point) and either the value of  $s$  at the current point is greater than the value of  $s$  at the previous point where  $q$  holds, or else if  $q$  holds at no previous points then the value of  $s$  is greater than its initial value.

At first sight these transition predicates pose a problem if they are to fit into the semantic framework established here. In order to detect the rise (fall or change) in the signal's value one needs a sequence of at least two states. The resolution of this dilemma lies in augmenting the atoms.

First, let us define a function **prev**( $s, q$ ), that computes the value of  $s$  at the previous point for which  $q$  holds. If no such point exists, it gives the initial value of  $s$ .

We can then define the following new propositions:

$$\begin{aligned}
 \mathbf{RISE}(s, q) &\stackrel{\text{def}}{=} \mathbf{prev}(s, q) < s \\
 \mathbf{FALL}(s, q) &\stackrel{\text{def}}{=} \mathbf{prev}(s, q) > s \\
 \mathbf{CHANGE}(s, q) &\stackrel{\text{def}}{=} \mathbf{prev}(s, q) \neq s
 \end{aligned}$$

Let

$$\mathcal{X} \stackrel{\text{def}}{=} \{\mathbf{RISE}(s, q), \mathbf{FALL}(s, q), \mathbf{CHANGE}(s, q) : s \text{ is a signal}, q \in \mathcal{E}\}$$

and extend  $\mathcal{A}$  to  $\mathcal{P} \cup \mathcal{E} \cup \mathcal{X}$ . So, we see that  $\mathbf{RISE}(s, q)$  is an atom no different from the atoms in  $\mathcal{P}$ .

Finally, the connection between  $\mathbf{rise}(s) @q$  of the concrete syntax and  $\mathbf{RISE}(s, q)$  is as follows:

$$\begin{aligned} \llbracket \mathbf{rise}(s) \rrbracket_q &= \mathbf{RISE}(s, q) @q \\ \llbracket \mathbf{fall}(s) \rrbracket_q &= \mathbf{FALL}(s, q) @q \\ \llbracket \mathbf{change}(s) \rrbracket_q &= \mathbf{CHANGE}(s, q) @q \end{aligned}$$

## 5.2 Discussion

Temporal  $e$  is thus a language of sequences of events  $e \in \mathcal{E}$ , and program state ( $p \in \mathcal{P}$ ), sampled by events (or clocks). Sequences may be composed in various ways via disjunction, conjunction, and repetition. The use of *negation* is restricted in the (concrete) language to the special case of **fail**  $t$  which has a clear operational interpretation if a rather cumbersome denotational one as it is given in Section 3.3.3: **fail**  $t$  specifies exactly the shortest sequences that cannot be extended to sequences satisfying  $t$ .

Disjunction is the source of nondeterminism in the language (conjunction, in contrast, is strongly deterministic). The other powerful operator of temporal  $e$  is the generalised *until* which we call ‘first match repeat’ (Section 3.3.2). Through this operator we are able in  $e$  to express bounded and unbounded eventualities in a natural way.

### 5.2.1 Temporal contexts revisited

Section 2.1 discussed the contexts in which temporal expression are used in  $e$  verification programs. It is useful to consider how these relate to temporal  $e$  itself:

**event**  $e$  **is**  $t$ —the event is ‘emitted’ whenever the (evaluation of the) temporal expression succeeds. This evaluation process commences anew each cycle. This is approximated by true match eventually:  $\blacklozenge t$  (approximated since  $\blacklozenge t$  necessarily ends in a suffix satisfying  $t$ , but the event definition makes no such commitment).

**expect**  $t$ —the informal explanation was that the temporal expression represents an invariant. In fact the evaluation of the temporal expression is evaluated anew on each cycle, and what we are interested in is the *failure* of  $t$ . This makes the ‘expect’ an application of first match eventually:  $\blacklozenge(\mathbf{fail} t)$ .

In a similar vein **wait**  $t$  is a simple application of first match eventually. The presence of sampling events, however, do complicate the above explanations slightly.

### 5.2.2 Sampling @ sampling

Much of the complexity in the discussion around  $\|t@q\|$  arises because of the possibility of nested clocking events. One might well ask whether such complexity is necessary in the property language for  $a@q$  is already a powerful notion. In fact the ability to nest clocks is of considerable utility since this permits one to sample a subsequence of a larger sequence, at a (notionally) faster *or* slower rate. Thus evaluation of  $\{\dots; \{a;b\}@q_1; \dots\}@q_2$  ‘enters’ the inner sequence when the preceding expression succeeded on  $q_2$ ; thereafter the inner  $\{a;b\}$  is clocked by  $q_1$ , the outer clock  $q_2$  being ignored unless and until  $\{a;b\}@q_1$  succeeds. The insistence that (sampled) expressions are in normal form guarantees this behaviour.

### 5.2.3 Power of the negation-free language

Negation-free temporal  $e$  is an interesting and useful property language in itself, and indeed the syntax in Section 3.2.2 can be augmented with the *Sample* operator so as to exploit nesting of sampling events as discussed above. Transition predicates are also within the scope of this subset of temporal  $e$ , as is the first match eventually  $\diamond a$  as discussed in Section 3.2.5.

A limitation of the negation-free language is that it does not have the expressive power to capture the until operator *First Match Repeat*. This operator is found to be very much used in practice in the bounded and unbounded versions. Another weakness, from a specification standpoint, would be the omission of the yield operator (defined via **fail** in Table 6). While many applications of this operator are of the form *expect*  $(a \Rightarrow t)@q$ , and while ‘**fail**  $a@q$ ’ may be defined in the negation-free language, nested applications of  $t_1 \Rightarrow t_2$  under some sampling event are also very useful.

### 5.2.4 Strong eventually

In a simulation context first match eventually must be regarded as a *weak* eventuality. This is because simulation runs are necessarily finite sequences, but evaluation of  $\diamond t$  would not necessarily fail if time ran out before the expression  $t$  had been satisfied. In temporal  $e$  therefore a stronger ‘eventually’ is defined by appealing to a special event, *quit*, which marks the end of time. Then (loosely as this is not correct under sampling)

$$\mathbf{eventually} \ t \stackrel{\text{def}}{=} \overline{\text{quit}} \ \mathcal{U} \ t$$

is a temporal expression that will fail unless  $t$  is satisfied before the simulation ends.

---

## References

- [1] J. Halpern, Z. Manna, and B. Moszkowski. A hardware semantics based on temporal intervals. In *ICALP'83*, volume 143 of *LNCS*, pages 278–291. Springer Verlag, New York, 1983.
- [2] H. Hiraishi, K. Hamaguchi, H. Fujii, and S. Yajima. Regular temporal logic expressively equivalent to finite automata and its application to logic design verification. *Journal of Information Processing*, 15(1):129–138, 1992.
- [3] B. Moszkowski. A temporal logic for multilevel reasoning about hardware. *IEEE Computer*, 18(2):10–19, 1985.
- [4] R. Rosner and A. Pnueli. A choppy logic. In *LICS'86*, pages 306–313, 1986.

# Contents

<b>1</b>	<b>Introduction</b>	<b>1</b>
<b>2</b>	<b>Introducing temporal <i>e</i></b>	<b>2</b>
2.1	Temporal contexts . . . . .	2
2.1.1	Event definitions . . . . .	2
2.1.2	Expect declarations . . . . .	3
2.1.3	Wait actions . . . . .	3
2.2	Abstract syntax . . . . .	3
<b>3</b>	<b>Denotational semantics</b>	<b>5</b>
3.1	Preliminaries . . . . .	5
3.1.1	Domain of discourse . . . . .	5
3.1.2	Interpretations . . . . .	5
3.1.3	Models . . . . .	6
3.2	The negation-free language . . . . .	6
3.2.1	Abstract syntax . . . . .	6
3.2.2	Semantics . . . . .	6
3.2.3	Fixed repeat . . . . .	7
3.2.4	True match variable repeat . . . . .	7
3.2.5	Eventually . . . . .	8
3.3	First Match, and Failure . . . . .	8
3.3.1	Firstmatch . . . . .	9
3.3.2	First match variable repeat . . . . .	9
3.3.3	Fail . . . . .	10
<b>4</b>	<b>Sampling</b>	<b>11</b>
4.1	Atomic cases . . . . .	11
4.2	Sampled normal form . . . . .	13
4.3	Sampling non-atomic expressions . . . . .	14
<b>5</b>	<b>Summary</b>	<b>15</b>
5.1	Transition predicates or unnamed events . . . . .	15
5.2	Discussion . . . . .	17
5.2.1	Temporal contexts revisited . . . . .	17
5.2.2	Sampling @ sampling . . . . .	18
5.2.3	Power of the negation-free language . . . . .	18
5.2.4	Strong eventually . . . . .	18
<b>A</b>	<b>Formalities in brief</b>	<b>21</b>

## A Formalities in brief

1.  $\mathcal{P}$  is a set of *propositional variables*, and let  $p$  be a metavariable ranging over  $\mathcal{P}$ .
2.  $\mathcal{E}$  is a set of *events*, and let  $e$  be a metavariable over  $\mathcal{E}$ .
3. Let  $\mathcal{A} = \mathcal{P} + \mathcal{E}$  be the set of *atomic propositions and events*, and let  $a \in \mathcal{A}$ .
4. Let  $\text{TEXP}$  be the collection of all *temporal expressions* and  $t \in \text{TEXP}$  be generated by the abstract syntax

$$t ::= \varepsilon \mid \text{any} \mid a \mid \bigvee_{i \in I} t_i \mid \{t_1; t_2\} \mid t @ q \mid \neg t \mid \mathbf{pre} t$$

for  $a \in \mathcal{A}$ ,  $q \in \mathcal{E}$ , and where  $I$  is some index set.

5. A *state* is an element  $s \in 2^{\mathcal{A}}$  (equivalently it is a valuation  $\mathcal{A} \rightarrow \{0, 1\}$ ).
6. Let  $\sigma$  be a *finite sequence*  $\langle s_1, s_2, \dots, s_n \rangle$  for  $|\sigma| = n \geq 1$ . The empty sequence is  $\varepsilon$ , and  $\sigma_1 \sigma_2$  is the *concatenation* of sequences  $\sigma_1$  and  $\sigma_2$ :  $|\sigma_1 \sigma_2| = |\sigma_1| + |\sigma_2|$ .
7. Let  $\Sigma^*$  be the (infinite) set of *all finite sequences* over  $2^{\mathcal{A}}$ .
8. The *semantics* of  $t \in \text{TEXP}$  is the set  $\|t\| \subseteq \Sigma^*$  inductively defined in Table 7.

<i>Empty</i>	$\ \varepsilon\ $	$\stackrel{\text{def}}{=} \{\}$
<i>Cycle</i>	$\ \text{any}\ $	$\stackrel{\text{def}}{=} \{\sigma :  \sigma  = 1\}$
<i>Atom</i>	$\ a\ $	$\stackrel{\text{def}}{=} \{\sigma : \sigma = \langle s \rangle, a \in s\}$
<i>Disjunction</i>	$\ \bigvee_{i \in I} t_i\ $	$\stackrel{\text{def}}{=} \bigcup_{i \in I} \ t_i\ $
<i>Sequence</i>	$\ \{t_1; t_2\}\ $	$\stackrel{\text{def}}{=} \{\sigma : \exists \sigma_1 \sigma_2 \in \Sigma^*. \sigma = \sigma_1 \sigma_2, \sigma_1 \in \ t_1\ , \sigma_2 \in \ t_2\ \}$
<i>Sample</i>	$\ t @ q\ $	$\stackrel{\text{def}}{=} \boxed{\text{See Table 8}}$
<i>Negation</i>	$\ \neg t\ $	$\stackrel{\text{def}}{=} \Sigma^* - \ t\ $
<i>Prefix</i>	$\ \mathbf{pre} t\ $	$\stackrel{\text{def}}{=} \{\sigma : \exists \sigma' \in \Sigma^*. \sigma \sigma' \in \ t\ \}$

Table 7: Semantics of the basic temporal operators

9. Let  $q \in \mathcal{E} \cup \{\text{any}\}$  be a *sampling event*. There are several *sampling rules* (see Table 8) where the term  $t @ q$  must be in *sampled normal form* (Table 9).
10. There are a number of *derived operators* (see Table 10 and 11). The  $t$  may be omitted from the repeat operators, defaulting to ‘any’ – ie.,  $[n] \stackrel{\text{def}}{=} [n]\text{any}$ , etc..
11. The set  $\mathcal{P}$  is partitioned into two sets: *propositions*, and *transition predicates*. If  $x$  is a *variable* of some ordered type and  $e$  is an event, then  $x_\Delta \in \mathcal{X}$  is a transition predicate, where  $\mathcal{X} \stackrel{\text{def}}{=} \{\mathbf{RISE}(x, e), \mathbf{FALL}(x, e), \mathbf{CHANGE}(x, e) : x \text{ is a variable, } e \in \mathcal{E}\}$ .

$\ p@q\ $	$\stackrel{\text{def}}{=} \ \diamond q \wedge \blacklozenge p\ $	for $p \in \mathcal{P}$
$\ e@q\ $	$\stackrel{\text{def}}{=} \ (\bar{e} \wedge \bar{q}) \mathcal{U} ((e \wedge q) \vee \{(e \wedge \bar{q}); \diamond q\})\ $	for $e \in \mathcal{E}$
$\ t@q\ $	$\stackrel{\text{def}}{=} \ (t \wedge \blacklozenge q) \vee \{(t \wedge \blacklozenge \bar{q}); \diamond q\}\ $	otherwise

Table 8: The three semantics of the *Sampling* operator

$\llbracket \varepsilon \rrbracket_q$	$= \varepsilon$	$\llbracket \text{any} \rrbracket_q$	$= \text{any}@q$
$\llbracket a \rrbracket_q$	$= a@q$	$\llbracket t_1 \vee t_2 \rrbracket_q$	$= \llbracket t_1 \rrbracket_q \vee \llbracket t_2 \rrbracket_q$
$\llbracket \neg t \rrbracket_q$	$= \neg \llbracket t \rrbracket_q$	$\llbracket \text{pre } t \rrbracket_q$	$= \text{pre } \llbracket t \rrbracket_q$
$\llbracket \{t_1; t_2\} \rrbracket_q$	$= \{\llbracket t_1 \rrbracket_q; \llbracket t_2 \rrbracket_q\} @q$	$\llbracket t @q' \rrbracket_q$	$= \llbracket t \rrbracket_{q'} @q$

Table 9: Computing the sampled normal form  $t @q$  from  $\llbracket t @q \rrbracket_q$ 

$\forall t \forall n \geq 1$	$\llbracket n \rrbracket t \stackrel{\text{def}}{=} \{t; [n-1]t\}$	$\llbracket 0 \rrbracket t \stackrel{\text{def}}{=} \varepsilon$
$\forall n \geq m \geq 0$	$\llbracket m..n \rrbracket t \stackrel{\text{def}}{=} \bigvee_{k=m}^n \llbracket k \rrbracket t$	$t_1 \mathcal{U}_n^m t_2 \stackrel{\text{def}}{=} \mathbf{fm} \{[m..n]t_1; t_2\}$
$\forall m \geq 0$	$\llbracket m.. \rrbracket t \stackrel{\text{def}}{=} \bigvee_{k \geq m} \llbracket k \rrbracket t$	$t_1 \mathcal{U}^m t_2 \stackrel{\text{def}}{=} \mathbf{fm} \{[m..]t_1; t_2\}$
$\forall n \geq 0$	$\llbracket ..n \rrbracket t \stackrel{\text{def}}{=} \llbracket 0..n \rrbracket t$	$t_1 \mathcal{U}_n t_2 \stackrel{\text{def}}{=} \mathbf{fm} \{[..n]t_1; t_2\}$
	$\llbracket .. \rrbracket t \stackrel{\text{def}}{=} \llbracket 0.. \rrbracket t$	$t_1 \mathcal{U} t_2 \stackrel{\text{def}}{=} \mathbf{fm} \{[..]t_1; t_2\}$

Table 10: Derived *Fixed Repeat* and *True* and *First Match Repeat* operators

<i>First Match</i>	$\mathbf{fm } t \stackrel{\text{def}}{=} t \wedge \neg \{t; [1..]\}$
<i>Fail</i>	$\mathbf{fail } t \stackrel{\text{def}}{=} (\mathbf{fm } \neg \text{pre } t) \wedge \neg \{t; [1..]\}$
<i>First Match Eventually</i>	$\diamond t \stackrel{\text{def}}{=} \mathbf{fm} \{[..]; t\}$
<i>True Match Eventually</i>	$\blacklozenge t \stackrel{\text{def}}{=} \{[..]; t\}$
<i>Atomic Complement</i>	$\bar{a} \stackrel{\text{def}}{=} \mathbf{fail } a$
<i>Conjunction</i>	$t_1 \wedge t_2 \stackrel{\text{def}}{=} \neg(\neg t_1 \vee \neg t_2)$
<i>Yield</i>	$t_1 \Rightarrow t_2 \stackrel{\text{def}}{=} (\mathbf{fail } t_1) \vee \{t_1; t_2\}$

Table 11: Other derived operators: *First Match*, *Fail*, *Eventually*, etc.