# **Appendix C: FSP Semantics**

The semantics of basic *FSP* are defined in terms of Labelled Transition Systems (*LTSs*). In the body of the book, we have depicted the *LTS* that corresponds to an *FSP* process as a graph. In the following, we formally define what an *LTS* is and then describe the correspondence between *FSP* process expressions and *LTS*s. This correspondence is defined by the function:

where  is the set of *FSP* process expressions, and  the set of *LTS*s. The function  is defined inductively on the structure of *FSP* process expressions.

## C.1 Labeled Transition System (LTS)

Let be the universal set of states including  a designated *error* state,  be the universal set of labels, and , where  is used to denote an internal action that cannot be observed by the environment of an *LTS*.

A finite *LTS*  is a quadruple where:

* where  denotes the alphabet of .
* , denotes a transition relation that maps from a state and an action onto another state.
* indicates the initial state of .

The only *LTS* that is allowed to have the error state  as its initial state is *,* named . The alphabet of this process .

An LTS transitswith action  into an LTS , denoted as , if:

* where and , or
* and

We use  to mean that  such that , and  to mean that  such that .

We define a set of designated end states  such that an LTS is terminating if there is a state  and  and  for all .

## C.2 Processes

In the following,  ranges over FSP process expressions,  ranges over process identifiers, and  range over sets of observable actions (i.e and ).

### Process Definition

means that .

### Process Constants

where

### Prefix

If and  is not

Then , where .

, where

### Choice |

Let and

Then where .

If is ERROR then .

### Alphabet Extension

If

Then .

### Recursion

We represent the *FSP* process defined by the recursive equation  as where  is a variable in . For example, the process defined by the recursive definition is represented as

We use to denote the FSP expression that is obtained by substituting for  in . Then is the smallest *LTS* that satisfies the following rule:

where is strong semantic equivalence defined in section C.6.1. Mutually recursive equations can be reduced to the simple form described above. For local processes, all occurrences of process variables are guarded by an action prefix and consequently, recursive definitions are guaranteed to have a fixed-point solution.

### Sequential Composition

If is terminating with end state and

Then where

# C.3 Composite Processes

Before defining the meaning of composition in FSP and of the priority operators on composite processes, we must first describe the meaning of composition and priority in the underlying LTS model.

### C.3.1 LTS Composition

The parallel composition  of two LTSs and is defined as follows:

If or , then .

For and , such that and ,

Where is the smallest relation satisfying the rules:

Let in

Parallel composition is both commutative and associative. Consequently, the order in which LTSs are composed is not significant.

In addition:

, , and .

### C.3.2 LTS Priority

The set of actions are high priority in the LTS where .

where is the smallest relation satisfying the rule:

Let in

The set of actions are low priority in the LTS where .

where is the smallest relation satisfying the rule:

Let in

### C.3.3 FSP Composition and Priority

Using the definitions for LTScomposition and priority, we can now simply define the meaning of composition and priority in FSP. In the following,  refers to FSP composition expressions of the form and  refers to the identifier of a process or composite process.

Parallel Composition

Priority High

Priority Low

## C.4 Common Operators

To define the *FSP* re-labeling, hiding and interface operators, which can be applied to processes and composite processes, we first describe the meaning of re-labeling and hiding in the underlying *LTS* model.

### C.4.1 Re-Labeling

Re-labeling applies a relation over action labels , to an LTS such that:

Where

### C.4.2 Hiding

The set of actions  are hidden in the *LTS* , where *.*

If  then

Otherwise

where Δ is the smallest relation satisfying the rule:

Let a ∈ Act in

### C.4.3 FSP Re-Labeling, Hiding and Interface

Re-labeling

Hiding

Interface

where

## C.5 Safety Properties

A safety property *Q* in *FSP* is represented by an image of the *LTS* of the process expression that defines the property. The image *LTS* has each state of the original *LTS* and has a transition from each state for every action in the alphabet of the original. Transitions added to the image *LTS* are to the error state.

property *Q* = *E*:

for an LTS ,

where

## C.6 Semantic Equivalences

Minimization of the *LTS* corresponding to an *FSP* process definition results in a semantically equivalent *LTS*. The equivalence relations used in performing minimization are defined in the following sections.

### C.6.1 Strong Equivalence

Strong semantic equivalence equates *LTS*s that have identical behavior when the occurrence of all their actions can be observed, including that of the silent action .

Let  be the universal set of LTSs. Strong semantic equivalence  is the union of all relations satisfying that implies:

* + implies  and .
  + implies  and .

1. iff

The *LTSA* tool performs minimization using strong equivalence if an *LTS* contains no silent actions (). For an *LTS P*, without -actions:

### C.6.2 Weak Equivalence

Weak semantic equivalence equates systems that exhibit the same behavior to an external observer who cannot detect the occurrence of -actions.

Let  denote , where  means a sequence of zero or more s. Then weak *(or* observational*)* semantic equivalence  is the union of all relations  satisfying that  implies:

1. where and is the empty sequence
   * implies  and .
   * implies  and .
2. iff

The *LTSA* tool performs minimization using weak equivalence if an *LTS* contains silent actions (). For an LTS P, with -actions:

Both strong and weak equivalence are congruences with respect to the composition, re-labeling, and hiding operators. This means that strongly or weakly equivalent components may substitute one another in any system constructed with these operators, without affecting the behavior of the system with respect to strong or weak equivalence, respectively.

## C.7 Fluent Linear Temporal Logic (FLTL)

### C.7.1 Linear Temporal Logic

Given a set of atomic propositions , a well-formed LTL formula is defined inductively using the standard Boolean operators  (not),  (or),  (and), and the temporal operators  (next) and  (strong until) as follows:

* each member of  is a formula,
* if  and  are formulae, then so are , , , ,.

An interpretation for an LTL formula is an infinite word  over . In other words, an interpretation maps to each instant of time a set of propositions that hold at that instant. We write  for the suffix of  starting at . LTL semantics is then defined inductively as follows:

* iff , for
* iff not
* iff or
* iff and
* iff
* iff , such that:  and

We introduce the abbreviations and . Implication  and equivalence  are defined as:

Temporal operators  (eventually),  (always), and  (weak until) are defined in terms of the main temporal operators:

### C.7.2 Fluents

We define a fluent  by a pair of sets, a set of initiating actions  and a set of terminating actions

In addition, a fluent  may initially be true or false at time zero as denoted by the attribute .

The set of atomic propositions from which FLTL formulae are built is the set of fluents . Therefore, an interpretation in FLTL is an infinite word over , which assigns to each time instant the set of fluents that hold at that time instant. An infinite word over  (i.e. an infinite trace of actions) also defines an FLTL interpretation   over  as follows:

* iff either of the following holds

In other words, a fluent holds at a time instant if and only if it holds initially or some initiating action has occurred, and, in both cases, no terminating action has yet occurred. Note that the interval over which a fluent holds is closed on the left and open on the right, since actions have immediate effect on the values of fluents. Since the sets of initiating and terminating actions are disjoint, the value of a fluent is always deterministic with respect to a system execution.

Every action  implicitly defines a fluent whose initial set of actions is the singleton and whose terminating set contains all other actions in the alphabet of a process :

becomes true the instant *a* occurs and becomes false with the first occurrence of a different action. It is often more succinct in defining properties to declare a fluent implicitly for a set of actions as in:

where

This is equivalent to  where  represents the implicitly defined