Processes and Threads

Abhik Roychoudhury
CS 3211
National University of Singapore

Modified from Kramer and Magee’s lecture notes.
Reading material: Chapter 2 of Textbook.

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, or to offload time-consuming tasks, or to manage communications or other devices. Designing concurrent software can be complex and error prone. A rigorous engineering approach is essential.

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

Sequential program
(also use the term process)
Constructs:
-> Prefixing of an action
| Choice
Iterative repetition
[These are the ones used in a modeling language like Promela or programming language like Java]

Parallel Composition
Relabeling of action names
(while connecting processes, connect the disparate set of action names)
Hiding of actions
(internal to a process, not visible to the concurrent composition)

Modelling 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 LTS4 analysis tool.

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 LTS4 analysis tool.

- LTS - graphical form (state machines)
- FSP - textual form close to state machines
- Promela - imperative form (closer to programming)
- Java - programming language

Modelling processes

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.

Concept of a process as a sequence of actions.

Model processes as finite state machines.

Program processes as threads in Java.
FSP - action prefix

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

ONESHOT = (once -> STOP).

ONESHOT state machine (terminating process)

Convention: actions begin with lowercase letters

PROCESSES begin with uppercase letters

animation using LTSA

The LTSA animator can be used to produce a trace.

Ticked actions are eligible for selection.

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

In Promela (discussed earlier)

```promela
proctype switch()
{
  bit on;
  do ::
    on = 1;     // equivalent to the action "on";
    on = 0;     // equivalent to the action "off";
  od
}
```

Reasonably close to Java implementation.
Yet supported by formal analysis!!

FSP - action prefix & recursion

Repetitive behaviour uses recursion:

\[
\begin{align*}
\text{SWITCH} &= \text{OFF}, \\
\text{OFF} &= (\text{on} \rightarrow \text{ON}), \\
\text{ON} &= (\text{off} \rightarrow \text{OFF}) .
\end{align*}
\]

Substituting to get a more succinct definition:

\[
\begin{align*}
\text{SWITCH} &= \text{OFF}, \\
\text{OFF} &= (\text{on} \rightarrow (\text{off} \rightarrow \text{OFF})).
\end{align*}
\]

And again:

\[
\begin{align*}
\text{SWITCH} &= (\text{on} \rightarrow \text{off} \rightarrow \text{SWITCH}).
\end{align*}
\]

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:

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

FSP - choice

If \( x \) and \( y \) are actions 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 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:

\[ \text{DRINKS} = \{ \text{red} \rightarrow \text{coffee} \rightarrow \text{DRINKS} \mid \text{blue} \rightarrow \text{tea} \rightarrow \text{DRINKS} \} \]

LTS generated using LTS4

Possible traces?

Spot Exercise

Input and output

Input actions: red, blue
Output actions: coffee, tea

\[ \text{DRINKS} = \{ \text{red} \rightarrow \text{coffee} \rightarrow \text{DRINKS} \mid \text{blue} \rightarrow \text{tea} \rightarrow \text{DRINKS} \} \]

\textit{proctype DRINKS()}

\{ do
\text{ch_in? color;}
\text{if color == red -> ch_out!coffee;}
\text{color == blue -> ch_out!tea;}
\text{fi}
\text{od} \}

Promela description

Non-deterministic choice

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

\[ \text{COIN} = \{ \text{toss} \rightarrow \text{HEADS} \mid \text{toss} \rightarrow \text{TAILS} \}, \]

\[ \text{HEADS} = \{ \text{heads} \rightarrow \text{COIN} \}, \]

\[ \text{TAILS} = \{ \text{tails} \rightarrow \text{COIN} \}. \]

Tossing a coin.

Possible traces?

Another encoding

\[ \text{COIN} = \{ \text{toss} \rightarrow \text{OUTCOME} \}, \]

\[ \text{OUTCOME} = \{ \text{heads} \rightarrow \text{COIN} \mid \text{tails} \rightarrow \text{COIN} \}. \]

Possible traces (as sequence of action labels)
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...

\[
\text{CHAN} = (\text{in} -> \text{CHAN} | \text{in} -> \text{out} -> \text{CHAN}).
\]

Non-determinism in the physical world modeled by non-determinism in process description.

FSP - indexed processes and actions

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

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

equivalent to

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

or using a process parameter with default value:

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

Input and output actions are clarified at the initiative of the programmer!!

FSP - constant & range declaration

Index expressions to model calculation:

\[
\text{const } N = 1 \\
\text{range } T = 0..N \\
\text{range } R = 0..2*N
\]

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

Computation is described, apart from control flow!

FSP - guarded actions

The choice \((\text{when } B x -> P | y -> Q)\) 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.

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

Traces of COUNTDOWN

We will discuss the Java implementation later!!

View it as a parameterized process, which will be implemented as a Java class.
FSP - guarded actions
What is the following FSP process equivalent to?

const False = 0
P = (when (False) do anything -> P).

Answer:
STOP

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

Alphabet extension can be used to extend the \textit{implicit} alphabet of a process:

\begin{verbatim}
\end{verbatim}

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

Exercise
In FSP, model a process \textit{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.

\begin{verbatim}
FILTER = (in[v:0..5] -> DECIDE[v]),
DECIDE[v:0..5] = ( ? ).
\end{verbatim}

Organization
Modeling processes (so far)
Implementing Processes in Java (now)

Implementing processes - the OS view
A (heavyweight) process in an operating system is represented by its code, data and the state of the machine registers, given in a descriptor. In order to support multiple (lightweight) \textit{threads of control}, it has multiple stacks, one for each thread.

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

Implementing processes
Modeling processes as finite state machines using FSP/LTS.

Implementing threads in Java.
Threads and Processes

A Java Virtual Machine (JVM) usually runs as an OS process.
The JVM runs a multi-threaded Java program which has several threads. The thread scheduling may or may not be done by the JVM.

A thread is created by the keyword new, somewhat similar to the creation of other Java objects.

threads in Java

Since Java does not permit multiple inheritance, we often implement the run() method in a class not derived from Thread but from the interface Runnable.

```
class MyThread extends Thread {
    public void run() {
        //......
    }
}
```

Creating a thread object:

```
Thread a = new MyThread();
```

thread life-cycle in Java

An overview of the life-cycle of a thread as state transitions:

```
new Thread() -> start() -> Alive
start() causes the thread to call its run() method.

stop(), or
run() returns

The predicate isAlive() can be used to test if a thread has been started but not terminated. Once terminated, it cannot be restarted.
```

thread alive states in Java

Once started, an alive thread has a number of substates:

```
start() causes the thread to run

Running

sleep()

yield()

dispatch

Runnable

suspend()

Non-Runnable

resume()

stop()

wait() also makes a Thread Non-Runnable, and notify() Runnable
```

Java thread lifecycle - an FSP specification

```
THREAD = CREATED, CREATED = (start,stop)->RUNNING
RUNNING = (suspend,sleep,dispatch)->NON_RUNNABLE

yield, end ->RUNNABLE

stop, end ->TERMINATED

RUNNABLE = (suspend,dispatch)->NON_RUNNABLE

stop, end ->TERMINATED

NON_RUNNABLE = (resume)->RUNNABLE

stop, end ->TERMINATED

TERMINATED = STOP.
```
Java thread lifecycle - an FSP specification

States 0 to 4 correspond to CREATED, TERMINATED, RUNNING, NON-RUNNABLE, and RUNNABLE respectively.

CountDown timer example

Countdown class - start(), stop() and run()
Summary

- **Concepts**
  - **process** - unit of concurrency, execution of a program

- **Models**
  - **LTS** to model processes as state machines - sequences of atomic actions
  - **FSP** to specify processes using
    - prefix "->"
    - choice " | 
    - recursion.

- **Practice**
  - **Java threads** to implement processes.
  - Thread lifecycle - created, running, runnable, ...