Introduction to modelling in FSP (Finite State Processes) and its application for analysis of Java Mutex problems
Modeling Processes

Models are described using state machines, known as Labelled Transition Systems LTS. These are described textually as finite state processes (FSP) and displayed and analysed by the LTSA analysis tool.

♦ LTS - graphical form
♦ FSP - algebraic (textual) form
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.

A light switch LTS

on \rightarrow off \rightarrow on \rightarrow off \rightarrow on \rightarrow off \rightarrow \ldots\ldots\ldots

a sequence of actions or \textit{trace}
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 $P$.

\[
\text{ONESHOT} = (\text{once} \rightarrow \text{STOP}) .
\]

\text{ONSESHOT state machine}
\[(\text{terminating process)}\]

Convention: actions begin with lowercase letters

\text{PROCESSES} begin with uppercase letters
Repetitive behaviour uses recursion:

\[
\text{SWITCH} = \text{OFF},
\quad \text{OFF} = (\text{on} \to \text{ON}),
\quad \text{ON} = (\text{off} \to \text{OFF}).
\]

Substituting to get a more succinct definition:

\[
\text{SWITCH} = \text{OFF},
\quad \text{OFF} = (\text{on} \to (\text{off} \to \text{OFF})).
\]

And again:

\[
\text{SWITCH} = (\text{on} \to \text{off} \to \text{SWITCH}).
\]
The LTS atranimator 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:

\[ \text{TRAFFICLIGHT} = (\text{red} \rightarrow \text{orange} \rightarrow \text{green} \rightarrow \text{orange} \rightarrow \text{red} \rightarrow \text{orange} \rightarrow \text{green} \rightarrow \text{red} \rightarrow \text{orange} \rightarrow \text{green} \rightarrow \ldots) \]

LTS generated using LTS\(\text{A}\):

Trace:

\[ \text{red} \rightarrow \text{orange} \rightarrow \text{green} \rightarrow \text{orange} \rightarrow \text{red} \rightarrow \text{orange} \rightarrow \text{green} \rightarrow \ldots \]
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 - choice

FSP model of a drinks machine:

\[
\text{DRINKS} = (\text{red} \rightarrow \text{coffee} \rightarrow \text{DRINKS} \\
| \text{blue} \rightarrow \text{tea} \rightarrow \text{DRINKS})
\]

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 \(Q\).

\[
\begin{align*}
\text{COIN} &= (\text{toss} \rightarrow \text{HEADS} \mid \text{toss} \rightarrow \text{TAILS}), \\
\text{HEADS} &= (\text{heads} \rightarrow \text{COIN}), \\
\text{TAILS} &= (\text{tails} \rightarrow \text{COIN}).
\end{align*}
\]

Tossing a coin.

Possible traces?
Modeling failure

How do we model an unreliable communication channel which accepts in actions and if a failure occurs produces no output, otherwise performs an 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:

\[
BUFF = (\text{in}[i:0..3] -> \text{out}[i] -> \text{BUFF}).
\]

equivalent to

\[
BUFF = (\text{in}[0] -> \text{out}[0] -> \text{BUFF} \\
| \text{in}[1] -> \text{out}[1] -> \text{BUFF} \\
| \text{in}[2] -> \text{out}[2] -> \text{BUFF} \\
| \text{in}[3] -> \text{out}[3] -> \text{BUFF})
\]

or using a process parameter with default value:

\[
BUFF(\text{N}=3) = (\text{in}[i:0..\text{N}] -> \text{out}[i] -> \text{BUFF}).
\]
FSP - constant & range declaration

index expressions to model calculation:

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

\[
\text{SUM} = (\text{in}[a:T][b:T]\rightarrow\text{TOTAL}[a+b]),
\]
\[
\text{TOTAL}[s:R] = (\text{out}[s]\rightarrow\text{SUM}).
\]
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.

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

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

\[
\text{COUNTDOWN (N=3)} = (\text{start} \rightarrow \text{COUNTDOWN}[N]), \\
\text{COUNTDOWN}[i:0..N] = \\
(\text{when}(i>0) \text{ tick} \rightarrow \text{COUNTDOWN}[i-1] \\
| \text{when}(i==0) \text{ beep} \rightarrow \text{STOP} \\
| \text{stop} \rightarrow \text{STOP} \\
).
\]
The alphabet of a process is the set of actions in which it can engage.

Alphabet extension can be used to extend the implicit alphabet of a process:

\[
\text{WRITER} = (\text{write}[1] \rightarrow \text{write}[3] \rightarrow \text{WRITER}) + \{\text{write}[0..3]\}.
\]

Alphabet of \text{WRITER} is the set \{\text{write}[0..3]\}

(we make use of alphabet extensions in later chapters)
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.

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

\text{think-} \rightarrow \text{talk-} \rightarrow \text{scratch} \quad \text{Possible traces as a result of action interleaving.}
\text{think-} \rightarrow \text{scratch-} \rightarrow \text{talk}
\text{scratch-} \rightarrow \text{think-} \rightarrow \text{talk}
parallel composition - action interleaving

ITCH
0
2 states

CONVERSE
0
1
2
3 states

CONVERSE_ITCH
0
1
2
3
4
5
2 x 3 states

from ITCH
MVP3

from CONVERSE

(0,0) (0,1) (0,2) (1,2) (1,1) (1,0)

0
scratch

1
think

2
talk

3
think

4
talk

5
think

2 x 3 states
parallel composition - algebraic laws

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

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

Clock radio example:

\[\begin{align*}
\text{CLOCK} & = (\text{tick} \rightarrow \text{CLOCK}). \\
\text{RADIO} & = (\text{on} \rightarrow \text{off} \rightarrow \text{RADIO}). \\
||\text{CLOCK} \_ \text{RADIO} & = (\text{CLOCK} || \text{RADIO}).
\end{align*}\]

*LTS?  Traces?  Number of states?*
modeling interaction - shared actions

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**.
modeling interaction - handshake

A handshake is an action acknowledged by another:

\[
\text{MAKER}_v^2 = (\text{make} \rightarrow \text{ready} \rightarrow \text{used} \rightarrow \text{MAKER}_v^2).
\]
\[
\text{USER}_v^2 = (\text{ready} \rightarrow \text{use} \rightarrow \text{used} \rightarrow \text{USER}_v^2).
\]
\[
\text{| | MAKER}_v^2 \text{USER}_v^2 = (\text{MAKER}_v^2 \text{ | | USER}_v^2).
\]

Interaction constrains the overall behaviour.
modeling interaction - multiple processes

Multi-party synchronization:

\[
\text{MAKE}_A = (\text{makeA} \rightarrow \text{ready} \rightarrow \text{used} \rightarrow \text{MAKE}_A).
\]

\[
\text{MAKE}_B = (\text{makeB} \rightarrow \text{ready} \rightarrow \text{used} \rightarrow \text{MAKE}_B).
\]

\[
\text{ASSEMBLE} = (\text{ready} \rightarrow \text{assemble} \rightarrow \text{used} \rightarrow \text{ASSEMBLE}).
\]

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

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

\[
||\text{MAKERS} = (\text{MAKE}_A \; || \; \text{MAKE}_B).
\]

\[
||\text{FACTORY} = (\text{MAKERS} \; || \; \text{ASSEMBLE}).
\]

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.

\[
||\text{FACTORY} = (\text{MAKE}_A \; || \; \text{MAKE}_B \; || \; \text{ASSEMBLE}).
\]
a:P prefixes each action label in the alphabet of P with a.

Two instances of a switch process:

\[ \text{SWITCH} = (\text{on} \rightarrow \text{off} \rightarrow \text{SWITCH}). \]

\[ ||\text{TWO\_SWITCH} = (a:\text{SWITCH} || b:\text{SWITCH}). \]

An array of instances of the switch process:

\[ ||\text{SWITCHES}(N=3) = (\text{forall}[i:1..N] s[i]:\text{SWITCH}). \]

\[ ||\text{SWITCHES}(N=3) = (s[i:1..N]:\text{SWITCH}). \]
Process prefixing is useful for modeling shared resources:

- `RESOURCE = (acquire→release→RESOURCE).`
- `USER = (acquire→use→release→USER).`
- `||RESOURCE_SHARE = (a:USER || b:USER || {a,b}::RESOURCE).`
process prefix labels for shared resources

How does the model ensure that the user that acquires the resource is the one to release it?
Relabeling functions are applied to processes to change the names of action labels. The general form of the relabeling function is:

\[\{\text{newlabel}_1/\text{oldlabel}_1, \ldots \text{newlabel}_n/\text{oldlabel}_n\}\].

Relabeling to ensure that composed processes synchronize on particular actions.

\begin{align*}
\text{CLIENT} &= (\text{call} \to \text{wait} \to \text{continue} \to \text{CLIENT}) . \\
\text{SERVER} &= (\text{request} \to \text{service} \to \text{reply} \to \text{SERVER}) .
\end{align*}
action relabeling

\[ \text{CLIENT} \cup \text{SERVER} = (\text{CLIENT} \cup \text{SERVER}) \]
\[ \{/\text{call/request, reply/wait}\}. \]
action relabeling - prefix labels

An alternative formulation of the client server system is described below using qualified or prefixed labels:

```
SERVERv2 = (accept.request
             ->service->accept.reply->SERVERv2).
CLIENTv2 = (call.request
             ->call.reply->continue->CLIENTv2).

||CLIENT_SERVERv2 = (CLIENTv2 || SERVERv2)
             /{call/accept}.
```
action hiding - abstraction to reduce complexity

When applied to a process P, the hiding operator \( \{a_1..a_x\} \) removes the action names \( a_1..a_x \) from the alphabet of P and makes these concealed actions "silent". These silent actions are labeled \( \tau \). Silent actions in different processes are not shared.

Sometimes it is more convenient to specify the set of labels to be exposed....

When applied to a process P, the interface operator \( @\{a_1..a_x\} \) hides all actions in the alphabet of P not labeled in the set \( a_1..a_x \).
action hiding

The following definitions are equivalent:

\[
\text{USER} = (\text{acquire} \to \text{use} \to \text{release} \to \text{USER}) \setminus \{\text{use}\}.
\]

\[
\text{USER} = (\text{acquire} \to \text{use} \to \text{release} \to \text{USER}) @ \{\text{acquire}, \text{release}\}.
\]

Minimization removes hidden \(\tau\) actions to produce an LTS with equivalent observable behavior.
structure diagrams

Process $P$ with alphabet $\{a,b\}$.

Parallel Composition $(P || Q) / \{m/a, m/b, c/d\}$

Composite process $||S = (P || Q) @ \{x,y\}$
structure diagrams

We use structure diagrams to capture the structure of a model expressed by the static combinators: *parallel composition, relabeling* and *hiding*.

\[
\text{range } T = 0..3 \\
\text{BUFF} = (\text{in}[i:T] \rightarrow \text{out}[i] \rightarrow \text{BUFF}).
\]

\[ ||\text{TWOBUFF} = ? \]
structure diagrams

Structure diagram for CLIENT_SERVER?

```
CLIENT
  call
  continue
  wait

SERVER
  request
  reply
  service
```

Structure diagram for CLIENT_SERVERv2?

```
CLIENTv2
  call
  continue

SERVERv2
  accept
  service
```
structure diagrams - resource sharing

RESOURCE = (acquire->release->RESOURCE).
USER =  (printer.acquire->use
       ->printer.release->USER).

||PRINTER_SHARE
    = (a:USER||b:USER||{a,b}::printer:RESOURCE).

MVP3
Shared Objects & Mutual Exclusion

**Concepts:** process interference.
mutual exclusion.

**Models:** model checking for interference
modeling mutual exclusion

**Practice:** thread interference in shared Java objects
mutual exclusion in Java
(synchronized objects/methods).
Ornamental garden problem:

People enter an ornamental garden through either of two turnstiles. Management wish to know how many are in the garden at any time.

The concurrent program consists of two concurrent threads and a shared counter object.
The **Turnstile** thread simulates the periodic arrival of a visitor to the garden every second by sleeping for a second and then invoking the `increment()` method of the counter object.
ornamental garden program

The `Counter` object and `Turnstile` threads are created by the `go()` method of the Garden applet:

```java
private void go() {
    counter = new Counter(counterD);
    west = new Turnstile(westD, counter);
    east = new Turnstile(eastD, counter);
    west.start();
    east.start();
}
```

Note that `counterD`, `westD` and `eastD` are objects of `NumberCanvas` used in chapter 2.
Turnstile class

class Turnstile extends Thread {
    NumberCanvas display;
    Counter people;

    Turnstile(NumberCanvas n,Counter c)
        { display = n; people = c; }

    public void run() {
        try{
            display.setvalue(0);
            for (int i=1;i<=Garden.MAX;i++){
                Thread.sleep(500); //0.5 second between arrivals
                display.setvalue(i);
                people.increment();
            }
        } catch (InterruptedException e) {}
    }
}
Counter class

```java
class Counter {
    int value=0;
    NumberCanvas display;

    Counter(NumberCanvas n) {
        display=n;
        display.setValue(value);
    }

    void increment() {
        int temp = value;   //read value
        Simulate.HWinterrupt();
        value=temp+1;       //write value
        display.setValue(value);
    }
}
```

Hardware interrupts can occur at arbitrary times.

The counter simulates a hardware interrupt during an increment(), between reading and writing to the shared counter value. Interrupt randomly calls Thread.yield() to force a thread switch.
After the East and West turnstile threads have each incremented its counter 20 times, the garden people counter is not the sum of the counts displayed. Counter increments have been lost. Why?
concurrent method activation

Java method activations are not atomic - thread objects east and west may be executing the code for the increment method at the same time.

```
west
PC
program counter

shared code

increment:

read value

write value + 1

east
PC
program counter
```
ornamental garden Model

Process \textsc{var} models read and write access to the shared counter \textit{value}.

Increment is modeled inside \textsc{turnstile} since Java method activations are not atomic i.e. thread objects \textit{east} and \textit{west} may interleave their \textit{read} and \textit{write} actions.
ornamental garden model

const N = 4
range T = 0..N
set VarAlpha = { value.{read[T],write[T]} } }

VAR = VAR[0],
VAR[u:T] = (read[u] ->VAR[u]
   |write[v:T]->VAR[v]).

TURNSTILE = (go -> RUN),
RUN = (arrive-> INCREMENT
   |end  -> TURNSTILE),
INCREMENT = (value.read[x:T]
   -> value.write[x+1]->RUN
 )+VarAlpha.

||GARDEN = (east:TURNSTILE || west:TURNSTILE
 || { east,west,display} ::value:VAR)
  /{| go /{ east,west} .go,
     end/{ east,west} .end} .
Checking for errors - animation

Scenario checking - use animation to produce a trace.

Is this trace correct?
Exhaustive checking - compose the model with a TEST process which sums the arrivals and checks against the display value:

\[
\begin{align*}
\text{TEST} & = \text{TEST}[0], \\
\text{TEST}[v:T] & = \\
& \quad \text{(when (v<N) \{east.arrive, west.arrive\} \rightarrow \text{TEST}[v+1]} \\
& \quad \text{| end \rightarrow \text{CHECK}[v]} \\
, \\
\text{CHECK}[v:T] & = \\
& \quad \text{(display.value.read[u:T] \rightarrow)} \\
& \quad \quad \text{(when (u==v) \text{right} \rightarrow \text{TEST}[v]} \\
& \quad \quad \quad \text{| when (u!=v) \text{wrong} \rightarrow \text{ERROR})} \\
& \quad \text{)+\{display.VarAlpha\}.}
\end{align*}
\]

Like STOP, ERROR is a predefined FSP local process (state), numbered -1 in the equivalent LTS.
ornamental garden model - checking for errors

||TESTGARDEN = (GARDEN || TEST) .

Use **LTSA** to perform an exhaustive search for **ERROR**.

Trace to property violation in TEST:

go
east.arrive
east.value.read.0
west.arrive
west.value.read.0
east.value.write.1
west.value.write.1
end
display.value.read.1
wrong

**LTSA** produces the shortest path to reach **ERROR**.
Interference and Mutual Exclusion

Destructive update, caused by the arbitrary interleaving of read and write actions, is termed *interference*.

Interference bugs are extremely difficult to locate. The general solution is to give methods *mutually exclusive* access to shared objects. Mutual exclusion can be modeled as atomic actions.
Mutual exclusion in Java

Concurrent activations of a method in Java can be made mutually exclusive by prefixing the method with the keyword `synchronized`.

We correct `COUNTER` class by deriving a class from it and making the increment method `synchronized`:

```java
class SynchronizedCounter extends Counter {
    SynchronizedCounter(NumberCanvas n) {
        super(n);
    }
    synchronized void increment() {
        super.increment();
    }
}
```
Java associates a *lock* with every object. The Java compiler inserts code to acquire the lock before executing the body of the synchronized method and code to release the lock before the method returns. Concurrent threads are blocked until the lock is released.
**Java synchronized statement**

Access to an object may also be made mutually exclusive by using the `synchronized` statement:

```
  synchronized (object) { statements }
```

A less elegant way to correct the example would be to modify the `Turnstile.run()` method:

```
  synchronized(counter) {counter.increment();}
```

*Why is this “less elegant”?*

To ensure mutually exclusive access to an object, all object methods should be synchronized.
Modeling mutual exclusion

To add locking to our model, define a LOCK, compose it with the shared VAR in the garden, and modify the alphabet set:

\[
\text{LOCK} = (\text{acquire} \rightarrow \text{release} \rightarrow \text{LOCK}). \\
\text{LOCKVAR} = (\text{LOCK} \ || \ \text{VAR}). \\
\text{set VarAlpha} = \{\text{value.}\{\text{read}[T],\text{write}[T], \text{acquire}, \text{release}\}\}
\]

Modify TURNSTILE to acquire and release the lock:

\[
\text{TURNSTILE} = (\text{go} \rightarrow \text{RUN}), \\
\text{RUN} = (\text{arrive} \rightarrow \text{INCREMENT} \\
\text{|end} \rightarrow \text{TURNSTILE}), \\
\text{INCREMENT} = (\text{value.acquire} \\
\rightarrow \text{value.read}[x:T] \rightarrow \text{value.write}[x+1] \\
\rightarrow \text{value.release} \rightarrow \text{RUN} \\
)+\text{VarAlpha}.
\]
Revised ornamental garden model - checking for errors

A sample animation execution trace

```
go
east.arrive
east.value.acquire
east.value.read.0
east.value.write.1
east.value.release
west.arrive
west.value.acquire
west.value.read.1
west.value.write.2
west.value.release
end
display.value.read.2
right
```

Use TEST and LTSA to perform an exhaustive check.

*Is TEST satisfied?*
COUNTER: Abstraction using action hiding

To model shared objects directly in terms of their synchronized methods, we can abstract the details by hiding.

For SynchronizedCounter we hide read, write, acquire, release actions.

const N = 4
range T = 0..N

VAR = VAR[0],
VAR[u:T] = ( read[u]->VAR[u]

LOCK = (acquire->release->LOCK).

INCREMENT = (acquire->read[x:T]
-> (when (x<N) write[x+1]
->release->increment->INCREMENT
)
)+{read[T],write[T]}.

||COUNTER = (INCREMENT||LOCK||VAR)@{increment}.
Minimized LTS:

We can give a more abstract, simpler description of a COUNTER which generates the same LTS:

\[
\text{COUNTER} = \text{COUNTER}[0] \\
\text{COUNTER}[v:T] = (\text{when } (v<N) \text{ increment } \rightarrow \text{COUNTER}[v+1]).
\]

This therefore exhibits “equivalent” behavior i.e. has the same observable behavior.
Typical efficiency measurement (of synchronized calls)

double start = new Date().getTime();
for(long i = ITERATIONS; --i >= 0 ;)
    tester.locking(0,0);
double end = new Date().getTime();
double locking_time = end - start;

start = new Date().getTime();
for(long i = ITERATIONS; --i >= 0 ;)
    tester.not_locking(0,0);
end = new Date().getTime();
double not_locking_time = end - start;

double time_in_synchronization =
    locking_time - not_locking_time;
Synchronized calls are 7-8% slower than normal calls. Alternative to synchronization: value assignment

Methods returning a simple value in a shared variable do not need synchronization due to the atomicity of assignment, e.g.:

```java
class some_class
{
    int some_field;
    void f( some_class arg ) // deliberately no synchronized
    {
        // ...
        some_field = new_value; // do this last.
    }
}
```
Alternative to synchronization: priorities?

According to JLS, threads may be assigned different priorities. However, the effect is highly platform dependent!

*Solaris* provides both cooperative (no implicit preemption) and preemptive thread execution, but without the users control. *Windows* provides only preemptive execution. Interference bugs are extremely difficult to locate.
Typical execution platform
Summary

◆ Concepts
  ● process interference
  ● mutual exclusion

◆ Models
  ● model checking for interference
  ● modeling mutual exclusion

◆ Practice
  ● thread interference in shared Java objects
  ● mutual exclusion in Java (synchronized objects/methods).
  ● priorities do not offer predictable thread execution