6SENG006W Concurrent Programming

Week 2

Introduction to Modelling Programs using FSP

Introduction to Modelling Programs using FSP

This lecture introduces the *modelling* approach we shall use to develop concurrent programs.

The topics we will cover are:

- Main difference between *sequential* & *concurrent* programs.
- Approach to *developing a concurrent program* using *modelling*.
- Introduce *Finite State Processes (FSP)* language, used to define processes.
- *Modelling* a concurrent program by a *Finite State Machine (FSM)*.
- FSP's modelling tool – *LTS Analyzer (`ltsa`)*, used to *animate* & *analyse* processes.
- Analyse a process's *behaviour attributes*: its *"alphabet"* & *"traces"*.

Note: See Chapters 1 & 2 of recommended book:
Concurrency: State Models & Java Programs, (2nd Edition)
J. Magee & J. Kramer, Wiley, 2006. (ISBN 978-0-470-09355-9)

PART I

Developing a Concurrent (Java) Program

by

Modelling Concurrency

Its a Concurrent World

In the *real world* complex systems & tasks can be broken down into a *set of simpler activities*.

For example, *house building* includes:

- bricklaying,
- carpentry,
- plumbing,
- wiring,
- roofing,
- making tea "*with 2 sugars*" &
- going to the betting shop.

Some activities do not always occur *strictly sequentially*, but can *overlap & take place concurrently*, e.g. plumbing & wiring.

Similarly, a program's activity is usually subdivided into simpler activities, each described by a *subprogram* (aka. *procedure* or *method*).

Sequential vs. Concurrent Programs

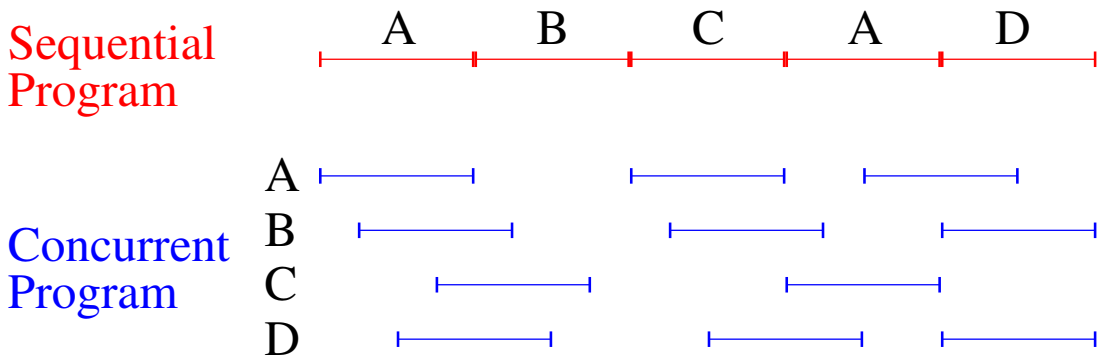

Figure : 2.1 Sequential vs. Concurrent program execution.

Sequential program — the subprograms (A, B, C & D) are executed sequentially & do not overlap in time, in a fixed order determined by the program & its input.

Concurrent program — the subprograms can be executed concurrently & can *overlap in time*, possibly with *no fixed order*.

E.g. Rik's bank account or Dining Philosophers.

The execution of a program (or subprogram) is termed a *process* & the execution of a concurrent program thus consists of *multiple processes*.

So What is a Process?

The general definition of a *process* is:

> **Definition:** *Process*
>
> A process is the execution of a sequential program.

The *state* of a process at any point in time consists of:

- the values of explicit variables, declared by the programmer &
- implicit variables such as the program counter & contents of data/address registers.

As a process executes, it transforms its state by executing *statements*.

Each statement consists of a sequence of one or more *atomic actions* that make *indivisible state changes*.

Examples of atomic actions are uninterruptible machine instructions that load & store registers.

So What is a Concurrent Program

The general definition of a *concurrent program* is:

Definition: *Concurrent Program*

A *concurrent program* consists of a collection of *subprograms*.

When the concurrent program executes each of the individual subprograms is executed by a separate *process*.

These processes generally execute at the *same time*, i.e. concurrently.

Therefore, the execution of a concurrent program thus consists of *multiple processes executing at the same time*.

In effect a *concurrent program* is viewed as a collection of concurrently executing processes, where each one is executing a sequential subprogram.

Developing a Concurrent Program: Goal & Problems

GOAL

Develop a concurrent (Java) program that satisfies the specified requirements & behaves in a safe manner.

PROBLEMS

- How should we design such a program?
- What software processes should we construct & how should we structure them to form a program?
- How can we ensure that our program provides the behaviour that we require while avoiding unsafe or undesirable behaviour?

Developing a Concurrent Program: Solution

SOLUTION?

Simply use previous development approaches & just use the appropriate Java concurrency constructs & rely on testing.

BUT

- Testing such software is difficult, far too many scenarios.
- Don't know when we have done sufficient number of tests.
- Testing is very difficult as it relies on executing the particular sequence of events & actions that cause a problem.
- Since concurrent events may occur in any order, or may not always occur, the problem sequences may never be tested, e.g. Therac-25 machine.

CONCLUSION

Need a better way to design, check & construct concurrent programs — so use a *"modelling"* approach.

The Modelling Approach

A *model* is a simplified representation of *"something"*, often a real world object or system.

It includes only those aspects of the system or objects *relevant to the problem at hand*.

For example, *aerodynamic properties of a plane* using a model:

- Used in wind tunnel tests, models only the external shape of the plane.
- But the: power of the plane's engines; number of seats; its cargo capacity; air crew's uniforms; what drinks are served – do not affect the plane's aerodynamic properties.
- The *simplification achieved by modelling* allows aeronautic engineers to *focus on a specific aspect* of the plane & *analyze specific properties* they are interested in & *ignore irrelevant details*.

Modern models tend to be mathematical in nature & as such can be analyzed using computers.

We will use a modelling approach to design concurrent programs.

Modelling Concurrent Systems

As we have seen a *model* is meant to be a simplified representation of *"something"*.

In our case the "something" we want to model is a *Concurrent System*.

So we need to be able to model both the *components* & the *dynamic behaviour & properties* of a concurrent system.

So our model must include *simplified representations* of the following:

- a *Process*,
- a *concurrent system (program)*, i.e. collection of processes executing concurrently,
- the *interactions* between a collection of processes,
- the *dynamic behaviour & properties* of a process & concurrent system.

CONCLUSION

If our model does **not** have these features then it is not **"fit-for-purpose"**!

Abstract Model of a Process & Concurrent System

Aim

We want a **good model**, but remember that a good model only includes those aspects of the system or objects *relevant to the problem at hand*.

Problem

The previous views of a process & concurrent system (program) are **far too detailed** if we want to **model** the **concurrency properties** of such systems.

This is especially the case if we want to deal with a concurrent system with a large number of processes,

Solution

Therefore, we need to simplify these views of a process & a concurrent system so that we are able to *focus solely on their concurrency properties*.

A “Model View” of a Process & concurrent System

So we need to ignore as much *irrelevant detail* as possible, e.g. program counters, memory allocation, etc, etc, etc, etc, ...

So we want our *model's view* of a process & concurrent system to be as simple as possible, **but** still have enough detail to represent what we are interested in.

Individual Process

We will use an *abstract model* of a *process*, where:

- a process has a *state*;
- its *state* is modified by *indivisible* or *atomic actions*;
- actions cause *transitions* from *one state to another*.

A Concurrent Program

A *concurrent program* is than modelled by a *collection of* these abstract processes executing in *parallel*.

PART II

Introduction to Finite State Processes (FSP)

What is Finite State Processes (FSP)?

Finite State Processes (FSP) is an *abstract language* that is used to develop concurrent systems of processes.

FSP is an example of a category of formal (mathematical) languages known as *process algebras*.

Other examples of process algebras are:

- *Communicating Sequential Processes* (CSP) – Hoare & Roscoe.
- *Calculus of Communicating Systems* (CCS) – R. Milner
- *Algebra of Communicating Processes* (ACP) – Baeten & Weijland.

FSP is *very small* (about 20 different constructs) when compared to a normal programming language with 100s or 1000s of constructs & features.

Using FSP to Develop Concurrent Systems

Finite State Processes (FSP) is used to *design*, *model* & *analyse* concurrent systems of processes.

Design concurrent system:

- in terms of *processes* & their *actions*,
- their *synchronisation* requirements &
- *interactions* between the processes.

Model designed system:

- using *Finite State Machines* (FSM),
- represented by *Labelled Transition System* (LTS graphs).

Analyse modelled system:

- check the system's *alphabet* is correct,
- check the system performs the correct, *actions*, i.e. *transitions*.
- check if it avoids *deadlock* & maintains *mutual exclusion*.

Developing a Concurrent (Java) Program using FSP

The diagram in Fig. 2.2 outlines the stages for developing a concurrent Java program we shall follow.

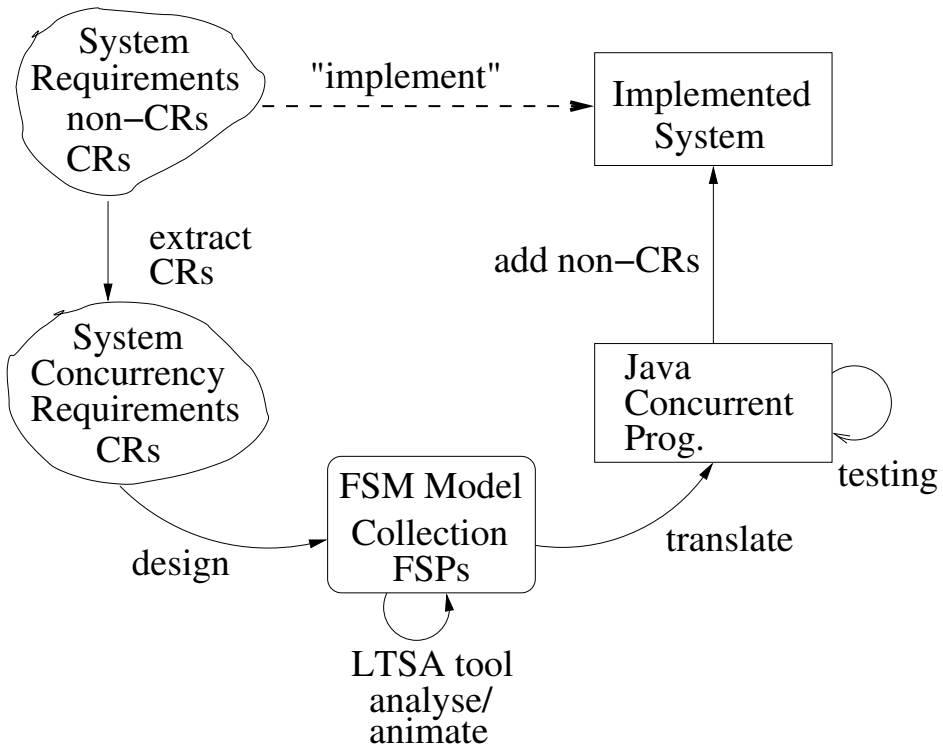

Figure : 2.2 Concurrent Java Program Development Process.

In subsequent lectures we will describe how these *abstract FSP processes* can be translated into a *concurrent Java program*, e.g. using *threads*.

Modelling Concurrent Programs & Processes using FSP

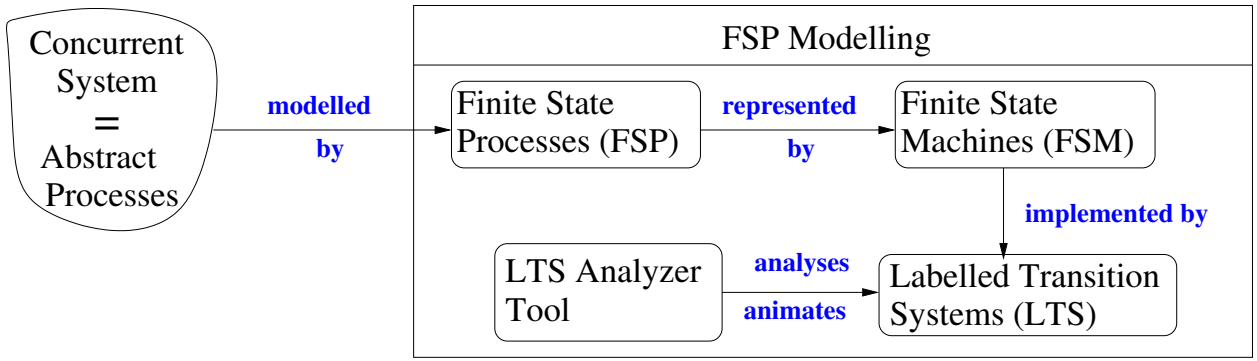

Figure : 2.3 Modelling an Abstract Process.

Concurrent system is a collection of *abstract processes*.

Abstract processes "defined/modelled" using *FSP language*.

FSP process "represented/modelled" by a *Finite State Machine* (FSM).

FSM model *abstracts away the details of a real program*, e.g. data representation, etc; & just *"focuses on the concurrency issues"*.

FSM are *"implemented"* by a *Labelled Transition System* (LTS).

Every FSP process has a *corresponding FMS/LTS description*.

Advantage of a *formal LTS model* is that it can be *animated* & its "behaviour" can be *analyzed*, using the LTS Analyzer tool.

FSP Abstract Model of Processes

FSP's *abstract model* of a process is as follows:

- Ignores details of *state representation* & *program/machine instructions*.
- Simply considers a process as having a *state* modified by *indivisible* or *atomic actions*.
- Each action causes a *transition* from the current state to the next state.
- The order in which actions are allowed to occur is determined by a *labelled transition graph*, i.e. an abstract representation of the program.

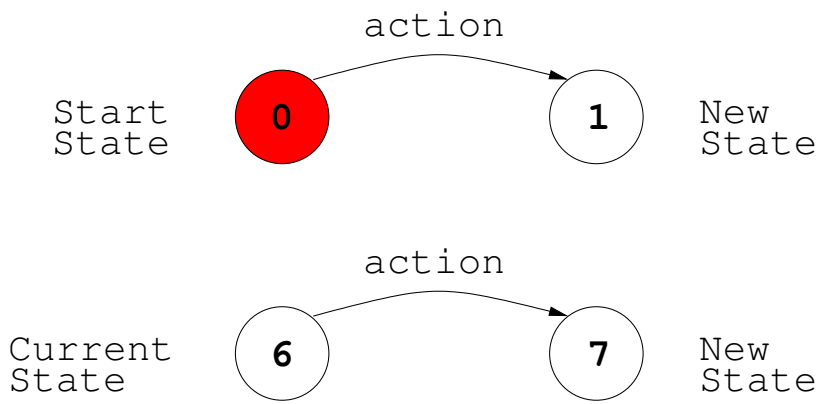

Figure : 2.4 A Label Transition Graphs (LTS).

These label transition graphs mean we will be modelling processes as *Finite State Machines (FSM)*.

Finite State Machines (FSM)

The state machine `SWITCH` models a light switch with the *actions* `on` & `off`.

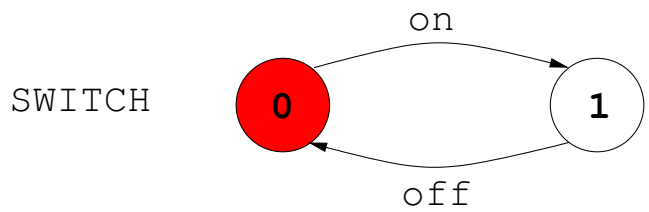

Figure : 2.5 Light switch state machine.

Use the following diagrammatic conventions:

- *Initial state* is always numbered 0.
- *Transitions* are always drawn in a clockwise direction.

For example, in Fig. 2.5, `on` causes a transition from state (**0**) to state (**1**) & `off` causes a transition from state (**1**) to state (**0**).

This form of state machine description is known as a *Labelled Transition System (LTS)*, since transitions are labelled with action names.

Diagrams of this form can be displayed in the LTS Analyzer tool, LTSA.

Although this representation of a process is *finite*, the behaviour described need *not be finite*.

Finite State Processes (FSP)

The graphical form of state machine description is excellent for simple processes.

However, it becomes unmanageable (& unreadable) for large numbers of states & transitions.

So a simple algebraic notation called Finite State Processes (FSP) is used to describe process models.

Every FSP description has a corresponding state machine (LTS) description.

We shall now introduce the language of FSP.

Note that we shall only be using the main features of FSP in the module. The full language definition of FSP can be found in Magee & Kramer's book, see Appendices A & B.

PART III

The Finite State Processes (FSP) Language

STOP – the really useful process!

Definition: *Process* STOP

STOP is a special predefined process that engages in no further actions.
It is used to represent the notion of a *"deadlocked"* process.

The following process *"does nothing"*, i.e. it does not engage in any actions at all & simply stops:

```
DOES_NOTHING = STOP .
```

Notation: process definitions are terminated by a *full stop "."*

The state machine for STOP, has **no** actions, & hence is **deadlocked**.

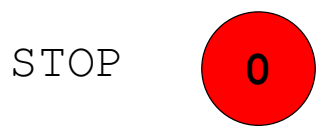

Figure : 2.6 STOP state machine.

This behaviour can be confirmed by animating (run) it in the LTSA animator – it does not offer us any actions to perform, i.e. *no out transitions*.

Action Prefix

Definition: *Action Prefix*

If `x` is an action & `P` a process then the action prefix
`( x -> P )`
describes a process that initially engages in the action `x` & then behaves exactly as described by `P`.

The action prefix operator "`->`" always has an *action on its left* & a *process on its right*.

Notation: In FSP, *identifiers* beginning with a:

- lowercase letter denotes an *action*,
- UPPERCASE letter denotes a *process*.

Action Prefix Example

The ONESHOT process illustrates a process that engages in the action once & then stops:

```
ONESHOT = ( once -> STOP ) .
```

Figure 2.7 illustrates the equivalent LTS state machine description for ONESHOT.

It shows that the action prefix in FSP describes a transition in the corresponding state machine description.

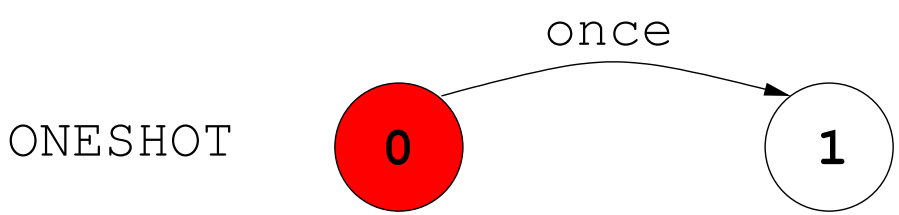

Figure : 2.7 ONESHOT state machine.

Processes with Repetitive Behaviour

Repetitive behaviour is described in FSP using *recursion*.

The following FSP process describes the light switch of Figure 2.5:

```
SWITCH = OFF ,
OFF    = ( on  -> ON ) ,
ON     = ( off -> OFF ) .
```

Notation: as indicated by the "," separators, the process definitions for `ON` & `OFF` are part of & local to the definition of the `SWITCH` process.

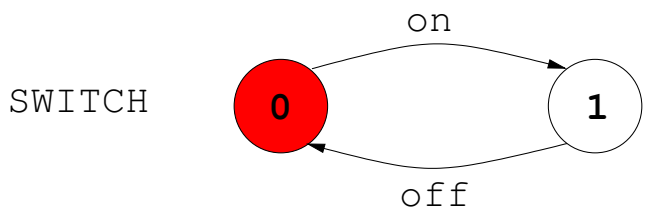

Figure : 2.5 Light switch state machine.

It should be noted that these local process definitions correspond to states in Figure 2.5:

- `OFF` defines state (0);
- `ON` defines state (1).

Alternative Definitions of SWITCH

Given the original definition of SWITCH:

```
SWITCH = OFF ,
OFF    = ( on  -> ON ) ,
ON     = ( off -> OFF ) .
```

We can define a more succinct definition of SWITCH by substituting the definition of ON in the definition of OFF

```
SWITCH2 = OFF ,
OFF     = ( on -> ( off -> OFF ) ) .
```

Finally, by substituting SWITCH for OFF, since they are defined to be equivalent, & dropping the internal parentheses we get:

```
SWITCH3 = ( on -> off -> SWITCH3 ) .
```

These three definitions for SWITCH generate identical state machines, i.e. the one given in Figure 2.8.

Exercise: Create the three version of SWITCH & verify this using the LTSA tool. Use it to draw the state machines for each one & confirm this is the case.

Example: TRAFFICLIGHT

The following process models the light sequence for a set of traffic lights.

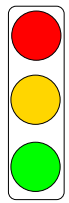

Figure : 2.8 Traffic Lights.

The FSP process definition is called TRAFFICLIGHT & is defined as:

```
TRAFFICLIGHT = ( red -> amber -> green
                     -> amber -> TRAFFICLIGHT ) .
```

Its equivalent state machine representation is in Figure 2.10.

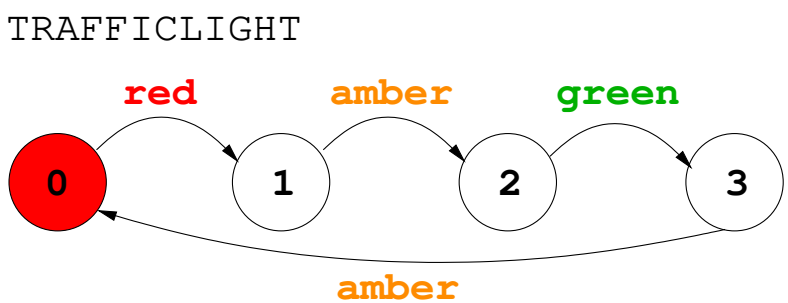

Figure : 2.9 TRAFFICLIGHT state machine.

Action Choice

If the only processes we could define have a single sequence of actions, like the `TRAFFICLIGHT` process, then we could only define very trivial processes.

What is missing is the ability for a process to *perform different sequences of actions*, i.e. produce different traces.

So to allow a process to describe more than a single execution trace, we introduce the *choice* operator.

Definition: *Choice*

If `x` & `y` are actions; `P` & `Q` are processes then

```
( x -> P | y -> Q )
```

describes a process which initially engages in either of the actions `x` or `y`.
After the first action has occurred, the subsequent behaviour is described by `P` if the first action was `x` & `Q` if the first action was `y`.

Note the choices must **all be actions, not processes**, e.g.
`( x -> P | Q )` is an error.

Choice Example: DRINKS Vending Machine

The DRINKS vending machine dispenses hot coffee if the **red** button is pressed & iced tea if the **blue** button is pressed.

```
DRINKS = (   red  -> coffee -> DRINKS
           | blue -> tea    -> DRINKS  ) .
```

Figure 2.10 is the state machine description of the drinks dispenser.

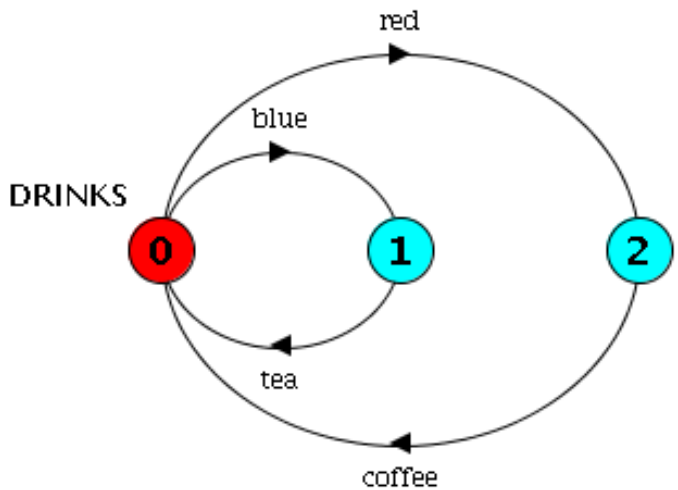

Figure : 2.10 DRINKS state machine.

Choice is represented as a *state with more than one outgoing transition*.

The *initial state* has two possible outgoing transitions labelled **red** & **blue**; in ltsa both **red** & **blue** actions are *ticked* (✓) as both are possible.

Input Actions vs. Output Actions

So far, we have not made any distinction between input & output actions, in our FSP processes.

Within this model, there is no semantic difference between an input action & an output action.

However, input actions are usually distinguished by forming part of a choice offered to the environment while outputs offer no choice.

In the `DRINKS` example,

- **`red`** & **`blue`** model input actions – *"pressing the buttons"*;
- `coffee` & `tea` model output actions – *"delivering the drinks"*.

Multiple Action Choices

A state may have *more than two outgoing transitions*.

The *choice operator* "|" can express a choice of more than two actions.

For example, process `FAULTY`, describes a chocolate vending machine that has four coloured buttons only one of which delivers a chocolate bar.

```
FAULTY = (   red    -> FAULTY
           | blue   -> FAULTY
           | green  -> FAULTY
           | orange -> kitkat -> FAULTY ) .
```

The *order of elements in the choice has no significance*, so the following process `FAULTY2` is equivalent:

```
FAULTY2 = (   orange -> kitkat -> FAULTY2
            | red    -> FAULTY2
            | green  -> FAULTY2
            | blue   -> FAULTY2 ) .
```

Note: for *"choice"* processes the LTSA tool may draw the LTS diagrams with the labels in a different order, but this is *not significant*.

Action Set Choices

The FAULTY process may be expressed more succinctly using a *set of action labels*.

```
FAULTY3 = (    { red, blue, green } -> FAULTY3
           | orange  -> kitkat     -> FAULTY3  ) .
```

The set of actions – { **red**, **blue**, **green** } is interpreted as being a choice of one of its members.

All three versions of FAULTY generate exactly the same state machine graph as depicted in Figure 2.11.

Note that **red**, **blue** & **green** label the same transition back to state (0).

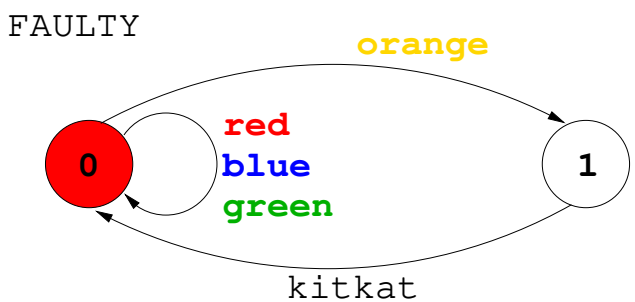

Figure : 2.11 FAULTY state machine.

Determinism vs. Nondeterminism

A question arises when animating one of these processes:

> *"Who or what makes the choice as to which action is executed?"*

In the `DRINKS` example, the *environment* makes the choice.

In this case the environment is *"the customer who presses a button"*.

If the environment makes the choice, it is known as an *external* or *deterministic choice*.

However, a choice may also be made *internally* within a process, without the intervention of the environment.

If the environment *does not make the choice*, it is known as an *internal* or *nondeterministic choice*.

A process that only contains *deterministic choices* is called a *deterministic* process & exhibits *deterministic behaviour* – *"usually good"*.

A process that contains one or more *nondeterministic choices* is called a *nondeterministic* process & **may** exhibit *nondeterministic behaviour* – *"usually bad"*.

Non-Deterministic Choice

Definition: *Non-Deterministic Choice*

The process

```
( x -> P | x -> Q )
```

is said to be *non-deterministic* since after the action `x`, it may behave as either `P` or `Q`.

The following `COIN` process is an example of a *non-deterministic* process.

```
COIN = (   toss -> heads -> COIN
         | toss -> tails -> COIN  ) .
```

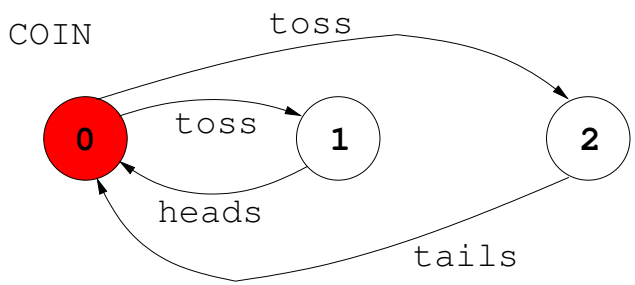

Figure : 2.12 `COIN` state machine.

After a toss action, the next action may be either `heads` or `tails`, but the environment has no control over which it will be.

Producing an FSP Program from an FSM

Given any LTS graph representation of a FSM we can *"reverse-engineer"* it to produce an FSP program that corresponds to it.

For example, consider the `DRINKS` process from Figure 2.13:

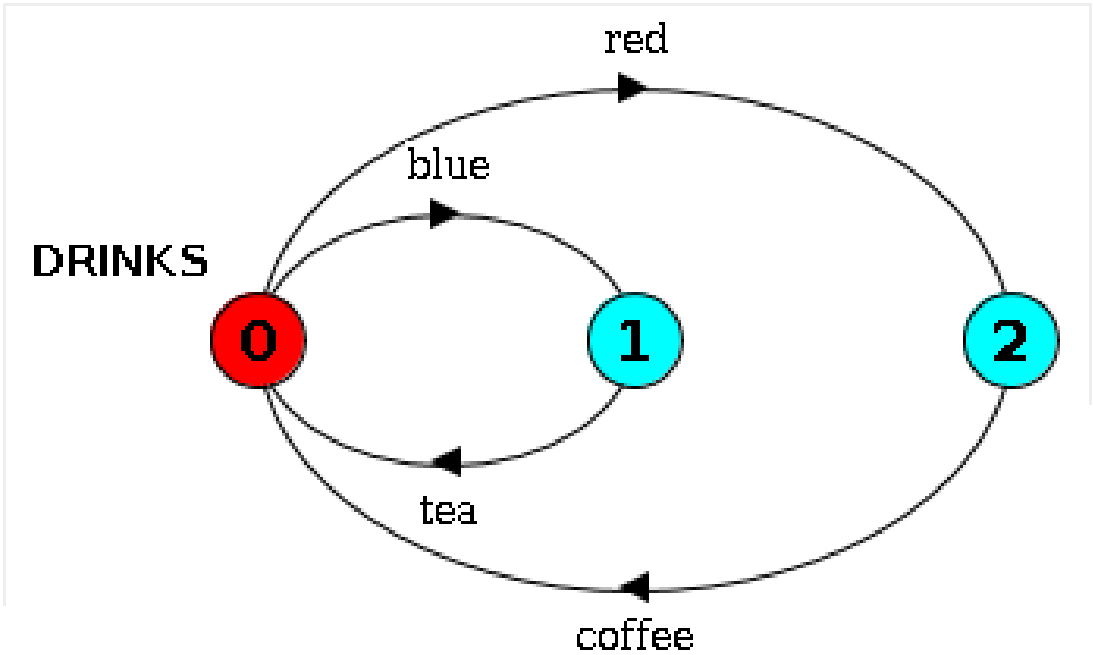

Figure : 2.13 `DRINKS` state machine.

Producing the FSP Program from the DRINKS LTS

We do this by defining *each state*: S0, S1, S2 as a *collection of recursive processes*.

Where each one simply captures the *outgoing transitions from it*, as follows:

```
PROCESS  =  S0 ,

S0  = ( red  -> S2 | blue  ->  S1 ) ,

S1  = ( tea -> S0 ) ,

S2  = ( coffee -> S0 ) .
```

NOTE: this is what is shown in the *"Transitions"* tab in the ltsa.

But ltsa uses Q0, Q1, Q2, ... rather than S0, S1, S2, ...

Simplifying the Process

We can simplify this by substituting the process definitions for `S1` & `S2` into `S0` as follows:

```
PROCESS  =  S0 ,
S0       = (   red  -> ( coffee -> S0 )
             | blue -> ( tea    -> S0 )  ) .
```

Finally, simplify further by substituting the process definition for `S0` into `PROCESS` as follows:

```
PROCESS = (   red  -> ( coffee -> PROCESS )
            | blue -> ( tea    -> PROCESS )  ) .
```

Now this is the *same process* as the original `DRINKS` process:

```
DRINKS = (   red  -> coffee -> DRINKS
           | blue -> tea    -> DRINKS  ) .
```

Apart from the different process name & braces, but this does not matter.

PART IV

FSP Process: Alphabets & Traces

Process Alphabets

Definition: *Process Alphabet*

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

For example, the alphabets of the `DRINKS` & `COIN` processes are:

```
alphabet( DRINKS ) = { red, blue, coffee, tea }
alphabet( COIN )   = { toss, heads, tails }
```

A process can **only do** the actions in its alphabet, but can have actions in its alphabet that it **never performs**.

For example, a program that writes to an integer variable may potentially write any (32/64-bit) integer value to the variable.

But, usually only writes a few integers out of this very large set of values.

In FSP, the alphabet of a process is determined implicitly by the set of actions referenced in its definition.

NOTE: this is what is shown in the *"Alphabet"* tab in `ltsa`.

The Trace of a Process

We have now introduced several of the basic operators of the FSP language & can use them to define quite sophisticated processes.

To help us understand the behaviour of these processes, it is useful to examine the sequences of actions, i.e. *traces* they can perform.

Definition: *Process Trace*

The sequence of actions produced by the execution of a process (or set of processes) is called a *trace*.

We shall now illustrate two ways we can represent the possible traces that a process can perform by representing them by:

- an *action labelled tree*; &
- a *set of sequences of actions*, i.e. a set of traces.

Representing a Process's Traces as a Tree

A simple way to visualise a process's traces is to represent them as a *tree*.

An *actual execution* of the process is then represented by following *one branch of the tree*.

A process's traces can be very long & so it is not always feasible to draw a complete trace tree; so will use a number of notational conventions.

Where:

- The *root node* of the tree represents the *initial state* of the process, (labelled "0" in the LTSA diagrams).
  (We will represent this using a small *red* circle.)
- The *branches* of the tree are labelled with the *actions* that the process performs & represent the *transitions*.
- The *non-terminal* (or *non-leaf*) *nodes* represent the state of the process after it has performed the action.
  (We will represent these using a small hollow black circle.)
- The *terminal* (or *leaf*) *nodes* represent the `STOP` (*deadlocked*) process.
  (We will represent these using a small solid black circle.)

An Execution of a Process in terms of a Tree

An *actual execution* of the process is then represented by following *one branch of the tree*.

Note that a process's traces can be very long & so it is not always feasible to draw a complete trace tree.

So we will use a number of notational conventions to represent this, see the following examples.

Sequential Process's Traces as a Tree

For example, if we consider the following two simple sequential processes:

```
P1 = ( a -> b -> c -> STOP ) .
P2 = ( a -> b -> c -> P2 ) .
```

Figure 2.14 depicts the trace trees for these two processes:

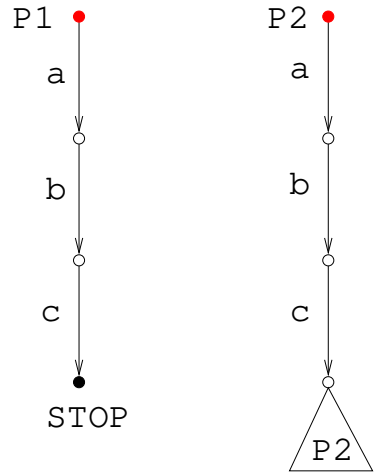

Figure : 2.14 Trace Trees for `P1` & `P2`

Process `P2` would have an *infinite trace* produced by repeating the sequence of actions `a, b, c`.

Obviously, cannot draw the trace tree for `P2`, so we use the process name (`P2`) inside a triangle to represent the *recursive* nature of the process.

Traces of Processes with Choice

Consider the following processes that include the *choice* operator:

```
P3 = ( a -> b -> c -> STOP | d -> e -> f -> STOP ) .

P4 = ( a -> b -> P4 | c -> d -> P4 ) .
```

Figure 2.15 depicts the trace trees for these two processes:

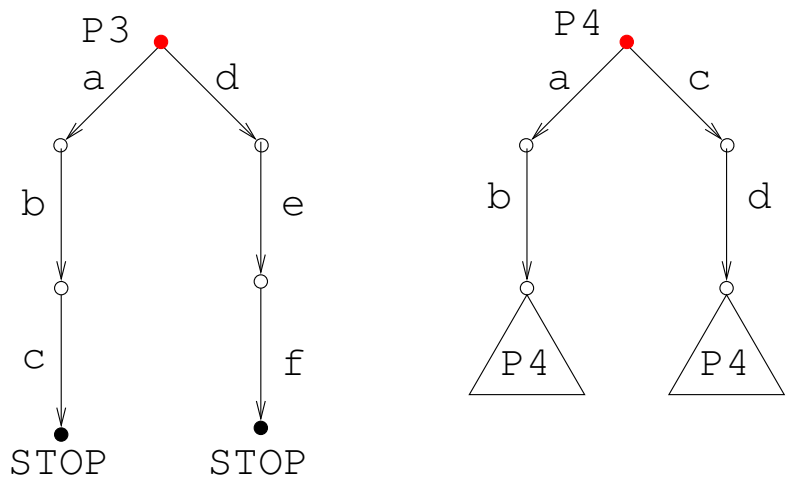

Figure : 2.15 Trace Trees for `P3` & `P4`

Process `P4` has *infinite traces*, produced by repeating & alternating the two sequences of actions `a, b` & `c, d`.

Traces of Processes with Choice & Non-deterministic Choice

Consider the following processes that include the choice operator & `P6` is a *nondeterministic choice*:

```
P5 = (   a -> ( b -> STOP | c -> STOP )
       | d -> e -> STOP ) .

P6 = ( a -> b -> STOP  |  a -> c -> STOP ) .
```

Figure 2.16 depicts the trace trees for these two processes:

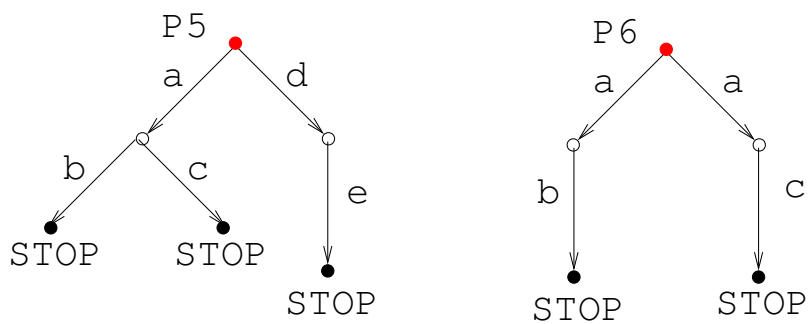

Figure : 2.16 Trace Trees for `P5` & `P6`

Process `P6` has a *nondeterministic choice* between the two `a` actions, this is represented by *two branches* from the root node *both labelled* with `a`.

Representing a Process's Traces as a Set

As we have seen the traces of a process can be represented by a *tree*.

Can also represent them as *set of sequences of actions*, i.e. a *set of traces*.

Note that *every process* can perform the *empty sequence of actions* or the *empty trace*, even `STOP`.

The trace set for `STOP` is simply:

```
traces( STOP ) = { <> }
```

The traces for processes `P1` & `P3` are as follows:

```
traces( a -> b -> c -> STOP )
    = { <>, <a>, <a, b>, < a, b, c> }
```

```
traces(   a -> b -> c -> STOP
        | d -> e -> f -> STOP )
    = { <>,
        <a>, <a, b>, < a, b, c>,
        <d>, <d, e>, < d, e, f>  }
```

DRINKS Traces

Using this approach, the set of traces for the DRINKS process

```
DRINKS = (   red  -> coffee -> DRINKS
           | blue -> tea    -> DRINKS  ) .
```

is given by the incomplete set (traces up to 4 events long):

```
traces( DRINKS )
 = { <>,
     <red>,
     <blue>,
     <red, coffee>,
     <blue, tea>,
     <red, coffee, red>,
     <red, coffee, blue>,
     <blue, tea, red>,
     <blue, tea, blue>,
     <red, coffee, red, coffee>,
     <red, coffee, blue, tea>,
     <blue, tea, red, coffee>,
     <blue, tea, blue, tea>,   ...   }
```

DRINKS Trace Tree

The trace tree for the partial set of DRINKS traces is:

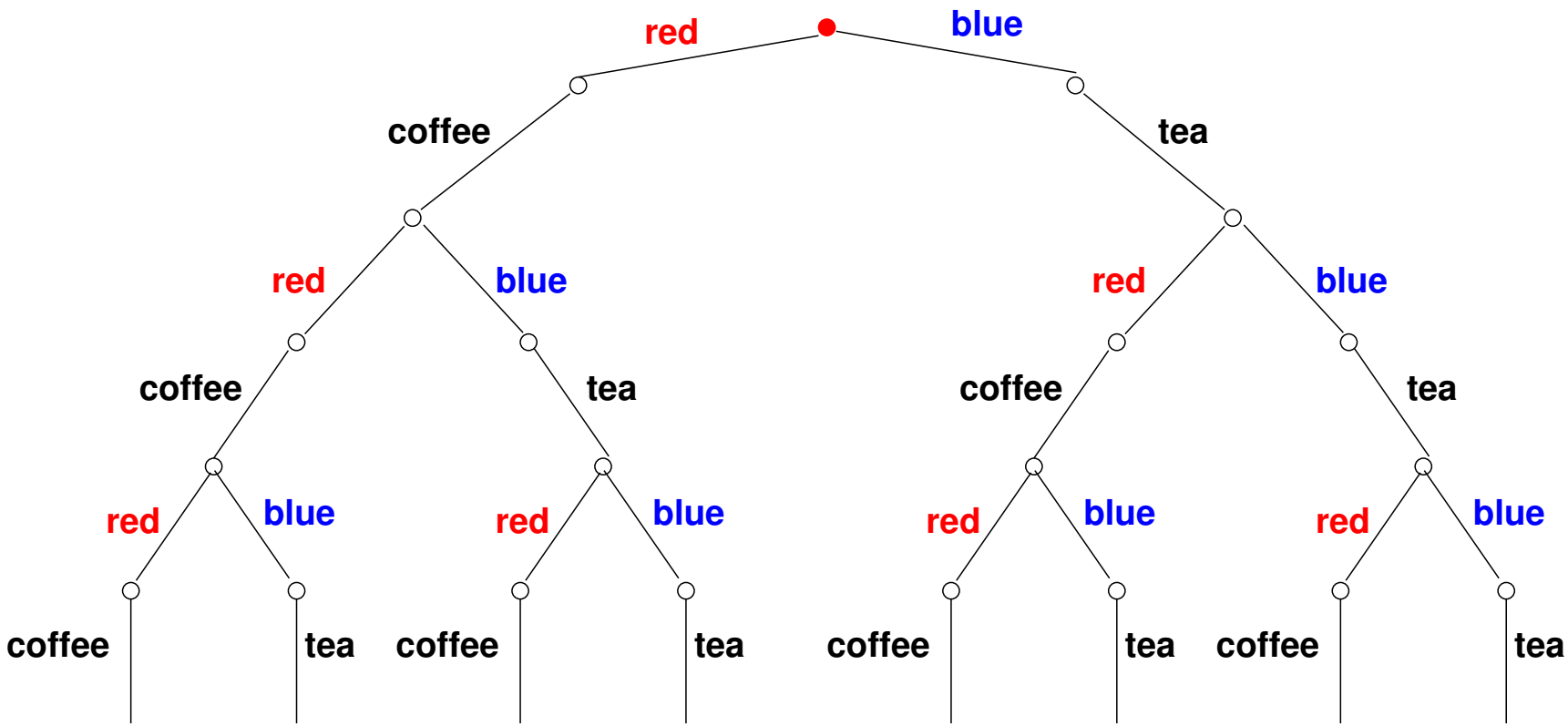

Figure : 2.17 DRINKS Trace Tree