The Calculus of Context-aware Ambients (CCA)

How to program: Capabilities and Processes (1)

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

Overview

Learning Outcomes

By the end of this lesson you should know:

- The syntax and the semantics of ccaPL programming language
- How to model IoT devices like smart lamp, smart door, and RFID tag and RFID reader
- How to build a network of IoT devices in CCA

Syntax of ccaPL

There are 3 main syntactic categories:

- Processes: which denote a computation (i.e. behaviour)
- Capabilities: which are elementary actions a process can perform
- Context expression: which is a formula expressing some property of a process' context.

Notations

In the syntax of processes, the following notations are used:

- P, Q: denote processes
- n, x: are names
- M: denotes a capability
- κ: denotes a context expression

Syntax of Processes

$P, Q ::=$	**Processes**
$\mathbf{0}$	inactivity
$M.P$	prefix
$P \mid Q$	parallel composition
$!P$	replication
$\texttt{new}\ n\ P$	name restriction
$n[P]$	ambient
$\{P\}$	brackets
$<\kappa>M.P$	context-guarded prefix
$\texttt{proc}\ x(y_1, y_2, \ldots, y_\ell)\ \{P\}$	process abstraction x
$\texttt{find}\ \tilde{y} : \kappa\ \texttt{for}\ P$	search
$\texttt{if}\ <\kappa_1>M_1.P_1\ \ldots\ <\kappa_\ell>M_\ell.P_\ell\ \texttt{fi}$	selection (if)
$\texttt{if}\ <\kappa_1>M_1.P_1\ \ldots\ <\kappa_\ell>M_\ell.P_\ell\ \texttt{else}\ P\ \texttt{fi}$	selection (if-else)
$\texttt{let}\ x_1 = e_1,\ \ldots,\ x_\ell = e_\ell\ \texttt{in}\ P$	'let' statement

DE MONTFORT UNIVERSITY LEICESTER

Syntax of Capabilities

$M ::=$	`skip`	a single transition
	`in` n	move into the ambient n
	`out`	move out of current location
	`del` n	delete the ambient n
	$\alpha\ x(z_1, z_2, \ldots, z_\ell)$	call the process x defined in location with actual parameters $z_1, z_2, \ldots, z_\ell$
	α `recv`$(y_1, y_2, \ldots, y_\ell)$	receive into $y_1, y_2, \ldots, y_\ell$ a list of messages from location α
	α `send`$(z_1, z_2, \ldots, z_\ell)$	send a list of messages $z_1, z_2, \ldots, z_\ell$ to location α

Syntax of Locations α

$\alpha ::=$	$@$	any parent ambient
	$n@$	parent n
	$\#$	any child ambient
	$n\#$	child n
	$::$	any sibling ambient
	$n ::$	sibling n
	ϵ	locally (empty string)

Context Property Language

Context expressions are described by the following language:

κ ::=	**Context Expressions**
$\mathbf{0}$	empty context
`true`	true
`false`	false
$n = m$	name match
`this`	hole
`not` κ	negation
$\kappa_1 \mid \kappa_2$	parallel composition
κ_1 `and` κ_2	conjunction
κ_1 `or` κ_2	disjunction
$n[\kappa]$	location
`next` κ	spatial next modality
`somewhere` κ	somewhere modality

DE MONTFORT UNIVERSITY LEICESTER

Inactive Process ($\mathbf{0}$)

The inactivity process or null process $\mathbf{0}$

- does nothing
- terminates immediately
- is the unit element for the parallel composition of processes, i.e.

$$\mathbf{0} \mid P \equiv P \mid \mathbf{0} \equiv P$$

where the symbol '$\equiv$' means 'is equivalent to'.

skip Transition

skip.P

- this process performs one transtion then continues like the process P.

Example

- skip.$\mathbf{0}$ // performs 1 transtion and terminates
- skip.skip.$\mathbf{0}$ // performs 2 transitions and terminates
- skip.skip.skip.$\mathbf{0}$ // performs 3 transitions and terminates

Sequential Composition ($M.P$)

$M.P$

- this process performs the capability M then continues like the process P.

Example

- `skip.out.`$\mathbf{0}$ // performs `skip` then the capability `out` and terminates
- `send(5).skip.`$\mathbf{0}$ // performs the capability `send` then `skip` and terminates
- `send(5).out.`$\mathbf{0}$ // performs the capability `send` then `out` and terminates

Parallel Composition ($P \mid Q$)

$P \mid Q$

- means that the process P and the process Q execute concurrently
- we also say P and Q execute in parallel i.e. P and Q execute at the same time
- during their execution, P and Q can communicate by message passing

Example

- $\texttt{out}.\mathbf{0} \mid \texttt{send}(5).\mathbf{0}$
- $\texttt{send}(5).\mathbf{0} \mid \texttt{recv}(x).\mathbf{0}$
- $\texttt{send}(5).\texttt{out}.\mathbf{0} \mid \texttt{recv}(x).\mathbf{0}$

DE MONTFORT UNIVERSITY LEICESTER

Parallel Composition ($P \mid Q$) (cont'd)

Properties of parallel composition of processes (the symbol $\equiv$ means is equivalent to.)

- $\mathbf{0}$ is the unit element

$$\mathbf{0} \mid P \equiv P \mid \mathbf{0} \equiv P$$

- Commutative (i.e. the syntactic order does not matter)

$$P \mid Q \equiv Q \mid P$$

- Associative (i.e. the execution order does not matter)

$$P \mid (Q \mid R) \equiv (P \mid Q) \mid R \equiv P \mid Q \mid R$$

Replication ($!P$)

- $!P$ denotes a process which can always creates a new copy of the process P
- $!P \equiv P \mid !P$
- Can be used to model iteration and recursion

Example

- This process always outputs/sends the value 2:

$$!\texttt{send}(2).\mathbf{0}$$

- This process always prompts to receive a message and then simply forwards the received message:

$$!\texttt{recv}(x).\texttt{send}(x).\mathbf{0}$$

DE MONTFORT UNIVERSITY LEICESTER

Lab Session 1 – Try it out (Concurrency)

1. $\texttt{skip.skip.}\mathbf{0}$
2. $\texttt{skip.}\mathbf{0} \mid \texttt{skip.}\mathbf{0}$
3. $\texttt{send}(5)\texttt{.skip.}\mathbf{0} \mid \texttt{recv}(x)\texttt{.skip.}\mathbf{0}$
4. $\texttt{skip.send}(5)\texttt{.}\mathbf{0} \mid \texttt{recv}(x)\texttt{.}\mathbf{0}$
5. $\texttt{skip.send}(5)\texttt{.}\mathbf{0} \mid \texttt{recv}(x)\texttt{.skip.}\mathbf{0}$
6. $\texttt{skip.send}(5)\texttt{.}\mathbf{0} \mid \texttt{skip.recv}(x)\texttt{.}\mathbf{0}$
7. $\texttt{send}(2,3)\texttt{.send}(6)\texttt{.}\mathbf{0} \mid \texttt{recv}(x,y)\texttt{.recv}(z)\texttt{.}\mathbf{0}$
8. $\texttt{send}(2)\texttt{.send}(6)\texttt{.}\mathbf{0} \mid \texttt{recv}(x,y)\texttt{.recv}(z)\texttt{.}\mathbf{0}$
9. $\texttt{send}(2)\texttt{.send}(6)\texttt{.}\mathbf{0} \mid \texttt{recv}(x)\texttt{.}\mathbf{0}$
10. $\texttt{send}(2)\texttt{.}\mathbf{0} \mid \texttt{recv}(x)\texttt{.}\mathbf{0} \mid \texttt{recv}(z)\texttt{.}\mathbf{0}$
11. $\texttt{send}(2)\texttt{.send}(6)\texttt{.}\mathbf{0} \mid !\texttt{recv}(x)\texttt{.}\mathbf{0}$
12. $!\texttt{send}(2)\texttt{.}\mathbf{0} \mid !\texttt{recv}(x)\texttt{.}\mathbf{0}$

DE MONTFORT UNIVERSITY LEICESTER

Example: a smart lamp

- A smart lamp is an actuator.
- It can be switched to: off, dim, or on.
- It can be controlled by a server.
- It receives commands as numbers: 0 for off; 1 for dim, and 2 for on.



Lamp is off



Lamp is dim



Lamp is on

DE MONTFORT UNIVERSITY LEICESTER

Example: a smart lamp (Cont'd)

The behaviour of the lamp is as follows:

- It receives a command from the server (home gateway) and changes state accordingly.
- Let state(n) = somewhere (this | n[0] | true)
- state(n) means my current state is n.

```
lamp[
    !recv().home_gateway::recv(x).if
<x=0 and not state(off)>skip.find y:state(y) for del y.send().off[0]
<x=1 and not state(dim)>skip.find y:state(y) for del y.send().dim[0]
<x=2 and not state(on)>skip.find y:state(y) for del y.send().on[0]
      else send().0
      fi.0 | send().0
| off[0] // Initially, the light is  off.
]
```

Brackets ($\{P\}$)

- $\{P\}$ behaves just like P.
- Brackets are used to describe complex processes.
- Used only as suffix process, i.e. in the form $P.\{Q\}$

Example

- $\{P \mid Q\}$
- $P.\{Q \mid R\}$
- $!\{P \mid Q\}$

Example

Counter examples

- $\{P \mid Q\}.S$
- $P.\{Q \mid R\}.S$

DE MONTFORT UNIVERSITY LEICESTER

Ambient process ($n[P]$)

$n[P]$

- represents an ambient named n with behaviour described by the process P.
- P specifies the capabilities that the ambient n can perform.
- it is also said that the ambient n executes the process P.

Example

- $n[!\texttt{send}(2).\mathbf{0} \mid !\texttt{recv}(x).\mathbf{0}]$
- $n[!\texttt{recv}(x).\texttt{send}(x).\mathbf{0}]$
- $n[\texttt{in}\ m.\texttt{out}.\mathbf{0}] \mid m[\texttt{in}\ n.\texttt{out}.\mathbf{0}]$

DE MONTFORT UNIVERSITY LEICESTER

Restriction (new n P)

The restriction new n P

- states that n is a new name visible only in the process P.
- So the name n cannot be referred to outside P.
- Similar to the declaration of local variables in programming languages like C or Java.
- Also use to hide the name n in P so that no other process can 'see' it.

Example

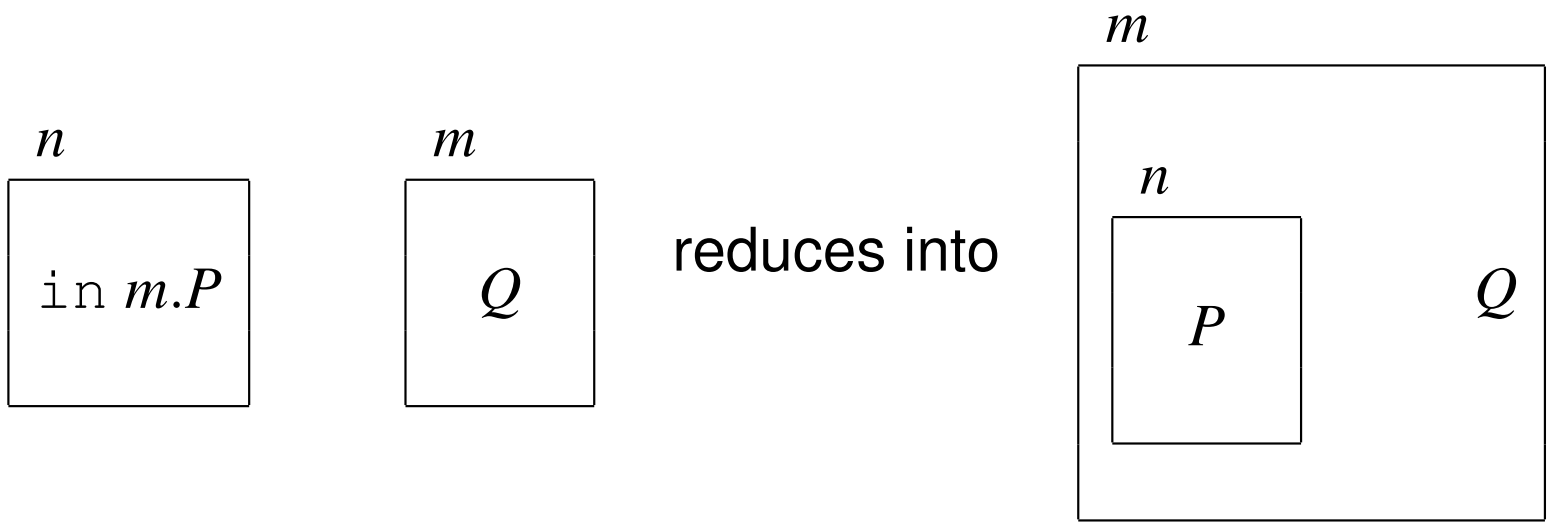
- new n $\{$del $n.\mathbf{0} \mid n[\mathbf{0}]\}$
- new n del $n.\mathbf{0} \mid n[\mathbf{0}]$
- Do these processes have the same behaviour?

DE MONTFORT UNIVERSITY LEICESTER

Mobility Primitives (in)

in m

- allows an ambient to move into a sibling ambient m



- The corresponding **reduction rule**

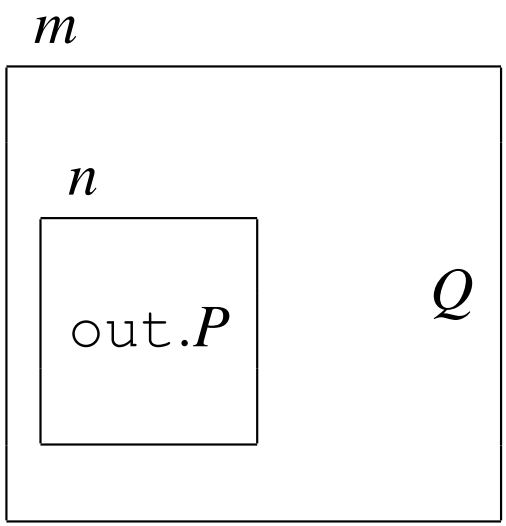
$$n[\texttt{in}\ m.P] \mid m[Q] \quad \longrightarrow \quad m[n[P] \mid Q]$$

The symbol $\longrightarrow$ means reduces to.

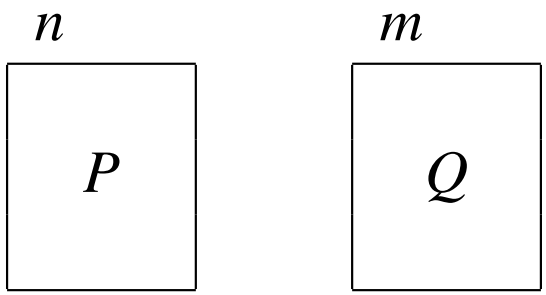
Mobility Primitives (out)

out

- allows an ambient to move out of its parent



reduces into



- The corresponding **reduction rule**

$$m[n[\texttt{out}.P] \mid Q] \quad \longrightarrow \quad n[P] \mid m[Q]$$

Mobility Primitives: Examples

- A software agent *ag* moves into computer *win*.

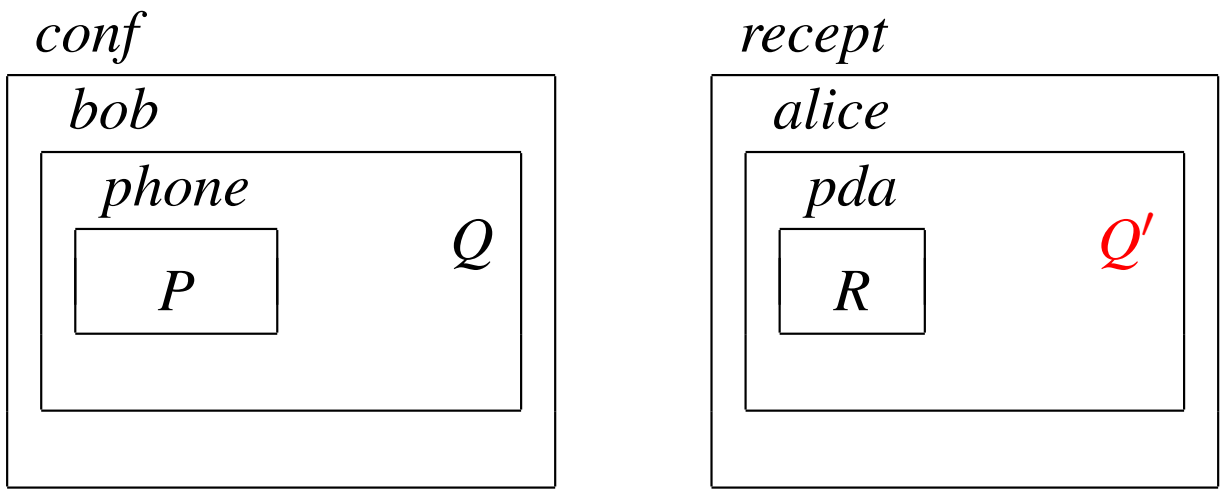
$$ag[\texttt{in}\ win.\texttt{out}.\mathbf{0}] \mid win[\mathbf{0}] \quad \rightarrow \quad win[ag[\texttt{out}.\mathbf{0}]]$$

- Agent *ag* exits computer *win*.

$$win[ag[\texttt{out}.\mathbf{0}]] \quad \rightarrow \quad ag[\mathbf{0}] \mid win[\mathbf{0}]$$

Mobility Primitives: Exercise

- Specify a process Q' that allows Alice to move from the reception room to the conference room



Delete an ambient (del)

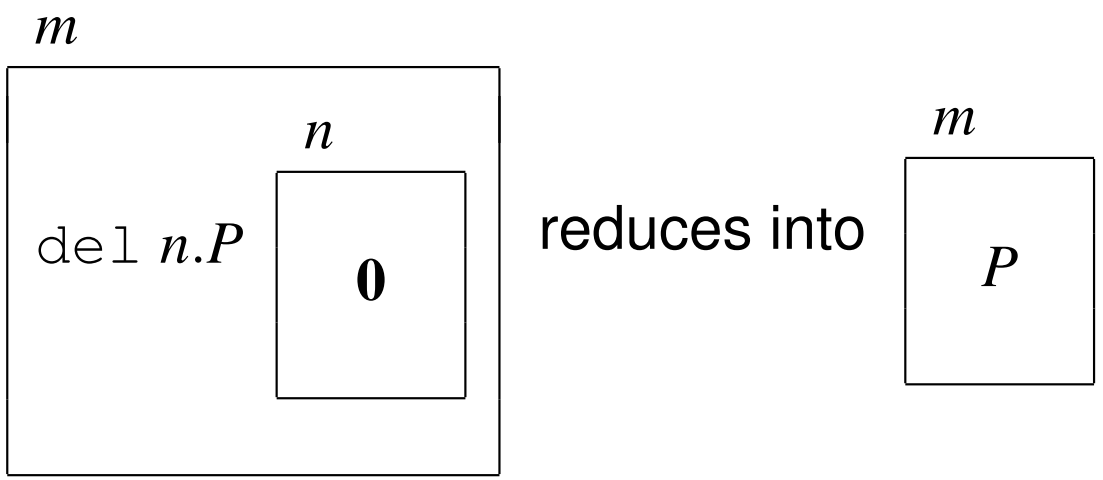
del n

- deletes the ambient n if n has no capabilities; i.e.

$$\texttt{del}\ n.P \mid n[\mathbf{0}] \longrightarrow P$$

$$\texttt{del}\ n.P \mid n[\texttt{out}.\mathbf{0}] \not\longrightarrow P$$

- Inside an ambient, it looks like this:



Lab Session 2 – Try it out (Mobility)

1. $\texttt{del}\ n.\mathbf{0} \mid n[\mathbf{0}]$
2. $\texttt{del}\ n.n[\mathbf{0}]$
3. $\texttt{del}\ n.\mathbf{0} \mid n[\texttt{send}().\mathbf{0}]$
4. $\texttt{del}\ n.\mathbf{0} \mid n[\texttt{send}().\mathbf{0} \mid \texttt{recv}().\mathbf{0}]$
5. $\texttt{del}\ n.\texttt{del}\ m.\mathbf{0} \mid n[m[\texttt{out}.\mathbf{0}]]$
6. $\texttt{del}\ n.\mathbf{0} \mid \texttt{del}\ m.\mathbf{0} \mid n[m[\texttt{out}.\mathbf{0}]]$
7. $n[\texttt{out}.\mathbf{0}]$
8. $m[n[\texttt{out}.\mathbf{0}] \mid \texttt{skip}.\mathbf{0}]$
9. $n[\texttt{in}\ m.\mathbf{0}]$
10. $n[\texttt{in}\ m.\mathbf{0}] \mid m[\mathbf{0}]$
11. $n[\texttt{in}\ m.\texttt{out}.\mathbf{0}] \mid m[\texttt{in}\ n.\texttt{out}.\mathbf{0}]$
12. $!n[\texttt{in}\ m.\texttt{out}.\mathbf{0}] \mid !m[\texttt{in}\ n.\texttt{out}.\mathbf{0}]$

DE MONTFORT UNIVERSITY LEICESTER

Example: RFID reader

- RFID card stores a positive number greater than 0.
- When the card is moved near the reader, the card transmits its ID number to the reader.
- The reader then sends the ID number to the server (home gateway).
- The server checks if ID number is valid or invalid and sends result to reader.
- The reader turns green (valid) or red (invalid).



RFID Card
IoT3



RFID Reader
IoT4

Example: RFID reader (Cont'd)

- RFID card is modelled simply as an ambient that stores an ID number.

```
RFID[ //RFID card
    !@send(102).0
]
```

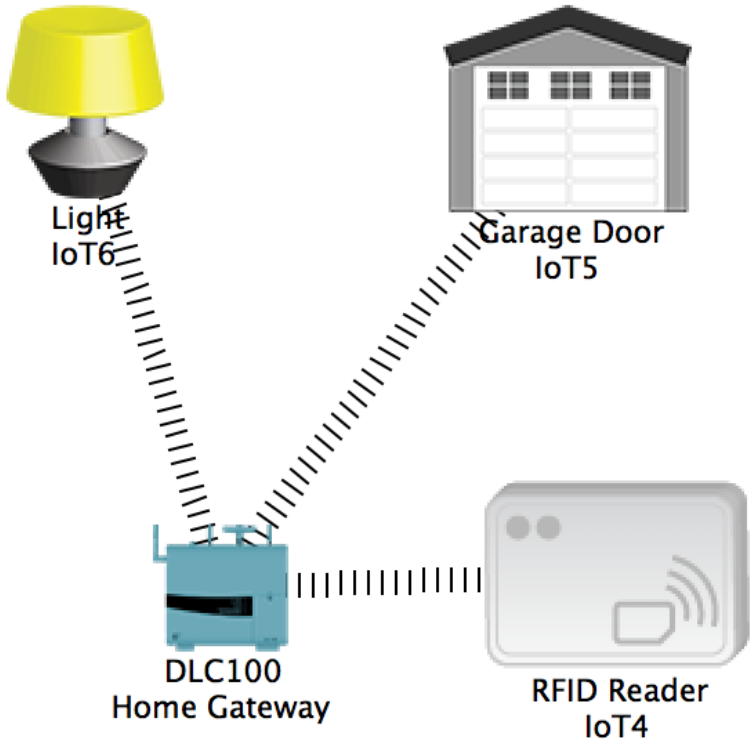
- The RFID reader is modelled as an ambient. The boundary of the ambient denotes the range of the reader. Therefore a RFID card is in range if it is a child ambient.

```
RFID_reader[
    ! recv().#recv(v).home_gateway::send(v).
       home_gateway::recv(status).send().0
  | send().0
  ]
```

Example: RFID reader (Cont'd)

Let's configure the server to control the lamp and the garage door automatically.
Suppose:

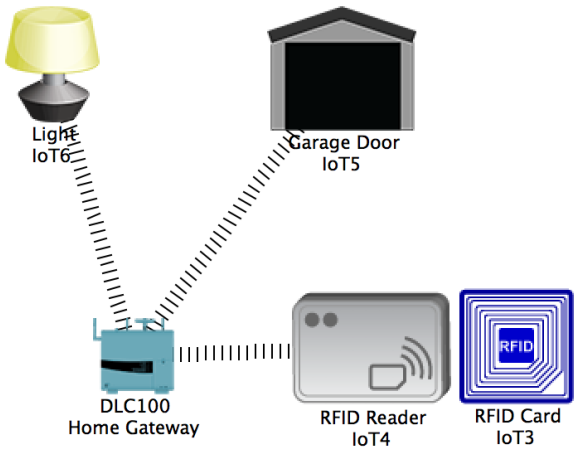
- A card ID is valid if it is between 100 and 200 exclusive.
- The garage door opens and the lamp is set to dim when a valid ID is detected.
- The devices are connected wirelessly to the server (home gateway).
- Download the program rfid_sim.cca from Blackboard.



Example: RFID reader (Cont'd)

- The server is specified as follows

```
home_gateway[
    !recv().RFID_reader::recv(v).if
       < v>100 and v<200 > RFID_reader::send(valid).if
          < not state(lamp, dim)> lamp::send(dim).0
          else skip.0
          fi. if < state(garage, closed)> garage::send(open).0
                 else skip.0 fi
     else RFID_reader::send(invalid).0  fi.send().0
   | send().0
]
```

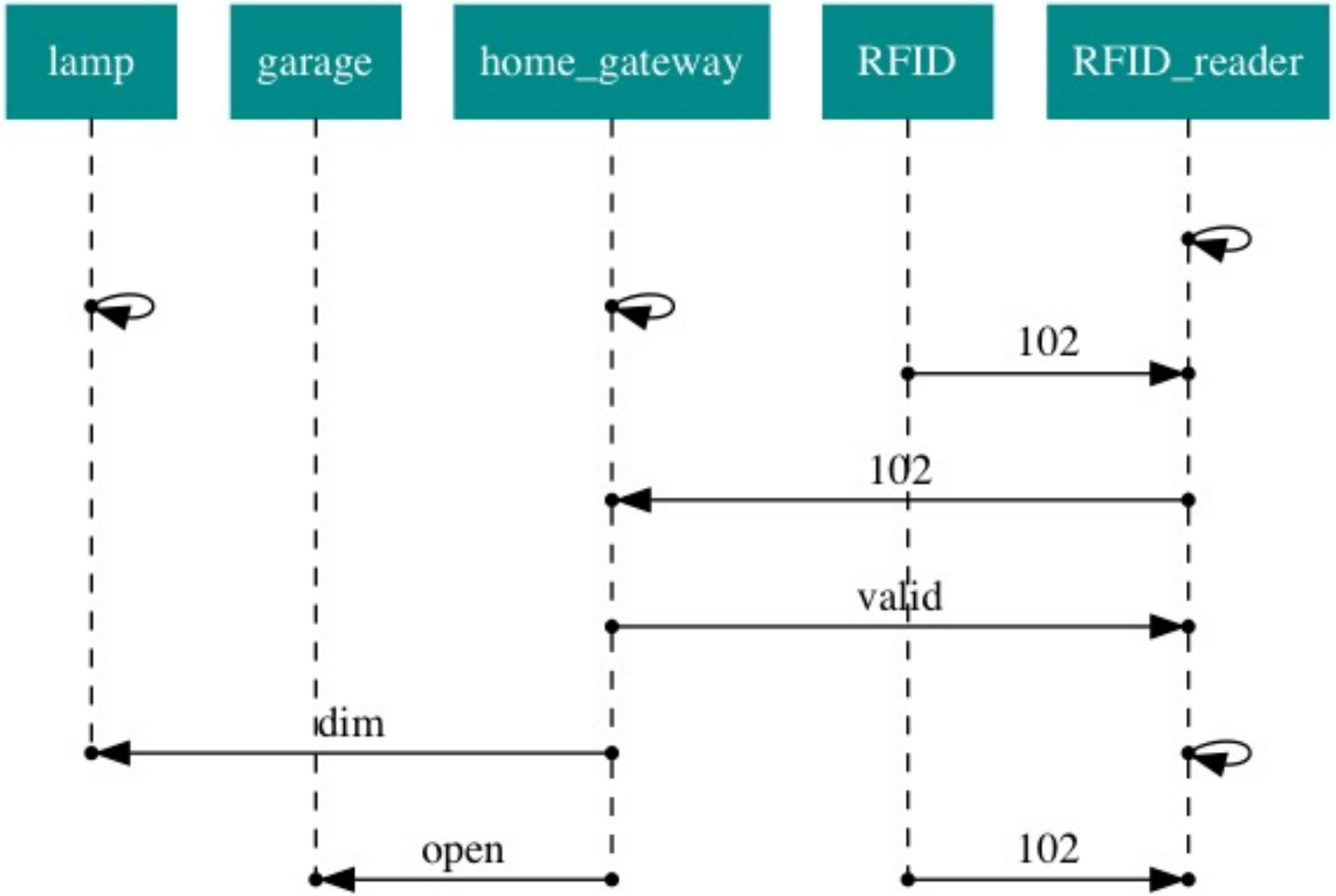


Example: RFID reader (Cont'd)

- The overall system system specification is the parallel composition of the components

```
garage[
  !::recv(x).if
     <x=closed and state(open)>del open.closed[0]
     <x=open and state(closed)>del closed.open[0]
  else skip.0
  fi.0
 | closed[0]  // Initially, the door is  closed.
]
|
lamp[ ... ]
|
RFID_reader[ ... ]
|
home_gateway[ ... ]
```

Example: RFID reader (Cont'd) – Simulation



Overview

Learning Outcomes

In this lesson you have learned:

- The syntax and the semantics of ccaPL programming language
- How to model IoT devices like smart lamp, smart door, and RFID tag and RFID reader
- How to build a network of IoT devices in CCA

Quiz

Answer by "true" or "false"

- The process {P} behaves like the process P.
- @is a capability.
- The process !P is the negation of the process P.
- The process 0 does nothing.
- The process skip does nothing.
- The process P | Q behaves like Q | P.
- The process P | Q | skip behaves like Q | P.
- The process !P behaves like P | !P.
- The process skip | skip behaves like skip.
- The process skip.skip behaves like skip.
- The process skip.**0** behaves like skip.
- The process skip | **0** behaves like skip.