Concurrent Execution

INF2140 Parallel Programming: Lecture 3

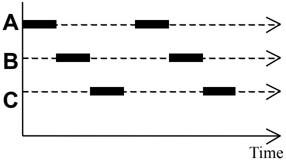
Feb. 01, 2012

Concurrent Execution

- Concepts
  - Processes - concurrent execution and interleaving
  - Process interaction
- Models
  - **Parallel composition** of asynchronous processes
    - Interleaving
  - **Interaction**
    - Shared actions
    - Process labeling
    - Action relabeling and hiding
  - **Structure diagrams**
- Practice
  - Multithreaded Java programs

Definitions

- Concurrency
  Logically simultaneous processing. Does not imply multiple processing elements (PEs). Requires **interleaved execution** on a single PE.



- Parallelism
  Physically simultaneous processing. Involves multiple PEs and/or independent device operations.

Both concurrency and parallelism require controlled access to shared resources . We use the terms parallel and concurrent interchangeably and generally do not distinguish between real and pseudo-concurrent execution.

Modeling Concurrency

- **How should we model process execution speed?**
  - arbitrary speed (we abstract away time)
- **How do we model concurrency?**
  - arbitrary relative order of actions from different processes
  - *interleaved execution* of processes, but *preserve* the order in each process
- **What is the result?**
  - a general model of execution, independent of scheduling (asynchronous model of execution)

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.

```
ITCH  = (scratch->STOP).
CONVERSE = (think->talk->STOP).
```

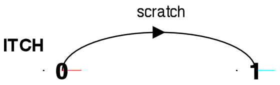
Disjoint alphabets!

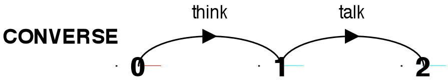
```
||CONVERSE_ITCH = (ITCH || CONVERSE).
```

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

Possible traces as a result of action interleaving.

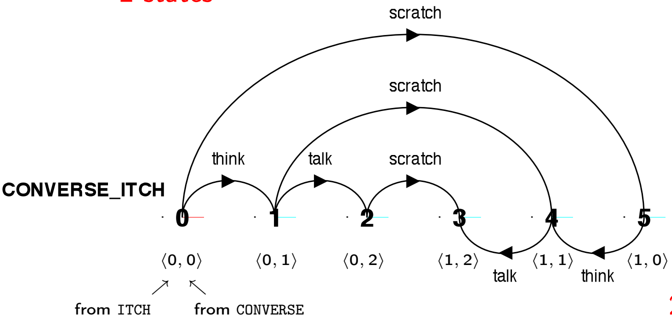
Parallel Composition - Action Interleaving





2 states

3 states



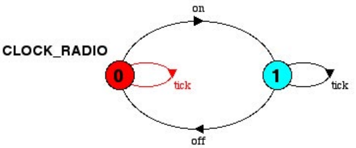
2×3 states

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).
```

Number of states? LTS? Traces?

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.

Modeling Interaction - Shared Actions

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

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

Non-disjoint alphabets!

MAKER synchronizes with USER when ready.

Traces? Number of states? LTS?

Modeling Interaction - Handshake

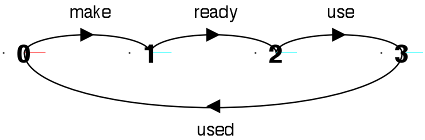
A handshake is an action acknowledged by another process

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

3 states
3 states

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

3×3 states?



Interaction constrains the overall behaviour.

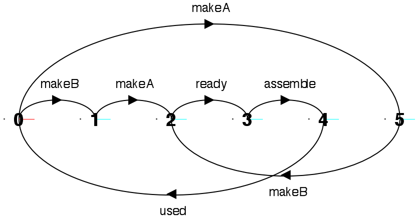
4 states

Modeling Interaction - Multiple Processes

Multi-party synchronization:

Many processes can participate in the shared action

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

Substituting the definition for MAKERS in FACTORY and applying the commutative and associative laws for parallel composition, we obtain the original definition for FACTORY:

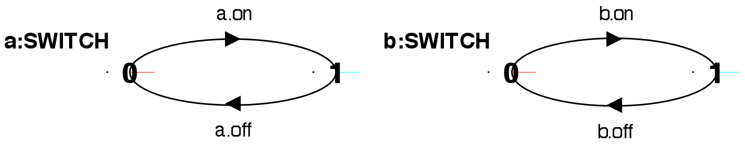
```
||FACTORY = (MAKE_A || MAKE_B || ASSEMBLE).
```

Process Instances and 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

Let P be a process and $\{a1, .., ax\}$ a set of labels.

Then $\{a1, .., ax\} :: P$ replaces

- every *action* with label n in the alphabet of P with the labels $a1.n, \ldots, ax.n$.
- every *transition* (n->X) in the definition of P with the transitions ($\{a1.n, \ldots, ax.n\}$->X).

Process prefixing is useful for modeling shared resources

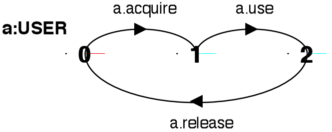
Process Labeling by a Set of Prefix Labels

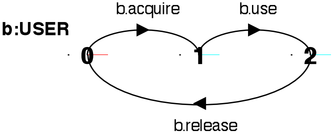
Example:

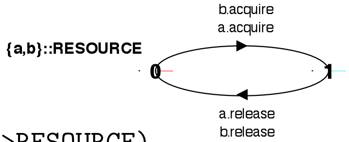
```
RESOURCE = (acquire->release->RESOURCE).

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

||RESOURCE_SHARE = (a:USER || b:USER
                    || {a,b}::RESOURCE).
```

Process Prefix Labels for Shared Resources

```
USER = (acquire->use->
         release->USER).

a:USER = (a.acquire->a.use->
           a.release->USER).

b:USER = (b.acquire->b.use->
           b.release->USER).

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

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

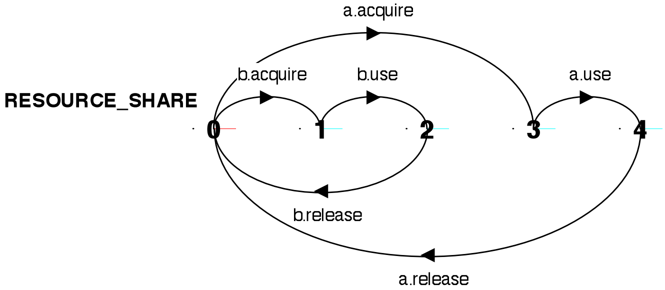






Process Prefix Labels for Shared Resources

How does the model ensure that the user that acquires the resource is the one to release it?



Action Relabeling

Relabeling functions are applied to processes to *change the names* of action labels.

The general form of the relabeling function is:

`P/{newlabel_1/oldlabel_1,... newlabel_n/oldlabel_n}.`

Relabeling is useful to ensure that composed processes *synchronize* on particular actions.

Note:
In `P/{newlabel/oldlabel}`, both `newlabel` and `oldlabel` can be *sets* of labels.

Action Relabeling

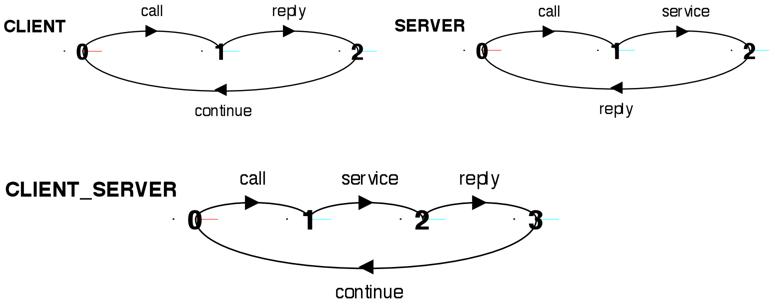
Example:

```
CLIENT = (call->wait->continue->CLIENT).

SERVER = (request->service->reply->SERVER).
```

We can use relabeling to make the processes synchronize

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.

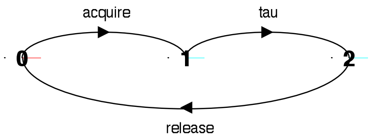
Sometimes it is more convenient to specify the set of labels to be *exposed*:

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`.

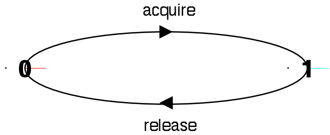
Action Hiding

These definitions are equivalent:

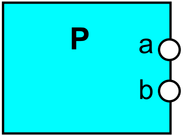
```
USER = (acquire->use->
          release->USER) \{use}.

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



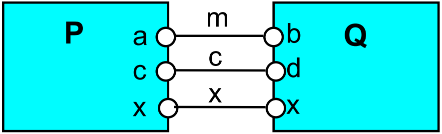
Minimization removes hidden tau actions to produce an LTS with equivalent observable behavior.



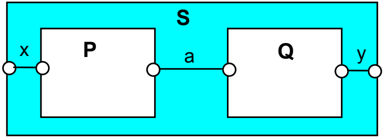
Structure Diagrams - Systems as Interacting Processes



Process P with alphabet `a,b`.



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



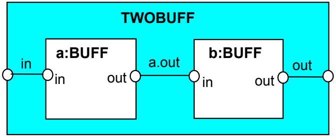
Composite process
`||S = (P||Q) @ {x,y}`

Structure Diagrams

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

Example

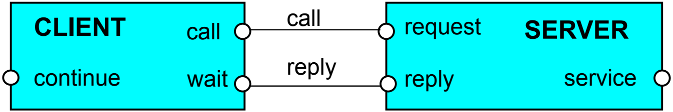
```
range T = 0..3

BUFF = (in[i:T]->
            out[i]->BUFF).

||TWOBUF = ?
```



Structure Diagrams

Structure diagram for CLIENT_SERVER ?

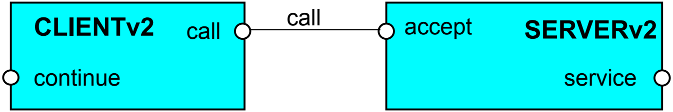
```
CLIENT = (call->wait->continue->CLIENT).

SERVER = (request->service->reply->SERVER).

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



Structure Diagrams

Structure diagram for CLIENT_SERVERv2 ?

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

CLIENTv2 = (call.request
            ->call.reply->continue->CLIENTv2).

||CLIENT_SERVERv2 = (CLIENTv2 || SERVERv2)/{call/accept}.
```



Structure Diagrams - Resource Sharing

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