Concurrent Programming
19530-V (WS01)

Lecture 2: Modeling Introduction

Dr. Richard S. Hall
rickhall@inf.fu-berlin.de

Our Approach to Concurrency

- Start with concurrency concepts
  - *Processes* are units of sequential execution
  - *Complex system* is a set of simpler activities, each represented as a process, that execute concurrently
- Use models to define concepts more clearly
  - Finite state machines / finite state processes / label transition systems
- Use Java to implement our models
  - We will discuss Java in a few weeks

Modeling

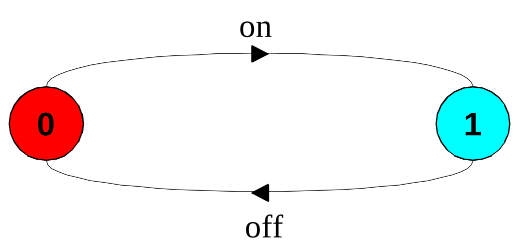
- Models are simplified representations of real-world entities
- We model something to better understand it
  - Focus on interesting aspects
  - Visualize potential outcomes
  - Create mechanisms to test and verify an approach
- We can use models in concurrent programs to achieve all of these goals

Modeling a Process

- A real process
  - Consists of
    - Explicit variables (e.g., programmer declarations)
    - Implicit variables (e.g., program counter, registers)
  - As a process executes, it transforms its state by executing program statements
    - Each statement is composed of a set of atomic actions
- Our simplified model of a process
  - Has a state that is modified by indivisible actions
  - An action transitions the current state to the next state
  - Allowable transitions from one state to the next must be completely specified

Modeling a Process

- A process model for a light switch (Lichtschalter)
  - There are two states for a light switch (OFF / ON), which we number consecutively
  - There are two allowable actions that transition back and forth between the two states



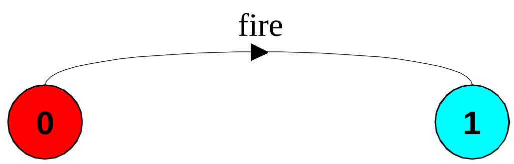
The LTS Modeling Technique

- In general, we have created a finite state machine model of a light switch
- Specifically we have a *label transition system* (LTS), since transition arcs have labels
- LTS is good for modeling because it is easy to understand and visualize
  - Unfortunately, LTS graphs do not scale for large processes
  - Instead, we will use an algebraic representation called *finite state processes* (FSP) that can be mapped to LTS

Finite State Process (FSP)

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 = (fire->STOP).
```



Convention:
Actions are written in lowercase,
processes are written in uppercase

FSP and Repetition

- Recursion is used to model repetition

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

- Substitution is used to simplify expressions

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

- Further substitution

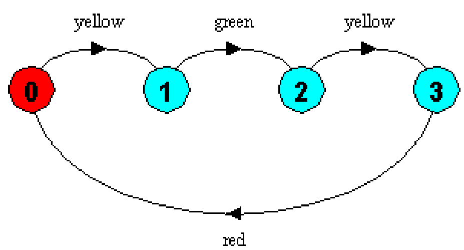
```
SWITCH = (on->(off->SWITCH)).
```

Another FSP Example

Modeling a traffic light

```
LIGHT = RED,
RED = (yellow->YELLOWGREEN),
YELLOWGREEN = (green->GREEN),
GREEN = (yellow->YELLOWRED),
YELLOWRED = (red->RED).
```

```
LIGHT = (yellow->green->yellow->red->LIGHT).
```



Expressing Choice in FSP

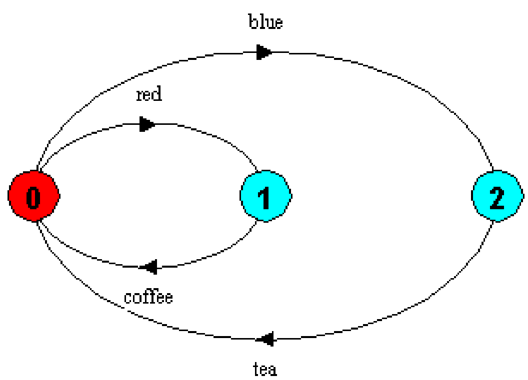
If **x** and **y** are actions then **(x->P | y->Q)** describes a process which initially engages in either of the actions **x** or **y**. After the first action has occurred, the subsequent behavior is described by **P** if the first action was **x** and **Q** if the first action was **y**.

- The "choice" is made by the environment or the process itself
- Order has no significance

Expressing Choice in FSP

Modeling a drink vending machine

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



Non-deterministic Choice in FSP

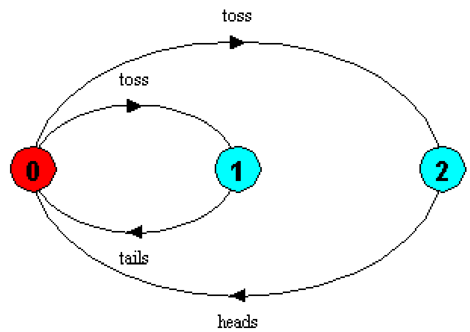
A choice in the form of **(x->P|x->Q)** is non-deterministic since after action **x**, the process may behave as either **P** or **Q**.

- It is not possible to predict the choice
- Can use to model non-deterministic failure

Non-deterministic Choice in FSP

Modeling a non-deterministic coin toss

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



Indexed Actions and Process Parameters in FSP

System that echoes its input from range 0 to 3

```
ECHO=(in0->out0->ECHO
     |in1->out1->ECHO
     |in2->out2->ECHO
     |in3->out3->ECHO).
```

Similar echo system using *indexed actions*

```
ECHO=(in[0]->out[0]->ECHO
     |in[1]->out[1]->ECHO
     |in[2]->out[2]->ECHO
     |in[3]->out[3]->ECHO).
```

Why have indexed actions?

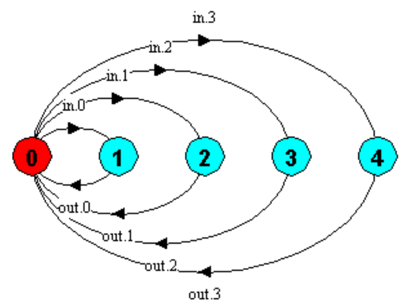
Indexed Actions and Process Parameters in FSP

Short-hand equivalent echo using index range

```
ECHO=(in[i:0..3]->out[i]->ECHO).
```

Also equivalent using *process parameters*

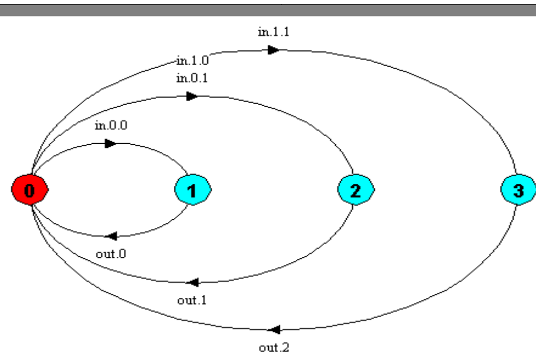
```
ECHO(N=3)=(in[i:0..N]->out[i]->ECHO).
```



Constants and Ranges in FSP

System that calculates the sum of two numbers

```
const N = 1
range T = 0..N
range R = 0..2*N
SUM         = (in[a:T][b:T]->TOTAL[a+b]),
TOTAL[s:R] = (out[s]->SUM).
```

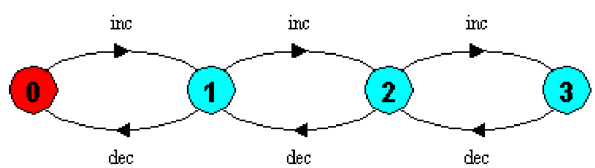


Guarded Actions in FSP

The choice **(when B x->P | y->Q)** means that **x** cannot be chosen unless **B** is true. If **B** is true then either **x** or **y** are eligible to be chosen.

System that increments/decrements from 0 to 3

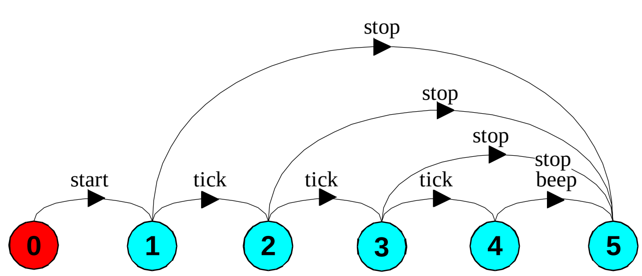
```
COUNT(N=3) = COUNT[0],
COUNT[i:0..N] = (when(i<N) inc->COUNT[i+1]
               |when(i>0) dec->COUNT[i-1]).
```



Guarded Actions in FSP

Timer that can be stopped and beeps at zero

```
TIMER(N=3) = (start->TIMER[N]),
TIMER[i:0..N] = (when(i>0) tick->TIMER[i-1]
               |when(i==0) beep->STOP
               |stop->STOP).
```



Process Alphabet

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

Timer example again

```
TIMER(N=3) = (start->TIMER[N]),
TIMER[i:0..N] = (when(i>0) tick->TIMER[i-1]
                |when(i==0) beep->STOP
                |stop->STOP).
```

Alphabet of the process

{ start, stop, tick, beep }

Concurrency and Parallelism

- Concurrency is the logical simultaneous execution of multiple processes
  - This may or may not include multiple physical processors
  - Simultaneous execution can be approximated by interleaving process execution on a single processor (e.g., preemptive multitasking)
- Parallelism is the actual simultaneous execution of multiple processes
  - Multiple physical processors are required

Both concurrency and parallelism require controlled access to shared resources to avoid conflicts; for our intents these two concepts are interchangeable.

Concurrency Modeling Issues

- How do we model process execution speed?
  - Speed and time are abstracted
- How do we model concurrency?
  - Arbitrary relative order of actions from different processes (preserves order of each process' actions)
- What is the result?
  - A general model independent of scheduling (asynchronous model of execution)

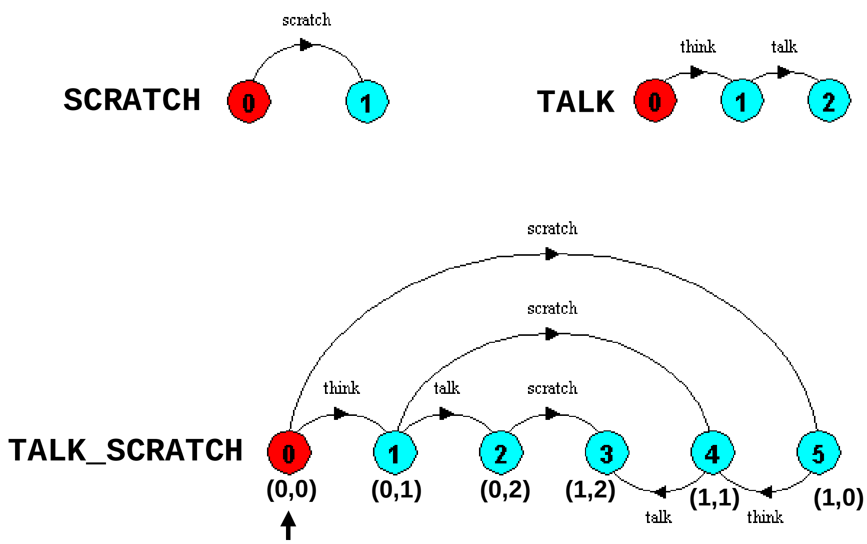
Parallel Composition in FSP

```
SCRATCH = (scratch->STOP).
TALK = (think->talk->STOP).
||TALK_SCRATCH = (SCRATCH || TALK).
```

<u>Possible traces</u>

```
think->talk->scratch
think->scratch->talk
scratch->think->talk
```

Interleaving of Parallel Actions



The states of the individual processes
SCRATCH and **ITCH**, respectively