The Calculus of Context-aware Ambients (CCA)

How to program: Capabilities and Processes (2)

François Siewe

Software Technology Research Laboratory (STRL)
Cyber Technology Institute (CTI)
De Montfort University
Leicester, UK
fsiewe@dmu.ac.uk or fsiewe@yahoo.fr
http://www.cse.dmu.ac.uk/∼fsiewe/

November 2022

DE MONTFORT UNIVERSITY LEICESTER

Learning Outcomes

By the end of this lesson you should know:

- Communication statements.
- Process abstractions.
- How to model and simulate IoT devices like sensors and actuators.
- How to model a micro-controller that controls IoT devices.

Communication Primitives (`send/recv`)

- Message passing communication: one process (**the sender**) sends a message n and another process (**the receiver**) receives it in a variable x.

$$\begin{array}{lcl} \textbf{Sender} & : & \texttt{send}(n).P \\ \textbf{Receiver} & : & \texttt{recv}(x).Q \end{array}$$

- The **reduction rule** for message passing is

$$\texttt{send}(n).P \mid \texttt{recv}(x).Q \quad \longrightarrow \quad P \mid Q\{x \leftarrow n\}$$

  where $Q\{x \leftarrow n\}$ denotes the substitution of n for each free occurrence of x in Q.

- Example
  - $\texttt{send}(n).\mathbf{0} \mid \texttt{recv}(x).x[\mathbf{0}] \quad \longrightarrow \quad n[\mathbf{0}]$
  - $\texttt{send}(5).\mathbf{0} \mid \texttt{recv}(x).\texttt{send}(x).\mathbf{0} \quad \longrightarrow \quad \texttt{send}(5).\mathbf{0}$

DE MONTFORT UNIVERSITY LEICESTER

Communication Primitives (`send/recv`)

- Can model both **synchronous** and **asynchronous** communications
- **Synchronous** communication: both the sender and the receiver wait for each other

$$\begin{array}{lcl} \textbf{Sender} & : & \texttt{send}(2).P \\ \textbf{Receiver} & : & \texttt{recv}(x).Q \end{array}$$

  P and Q cannot be executed until the communication has taken placed.

- **Asynchronous** communication: the sender does not wait for the communication to take place.

$$\begin{array}{lcl} \textbf{Sender} & : & \texttt{send}(2).\mathbf{0} \mid P \\ \textbf{Receiver} & : & \texttt{recv}(x).Q \end{array}$$

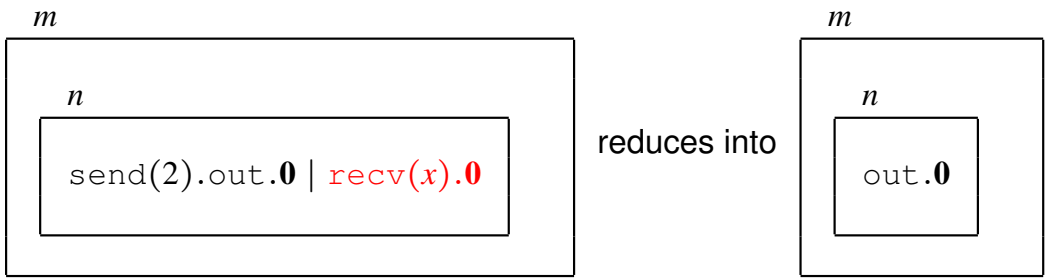
  Q cannot be executed until the communication takes place; but P can be executed before the communication takes place.

DE MONTFORT UNIVERSITY LEICESTER

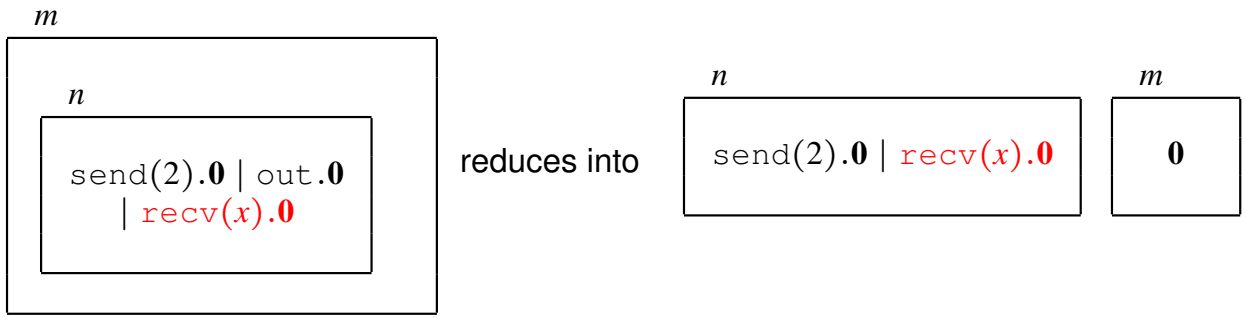
Communication Primitives (`send/recv`)

Examples: the receiver is in red.

- **Synchronous** communication



- **Asynchronous** communication



Communication Primitives (`send/recv`)

Capabilities to send a message:

- $\texttt{send}(a)$: capability to send a message a to self
- $n@\texttt{send}(a)$: capability to send a message a to the parent ambient n
- $@\texttt{send}(a)$: capability to send a message a to any parent ambient
- $n\#\texttt{send}(a)$: capability to send a message a to the child ambient n
- $\#\texttt{send}(a)$: capability to send a message a to any child ambient
- $n :: \texttt{send}(a)$: capability to send a message a to the sibling ambient n
- $:: \texttt{send}(a)$: capability to send a message a to any sibling ambient

DE MONTFORT UNIVERSITY LEICESTER

Communication Primitives (`send/recv`)

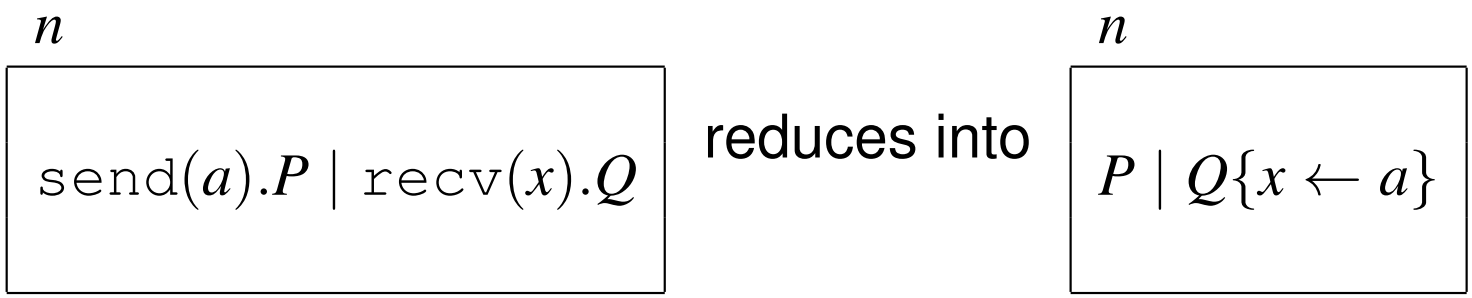
Capabilities to receive a message:

- `recv`(x): capability to receive a message a from self
- n@`recv`(x): capability to receive in the variable x a message from the parent ambient n
- @`recv`(x): capability to receive in the variable x a message from any parent ambient
- n#`recv`(x): capability to receive in the variable x a message from the child ambient n
- #`recv`(x): capability to receive in the variable x a message from any child ambient
- n :: `recv`(x): capability to send receive in the variable x a message from the sibling ambient n
- :: `recv`(x): capability to receive in the variable x a message from any sibling ambient

DE MONTFORT UNIVERSITY LEICESTER

Communication Primitives (`send/recv`)

- Local communication:



- The corresponding **reduction rule**

$$n[\texttt{send}(a).P \mid \texttt{recv}(x).Q] \longrightarrow n[P \mid Q\{x \leftarrow a\}]$$

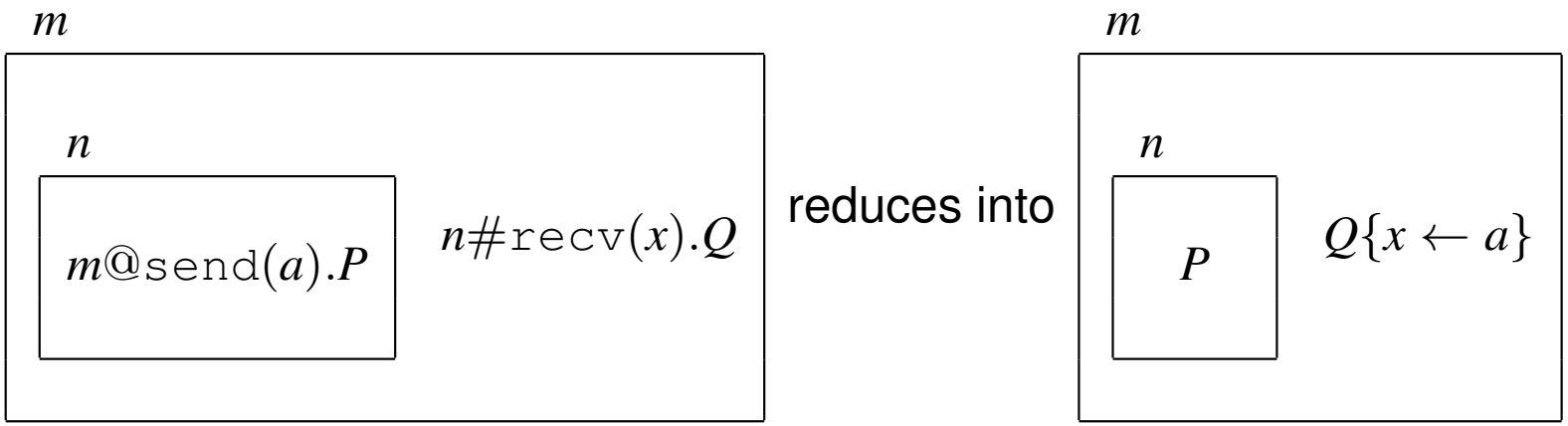
- Example

$$n[\texttt{send}(5).\mathbf{0} \mid \texttt{recv}(x).\texttt{send}(x).\mathbf{0}] \longrightarrow n[\texttt{send}(5).\mathbf{0}]$$

DE MONTFORT UNIVERSITY LEICESTER

Communication Primitives (`send/recv`)

- Child-parent communication: specific child and specific parent

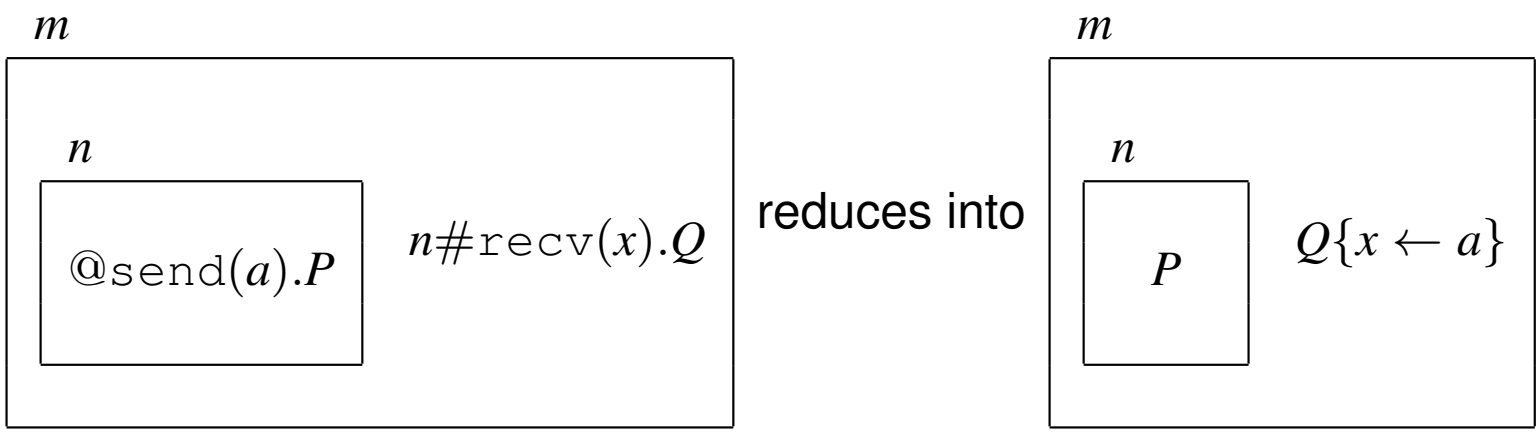


- The corresponding **reduction rule**

$$m[n[m@\texttt{send}(a).P] \mid n\#\texttt{recv}(x).Q] \longrightarrow m[n[P] \mid Q\{x \leftarrow a\}]$$

Communication Primitives (`send/recv`)

- Child-parent communication: specific child and any parent

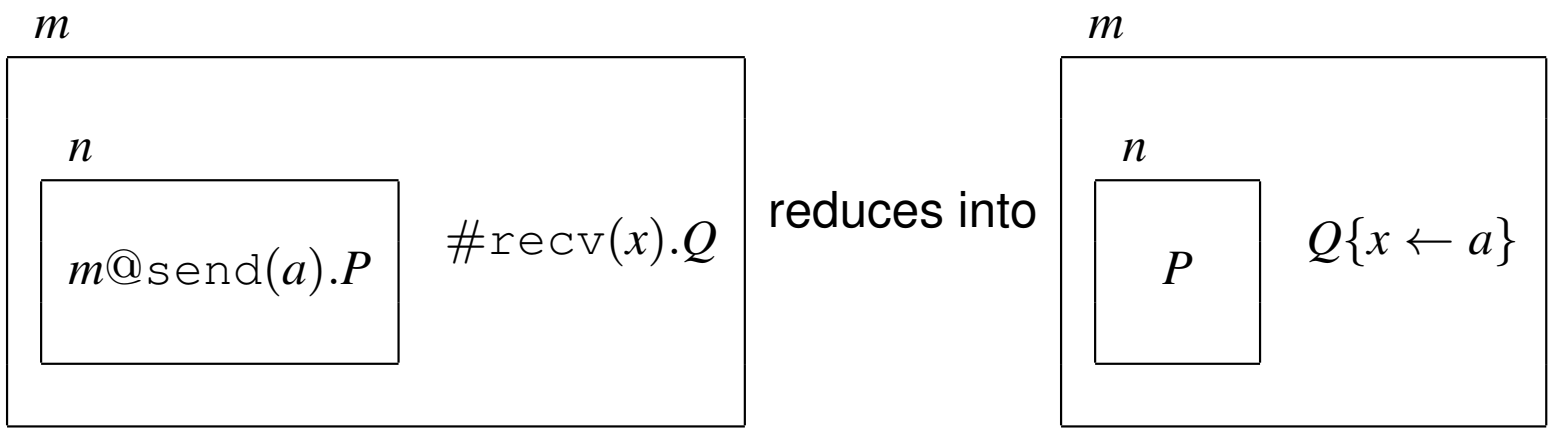


- The corresponding **reduction rule**

$$m[n[@\texttt{send}(a).P] \mid n\#\texttt{recv}(x).Q] \longrightarrow m[n[P] \mid Q\{x \leftarrow a\}]$$

Communication Primitives (`send/recv`)

- Child-parent communication: any child and specific parent

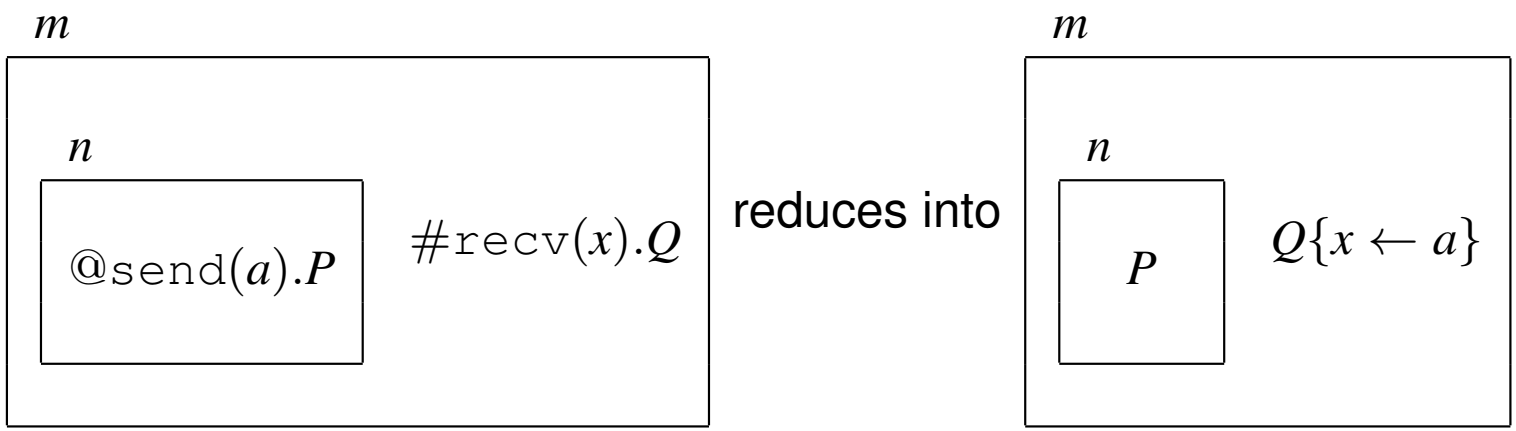


- The corresponding **reduction rule**

$$m[n[m@\texttt{send}(a).P] \mid \#\texttt{recv}(x).Q] \longrightarrow m[n[P] \mid Q\{x \leftarrow a\}]$$

Communication Primitives (`send/recv`)

- Child-parent communication: any child and any parent

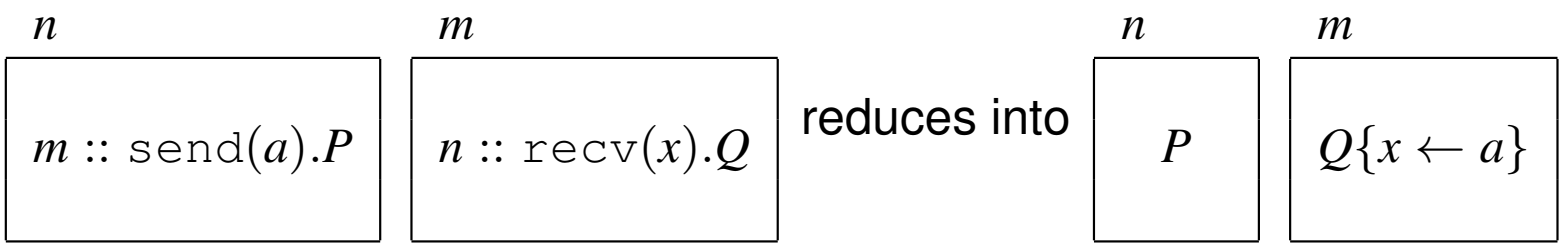


- The corresponding **reduction rule**

$$m[n[@\texttt{send}(a).P] \mid \#\texttt{recv}(x).Q] \longrightarrow m[n[P] \mid Q\{x \leftarrow a\}]$$

Communication Primitives (`send/recv`)

- Sibling-sibling communication: specific siblings

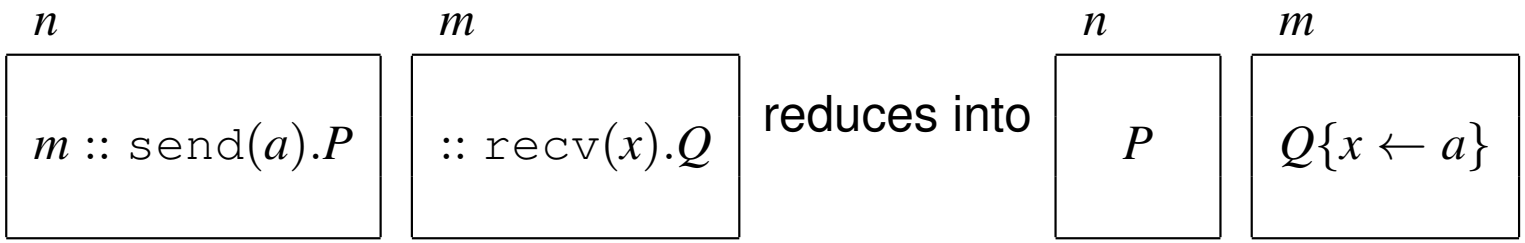


- The corresponding **reduction rule**

$$n[m :: \texttt{send}(a).P] \mid m[n :: \texttt{recv}(x).Q] \longrightarrow n[P] \mid m[Q\{x \leftarrow a\}]$$

Communication Primitives (`send/recv`)

- Sibling-sibling communication: a specific sibling and any other sibling

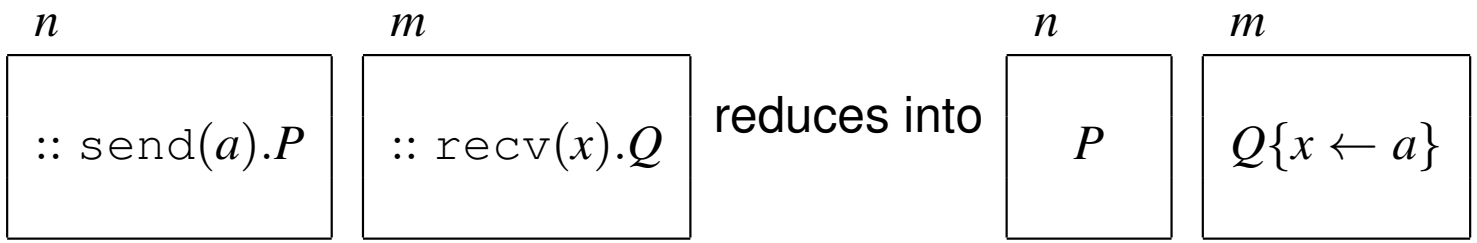


- The corresponding **reduction rule**

$$n[m :: \texttt{send}(a).P] \mid m[:: \texttt{recv}(x).Q] \longrightarrow n[P] \mid m[Q\{x \leftarrow a\}]$$

Communication Primitives (`send/recv`)

- Sibling-sibling communication: any siblings



n

- The corresponding **reduction rule**

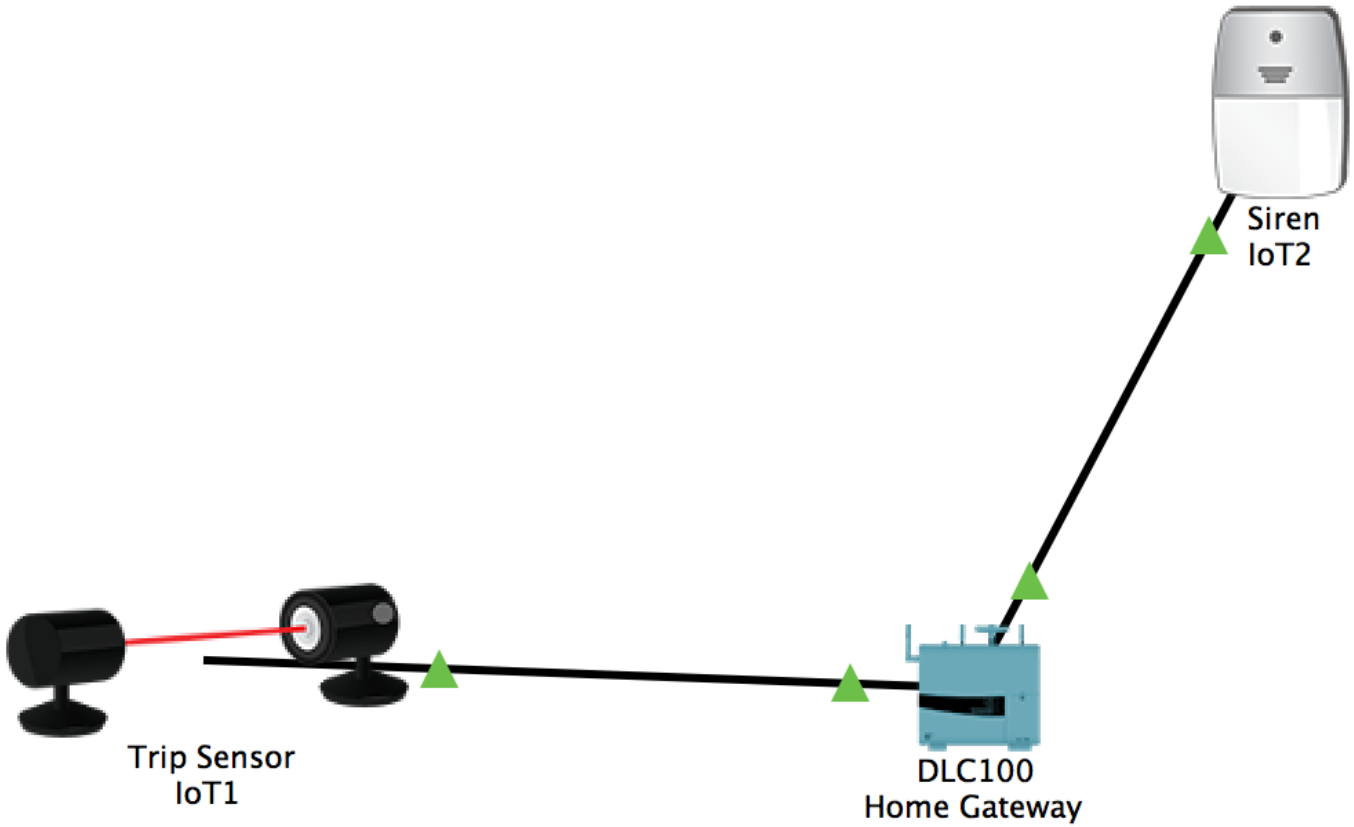
$$n[:: \texttt{send}(a).P] \mid m[:: \texttt{recv}(x).Q] \longrightarrow n[P] \mid m[Q\{x \leftarrow a\}]$$

Lab Session 3 – Try it out (Communication)

1. $\texttt{send}(8).\mathbf{0} \mid \texttt{recv}().\mathbf{0}$
2. $\texttt{send}(8).\mathbf{0} \mid \texttt{recv}(x, y).\mathbf{0}$
3. $\texttt{send}(8).\mathbf{0} \mid \texttt{recv}(x).\mathbf{0}$
4. $\texttt{send}(8, 6).\mathbf{0} \mid \texttt{recv}().\mathbf{0}$
5. $\texttt{send}(8, 6).\mathbf{0} \mid \texttt{recv}(x).\mathbf{0}$
6. $\texttt{send}(8).\texttt{send}(6).\mathbf{0} \mid \texttt{recv}(x, y).\mathbf{0}$
7. $\texttt{send}(8, 6).\mathbf{0} \mid \texttt{recv}(x).\texttt{recv}(y).\mathbf{0}$
8. $\texttt{send}(8, 6).\mathbf{0} \mid \texttt{recv}(x, y).\mathbf{0}$
9. $\#\texttt{send}(1).\mathbf{0} \mid n[@\texttt{recv}(x).\mathbf{0}]$
10. $n\#\texttt{send}(1).\mathbf{0} \mid n[@\texttt{recv}(x).\mathbf{0}]$
11. $m\#\texttt{send}(1).\mathbf{0} \mid n[@\texttt{recv}(x).\mathbf{0}]$
12. $m[::\ \texttt{send}(1).\mathbf{0}] \mid n[::\ \texttt{recv}(x).\mathbf{0}]$
13. $m[::\ \texttt{send}(1).\mathbf{0}] \mid n[::\ \texttt{recv}(x).\mathbf{0}] \mid w[::\ \texttt{recv}(x).\mathbf{0}]$

DE MONTFORT UNIVERSITY LEICESTER

Example: A simple intrusion detection system



Example: A simple intrusion detection system (Cont'd)

- Trip sensor is a type of motion detector that can detect movement across a laser beam.
- It can be used to alert a user of intrusion into a controlled area; e.g. by raising the alarm.
- The simple intrusion detection system comprises a trip sensor which is connected to a server (home gateway) that raises the alarm when movement is detected.
- The system can be modelled in ccaPL as follows.

DE MONTFORT UNIVERSITY LEICESTER

Example: A simple intrusion detection system (Cont'd)

- The trip sensor is modelled as an ambient.
- The trip sensor is triggered when it receives a signal from a sibling ambient.
- It then tells the server that a movement has been detected.

```
trip_sensor[
  !recv(). ::recv().home_gateway::send().send().0
| send().0
]
```

- A user, say Jack, can trigger the trip sensor (by interacting with it)

```
jack[
    trip_sensor::send().0
]
```

- This will raise the alarm ...

DE MONTFORT UNIVERSITY LEICESTER

Example: A simple intrusion detection system (Cont'd)

- The server (home gateway) can be modelled as an ambient
- When it receives a signal from the trip sensor (movement detected), it turns on the alarm for some time (`skip`), then turns off the alarm.
- `skip` is used to model an abstract time unit.

```
home_gateway[
    !recv().trip_sensor::recv().siren::send(on).
     skip.skip.siren::send(off).send()
| send().0
]
```

DE MONTFORT UNIVERSITY LEICESTER

Example: A simple intrusion detection system (Cont'd)

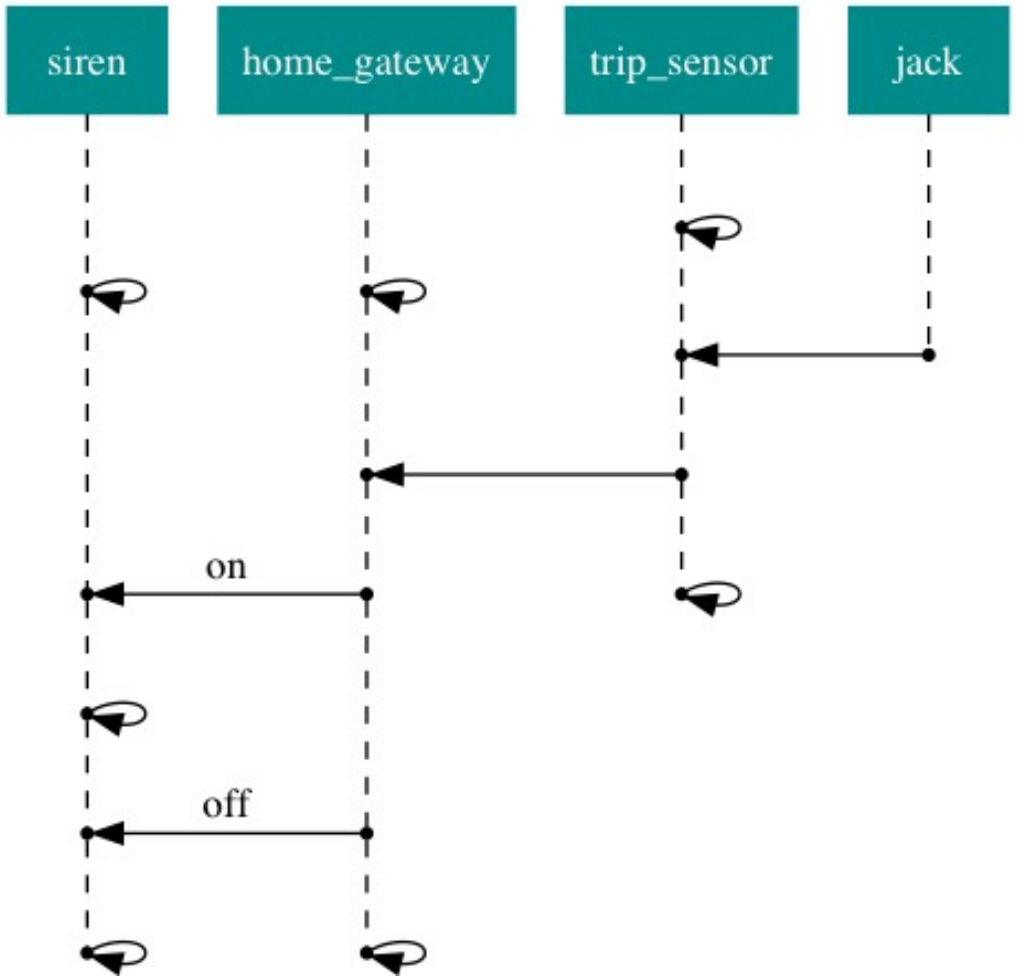
- The siren is also modelled as an ambient.
- It has 2 possible states: on and off.
- Its current state is indicated by a child ambient `on[0]` or `off[0]`.
- It is controlled by the server (home gateway).
- Initially the siren is off.

```
siren[
 !recv().home_gateway::recv(x).if
   < x=off and state(on)>del on.send().off[0]
   < x=on and state(off)>del off.send().on[0]
 else send().0
 fi | send().0
| off[0] // Initially, the siren is off.
]
```

DE MONTFORT UNIVERSITY LEICESTER

Example: A simple intrusion detection system (Cont'd)

- Simulation of the intrusion detection system



'let' statement

$\texttt{let}\ x_0 = e_0, x_1 = e_1, \cdots, x_n = e_n\ \texttt{in}\ P$

- Behaves like $P\{x_0 \leftarrow v_0, \ldots, x_n \leftarrow v_n\}$ where v_i is the value of the arithmetic expression e_i.
- Arithmetic expressions e:

e	$::=$	x	a name: string or integer
		$\mid -e$	unary minus
		$\mid e_1 + e_2$	addition of integers or concatenation of strings
		$\mid e_1 - e_2$	minus
		$\mid e_1 * e_2$	multiplication
		$\mid e_1 * *e_2$	power
		$\mid e_1 \% e_2$	modulo
		$\mid e_1 / e_2$	division

DE MONTFORT UNIVERSITY LEICESTER

'let' statement – Examples

Example

- $\texttt{let}\ x = 3 + 4\ \texttt{in}\ \texttt{send}(x).\mathbf{0}$ behaves like $\texttt{send}(7).\mathbf{0}$
- $\texttt{recv}(x, y).\texttt{let}\ z = x + y\ \texttt{in}\ \texttt{send}(z).\mathbf{0}$receives two integers (resp. strings) and outputs their sum (res. concatenation).
- $\texttt{recv}(x).\texttt{let}\ z = x\%2\ \texttt{in}\ \texttt{send}(z).\mathbf{0}$
- $\texttt{recv}(x).\texttt{let}\ y = x + 1, z = x\%2\ \texttt{in}\ \texttt{send}(y, z).\mathbf{0}$
- $\texttt{recv}(x).\texttt{let}\ z = x\%2\ \texttt{in}\ \texttt{let}\ y = x * z + 1\ \texttt{in}\ \texttt{send}(y, z).\mathbf{0}$

DE MONTFORT UNIVERSITY LEICESTER

Lab Session 4 – Try it out (Let statement)

1. $\texttt{send}(2,3).\texttt{recv}(res).\mathbf{0} \mid$
   $\texttt{recv}(x,y).\texttt{let}\ z = x+y\ \texttt{in}\ \texttt{send}(z).\mathbf{0}$
2. $\texttt{send}(2,3).\texttt{recv}(sum,dif).\mathbf{0} \mid$
   $\texttt{recv}(x,y).\texttt{let}\ z = x+y, t = x-y\ \texttt{in}\ \texttt{send}(z,t).\mathbf{0}$
3. $\texttt{send}(2,3,7).\texttt{recv}(res).\mathbf{0} \mid$
   $\texttt{recv}(x,y,z).\texttt{let}\ t = (x+y+z)/3\ \texttt{in}\ \texttt{send}(t).\mathbf{0}$
4. $\#\texttt{send}(2,3).\#\texttt{recv}(res).\mathbf{0} \mid$
   $adder[!@\texttt{recv}(x,y).\texttt{let}\ z = x+y\ \texttt{in}\ @\texttt{send}(z).\mathbf{0}]$
5. $\#\texttt{send}(2,3,7).\#\texttt{recv}(res).\mathbf{0} \mid$
   $mean[!@\texttt{recv}(x,y,z).\texttt{let}\ t = (x+y+z)/3\ \texttt{in}\ @\texttt{send}(t).\mathbf{0}]$

DE MONTFORT UNIVERSITY LEICESTER

Process Abstraction/Call

- A **process abstraction** links a name x to a process P using the syntax:

$$\texttt{proc}\ x(y_1, y_2, \ldots, y_n)\ \{P\}$$

  where $y_1, y_2, \ldots, y_n$ are formal parameters.

- A **process call** invokes a process abstraction x using the syntax:

$$\alpha x(z_1, z_2, \ldots, z_n).$$

  where $z_1, z_2, \ldots, z_n$ are actual parameters and α indicates the location of the process abstraction called and is defined as follows:

$$\alpha ::= @ \mid n@ \mid \# \mid n\# \mid \ :: \ \mid n :: \ \mid \epsilon$$

DE MONTFORT UNIVERSITY LEICESTER

Process Abstraction/Call: Semantics

- Calling a local process

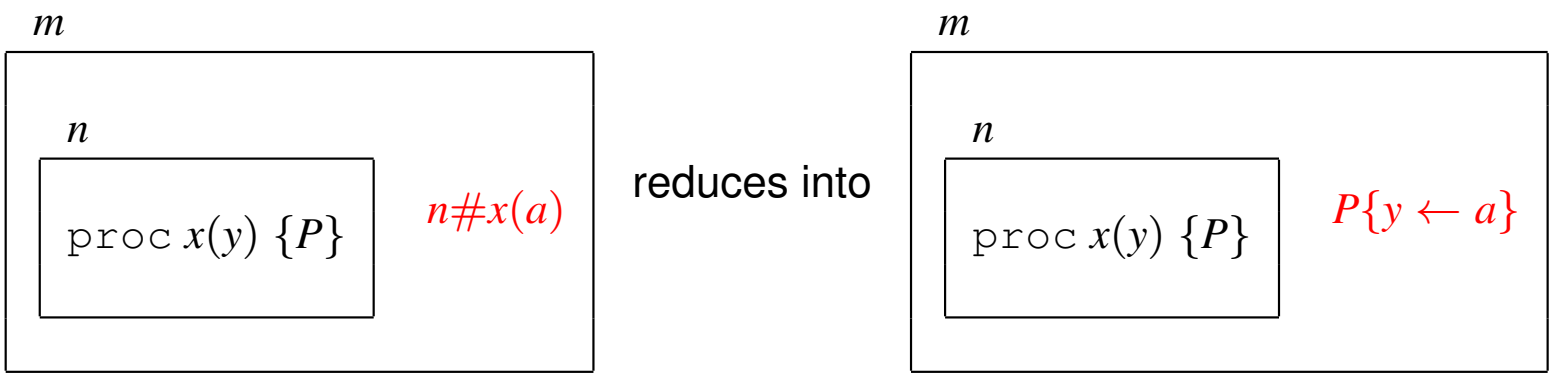


- The corresponding **reduction rule**

$$n[\texttt{proc}\, x(y)\ \{P\} \mid x(a)] \longrightarrow n[\texttt{proc}\, x(y)\ \{P\} \mid P\{y \leftarrow a\}]$$

Process Abstraction/Call: Semantics

- Calling a process abstraction defined in a child ambient

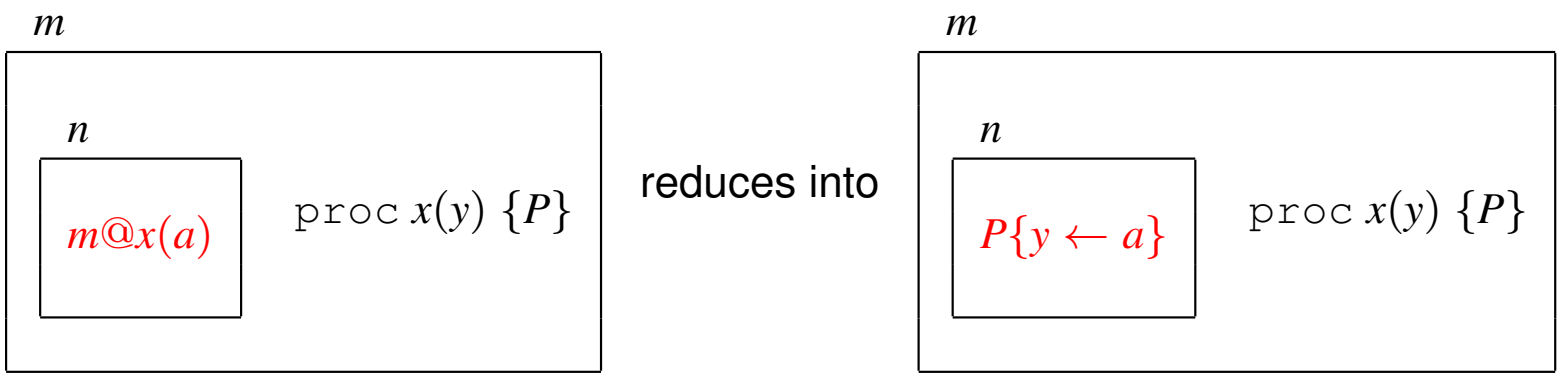


- The corresponding **reduction rule**

$$m[n[\texttt{proc}\ x(y)\ \{P\}] \mid n\#x(a)] \longrightarrow m[n[\texttt{proc}\ x(y)\ \{P\}] \mid P\{y \leftarrow a\}]$$

Process Abstraction/Call: Semantics

- Calling a process abstraction defined in a child ambient

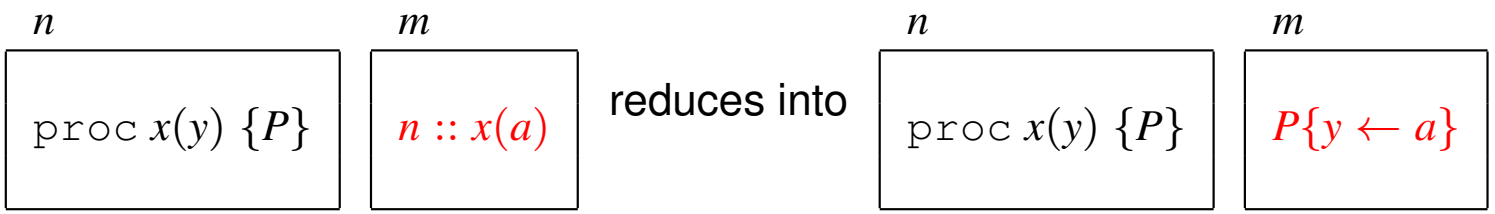


- The corresponding **reduction rule**

$$m[n[m@x(a)] \mid \texttt{proc}\ x(y)\ \{P\}] \longrightarrow m[n[P\{y \leftarrow a\}] \mid \texttt{proc}\ x(y)\ \{P\}]$$

Process Abstraction/Call: Semantics

- Calling a process abstraction defined in a sibling ambient



- The corresponding **reduction rule**

$$n[\texttt{proc}\ x(y)\ \{P\}] \mid m[n :: x(a)] \longrightarrow n[\texttt{proc}\ x(y)\ \{P\}] \mid m[P\{y \leftarrow a\}]$$

Process Abstraction/Call: Example

Context Provision using Process Abstraction

- User preferences, Available services, Device configuration
- My favourite text editor on Windows is `notepad`

$$win[\texttt{proc edit}(file)\ \{\texttt{notepad}(file).\mathbf{0}\}]$$

- My favourite text editor on Linux is `emacs`

$$lin[\texttt{proc edit}(file)\ \{\texttt{emacs}(file).\mathbf{0}\}]$$

A Context-aware file editing agent can be specified as

$$ag[@\texttt{edit}(foo).\mathbf{0}]$$

DE MONTFORT UNIVERSITY LEICESTER

Context Acquisition using Process Call

When the file editing agent runs

- on my Windows machine, `notepad` is used

$$win[\texttt{proc edit}(file)\ \{\texttt{notepad}(file).\mathbf{0}\} \mid ag[@\texttt{edit}(foo).\mathbf{0}]] \rightarrow win[\texttt{proc edit}(file)\ \{\texttt{notepad}(file).\mathbf{0}\} \mid ag[\texttt{notepad}(foo).\mathbf{0}]]$$

- on my Linux machine, `emacs` is used

$$lin[\texttt{proc edit}(file)\ \{\texttt{emacs}(file).\mathbf{0}\} \mid ag[@\texttt{edit}(foo).\mathbf{0}]] \rightarrow lin[\texttt{proc edit}(file)\ \{\texttt{emacs}(file).\mathbf{0}\} \mid ag[\texttt{emacs}(foo).\mathbf{0}]]$$

DE MONTFORT UNIVERSITY LEICESTER

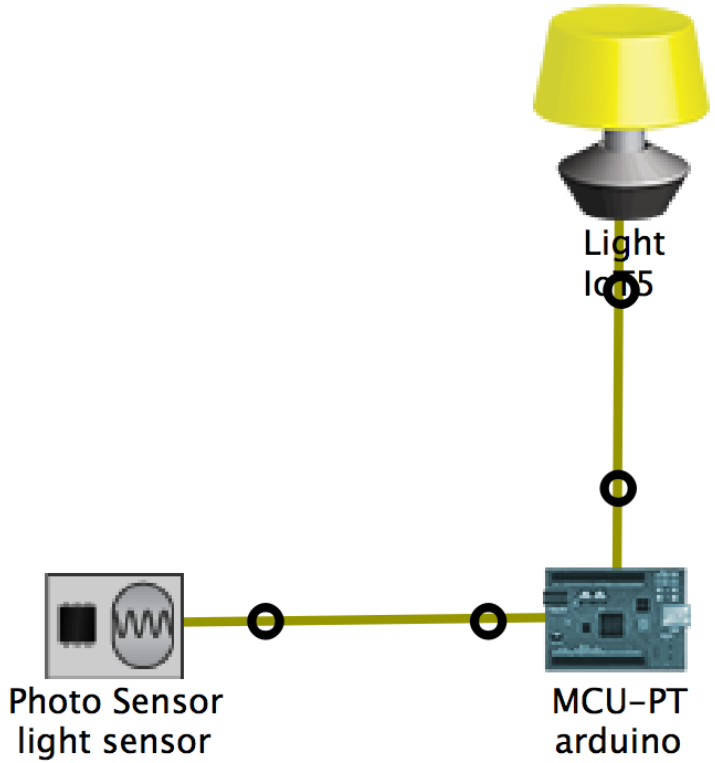
Lab Session 5 – Try it out (Process Abstraction)

1. $add(2,3).\mathbf{0} \mid \texttt{recv}(res).\mathbf{0} \mid$
   $\texttt{proc}\ add(x,y)\ \{$
   $\quad \texttt{let}\ z = x + y\ \texttt{in}\ \texttt{send}(z).\mathbf{0}$
   $\}$
2. $\texttt{send}(2,3).\texttt{recv}(sum, dif).\mathbf{0} \mid calc().\mathbf{0}$
   $\texttt{proc}\ calc()\ \{$
   $\quad \texttt{recv}(x,y).\texttt{let}\ z = x + y, t = x - y\ \texttt{in}\ \texttt{send}(z,t).\mathbf{0}$
   $\}$
3. $n\#add(2,3).\mathbf{0} \mid \texttt{recv}(res).\mathbf{0} \mid$
   $n[$
   $\quad \texttt{proc}\ add(x,y)\ \{$
   $\quad\quad \texttt{let}\ z = x + y\ \texttt{in}\ \texttt{send}(z).\mathbf{0}$
   $\quad \}$
   $]$

DE MONTFORT UNIVERSITY LEICESTER

Example: Smart Lamp Control System

- The smart lamp is controlled by an Arduino-like micro-controller unit (MCU).
- The light sensor detects light when the light level is above a 20% threshold.
- The controller turns the lamp on when no light is detected and off otherwise.
- This is the lamp is switched on in night time and off during the day, automatically.
- Download the program smartlamp.cca from Blackboard.



Example: Smart Lamp Control System

- The lamp is modelled as in the example above.
- Suppose that the light level goes up and down randomly.

```
light_sensor[
!recv(v). if
    < v>20 > mcu::send(1).send(v).0
    < v<=20 > mcu::send(0).send(v).0
    < v<100 > let x=v+1 in send(x).0
    < v>0 > let x=v-1 in send(x).0
  fi

| send(20).0// Initial light level is 20%.
]
```

Example: Smart Lamp Control System

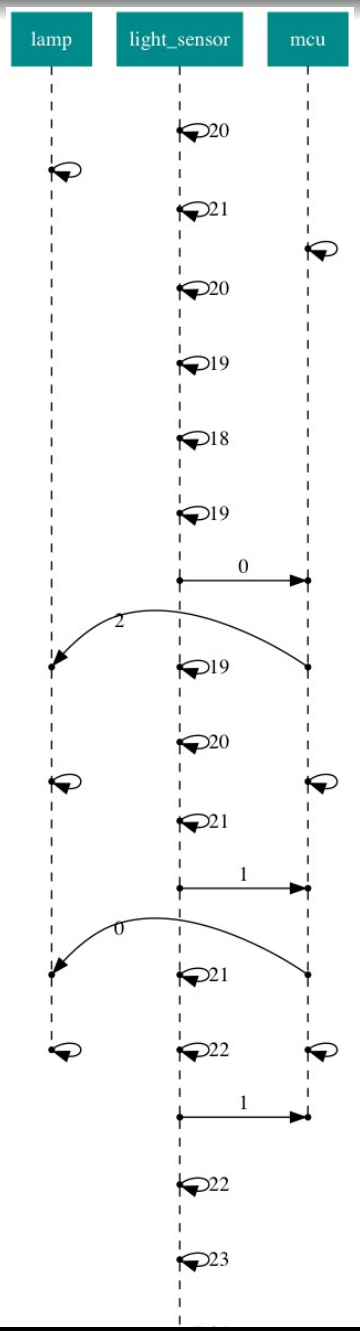
- The MCU is an ambient that implements the process abstraction loop().

```
mcu[
  proc loop() { // definition
     !recv().light_sensor::recv(t).
      if < t=0 and state(lamp, off) > lamp::send(2).0
         < t=1 and state(lamp, on) > lamp::send(0).0
      fi.send().0
   | send().0
  }

  | loop() // call
]
```

DE MONTFORT UNIVERSITY LEICESTER

Example: Smart Lamp Control System (Cont'd)



Learning Outcomes

In this lesson you have learned:

- Communication statements.
- Process abstractions.
- How to model and simulate IoT devices like sensors and actuators.
- How to model a micro-controller that controls IoT devices.

Quiz

Answer by "true" or "false"

- The ambient $n[\#\texttt{send}(2)]$ sends the value 2 to a parent ambient.
- The ambient $n[::\texttt{recv}()]$ receives a signal from a sibling ambient.
- The ambient $n[\#\texttt{send}(2)]$ sends the value 2 to a child ambient.
- The ambient $n[::\texttt{recv}()]$ receives a signal from a parent ambient.
- The ambient $n[\texttt{send}(2)]$ sends the value 2 to a parent ambient.
- The ambient $n[\texttt{recv}()]$ receives a signal from a sibling ambient.
- The process $\texttt{send}().0|\texttt{recv}(x).0$ reduces to $\mathbf{0}$.
- The process $\texttt{send}(2).0|\texttt{recv}(x).0$ reduces to $\mathbf{0}$.
- The process $\texttt{send}(2,5).0|\texttt{recv}(x).0$ reduces to $\mathbf{0}$.

DE MONTFORT UNIVERSITY LEICESTER

Exercise 1

Give the reductions corresponding to the execution of the following processes:

1. $\texttt{send}(5).\mathbf{0} \mid \texttt{recv}(x).\mathbf{0}$
2. $\texttt{send}(5).\mathbf{0} \mid \texttt{recv}(x).\texttt{send}(x).\mathbf{0}$
3. $n[\texttt{send}(5).\mathbf{0} \mid \texttt{recv}(x).\mathbf{0}]$
4. $m[\texttt{send}(5).\mathbf{0} \mid \texttt{recv}(x).\texttt{send}(x).\mathbf{0}]$
5. $n[m :: \texttt{send}(5).\mathbf{0}] \mid m[n :: \texttt{recv}(x).\mathbf{0}]$
6. $n[:: \texttt{send}(5).\mathbf{0}] \mid m[n :: \texttt{recv}(x).\mathbf{0}]$
7. $n[m :: \texttt{send}(5).\mathbf{0}] \mid m[:: \texttt{recv}(x).\mathbf{0}]$
8. $n[:: \texttt{send}(5).\mathbf{0}] \mid m[:: \texttt{recv}(x).\mathbf{0}]$
9. $n[:: \texttt{send}(5).\texttt{in}\ m.\mathbf{0}] \mid m[:: \texttt{recv}(x).\texttt{del}\ n.\mathbf{0}]$
10. $n[@\texttt{send}(5).\mathbf{0} \mid \texttt{in}\ m.\mathbf{0}] \mid m[\#\texttt{recv}(x).\texttt{del}\ n.\mathbf{0}] \mid \texttt{del}\ m.\mathbf{0}$

DE MONTFORT UNIVERSITY LEICESTER

Exercise 2

1. The following process send a hello message and receives an acknowledgement:

$$\texttt{send}(\mathit{hello}).\texttt{recv}(x).\mathbf{0} \mid P$$

   Give the specification of P.

2. An ambient m sends an hello message to another ambient n and receives an acknowledgement from n. Give the specification of the ambients m and n.

3. Explain the behaviour of the following process

$$m[\texttt{in}\ n.\texttt{out}.\mathbf{0}] \mid n[\texttt{in}\ m.\texttt{out}.\mathbf{0}]$$

4. Specify a network hub that has 4 output ports.

5. Specify a simple network switch with four output ports connected to specific devices.

6. Specify a server that receives a service request, invokes the requested service, and sends the output to the requester. Services can be modelled as process abstractions.

DE MONTFORT UNIVERSITY LEICESTER