Concurrent Programming 19530-V (WS01)

Lecture 3: Modeling Continued

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

Parallel Composition in FSP

If **P** and **Q** are processes then **(P || Q)** represents the concurrent execution of **P** and **Q**. The operator **||** is the parallel composition operator.

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

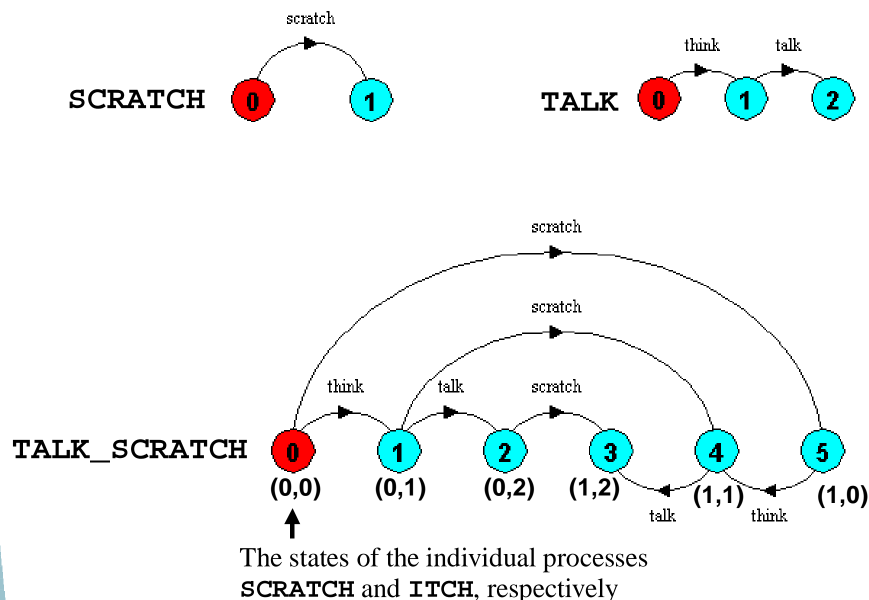
Parallel Composition in FSP

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

Possible traces

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

Interleaving of Concurrent Actions



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

Another Parallel Composition

Clock radio example

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

LTS?   Traces?   Number of states?

Process Interaction in FSP

If processes in a composition have actions in common, these actions are said to be ***shared*** -- shared actions are used to model process interaction.

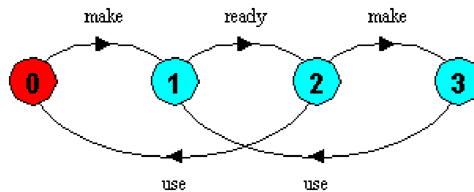
- How is a shared action different than a non-shared action?
  - The execution of non-shared actions may be arbitrarily interleaved
  - Shared actions must be executed at the same time by all processes that share the action

Process Interaction in FSP

Producer/consumer example

```
MAKE = (make->ready->MAKE).
USE = (ready->use->USE).
||MAKE_USE = (MAKE || USE).
```

MAKE synchronizes with **USE** when **ready**.

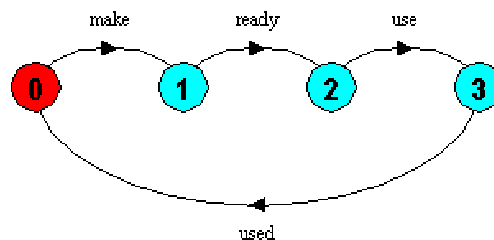


Process Handshaking in FSP

Handshaking to acknowledge an action by another

```
MAKEv2 = (make->ready->used->MAKEv2).
USEv2 = (ready->use->used->USEv2).
||MAKE_USEv2 = (MAKEv2 || USEv2).
```

3 states
3 states
3 x 3
states?



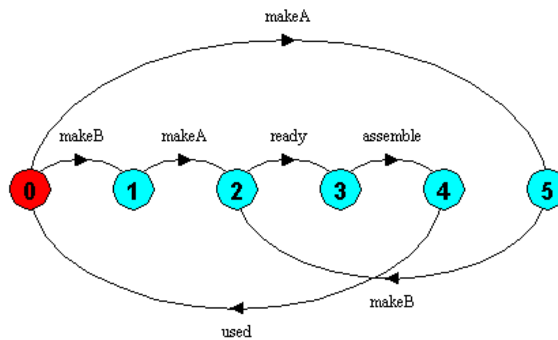
4 states

Interaction constrains the overall behavior

Process Handshaking in FSP

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 in FSP

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).
```

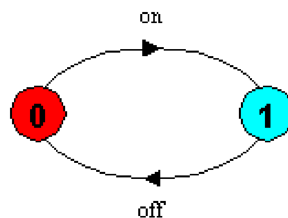
Using Duplicate Processes in FSP

Modeling two light switch processes

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

This does not work, why?

All actions are shared, thus the resulting process composition reduces to a single light switch.



Process Labeling in FSP

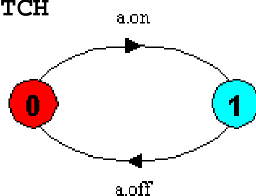
a:P prefixes each action label in the alphabet of **P** with **a**.

Correctly modeling two light switch processes

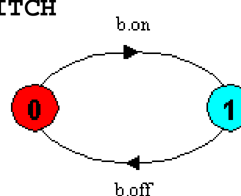
```
SWITCH = (on->off->SWITCH).
||TWO_SWITCH = (a:SWITCH || b:SWITCH).
```

The actions of the process are relabeled with the prefix

a:SWITCH

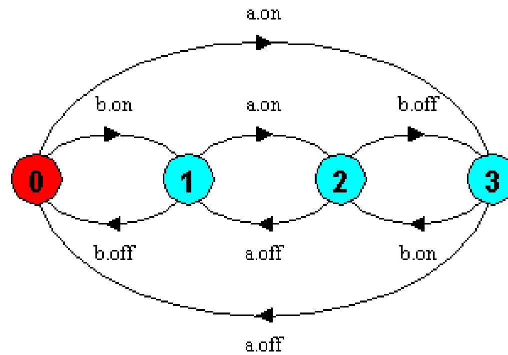


b:SWITCH



Process Labeling in FSP

LTS graph of two light switches



An array of light switches

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

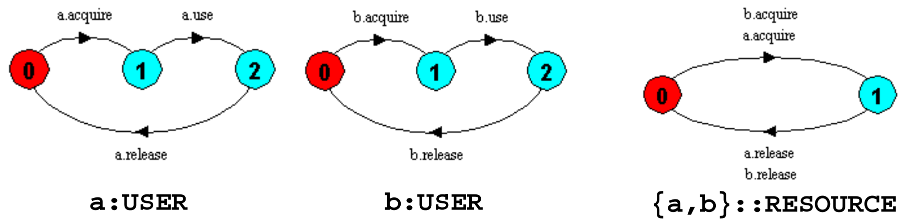
Labeling with a Set of Prefixes

`{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)`.

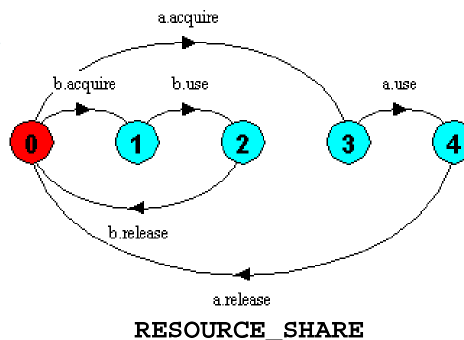
Labeling with a set of prefixes are useful for modeling ***shared*** resources in a program

```
RESOURCE = (acquire->release->RESOURCE).
USER = (acquire->use->release->USER).
||RESOURCE_SHARE = (a:USER || b:USER
                   || {a,b}::RESOURCE).
```

Labeling with a Set of Prefixes



How does the model ensure that the user that acquires the resource is the same one that releases it?



Relabeling Actions in FSP

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

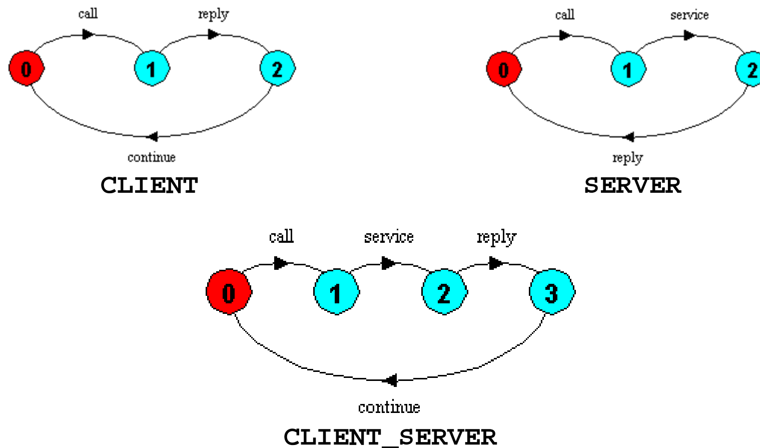
`/{newlabel1/oldlabel1,… newlabeln/oldlabeln}.`

Relabeling to ensure that composed processes synchronize on particular actions.

```
CLIENT = (call->wait->continue->CLIENT).
SERVER = (request->service->reply->SERVER).
||CLIENT_SERVER = (CLIENT || SERVER)
                  /{call/request, reply/wait}.
```

Relabeling Actions in FSP

```
||CLIENT_SERVER = (CLIENT || SERVER)
                  /{call/request, reply/wait}.
```



Action Hiding in FSP

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....

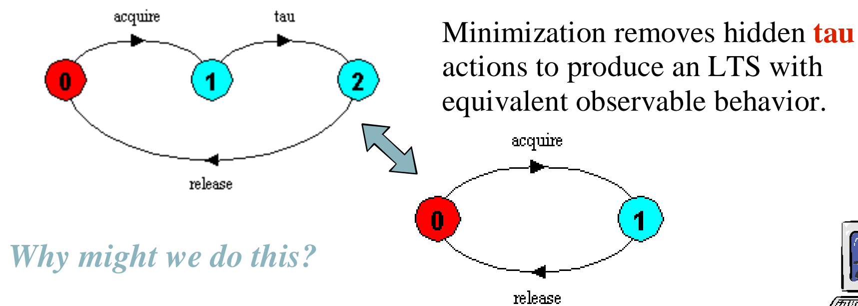
When applied to a process **P**, the interface operator **@{a1..ax}** hides all actions in the alphabet of **P** not labeled in the set **a1..ax**.

Action Hiding in FSP

These processes are equivalent

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

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



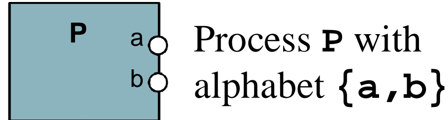
Minimization removes hidden **tau** actions to produce an LTS with equivalent observable behavior.

Why might we do this?

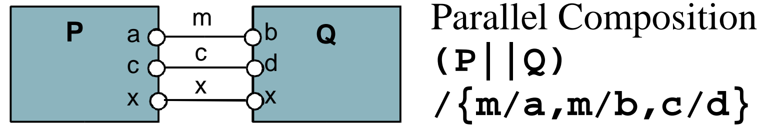
Structure Diagrams

- State machine diagrams to depict the dynamic behavior of processes
- Structure diagrams capture the structure of a model expressed by the static combination operators: ***parallel composition***, ***relabeling***, and ***hiding***

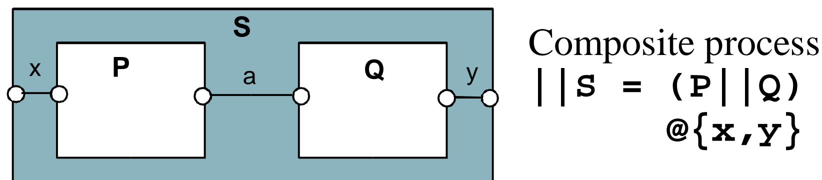
Structure Diagrams



Process **P** with alphabet **{a,b}**



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

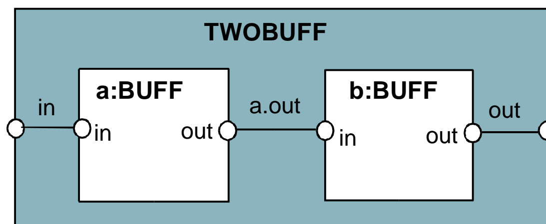


Composite process
||S = (P||Q)
@{x,y}

Structure Diagrams

```
range T = 0..3
BUFF = (in[i:T]->out[i]->BUFF).
||TWOBUFF = ?
```

It can be easier to understand relabeling with structure diagrams.

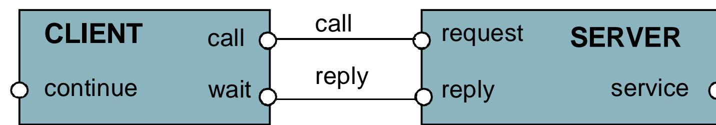



```
||TWOBUFF = (a:BUFF || b:BUFF)
            /{in/a.in,a.out/b.in,out/b.out}
            @{in,out}.
```

Structure Diagrams

```
CLIENT = (call->wait->continue->CLIENT).
SERVER = (request->service->reply->SERVER).
||CLIENT_SERVER = (CLIENT || SERVER).
                  /{call/request, reply/wait}.
```

Structure diagram for client/server example



Structure Diagrams

```
RESOURCE        = (acquire->release->RESOURCE).
USER            = (printer.acquire->use
                   ->printer.release->USER).
||PRINTER_SHARE = (a:USER||b:USER
                   ||{a,b}::printer:RESOURCE).
```

Shared resources are depicted as a rounded rectangle.