SFDV3006
Concurrent Programming

Lecture 5 – Modelling Concurrent Programs using FSP

Based on lectures from the book *Concurrency:State Models and Java Programs*
by *Jeff Magee & Jeff Kramer*

Introduction

- As we have seen in the previous lectures writing concurrent programs can lead to violation of progress and safety properties.
- It is important we write correct programs and also verify that they do not have progress and safety violations
- To achieve this we first build and verify models and then implement the verified models in a language such as Java – this is a common practice in engineering
- Because it is very difficult to test concurrent programs due to their non-deterministic behaviour, verification of the model becomes even more significant
- Design → Model → Verify → Implement → Test

models

A model is a simplified representation of the real world.

Engineers use models to gain confidence in the adequacy and validity of a proposed design.

- focus on an aspect of interest - concurrency
- model animation to visualise a behaviour
- mechanical verification of properties (safety & progress)

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.

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 form

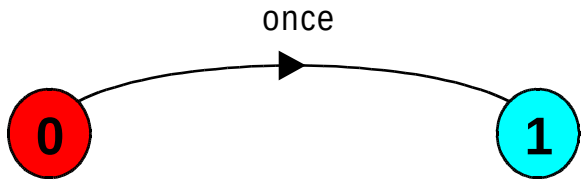
In FSP a process has *actions* and *states*, on executing an action a process moves from one state to another. In FSP actions are written in lower case and states in upper case

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



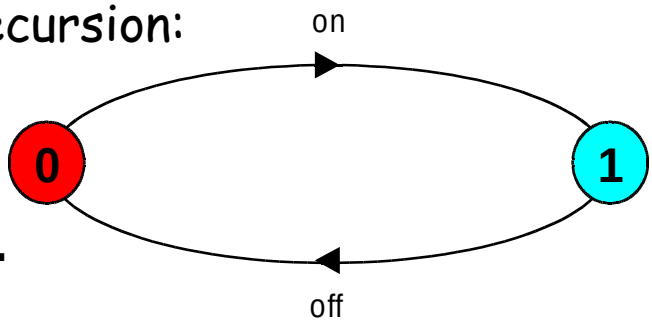
(terminating process)

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

FSP - action prefix & recursion

Repetitive behaviour uses recursion:

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



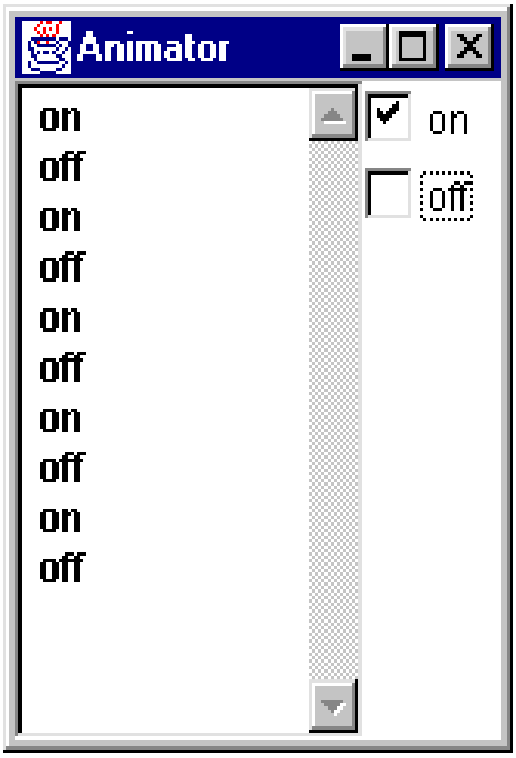
Substituting to get a more succinct definition:

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

And again:

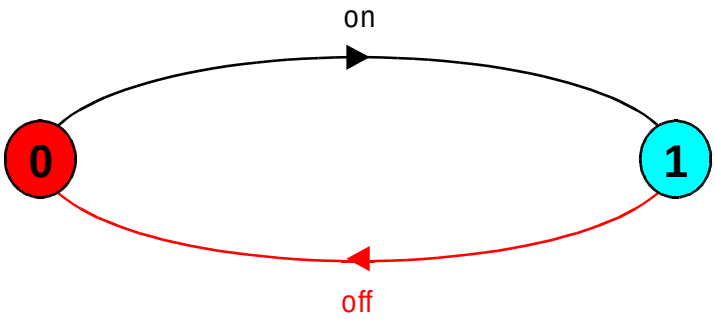
```
SWITCH = (on->off->SWITCH).
```

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.

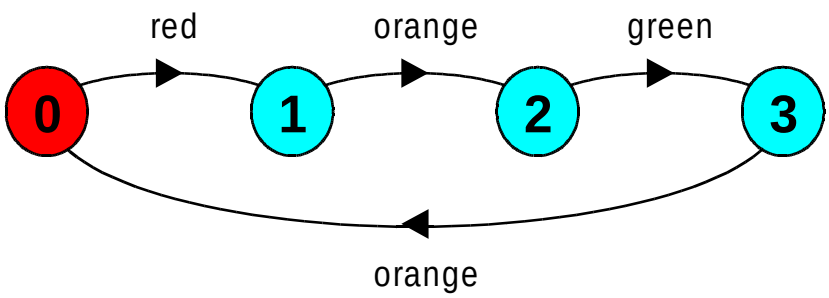


FSP - action prefix

FSP model of a traffic light :

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

LTS generated using *LTSA*:



Trace:

red→orange→green→orange→red→orange→green …

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**.

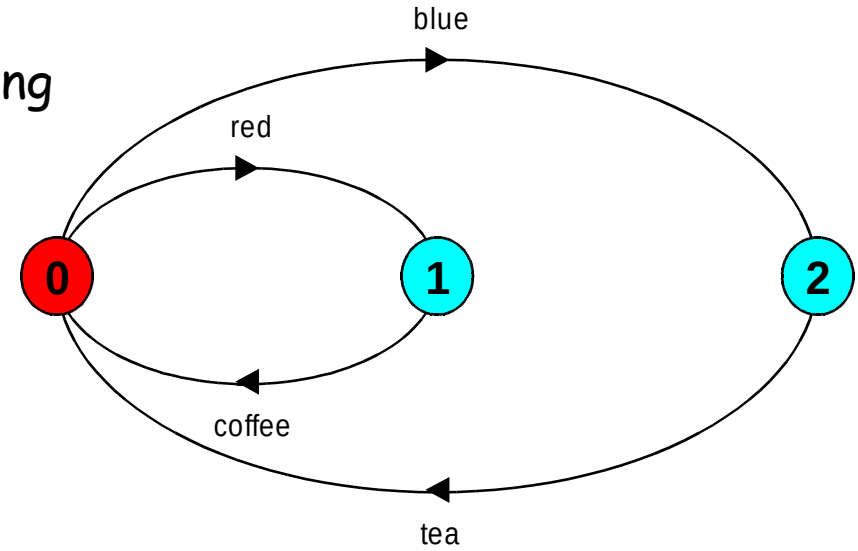
Who or what makes the choice?

FSP - choice

FSP model of a drinks machine :

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

LTS generated using *LTSA*:



Possible traces?

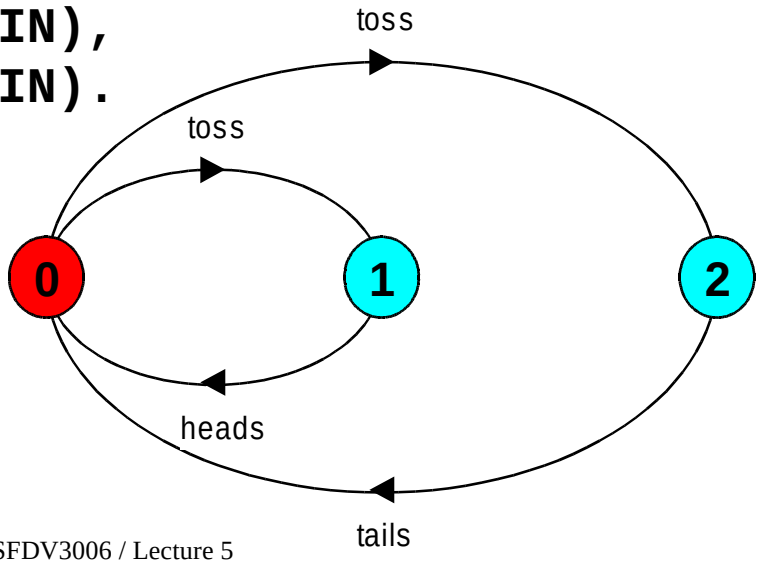
Non-deterministic choice

Process (x-> P | x -> Q) describes a process which engages in **x** and then behaves as either P or Q.

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

Tossing a coin.

Possible traces?



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]->out[i]-> BUFF).
```

equivalent to

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

or using a process parameter with default value:

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

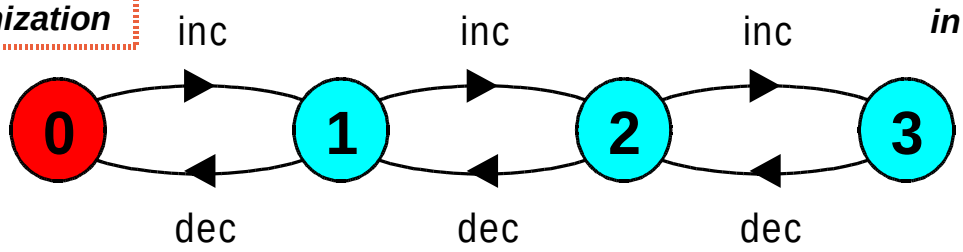
FSP - guarded actions

The choice **(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.

```
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 used to model Condition Synchronization

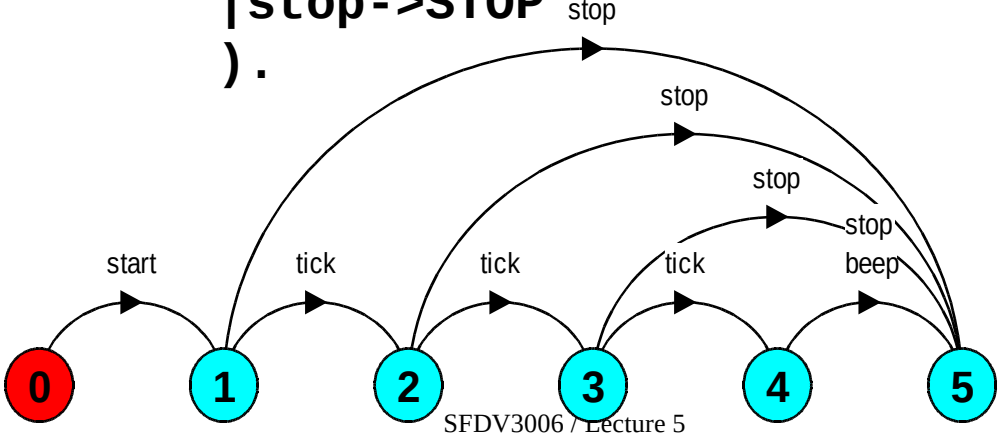
How to implement in Java?



FSP - guarded actions

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

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



FSP - process alphabets

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

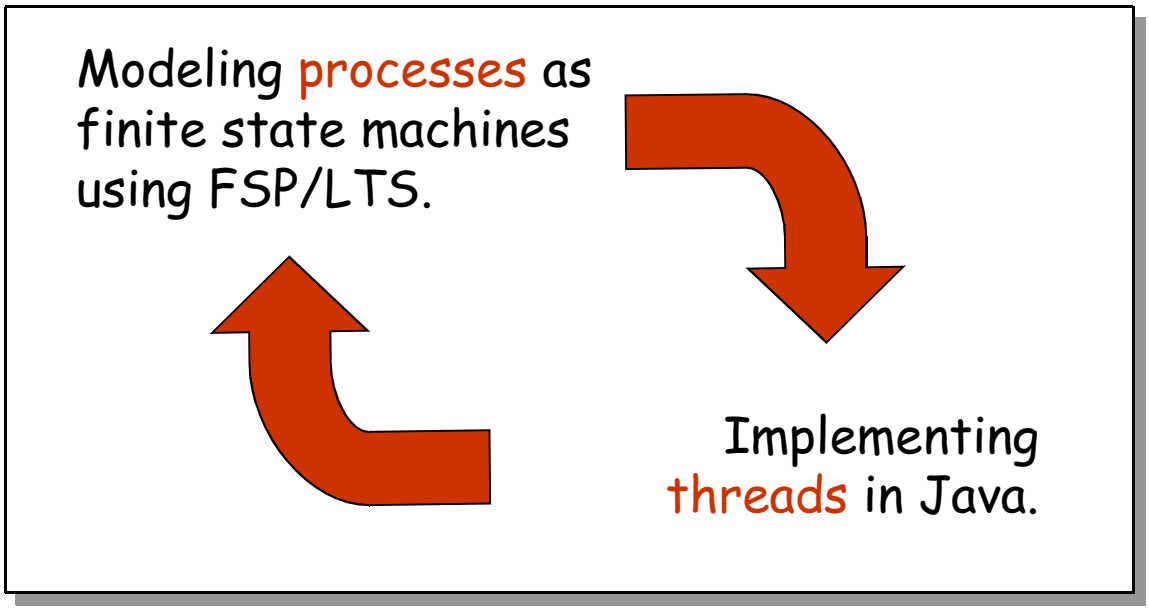
Alphabet extension can be used to extend the **implicit** alphabet of a process:

```
WRITER = (write[1]->write[3]->WRITER)
        +{write[0..3]}.
```

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

(we make use of alphabet extensions in later chapters)

Implementing the Model



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

Modelling concurrent execution using 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.

```
READ  = (read->STOP).
CONVERSE = (think->talk->STOP).

||CONVERSE_READ=(READ || CONVERSE).
```

LTS?

```
think→talk→read
think→read→talk
read→think→talk
```

Possible traces as a result of action interleaving.

parallel composition - algebraic laws

```
Commutative:   (P||Q) = (Q||P)
Associative:   (P||(Q||R))    = ((P||Q)||R)
                              = (P||Q||R).
```

Clock radio example:

```
CLOCK = (tick->CLOCK).
RADIO = (on->off->RADIO).

||CLOCK_RADIO = (CLOCK || RADIO).
```

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.

```
MAKER = (make->ready->MAKER).
USER  = (ready->use->USER).

||MAKER_USER = (MAKER || USER).
```

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

LTS? Traces? Number of states?

modeling interaction - handshake

A handshake is an action acknowledged by another:

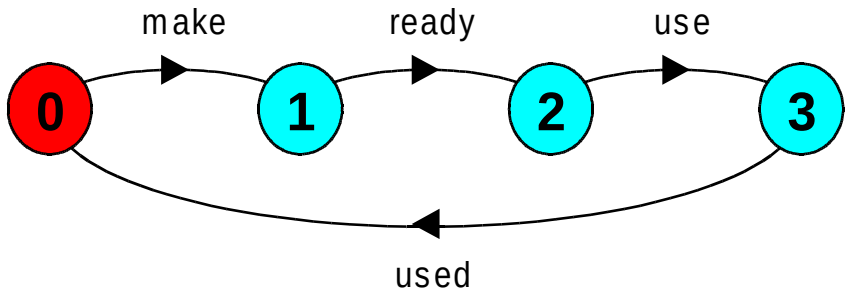
```
MAKERv2 = (make->ready->used->MAKERv2).
USERv2  = (ready->use->used ->USERv2).

||MAKER_USERv2 = (MAKERv2 || USERv2).
```

3 states

3 states

3 x 3 states?



4 states

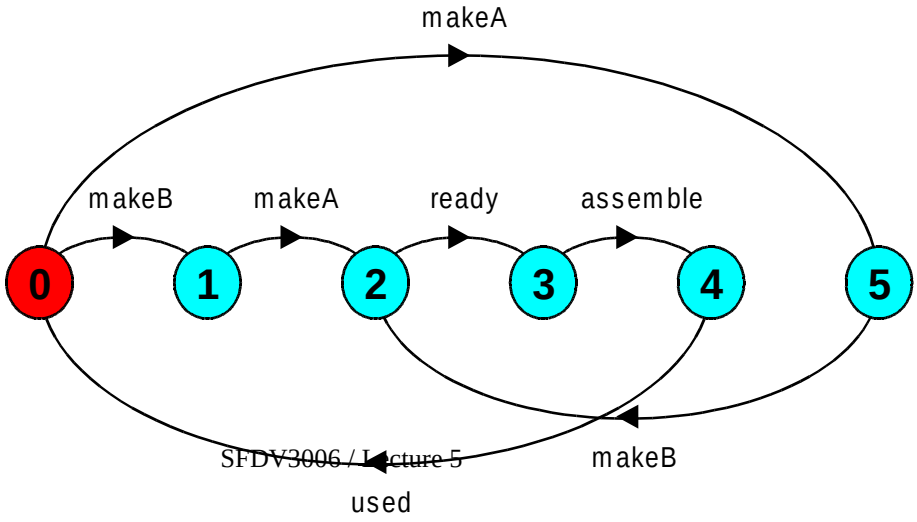
Interaction constrains the overall behaviour.

modeling interaction - multiple processes

Multi-party synchronization:

```
MAKE_A   = (makeA->ready->used->MAKE_A).
MAKE_B   = (makeB->ready->used->MAKE_B).
ASSEMBLE = (ready->assemble->used->ASSEMBLE).

||FACTORY =(MAKE_A || MAKE_B || 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.

```
||MAKERS = (MAKE_A || MAKE_B).

||FACTORY = (MAKERS || ASSEMBLE).
```

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

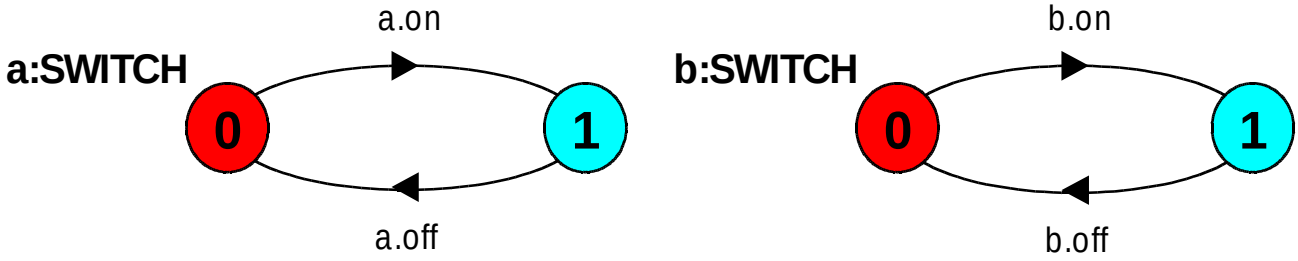
```
||FACTORY = (MAKE_A || MAKE_B ||
ASSEMBLE).
```

process labeling

a:P prefixes each action label in the alphabet of P with a.

Two **instances** of a switch process:

```
SWITCH =  (on->off->SWITCH).
||TWO_SWITCH = (a:SWITCH || b:SWITCH).
```



An array of **instances** of the switch process:

```
||SWITCHES(N=3) = (forall[i:1..N] s[i]:SWITCH).
||SWITCHES(N=3) = (s[i:1..N]:SWITCH).
```


process labeling by a set of prefix labels

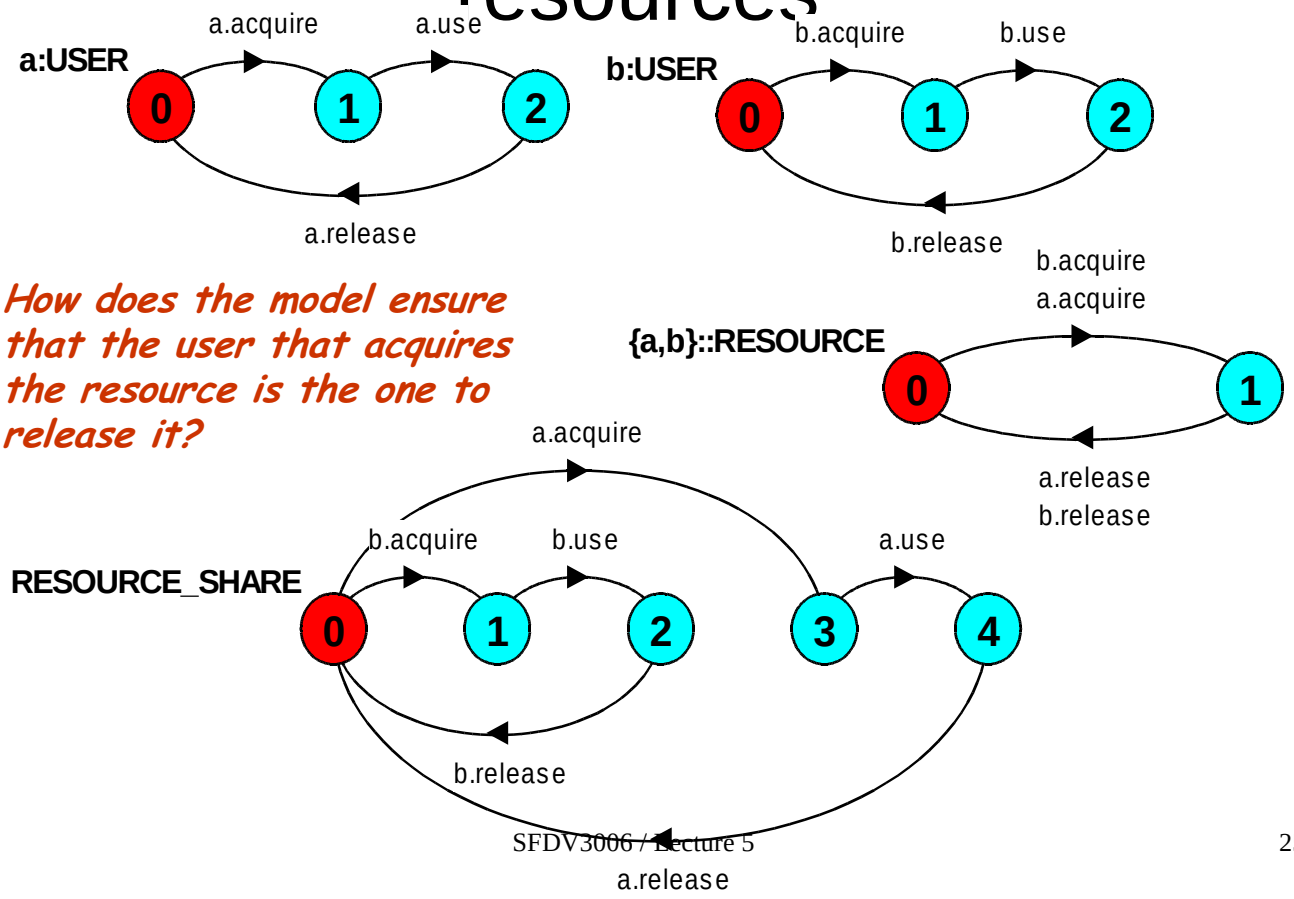
{a1,…,ax}::P replaces every action label n in the alphabet of P with the labels a1.n,…,ax.n. Further, every transition (n->X) in the definition of P is replaced with the transitions ({a1.n,…,ax.n} ->X).

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



action relabeling

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

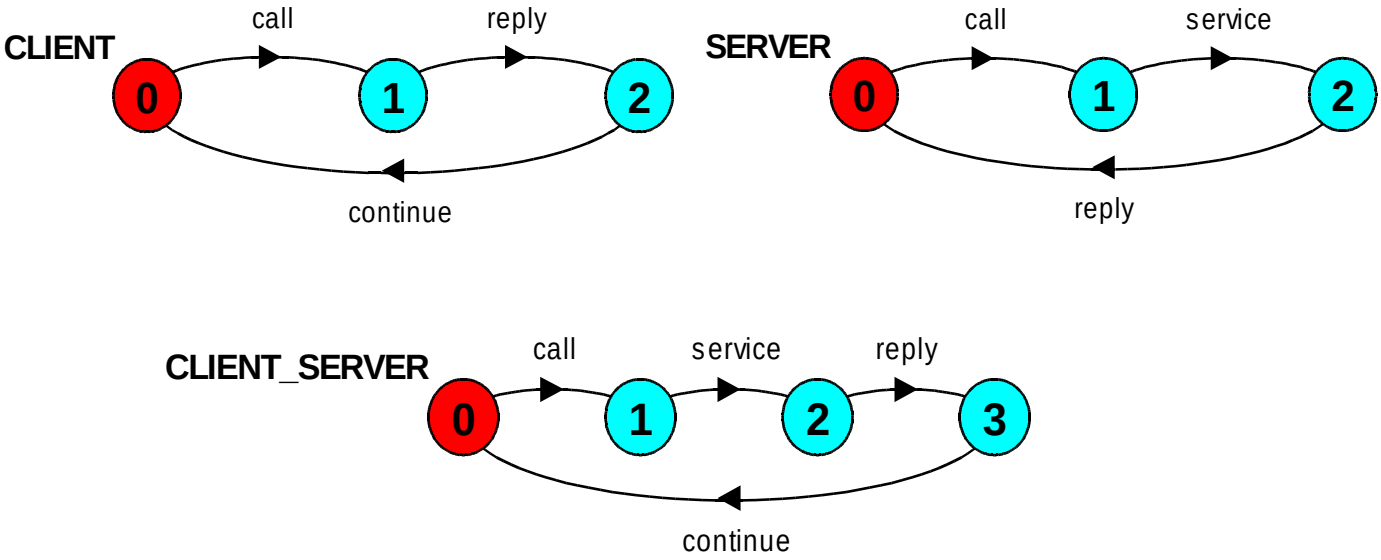
/{*newlabel_1*/ *oldlabel_1*,... *newlabel_n*/ *oldlabel_n*}.

Relabeling to ensure that composed processes synchronize on particular actions.

```
CLIENT = (call->wait->continue->CLIENT).
SERVER=(request->service->reply->SERVER).
```

action relabeling

```
||CLIENT_SERVER = (CLIENT || SERVER)
                          /{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 \{a1..ax} removes the action names a1..ax 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....

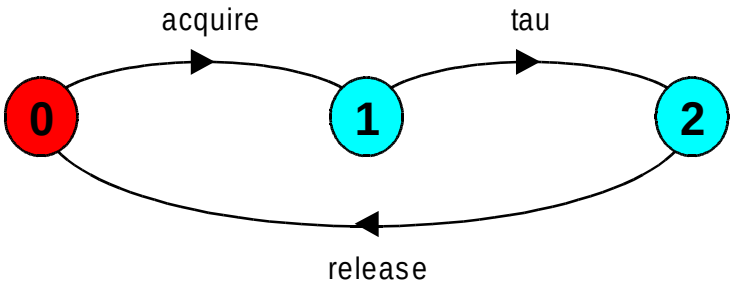
When applied to a process P, the interface operator @{a1..ax} hides all actions in the alphabet of P not labeled in the set a1..ax.

action hiding

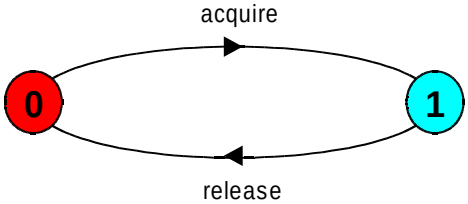
The following definitions are equivalent:

```
USER = (acquire->use->release->USER)
       \{use}.

USER = (acquire->use->release->USER)
       @{acquire,release}.
```



Minimization removes hidden **tau** actions to produce an LTS with equivalent observable behavior.



FSP for verification – an example

Consider the following FSP which shows two process P and Q using two shared resources – a printer and a scanner

```
RESOURCE = (get-> put-> RESOURCE).

P = (printer.get -> scanner.get -> copy -> printer.put -> scanner.put -> P).
Q = (scanner.get -> printer.get -> copy-> scanner.put -> printer.put -> Q).

||SYS = (p:P||q:Q ||{p,q}::printer:RESOURCE ||{p,q}::scanner:RESOURCE).
```

How to verify whether this will work correctly? We run the model using LTSA. LTSA tool will allow us to check the safety (deadlocks?) and progress of this model. This can even show you the trace to deadlock.

FSP for verification – example 2

Consider the following FSP for a bounded buffer which is shared by a Producer and a Consumer

How to convert this to Java?

```
//FSP for bounded buffer
const EMPTY = 0
const MAX_SIZE = 10

BUFF = BUFFER[EMPTY],
BUFFER[size:EMPTY..MAX_SIZE] = (when (size<MAX_SIZE) put -> BUFFER[size + 1]
                               |when (size > EMPTY) get -> BUFFER[size - 1]
                               ).
PRODUCER = (put -> PRODUCER).
CONSUMER = (get -> CONSUMER).

||PRODUCER_CONSUMER = (PRODUCER || CONSUMER || BUFF).
```

Can we model Unbounded buffer? How?

How to verify whether this will work correctly? We run the model using LTSA. LTSA tool will allow us to check the safety (deadlocks?) and progress of this model. This can even show you the trace to deadlock or other problems such as progress violations

Resources and references

- Concurrency: State Models & Java Programs, 2e, Jeff Magee & Jeff Kramer, Wiley
- *Concurrent State Models & Java Programs website - http://bit.ly/fspbook*
- *LTSA tool - http://bit.ly/ltsatool*
- *Course Website – http://sfdv3006.wikidot.com*