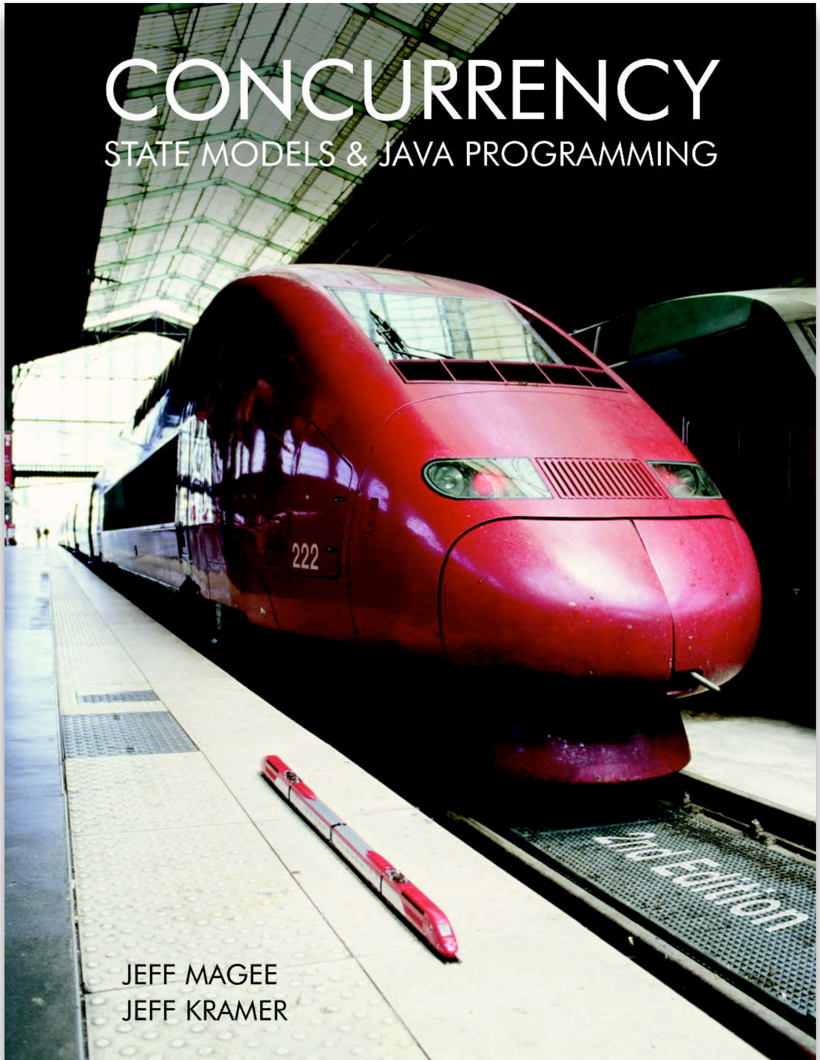
Exam Questions & Revision

University of Southern Denmark



The Main Aims Of The Course

Construct **models** from specifications of concurrency problems

The Main Aims Of The Course

Construct **models** from specifications of concurrency problems

Test, analyse, and compare **models' behaviour**

The Main Aims Of The Course

UNIVERSITY OF SOUTHERN DENMARK

Construct **models** from specifications of concurrency problems

Test, analyse, and compare **models' behaviour**

Define and verify models' **safety & liveness** properties

The Main Aims Of The Course

Construct **models** from specifications of concurrency problems

Test, analyse, and compare **models' behaviour**

Define and verify models' **safety & liveness** properties

Implement models in Java

The Main Aims Of The Course

University of Southern Denmark

Construct **models** from specifications of concurrency problems

Test, analyse, and compare **models' behaviour**

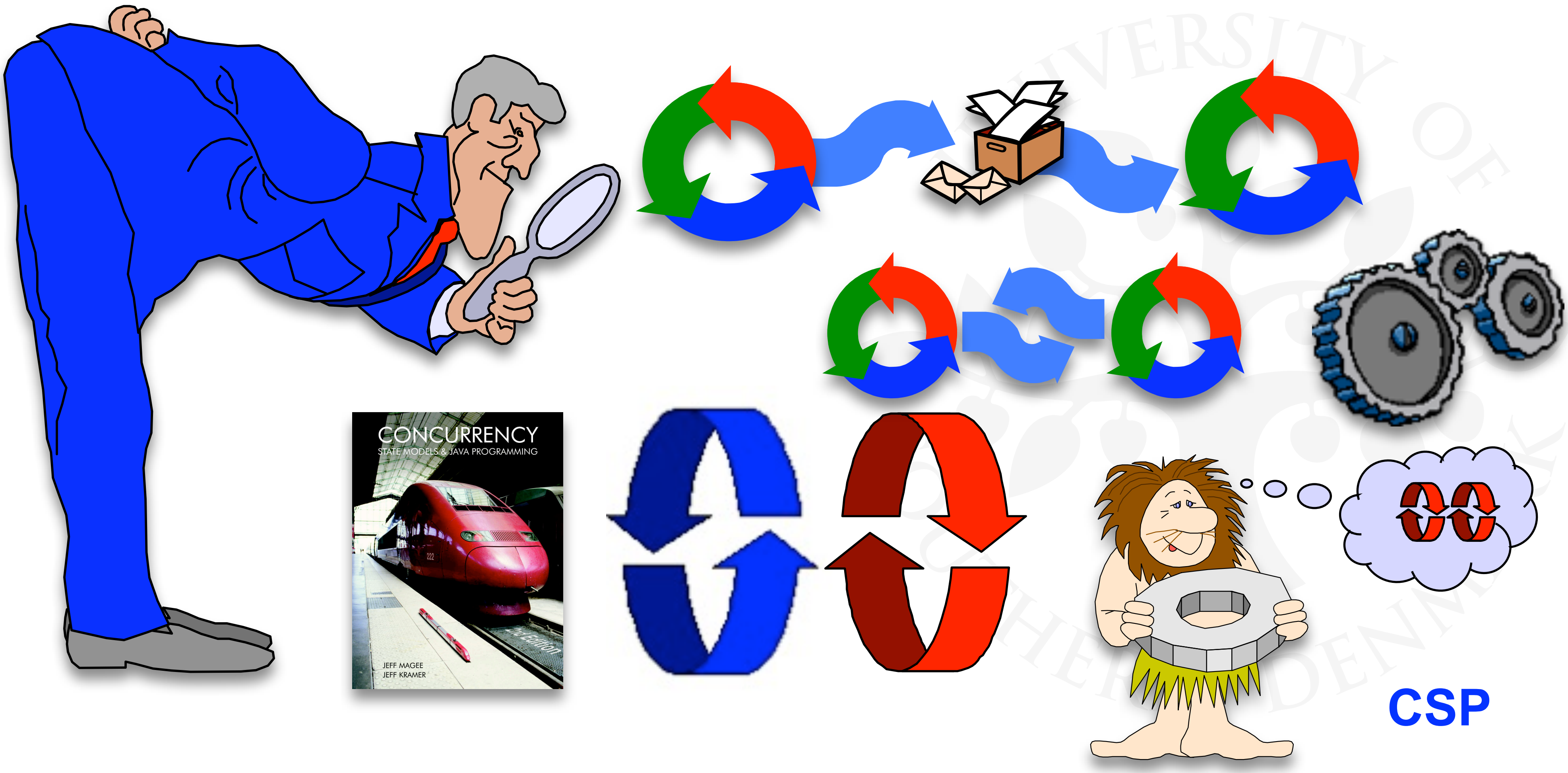
Define and verify models' **safety & liveness** properties

Implement models in Java

Relate models and implementations

Revision

The following is a **sample** of some of the covered topics



Outline Of Covered Chapters

2. Processes and Threads
3. Concurrent Execution
4. Shared Objects & Interference
5. Monitors & Condition Synchronisation
6. Deadlock
7. Safety and Liveness Properties
8. Model-based Design

The main basic

Concepts

Models

Practice

Advanced topics …

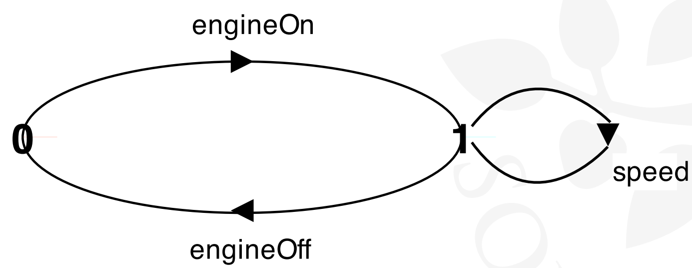
9. Dynamic systems
10. Message Passing

Models: FSP & LTS

Model = simplified representation of the real world

- Based on **Labelled Transition Systems** (LTS):

Focuses on concurrency aspects (of the program)
- everything else abstracted away



- Described textually as **Finite State Processes** (FSP):

```
EngineOff = (engineOn  -> EngineOn),
EngineOn  = (engineOff -> EngineOff
            |speed     -> EngineOn).
```

Finite State Processes (FSP)

University of Southern Denmark

FSPs can be defined using:

P =

- x -> Q // action
- Q // other process variable
- STOP // termination
- Q | R // choice
- when (...) x -> Q // guard
- ... + {write[0..3]} // alphabet extension
- X[i:0..N] =x[N-i] -> P // process & action index
- BUFF(N=3) // process parameter

const N = 3 // constant definitions
range R = 0..N // range definitions
set S = {a,b,c} // set definitions

Finite State Processes (FSP)

FSPs can be defined using:

P =

– x -> Q	// action
– Q	// other process variable
– STOP	// termination
– Q | R	// choice
– when (...) x -> Q	// guard
– ... + {write[0..3]}	// alphabet extension
– X[i:0..N] =x[N-i] -> P	// process & action index
– BUFF(N=3)	// process parameter

```
const N = 3         // constant definitions
range R = 0..N      // range definitions
set S = {a,b,c}     // set definitions
```

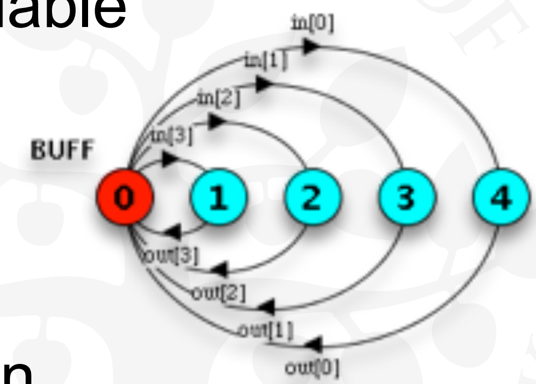
```
range T = 0..3
BUFF = (in[i:T]->out[i]->BUFF).
```

Finite State Processes (FSP)

FSPs can be defined using:

P =

- x -> Q // action
- Q // other process variable
- STOP // termination
- Q | R // choice
- when (...) x -> Q // guard
- ... + {write[0..3]} // alphabet extension
- X[i:0..N] =x[N-i] -> P // process & action index
- BUFF(N=3) // process parameter




```
const N = 3        // constant definitions
range R = 0..N     // range definitions
set S = {a,b,c}    // set definitions
```

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

Finite State Processes (FSP)

Finite State Processes (FSP)

FSP:

- P || Q // parallel composition
- a:P // process labelling (1 process/prefix)
- {…}::P // process sharing (1 process w/all prefixes)
- P / {x/y} // action relabelling
- P \ {…} // hiding
- P @ {…} // keeping (hide complement)

Finite State Processes (FSP)

FSP:

- P || Q // parallel composition
- a:P // process labelling (1 process/prefix)
- {…}::P // process sharing (1 process w/all prefixes)
- P / {x/y} // action relabelling
- P \ {…} // hiding
- P @ {…} // keeping (hide complement)

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

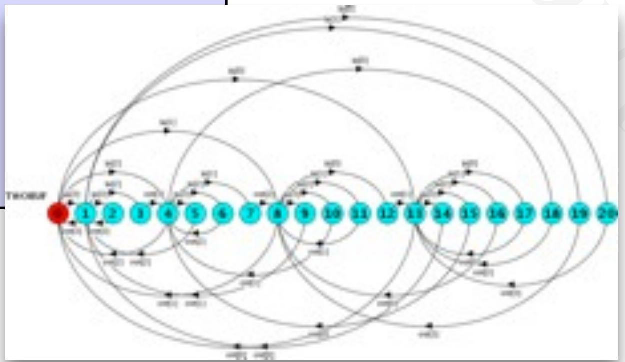
Finite State Processes (FSP)

UNIVERSITY OF SOUTHERN DENMARK

FSP:

- P || Q   // parallel composition
- a:P   // process labelling (1 process/prefix)
- {…}::P   // process sharing (1 process w/all prefixes)
- P / {x/y}   // action relabelling
- P \ {…}   // hiding
- P @ {…}   // keeping (hide complement)

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

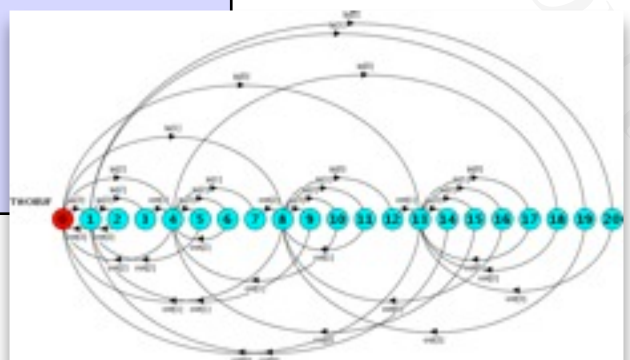


Finite State Processes (FSP)

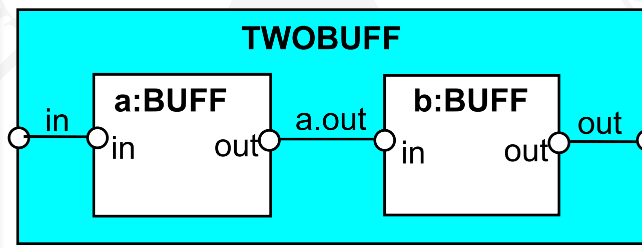
FSP:

- P || Q — // parallel composition
- a:P — // process labelling (1 process/prefix)
- {…}::P — // process sharing (1 process w/all prefixes)
- P / {x/y} — // action relabelling
- P \ {…} — // hiding
- P @ {…} — // keeping (hide complement)

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



Structure Diagrams:



Structure Diagrams - Resource Sharing


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

Structure Diagrams - Resource Sharing

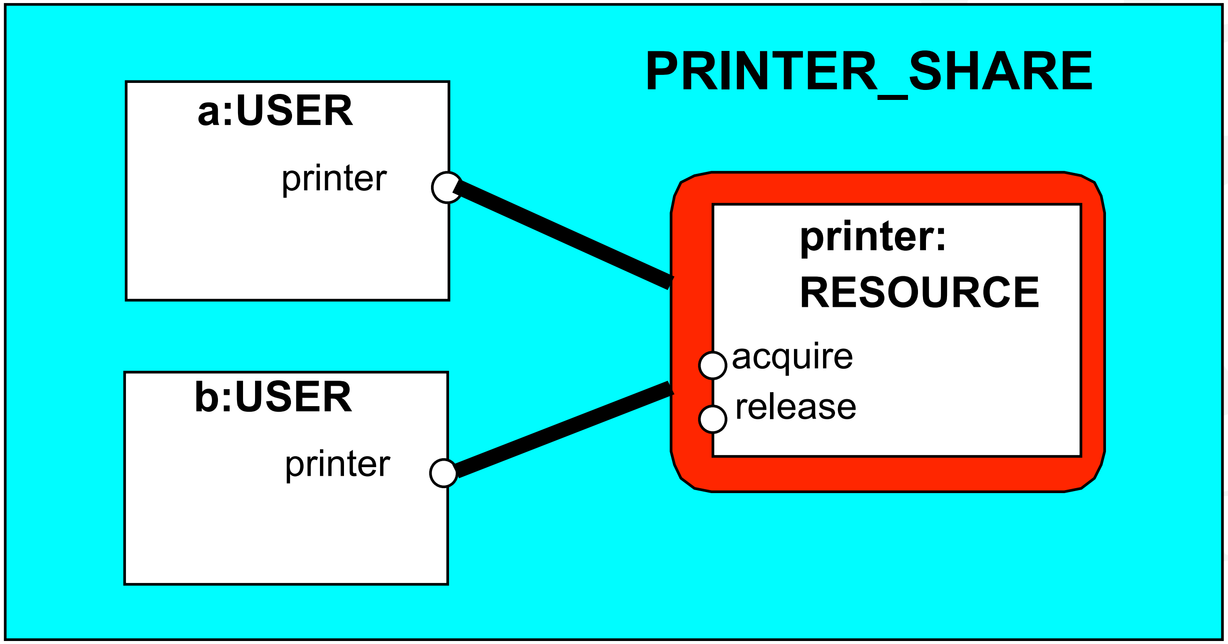
```
RESOURCE = (acquire->release->RESOURCE).
USER     = (printer.acquire->use->printer.release->USER).
```

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

Structure Diagrams - Resource Sharing

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

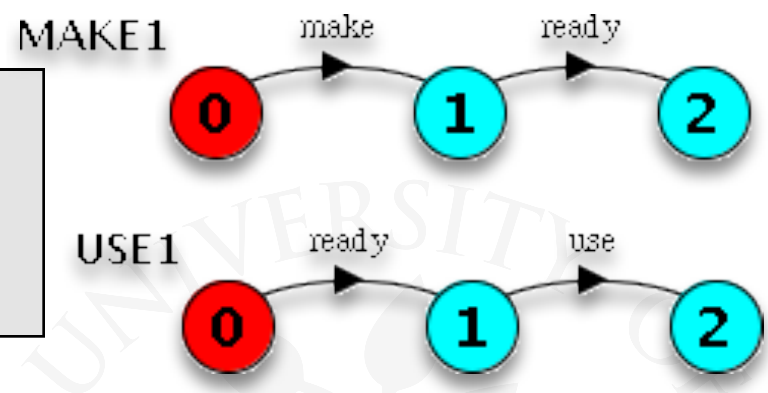
```
||PRINTER_SHARE =
    (a:USER || b:USER || {a,b}::printer:RESOURCE).
```



How To Create The Parallel Composed LTS

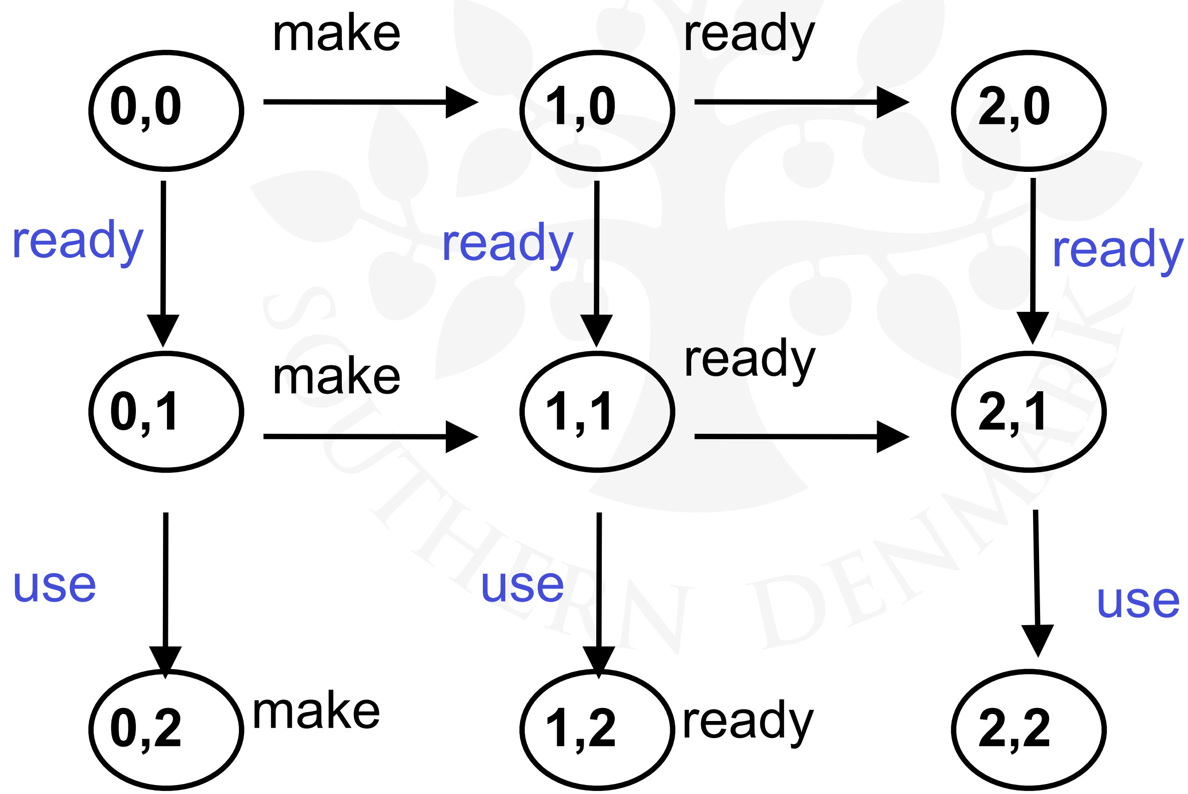

```
MAKE1 = (make->ready->STOP).
USE1  = (ready->use->STOP).

||MAKE1_USE1 = (MAKE1 || USE1).
```



For any state reachable from the initial state (0,0), consider the possible actions and draw edges to the corresponding new states (i,j).

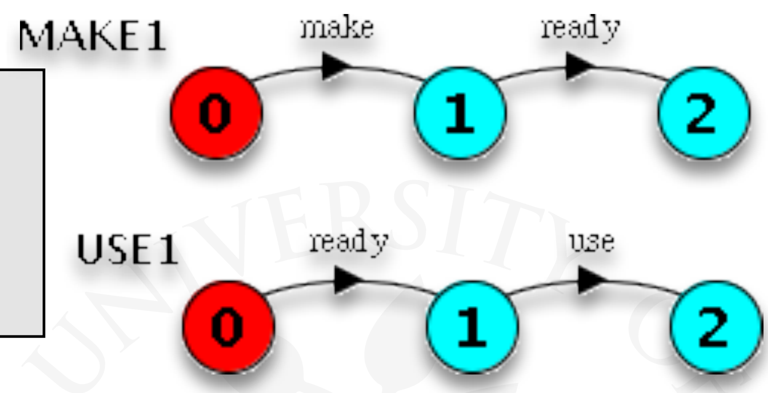
Remember to consider **shared** actions.



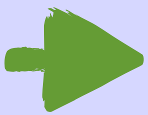
How To Create The Parallel Composed LTS


```
MAKE1 = (make->ready->STOP).
USE1  = (ready->use->STOP).

||MAKE1_USE1 = (MAKE1 || USE1).
```

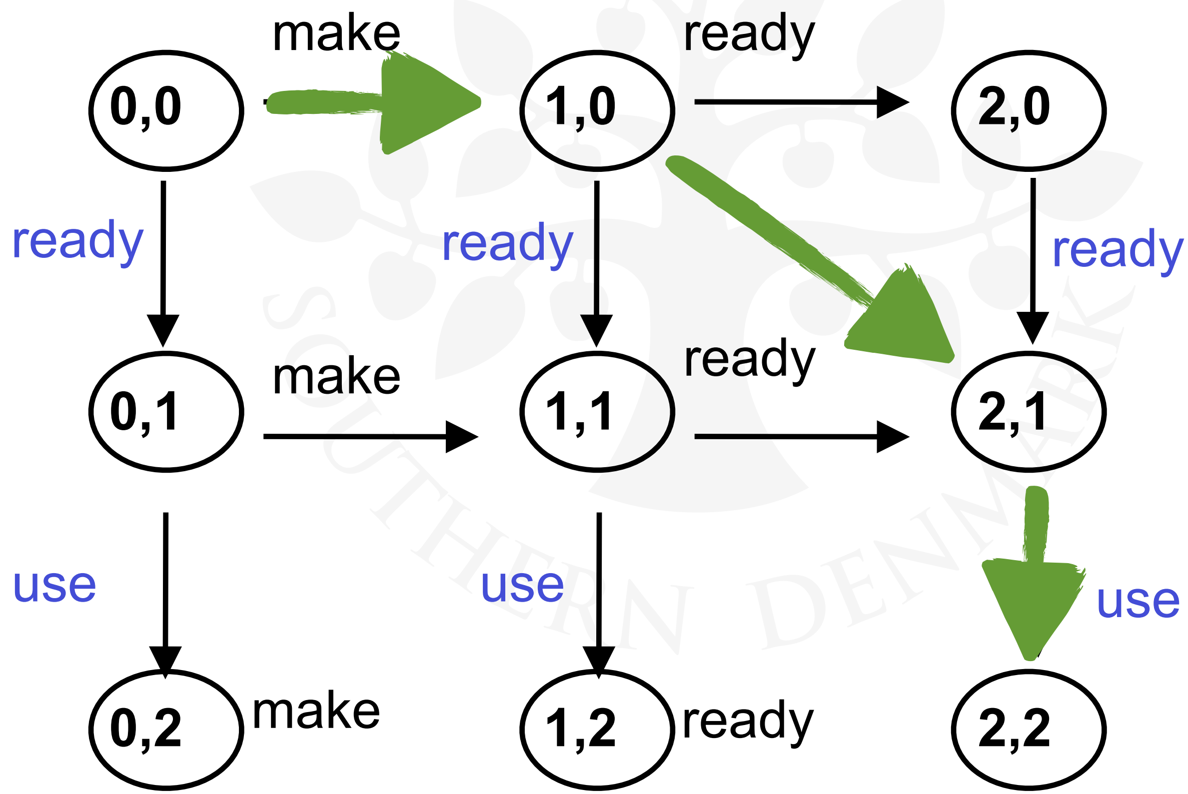


For any state reachable from the initial state (0,0), consider the possible actions and draw edges



to the corresponding new states (i,j).

Remember to consider **shared** actions.



Chapter 4: Shared Objects & Mutual Exclusion

- **Concepts:**
  - Process interference
  - Mutual exclusion
- **Models:**
  - Model-checking for interference
  - Modelling mutual exclusion
- **Practice:**
  - Thread interference in shared objects in Java
  - Mutual exclusion in Java
  - Synchronised objects, methods, and statements

Condition Synchronisation In FSP And Java

```
FSP:    when (cond) action -> NEWSTATE
```

```
synchronized void action() throws Int'Exc' {
    while (!cond) wait();
    // modify monitor data
    notifyAll();
}
```

The **while** loop is necessary to re-test the condition cond to ensure that cond is indeed satisfied when it re-enters the monitor.

notifyAll() is necessary to awaken other thread(s) that may be waiting to enter the monitor now that the monitor data has been changed.

Condition Synchronisation (in Java)


```
CONTROL(CAPACITY=4) = SPACES[CAPACITY],
SPACES[spaces:0..CAPACITY] =
            (when(spaces>0)          arrive -> SPACES[spaces-1]
            |when(spaces<CAPACITY) depart -> SPACES[spaces+1]).
```


```
class CarParkControl {
    protected int spaces, capacity;

    synchronized void arrive()
                  throws Int'Exc' {
        while (!(spaces>0)) wait();
        --spaces;
        notifyAll();
    }

    synchronized void depart()
                  throws Int'Exc' {
        while (!(spaces<capacity)) wait();
        ++spaces;
        notifyAll();
}   }
```

Condition Synchronisation (in Java)

```
CONTROL(CAPACITY=4) = SPACES[CAPACITY],
SPACES[spaces:0..CAPACITY] =
            (when(spaces>0)          arrive -> SPACES[spaces-1]
            |when(spaces<CAPACITY)   depart -> SPACES[spaces+1]).
```

```
class CarParkControl {
    protected int spaces, capacity;

    synchronized void arrive()
                  throws Int'Exc' {
        while (!(spaces>0)) wait();
        --spaces;
        notifyAll();
    }

    synchronized void depart()
                  throws Int'Exc' {
        while (!(spaces<capacity)) wait();
        ++spaces;
        notifyAll();
}   }
```

notify() instead of notifyAll() ?
1. Uniform waiters - everybody waits on the same condition
2. One-in, one-out

What goes wrong with notify and 8xDepartures, 5xArrivals?

Semaphores

Semaphores are widely used for dealing with inter-process synchronisation in operating systems.

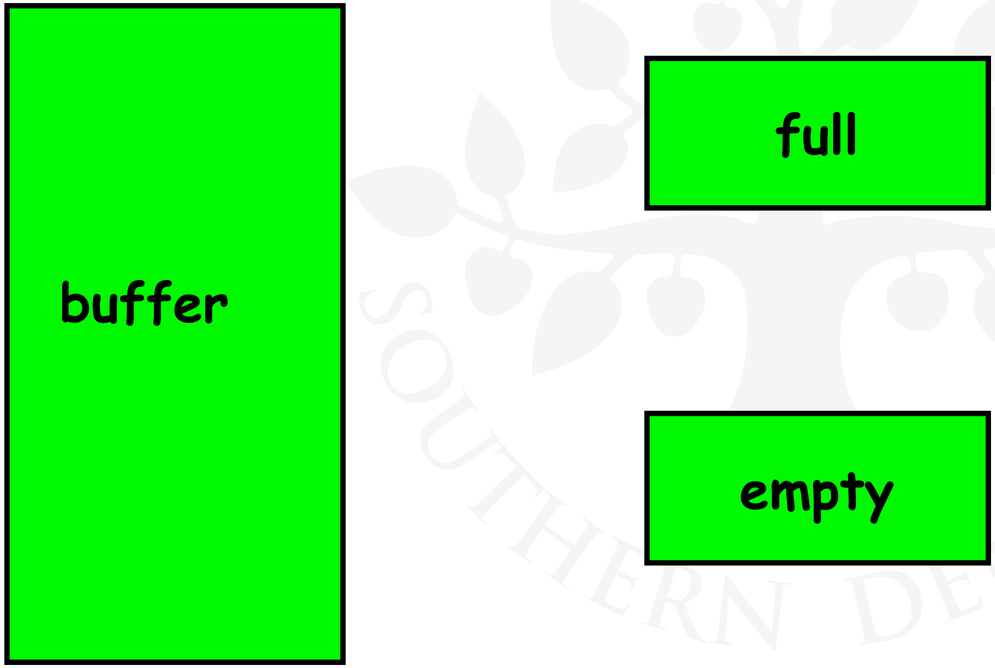
Semaphore s : integer var that can take only non-neg. values.

```
sem.down(); // decrement (block if counter = 0)
```

```
sem.up(); // increment counter (allowing one blocked thread to pass)
```

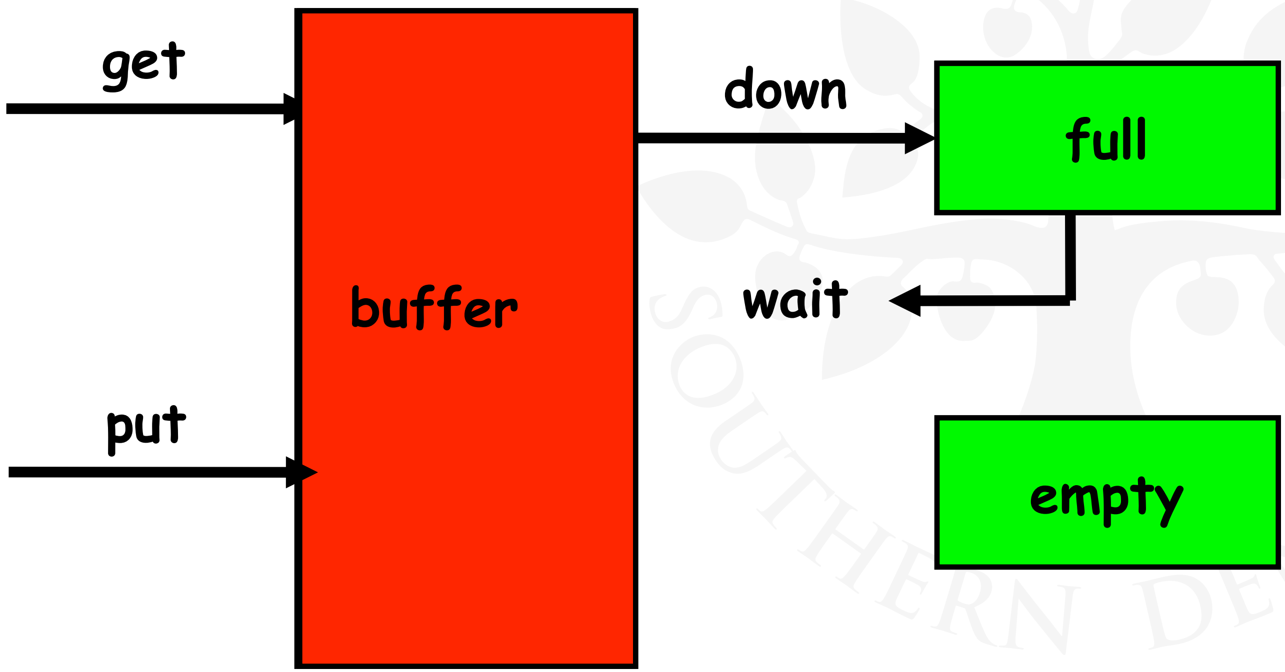
Nested Monitor Problem

```
synchronized public E get()
                throws InterruptedException{
    full.down(); // if no items, block!
    ...
  }
```



Nested Monitor Problem

```
synchronized public E get()
                   throws InterruptedException{
    full.down(); // if no items, block!
    ...
  }
```



Deadlock: 4 Necessary And Sufficient Conditions

1. Mutual exclusion condition (aka. "Serially reusable resources"):

the processes involved share resources which they use under mutual exclusion.

2. Hold-and-wait condition (aka. "Incremental acquisition"):

processes hold on to resources already allocated to them while waiting to acquire additional resources.

3. No preemption condition:

once acquired by a process, resources cannot be "pre-empted" (forcibly withdrawn) but are only released voluntarily.

4. Circular-wait condition (aka. "Wait-for cycle"):

a circular chain (or cycle) of processes exists such that each process holds a resource which its successor in the cycle is waiting to acquire.

Deadlock: 4 Necessary And Sufficient Conditions

1. Mutual exclusion condition (aka. “Serially reusable resources”):

the processes involved share resources which they use under mutual exclusion.

2. Hold-and-wait condition (aka. “Incremental acquisition”):

processes hold on to resources already allocated to them while waiting to acquire additional resources.

3. No preemption condition:

once acquired by a process, resources cannot be “pre-empted” (forcibly withdrawn) but are only released voluntarily.

4. Circular-wait condition (aka. “Wait-for cycle”):

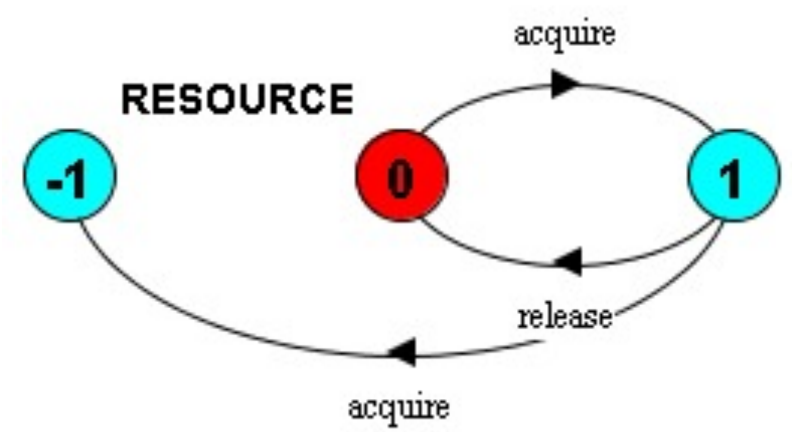
a circular chain (or cycle) of processes exists such that each process holds a resource which its successor in the cycle is waiting to acquire.

Deadlock avoidance:
“Break at least one of the deadlock conditions”.

7.1 Safety Properties

A **safety property** asserts that nothing bad happens.

- STOP or deadlocked state (no outgoing transitions)
- ERROR process (-1) to detect erroneous behaviour




```
RESOURCE =(acquire -> ACQUIRED),
ACQUIRED =(release -> RESOURCE
          |acquire -> ERROR).
```

- Analysis using LTSA:
  (shortest trace)

```
Trace to property violation in RESOURCE:
        acquire
        acquire
```

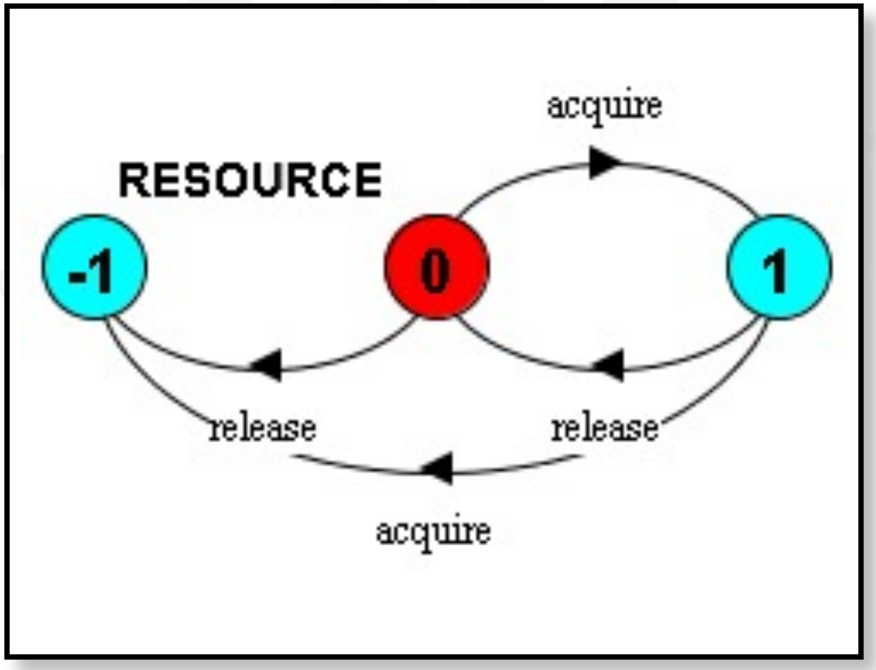
Safety - Property Specification

- ERROR conditions state what is **not** required (~ *exceptions*).
- In complex systems, it is usually better to specify **safety properties** by stating directly what **is** required.

```
property SAFE_RESOURCE =
    (acquire ->
        release ->
            SAFE_RESOURCE).
```

is equivalent to

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



7.3 Liveness Properties

A safety property **asserts** that nothing bad happens.

A liveness property **asserts** that something good **eventually** happens.

E.g., does every car eventually get an opportunity to cross the bridge, i.e., make **progress**?

A progress property **asserts** that it is always the case that an action is eventually executed.

Progress is the opposite of starvation (= the name given to a concurrent programming situation in which an action is never executed).

Progress Properties

UNIVERSITY OF SOUTHERN DENMARK

```
progress P = {a1, a2, ..., an}
```

This defines a **progress property**, P, which **asserts** that in an infinite execution, at least one of the actions $a_1, a_2, \ldots, a_n$ will be executed infinitely often.

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

progress HEADS = {heads} ? ☺

progress TAILS = {tails} ? ☺

LTSA check progress:

```
No progress violations detected
```

Dynamic Systems

Concepts: **dynamic** creation and deletion of **processes**

Resource allocation example – varying number of users and resources.

master-slave interaction

Models: **static - fixed populations with cyclic behavior**

interaction

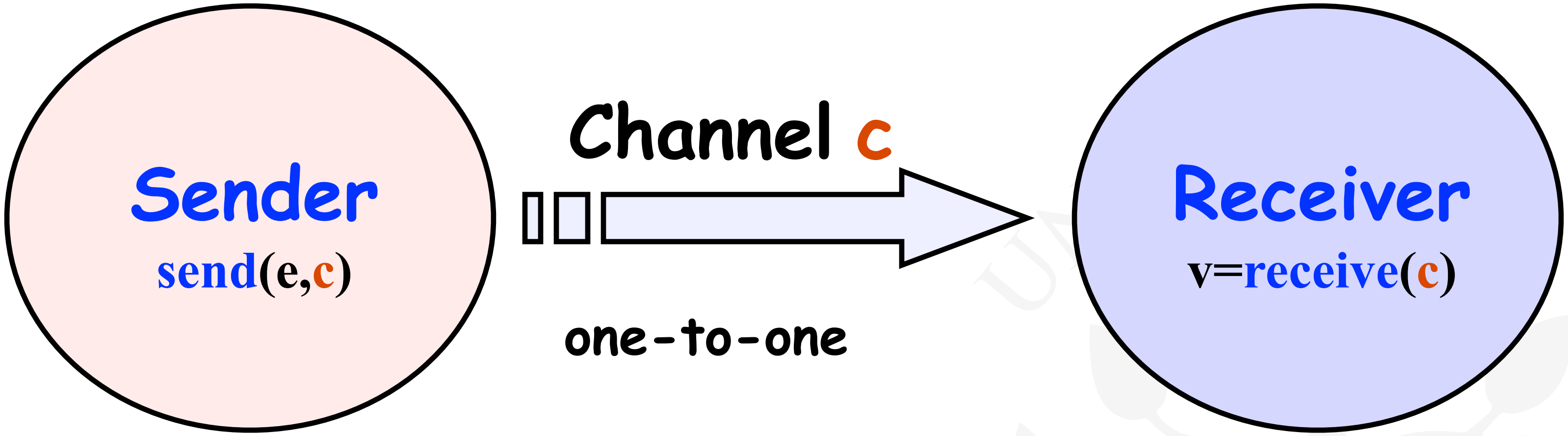
Practice: **dynamic** creation and deletion of **threads**

(# active threads varies during execution)

Resource allocation algorithms

Java join() method

10.1 Synchronous Message Passing - Channel



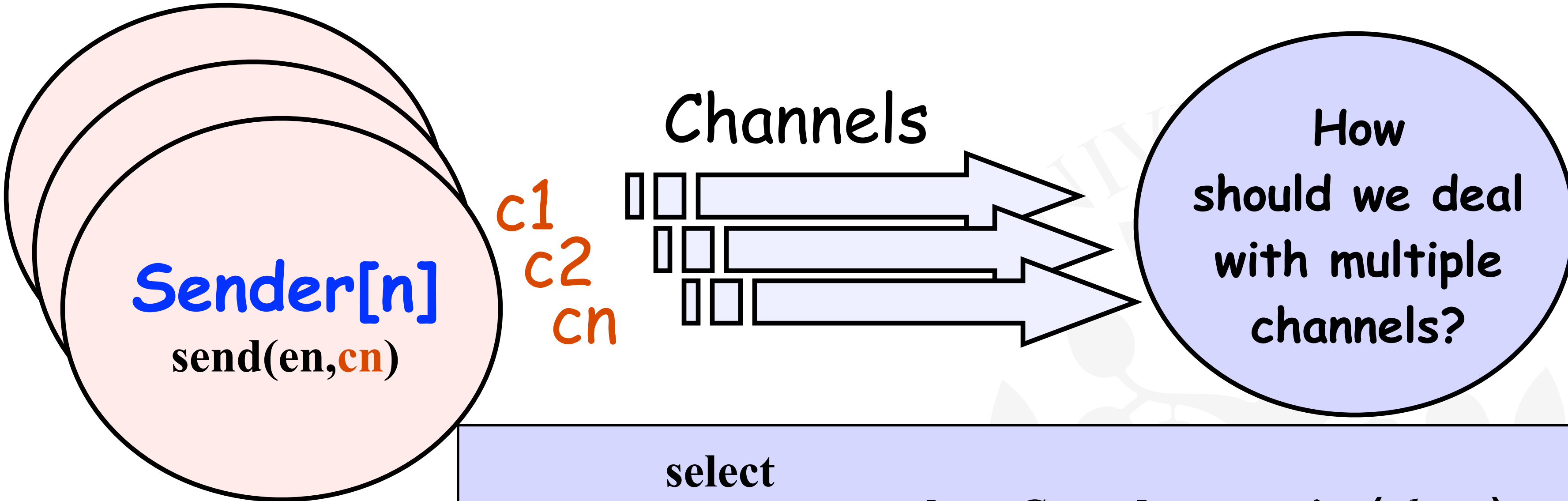
- **send(e,c)** - send e to channel c. The sender is **blocked** until the message is received from the channel.

- **v = receive(c)** - receive a value into local variable v from channel c. The calling process is **blocked** until a message is sent to the channel.

Channel has no buffering

Corresponds to "v = e"

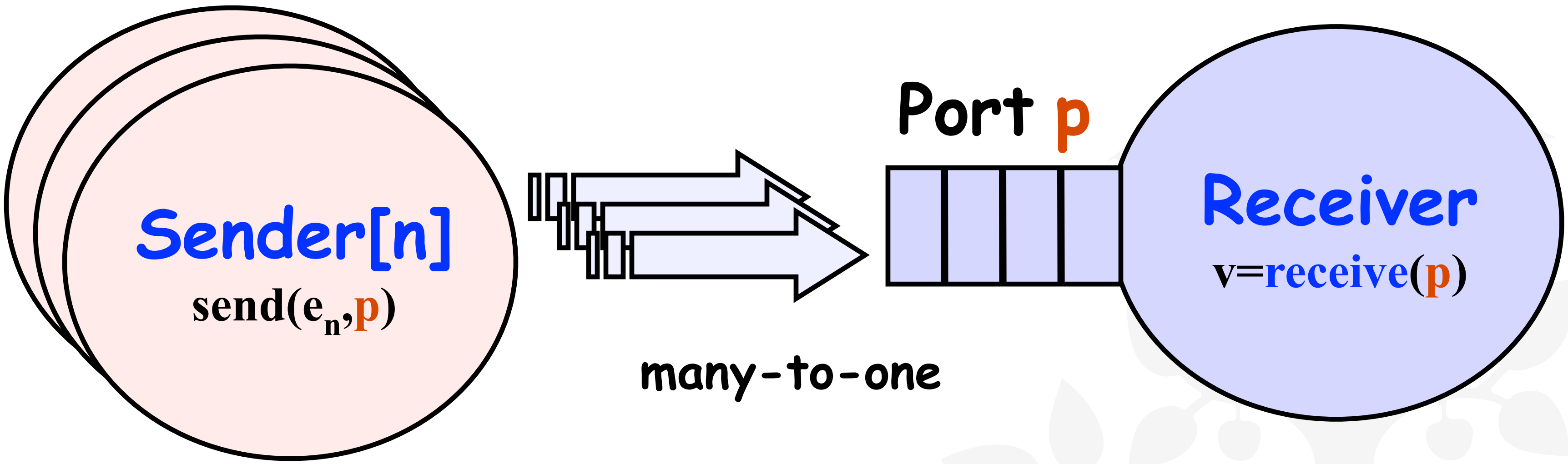
Selective Receive



Select statement...

select
  when G_1 **and** v_1**=receive(***$chan_1$***) =>** S_1;
or
  when G_2 **and** v_2**=receive(***$chan_2$***) =>** S_2;
or
  …
or
  when G_n **and** v_n**=receive(***$chan_n$***) =>** S_n;
end

10.2 Asynchronous Message Passing - Port

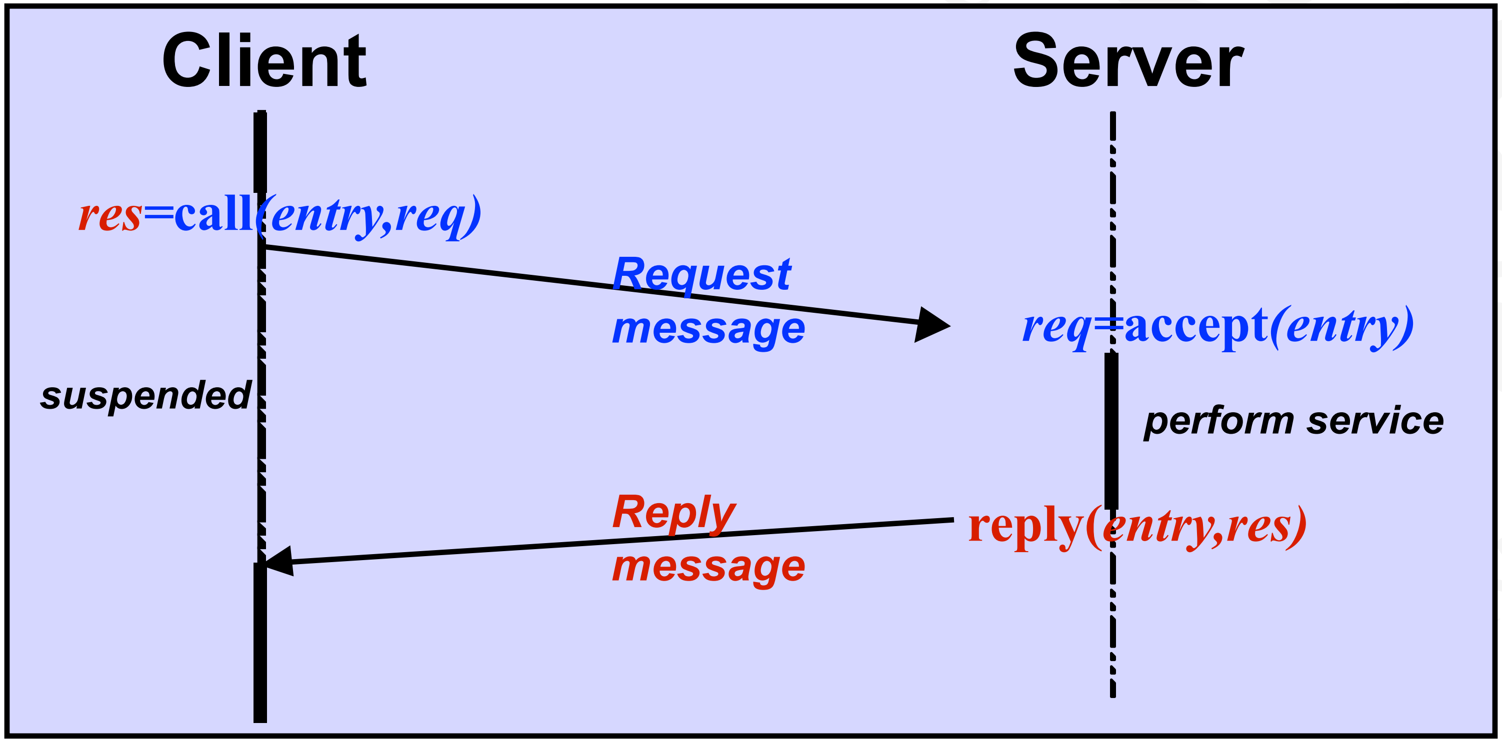


- **send(e,p)** - send e to port p. The calling process is **not blocked**. The message is queued at the port if the receiver is not waiting.

- *v* = **receive(*p*)** - receive a value into local variable v from port p. The calling process is **blocked** if no messages queued to the port.

10.3 Rendezvous - Entry

Rendezvous is a form of request-reply to support client server communication. Many clients may request service, but only one is serviced at a time.



The Main Aims Of The Course (Repetition)

UNIVERSITY OF SOUTHERN DENMARK

Construct **models** from specifications of concurrency problems

Test, analyse, and compare **models' behaviour**

Define and verify models' **safety & liveness** properties

Implement models in Java

Relate models and implementations