Finite State Process Algebra
and LTSA

Book

Concurrency: State Models & Java Programs,
2nd Edition

Jeff Magee &
Jeff Kramer

WILEY
**concurrent processes**

We structure complex systems as sets of simpler activities, each represented as a **sequential process**.

Processes can overlap or be concurrent, so as to reflect the concurrency inherent in the physical world.

Designing concurrent software can be complex and error prone. A rigorous engineering approach is essential.

**Concept of a process as a sequence of actions.**

**Model processes as finite state machines.**

**Program processes as threads in Java.**

---

**processes and threads**

**Concepts:** processes - units of sequential execution.

**Models:**
- **finite state processes (FSP)** to model processes as sequences of actions.
- **labelled transition systems (LTS)** to analyse, display and animate behavior.

**Practice:** Java threads
A process is the execution of a sequential program. It is modeled as a finite state machine which transits from state to state by executing a sequence of atomic actions.

Can finite state models produce infinite traces?
FSP - action prefix

If $x$ is an action and $P$ a process then $(x \rightarrow P)$ describes a process that initially engages in the action $x$ and then behaves exactly as described by $P$.

ONESHOT = (once $\rightarrow$ STOP).

ONESHOT state machine

(terminating process)

Convention: actions begin with lowercase letters
PROCESSES begin with uppercase letters

FSP - action prefix & recursion

Repetitive behaviour uses recursion:

$$\text{SWITCH} = \text{OFF}, \quad \text{OFF} = (\text{on} \rightarrow \text{ON}), \quad \text{ON} = (\text{off} \rightarrow \text{OFF}).$$

Substituting to get a more succinct definition:

$$\text{SWITCH} = \text{OFF}, \quad \text{OFF} = (\text{on} \rightarrow (\text{off} \rightarrow \text{OFF})).$$

And again:

$$\text{SWITCH} = (\text{on} \rightarrow \text{off} \rightarrow \text{SWITCH}).$$
The LTS animator can be used to produce a trace.

Ticked actions are eligible for selection.

In the LTS, the last action is highlighted in red.

FSP - action prefix

FSP model of a traffic light:

$$\text{TRAFFICLIGHT} = (\text{red} \rightarrow \text{orange} \rightarrow \text{green} \rightarrow \text{orange} \rightarrow \text{TRAFFICLIGHT}).$$

LTS generated using LTSA:

Trace:

$$\text{red} \rightarrow \text{orange} \rightarrow \text{green} \rightarrow \text{orange} \rightarrow \text{red} \rightarrow \text{orange} \rightarrow \text{green} \ldots$$
If $x$ and $y$ are actions then $(x \rightarrow P \mid y \rightarrow Q)$ describes a process which initially engages in either of the actions $x$ or $y$. After the first action has occurred, the subsequent behavior is described by $P$ if the first action was $x$ and $Q$ if the first action was $y$.

**Who or what makes the choice?**

**Is there a difference between input and output actions?**

---

FSP model of a drinks machine:

$$DRINKS = (red \rightarrow coffee \rightarrow DRINKS \mid blue \rightarrow tea \rightarrow DRINKS)$$

LTS generated using LTSA:

Possible traces?
Non-deterministic choice

Process \((x \rightarrow P \mid x \rightarrow Q)\) describes a process which engages in \(x\) and then behaves as either \(P\) or \(Q\).

\[
\text{COIN} = (\text{toss} \rightarrow \text{HEADS} | \text{toss} \rightarrow \text{TAILS}), \\
\text{HEADS} = (\text{heads} \rightarrow \text{COIN}), \\
\text{TAILS} = (\text{tails} \rightarrow \text{COIN}).
\]

**Tossing a coin.**

Possible traces?

---

Modeling failure

How do we model an unreliable communication channel which accepts \(\text{in}\) actions and if a failure occurs produces no output, otherwise performs an \(\text{out}\) action?

Use non-determinism...

\[
\text{CHAN} = (\text{in} \rightarrow \text{CHAN} | \text{in} \rightarrow \text{out} \rightarrow \text{CHAN}).
\]
**FSP - indexed processes and actions**

Single slot buffer that inputs a value in the range 0 to 3 and then outputs that value:

\[
BUFF = (in[i:0..3] \rightarrow out[i] \rightarrow BUFF).
\]

equivalent to

\[
BUFF = (in[0] \rightarrow out[0] \rightarrowBUFF \\
| in[1] \rightarrow out[1] \rightarrow BUFF \\
\].

or using a process parameter with default value:

\[
BUFF(N=3) = (in[i:0..N] \rightarrow out[i] \rightarrow BUFF).
\]

Local indexed process definitions are equivalent to process definitions for each index value.

**index expressions to model calculation:**

- const \( N = 1 \)
- range \( T = 0..N \)
- range \( R = 0..2*N \)

\[
SUM = (in[a:T][b:T] \rightarrow TOTAL[a+b]),
\]

\[
TOTAL[s:R] = (out[s] \rightarrow SUM).
\]
FSP - guarded actions

The choice \((\text{when } B \ x \rightarrow P | y \rightarrow Q)\) means that when the guard \(B\) is true then the actions \(x\) and \(y\) are both eligible to be chosen, otherwise if \(B\) is false then the action \(x\) cannot be chosen.

\[
\text{COUNT} \ (N=3) = \text{COUNT}[0], \\
\text{COUNT}[i:0..N] = (\text{when}(i<N) \ inc->\text{COUNT}[i+1] \\
| \text{when}(i>0) \ dec->\text{COUNT}[i-1]) .
\]

FSP - guarded actions

A countdown timer which beeps after \(N\) ticks, or can be stopped.

\[
\text{COUNTDOWN} \ (N=3) = (\text{start}->\text{COUNTDOWN}[N]), \\
\text{COUNTDOWN}[i:0..N] = (\text{when}(i>0) \ tick->\text{COUNTDOWN}[i-1] \\
| \text{when}(i==0) \ beep->\text{STOP} \\
| \text{stop}->\text{STOP}) .
\]
FSP - guarded actions

What is the following FSP process equivalent to?

\[
\begin{align*}
\text{const False} &= 0 \\
\text{P} &= \text{(when (False) doanything->P)}.
\end{align*}
\]

Answer:

STOP

FSP - process alphabets

The alphabet of a process is the set of actions in which it can engage.

Process alphabets are implicitly defined by the actions in the process definition.

The alphabet of a process can be displayed using the LTSA alphabet window.

Example:

Process: COUNTDOWN
Alphabet: { beep, start, stop, tick }
Revision & Wake-up Exercise

In FSP, model a process **FILTER**, that exhibits the following repetitive behavior:

inputs a value v between 0 and 5, but only outputs it if \( v \leq 2 \), otherwise it discards it.

\[
\text{FILTER} = (\text{in}[v:0..5] \rightarrow \text{DECIDE}[v]), \\
\text{DECIDE}[v:0..5] = (\ ? \ ).
\]

parallel composition - action interleaving

If P and Q are processes then \((P || Q)\) represents the concurrent execution of P and Q. The operator || is the parallel composition operator.

\[
\text{ITCH} = (\text{scratch} \rightarrow \text{STOP}). \\
\text{CONVERSE} = (\text{think} \rightarrow \text{talk} \rightarrow \text{STOP}). \\
\text{||CONVERSE_ITCH} = (\text{ITCH} || \text{CONVERSE}).
\]

Possible traces as a result of action interleaving:

- think \(\rightarrow\) talk \(\rightarrow\) scratch
- think \(\rightarrow\) scratch \(\rightarrow\) talk
- scratch \(\rightarrow\) think \(\rightarrow\) talk

Disjoint alphabets
**parallel composition - action interleaving**

ITCH

- States: 0, 1
- Transitions:
  - 0 → scratch (0,0)
  - 1 → scratch (0,1)

CONVERSE

- States: 0, 1, 2
- Transitions:
  - 0 → think (0,0)
  - 1 → talk (0,1)
  - 2 → scratch (0,2)
  - 1 → think (1,1)
  - 2 → talk (1,2)

CONVERSE_ITCH

- States: 0, 1, 2, 3, 4, 5
- Transitions:
  - 0 → think (0,0)
  - 1 → talk (0,1)
  - 2 → scratch (0,2)
  - 3 → think (1,1)
  - 4 → talk (1,2)
  - 5 → think (1,0)

From ITCH to CONVERSE

- Composition: 2 x 3 states

LTS? Traces? Number of states?

**parallel composition - algebraic laws**

**Commutative:** \((P || Q) = (Q || P)\)

**Associative:** \((P || (Q || R)) = ((P || Q) || R) = (P || Q || R)\).

Clock radio example:

- \(CLOCK = (\text{tick} \rightarrow \text{CLOCK})\).
- \(RADIO = (\text{on} \rightarrow \text{off} \rightarrow \text{RADIO})\).
- \(CLOCK \ || \ RADIO = (CLOCK \ || \ RADIO)\).
If processes in a composition have actions in common, these actions are said to be **shared**. Shared actions are the way that process interaction is modeled. While unshared actions may be arbitrarily interleaved, a shared action must be executed at the same time by all processes that participate in the shared action.

\[
\text{MAKER} = (\text{make} \rightarrow \text{ready} \rightarrow \text{MAKER}). \\
\text{USER} = (\text{ready} \rightarrow \text{use} \rightarrow \text{USER}). \\
||\text{MAKER}_\text{USER} = (\text{MAKER} \ || \ \text{USER}).
\]

**MAKER** synchronizes with **USER** when **ready**.

**LTS? Traces? Number of states?**

Concurrency: processes & threads

A handshake is an action acknowledged by another:

\[
\text{MAKER}_\text{v2} = (\text{make} \rightarrow \text{ready} \rightarrow \text{used} \rightarrow \text{MAKER}_\text{v2}). \\
\text{USER}_\text{v2} = (\text{ready} \rightarrow \text{use} \rightarrow \text{used} \rightarrow \text{USER}_\text{v2}). \\
||\text{MAKER}_\text{USER}_\text{v2} = (\text{MAKER}_\text{v2} \ || \ \text{USER}_\text{v2}).
\]

3 states 3 states 3 x 3 states?
modeling interaction - multiple processes

Multi-party synchronization:

\[
\begin{align*}
  \text{MAKE}_A & = (\text{makeA} \rightarrow \text{ready} \rightarrow \text{used} \rightarrow \text{MAKE}_A). \\
  \text{MAKE}_B & = (\text{makeB} \rightarrow \text{ready} \rightarrow \text{used} \rightarrow \text{MAKE}_B). \\
  \text{ASSEMBLE} & = (\text{ready} \rightarrow \text{assemble} \rightarrow \text{used} \rightarrow \text{ASSEMBLE}). \\
  \text{||FACTORY} & = (\text{MAKE}_A \ || \ \text{MAKE}_B \ || \ \text{ASSEMBLE}).
\end{align*}
\]

composite processes

A composite process is a parallel composition of primitive processes. These composite processes can be used in the definition of further compositions.

\[
\begin{align*}
  \text{||MAKERS} & = (\text{MAKE}_A \ || \ \text{MAKE}_B). \\
  \text{||FACTORY} & = (\text{MAKERS} \ || \ \text{ASSEMBLE}).
\end{align*}
\]

Substituting the definition for \text{MAKERS} in \text{FACTORY} and applying the commutative and associative laws for parallel composition results in the original definition for \text{FACTORY} in terms of primitive processes.

\[
\text{||FACTORY} = (\text{MAKE}_A \ || \ \text{MAKE}_B \ || \ \text{ASSEMBLE}).
\]
Action relabeling

Relabeling functions are applied to processes to change the names of action labels. The general form of the relabeling function is:

\[/(newlabel_1/oldlabel_1, ..., newlabel_n/oldlabel_n)\].

Relabeling to ensure that composed processes synchronize on particular actions.

\[\text{CLIENT} = (\text{call} \rightarrow \text{wait} \rightarrow \text{continue} \rightarrow \text{CLIENT}).\]
\[\text{SERVER} = (\text{request} \rightarrow \text{service} \rightarrow \text{reply} \rightarrow \text{SERVER}).\]

Note that both \(newlabel\) and \(oldlabel\) can be sets of labels.

\[\text{CLIENT} \oplus \text{SERVER} = (\text{CLIENT} \oplus \text{SERVER})\]
\[/(\text{call/request, reply/wait}).\]
2.2 Implementing processes

Modeling processes as finite state machines using FSP/LTS.

Implementing threads in Java.

Note: to avoid confusion, we use the term process when referring to the models, and thread when referring to the implementation in Java.

CountDown class - start(), stop() and run()

```java
public void start() {
    counter = new Thread(this);
    i = N; counter.start();
}

public void stop() {
    counter = null;
}

public void run() {
    while(true) {
        if (counter == null) return;
        if (i>0)  { tick(); --i; }
        if (i==0) { beep(); return; }
    }
}
```

COUNTDOWN Model

start ->

stop ->

COUNTDOWN[i] process

recursion as a while loop

'STOP

when(i>0) tick -> CD[i-1]

when(i==0) beep -> STOP

STOP when run() returns