V2.0 -Additions

FSP Language Specification

This document describes the additions that have been made to the *FSP* input notation to the LTSA tool since its initial release. It should be read in conjunction with Appendix B of *Concurrency: State Models and Java Programs*.

The additions are described in the following sections:

1) Sequential Processes

2) Graphical Animation

3) Miscellaneous extensions

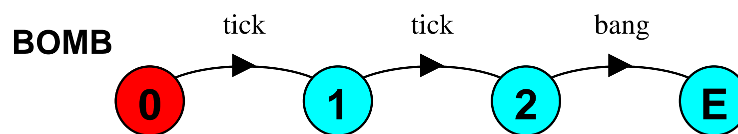
1 Sequential Processes

FSP divides processes into three types: local processes that define a state within a primitive process, primitive processes defined by a set of local processes and composite processes that use parallel composition, relabeling and hiding to compose primitive processes. A local process is defined using **STOP**, **ERROR**, action prefix and choice.

> A sequential process is a process that can terminate. A process can terminate if the local process **END** is reachable from its start state.

1.1 Local Process END

The local process **END** denotes the state in which a process successfully terminates. A process engages in no further actions after **END**. In this respect it has the same semantics as **STOP**. However, **STOP** denotes a state in which a process has halted prematurely, usually due to communication deadlock. In the book, we sometimes used **STOP** to indicate the successful termination of a process. These uses of **STOP** should now be replaced by **END**. With the introduction of a state describing successful termination, the need to use **STOP** explicitly in process description largely disappears. Figure 1 depicts an example of a sequential process together with its LTS:




```
BOMB = (tick -> tick -> bang -> END).
```

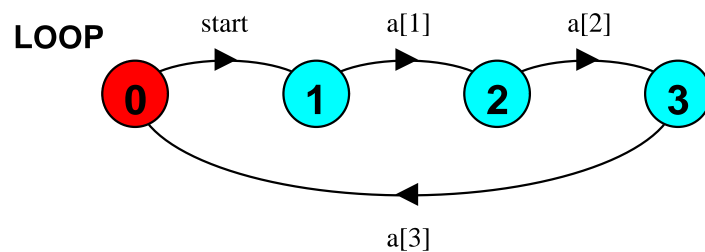
Figure 1 – Sequential Process BOMB

1.2 Sequential Composition ;

A sequential composition in FSP always takes the form:

```
SP1;SP2;..SPn;LP
```

where SP1,..,SPn are sequential processes and LP is a local process. A sequential composition can appear anywhere in the definition of a primitive process that a local process reference can appear e.g.




```
SP(I=0) = (a[I]->END).

P123 = (start -> SP(1);SP(2);SP(3);END).

LOOP = P123;LOOP.
```

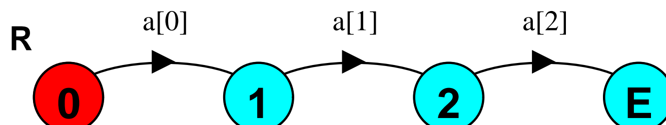
Figure 2 – Sequential Composition LOOP

A sequential composition `P;Q` means that when P terminates, `P;Q` becomes the process `Q`.

If we define a process `SKIP = END` then:

```
P;SKIP ≡ P.
SKIP;P ≡ P.
```

Sequential composition can be used in a recursive context as in:



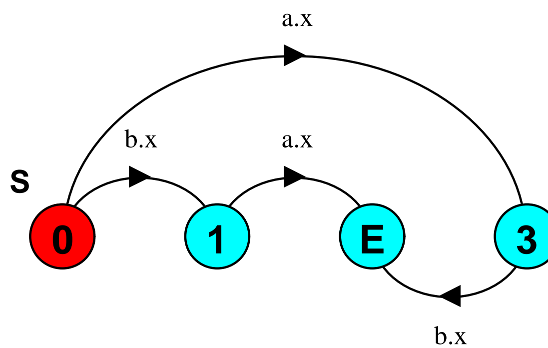
```
const N = 3
R(I =0) = if (I<N) then
              (a[I]->R(I+1);END)
          else
              END.
```

Figure 3 – Sequential Composition and Recursion

1.3 Parallel Composition and Sequential Processes

> The parallel composition `SP1||SP2` of two sequential processes `SP1` and `SP2` terminates when both of these processes terminate. If termination is reachable in `SP1||SP2` then it is a sequential process.

Note that a composite process that terminates can appear in the definition of a primitive process.

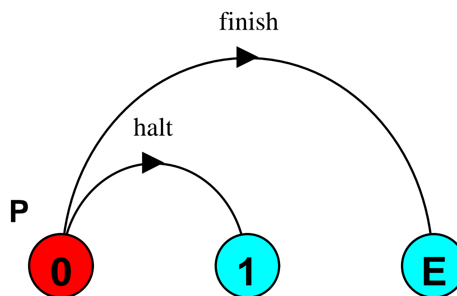



```
P = (x -> END).
||S = (a:P || b:P).
```

Figure 4 – Parallel Composition of Sequential Processes

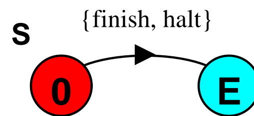
1.4 Sequential Processes and minimization

Minimization does not distinguish between **STOP** and **END**, consequently the process:




```
P =(halt->STOP | finish->END).
```

minimizes to:




```
P =({finish,halt} -> END).
```

1.5 Sequential Processes and Analysis

While a reachable **STOP** state is a safety violation resulting in the LTSA generating a counter example trace, a reachable **END** state is not a safety violation. However, a trace to a reachable **END** state will be generated during progress analysis since the **END** state violates all progress properties.

2 Graphical Animation

LTSA V2.0 supports graphical animation using SceneBeans. Here we describe only the extensions to FSP required to map a model to a graphical animations.

This mapping is defined by the **animation** construct that specifies the XML file that contains the description of the animation and two relations that describe the mapping of model actions to animation commands - **actions** and model actions to animation controls – **controls**. The following example describes the mapping for a channel animation:

```
CHAN = (in ->out->CHAN |in->fail->CHAN ).

animation FAILCHAN = "xml\channel.xml"
   actions { in  /channel.begin,
             fail/explode
           }
   controls { out /channel.end,
              fail/channel.fail,
              in  /send
            }
```

Figure 5 – Animation example.

The actions and controls relations are defined in exactly the same way as relabeling relations. The label on the left of a pair is the model label and the label to the right, the name of an animation command or control.

The animation construct may optionally specify the target composition to which it can be applied as in:

```
animation DINERS = "xml/diners.xml" target DINERS
 compose
  {PHILOS||FORKS
   /{forall [i:0..N-1] {
    {phil[i].left,phil[((i-1)+N)%N].right}/fork[i]}
    }
  }
```

Note that here, the mapping relations are formed by composing two other animations. See the draft paper "*Graphical Animation of Behaviour Models*" for further details.

3 Miscellaneous extensions

3.1 Build directives

The following new keywords can be used to prefix the definition of primitive and composite processes: **`deterministic, minimal, compose`**.

`deterministic`

If the process is non-deterministic, the standard NDFA – DFA automata transformation is applied.

`minimal`

Minimizes the primitive or composite process

`compose`

Forces composition when prefixing a composite process. Has no effect when prefixing a primitive process.

A property automaton can now be formed from a composite process simply by prefixing it with the keyword **`property`**. If the composite is non-deterministic, it is transformed into a deterministic automaton, before being converted to an image automaton.

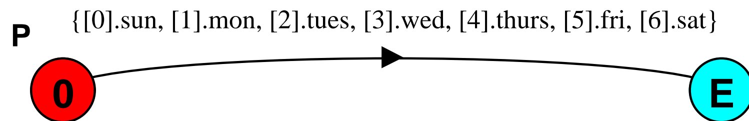
3.2 Set index operation @

The elements of a set can be accessed using an integer index expression.

`@(S,n)` – returns the nth label from the set `S`.

Example:

```
set Days = {sun, mon, tues, wed, thurs, fri, sat}
P = ([i:0..6][@(Days,i)]->END).
```



3.3 Set cardinality operation

The number of elements in a set can be found using:

`#S` – returns the number of elements in set S.