The Calculus of Context-aware Ambients (CCA)

How to install the ccaPL Software Tool

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:

- How to install ccaPL software tool
- How to write a hello program in ccaPL
- How to execute a ccaPL program
- How to interpret the execution outputs.
- The ccaPL website is here:
  http://www.cse.dmu.ac.uk/~fsiewe/CCA/cca.html

Introduction

- CCA stands for Calculus of Context-aware Ambients.
- It is a process calculus for modelling and reasoning about the behaviours of context-aware and mobile concurrent systems.
- A process calculus is a formalism for the high-level description of interactions, communications, and synchronisations between a collection of independent processes.
- The main features of CCA are: **mobility**, **context-awareness**, and **concurrency**. The syntax and semantics of CCA can be found in [1,2,3].
- ccaPL is the programming environment of CCA.

DE MONTFORT UNIVERSITY LEICESTER

Installing ccaPL

1. Create a work directory
2. Download from Black Board the JAR file "ccaPL.jar" into your work directory
3. Make sure the most recent java virtual machine (JVM) is installed on your computer
4. This works with any operating system that supports JVM
5. Download and install the graph visualisation software "graphviz" from "http://www.graphviz.org/"
6. That is all you need to execute ccaPL programs.

How to Execute a Program?

1. Create a program file "myprog.cca" using your favorite text editor, e.g. emacs, gedit, or notepad++.
2. Save your program in your work directory containing the JAR file "ccaPL.jar"
3. Open a Terminal Window and use command line to change to your work directory
4. You can now execute your ccaPL program using the command:

```
java -jar ccaPL.jar -e myprog.cca
```

Save the execution output in a file

- The execution output is sent to the standard output, which is your computer screen by default.
- To save the execution output in a file "myoutput.txt",

```
java -jar ccaPL.jar -e myprog.cca > myoutput.txt
```

- To append sequence of execution outputs in a file "myoutput.txt",

```
java -jar ccaPL.jar -e myprog.cca > > myoutput.txt
```

My hello program (locally in the root ambient)

- hello program

```
send(hello).0 | recv(x).0
```

- execution output

```
      send(hello).0
   |  recv(x).0

--->  {Local: root ===(hello)===> root}

0
```

My hello program (locally in ambient n)

- hello program

```
n[ send(hello).0 | recv(x).0 ]
```

- execution output

```
      n[
            send(hello).0
         |  recv(x).0
      ]

--->  {Local: n ===(hello)===> n}

      n[ 0 ]
```

My hello program (sibling to sibling)

- hello program

```
n[ m::send(hello).0] | m[ n::recv(x).0 ]
```

- execution output

```
      n[ m::send(hello).0 ]
   |
      m[ n::recv(x).0 ]

--->  {Sibling to sibling: n ===(hello)===> m}

      n[ 0 ]
   |
      m[ 0 ]
```

My hello program (parent to child)

- hello program

```
n[ #send(hello).0 | m[ @recv(x).0 ] ]
```

- execution output

```
      n[
            m[ @recv(x).0 ]
         |  #send(hello).0
      ]

--->  {Parent to child: n ===(hello)===> m}

      n[
            m[ 0 ]
      ]
```

My hello program (child to parent)

- hello program

```
n[ #recv(x).0 | m[ @send(hello).0 ] ]
```

- execution output

```
      n[
            m[ @send(hello).0 ]
         |  #recv(x).0
      ]

--->  {Child to parent: m ===(hello)===> n}

      n[
            m[ 0 ]
      ]
```

My hello+ack program

- hello+ack program

```
n[ ::send(hello).::recv(ack).0]
| m[ ::recv(x).::send(hello).0 ]
```

- execution output

```
      n[ ::send(hello).::recv(ack).0 ]
   |
      m[ ::recv(x).::send(hello).0 ]
--->  {Sibling to sibling: n ===(hello)===> m}

      n[ ::recv(ack).0 ]
   |
      m[ ::send(thanks).0 ]
--->  {Sibling to sibling: m ===(hello)===> n}

      n[ 0 ]
   |
      m[ 0 ]
```

Structure of a Program

```
BEGIN_DECLS

  // Definitions of context-expressions go here.
  // Also, the execution directives are specified here.

END_DECLS

//Body of the prgram goes here.
// The body of program is a process.
```

Example of Program

```
BEGIN_DECLS

  def isHello(n) = {n = hello}

END_DECLS

   send(hello).recv(ack).0
   | recv(x).if <isHello(x)>send(thanks).0
               else skip.0
               fi
```

Example of Program with execution directives

```
BEGIN_DECLS
  // Declaration of context expressions
  def has(n) = { somewhere (this | n[true] | true)}
  def isHello(n) = {n = hello}
 // Declaration of execution directives
  mode random
  display code
  display congruence
END_DECLS
  // main process
   send(hello).recv(ack).0
   | recv(x).if <isHello(x)>send(thanks).0
              else skip.0 fi
```

Example of Program without definitions nor directives

```
BEGIN_DECLS

END_DECLS
   send(hello).recv(ack).0
   | recv(x).if <x = hello>send(thanks).0
               else skip.0
               fi
```

OR JUST:

```
   send(hello).recv(ack).0
   | recv(x).if <x = hello>send(thanks).0
               else skip.0
               fi
```

Declaration Section

- begins with the keyword BEGIN_DECLS, all in capital
- ends with the keyword END_DECLS, all in capital
- this section is optional
- each declaration has the for:

$$\texttt{def}\ f(x_1, \ldots, x_n) = \{\kappa\}$$

where:

- def is a keyword
- f is the name of the function being declared
- $x_1, \ldots, x_n$ is a finite list of formal parameters; can be empty
- κ is a normalised context expression

Example of Declarations

```
BEGIN_DECLS
  def has(n) = { somewhere (this | n[true] | true) }
  def at(n) = { somewhere (n[next(this | true)] | true) }
  def with(n) = { somewhere (n[true] | next (this | true)) }
  def at(n, m) = { somewhere (n[m[true] | true] | true)}
  def with(n, m) = { somewhere (n[true] | m[true] | true) }
END_DECLS
```

Execution Directives

Two execution **modes** are implemented which ensure fair execution of concurrent processes.

- `mode random`
  For each reduction, the pair of process to be reduced is chosen randomly (with uniform distribution) among all the processes that are willing to execute.
- `mode interleaving`
  For each reduction, the pair of process to be reduced is chosen based on two criteria:
  - how long the pair has been willing to execute (first-in, first-out), and
  - in case of conflict, the sequential order as they appear in the program text is used.
- The default mode is `interleaving`.

Execution Directives (cont'd)

The directive **display** filters the output of process execution.

- `display code`
  Display the program code (except comments) after each reduction.
- `display congruence`
  Display in addition to reduction steps (denoted by `-->`), the congruence steps (denoted by `<-->`).
- By default code and congruence are not displayed in output.
- `length = xx`
  Limit the length of execution to xx transitions (ccaPL version 5.0 or later).

Body and Comments

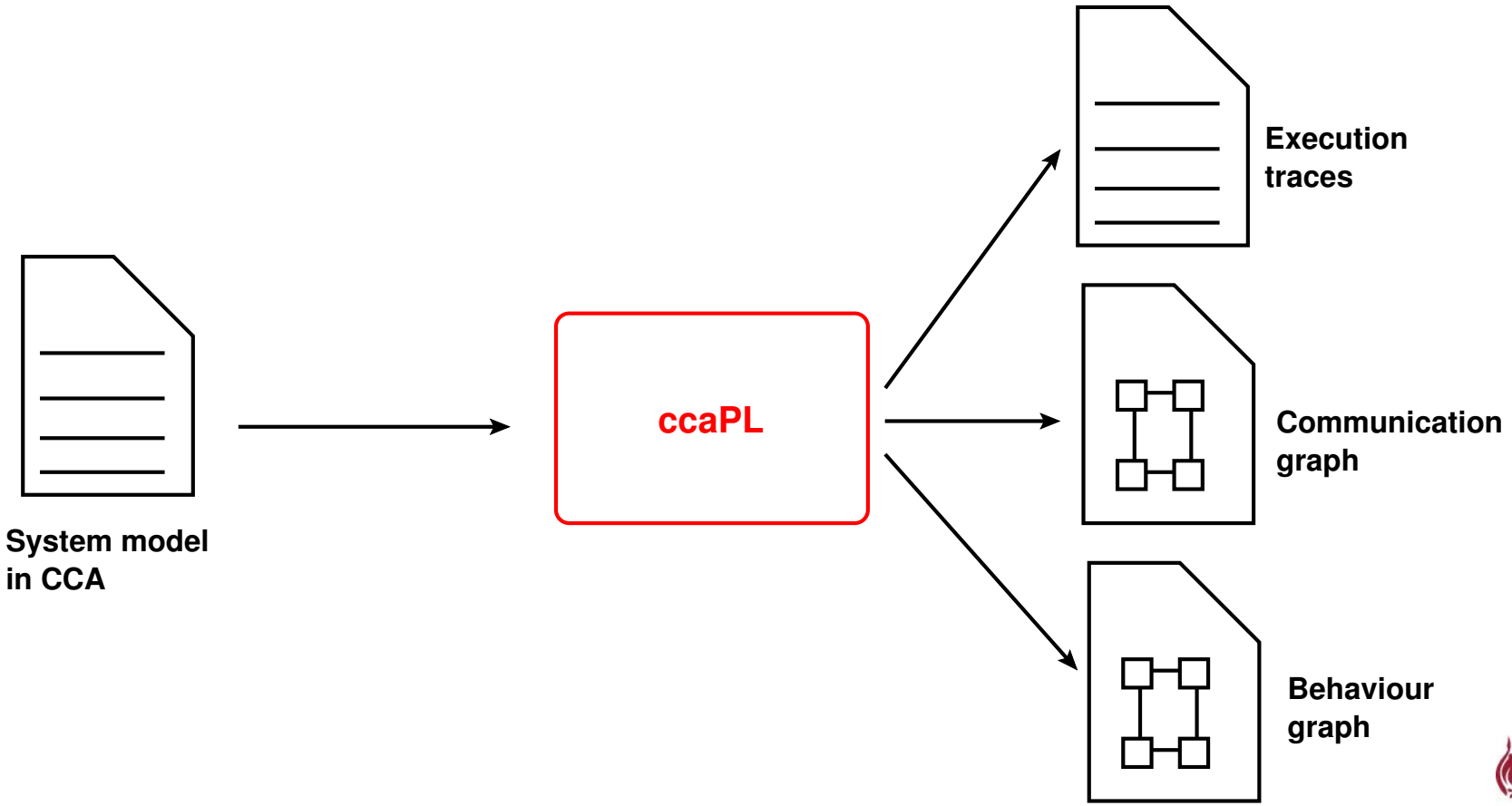
- The body of a program is a single process
- A multiline comment is enclosed between “/*” and “*/”
- A single line comment follows “//”
- A single line comment must end with a new line or a carriage return
- Finally, the ccaPL parser is CASE SENSITIVE, i.e. “hello” and “Hello” are treated as different messages, for example.

Example of Program with comments

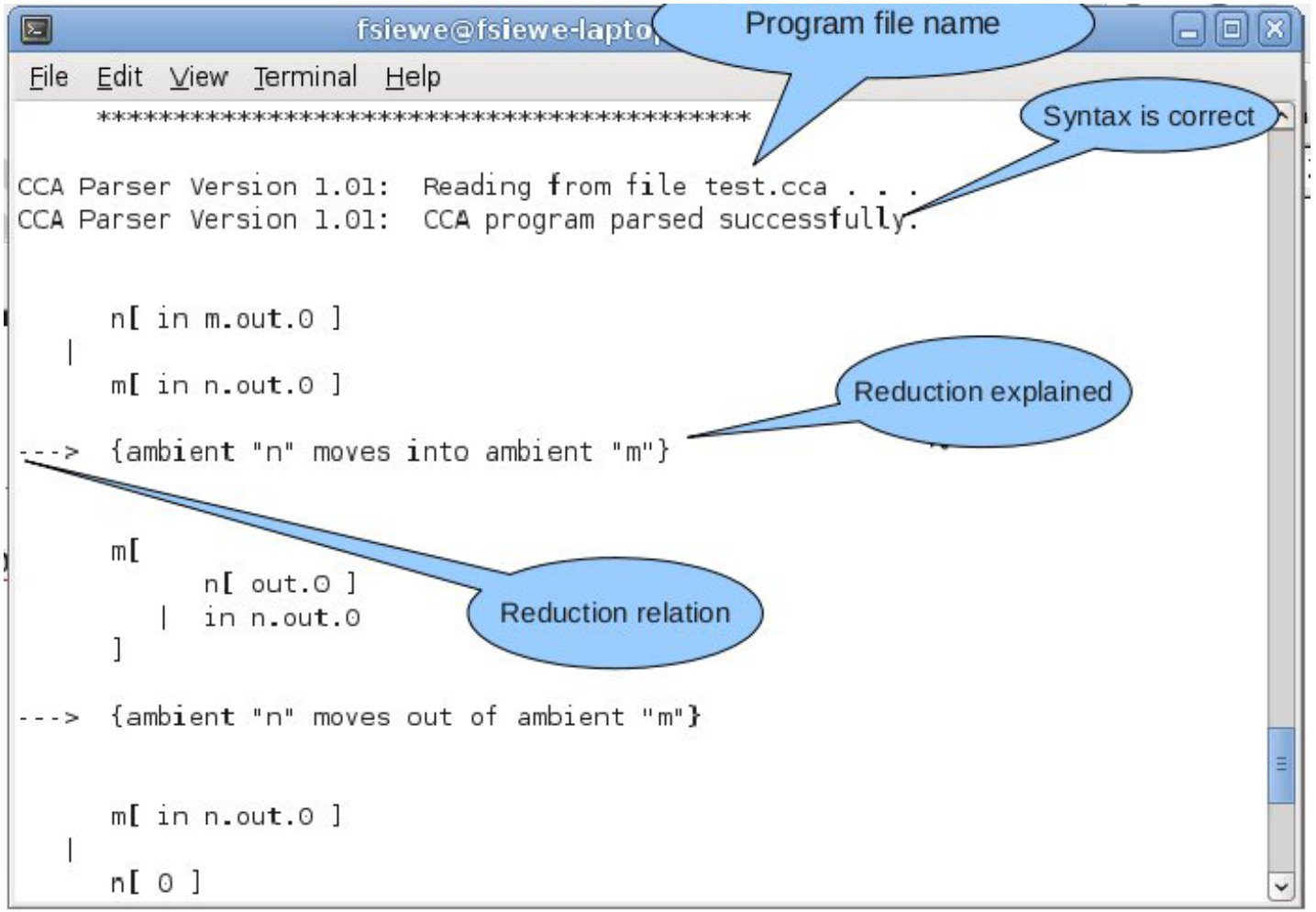
```
/* this is a multi-line comments, i.e.
   a comment that expands across many lines.
*/
BEGIN_DECLS // this section is optional
  def has(n) = {somewhere (this | n[true] | true)}
  def at(n)  = {somewhere (n[next (this | true)] | true)}
END_DECLS
//put here the body of the program
n[
  find x: has(x) for x#send(hello).x#recv(y).0
| m[ <at(n)>n@recv(x).
      <x=hello>@send(thanks).0 ]
]
```

Execution Output

ccaPL generates the execution output in textual format and graphical format.



Understanding Execution Output



Understanding Execution Output (cont'd)

- The symbol '<-->' corresponds to the structural congruence relation as formally defined in [1, 2]
- '-->' represents the reduction relation (see [1, 2])
- The explanation of each transition is given between a pair of curly brackets.
- The notation 'A ==X==> B' means that an ambient 'A' has sent a message 'X' to another ambient 'B'.
- Other notations such as 'Child to parent' and 'Sibling to sibling' provide information about the relationship between the sender 'A' and the receiver 'B'.
- the notation {binding: n -> IN1} corresponds to the execution of the operator find and means that the name IN1 has been found for n.
- ...

Understanding Execution Output (cont'd)

Summary of Execution Output Notations	
Notation	Description
`<-->`	is equivalent to
`-->`	reduces to (i.e. one execution step)
{ ... }	explanation of each execution step
`A ==X==> B`	'A' sends 'X' to 'B'
`Child to parent`	Child to parent communication
`Parent to child`	Parent to child communication
`Sibling to sibling`	communication between siblings
`{binding: n -> IN1}`	execution of the operator `find`
...	other messages are self-explaining.

Error Messages

- Programs are first parsed for syntax errors and then executed if no error is detected.
- When an error is detected, the parser displays an error message then terminates. Error messages have the following form:

```
CCA Parser Version 3.0 :  Reading from file prg2.cca . . .
CCA Parser Version 3.0 :  Encountered errors during parse.
ParseException: Encountered " <NAME> "y "" at line 8, column 3.
Was expecting one of:
    "def" ...
    "END_DECLS" ...
    "display" ...
    "mode" ...
```

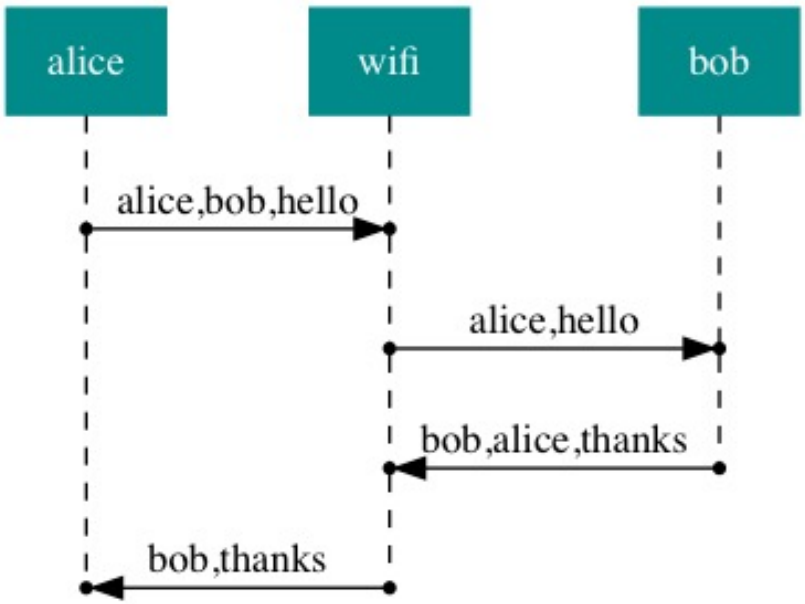
- Note that line number and column at which the error occurs are given, as well as a list of expected tokens.

Generating the Execution Graphs

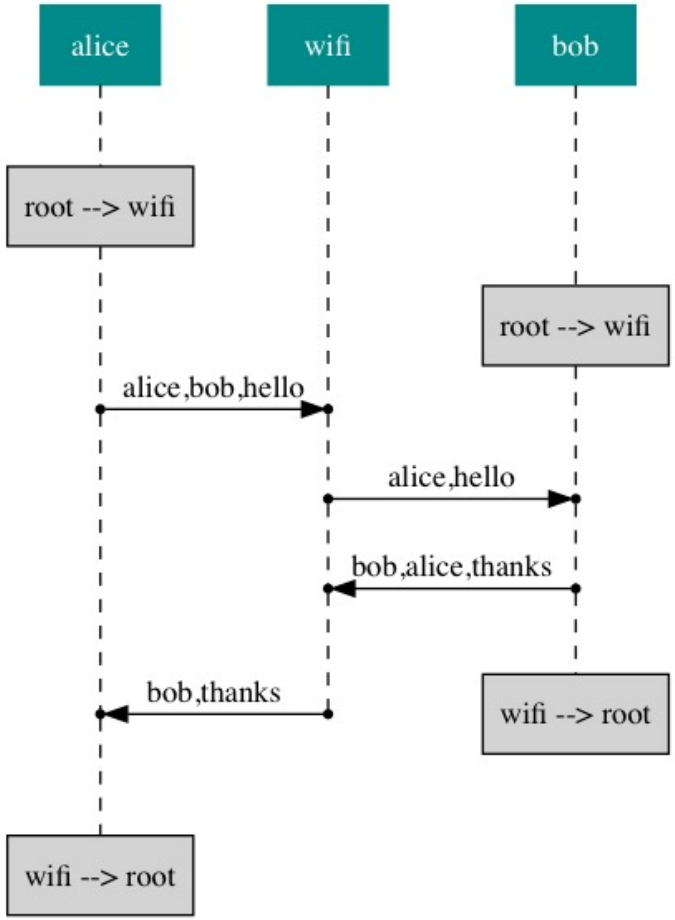
The simulator ccaPL also displays the execution output graphically in the form of a diagram, which enables the user to visualise the concurrent behaviours of the system.
Two types of graphs can be produced:

- a communication graph depicting message passing between ambients;
- a behaviour graph showing both the movements of ambients and the communications between them.

Generating the Execution Graphs (Cont'd)



Communication graph



Behaviour graph

Generating the Execution Graphs (Cont'd)

- The top row in both diagrams lists the names of the ambients involved in the system being executed.
- The vertical dashed line represents the timeline for each ambients (time increases from top to bottom).
- A solid directed line from the timeline of an ambient B to that of an ambient C indicates a message passing from the ambient B to the ambient C at a specific time point, with the content of the message carried as a label to that line.
- A behaviour graph provides additional information about the mobility of ambients. A box labelled as "X - -> Y" on the timeline of an ambient B indicates that the ambient B has moved from location X to location Y.

Generating the Execution Graphs (Cont'd)

- Download and install the graph visualisation software "graphviz" from "http://www.graphviz.org/"
- To generate only the communication graph, use the command:

```
java -jar ccaPL.jar -g myprog.cca
```

  The generated graph is stored in the file myprog.jpg.
- To generate both the communication graph and the behaviour graph, use the command:

```
java -jar ccaPL.jar -gx myprog.cca
```

  The generated graphs are stored in the file myprog_0.jpg and myprog_1.jpg respectively.

Generating the Execution Graphs (Cont'd)

- From ccaPL 5.06 other image formats can be generated as follows, where XXX can be ps, eps, jpg, pdf, png, tiff, and so on.
- To generate only the communication graph, use the command:

```
java -jar ccaPL.jar -gXXX myprog.cca
```

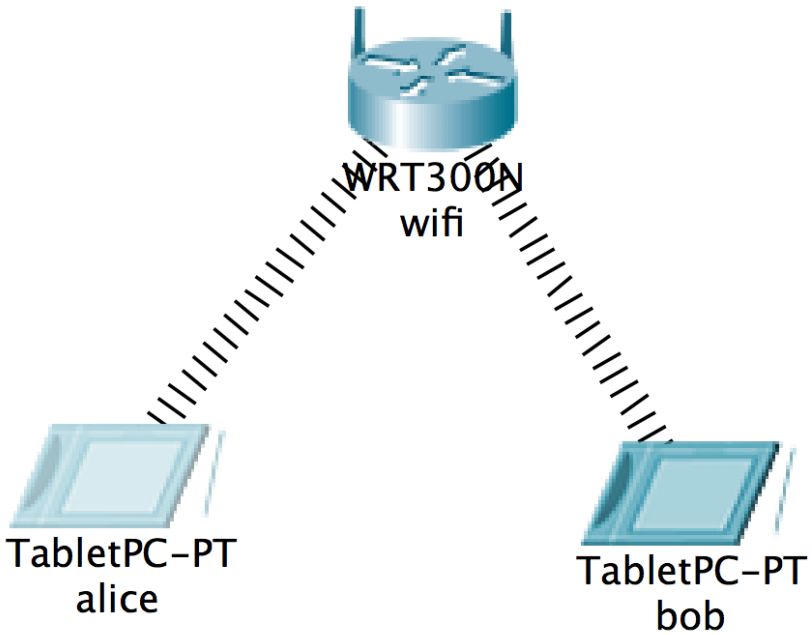
  The generated graph is stored in the file myprog.XXX.
- To generate both the communication graph and the behaviour graph, use the command:

```
java -jar ccaPL.jar -gxXXX myprog.cca
```

  The generated graphs are stored in the file myprog_0.XXX and myprog_1.XXX respectively.

Example: a simple wifi access point

- The wifi access point can be modelled as an ambient named wifi.
- The boundary of the ambient denotes the range of wifi access point.
- Therefore a device is in the range of the wifi if that device is a child ambient to wifi.



Exercise: a simple wifi access point

- Download the program wifi.cca
- Inspect the program code and try to understand what it does
- Execute the program to generate the textual execution output
- Interpret the execution output
- Execute the program to generate the communication graph
- Execute the program to generate the behaviour graph
- Modify the program to generate an error

Summary

Learning Outcomes

In this lesson you have learned:

- How to install ccaPL software tool
- How to write a hello program in ccaPL
- How to execute a ccaPL program
- How to interpret the execution outputs.
- The ccaPL website is here:
  http://www.cse.dmu.ac.uk/~fsiewe/CCA/cca.html

Quiz

- What does CCA stand for?
- What is ccaPL?
- Answer by "true" or "false"
  - ccaPL is a java program.
  - CCA is a java program.
  - ccaPL is a C++ program.
  - CCA is a process caluclus.
- How to generate a communication graph?
- How to generate a behaviour graph?
- What is the difference between the 2 types of graphs?

References

1. http://www.cse.dmu.ac.uk/~fsiewe/CCA/cca.html
2. F. Siewe et al. (2009). CCA: a Calculus of Context-aware Ambients. *IEEE 23rd International Conference on Advanced Information Networking and Applications (AINA-09)*, University of Bradford, Bradford, UK, May 26-29.
3. F. Siewe et al. (2010). The Calculus of Context-aware Ambients. *Journal of Computer and Systems Sciences*, 77(4), 597-620.
4. F. Siewe et al. (2016). Analysing Use Case Diagrams in a Calculus of Context-aware Ambients. *International Journal of Intelligent Computing Research, Infonomics Society*, 7(1), 655-662.
5. F. Siewe et al. (2016). Privacy Protection by Typing in Ubiquitous Computing Systems. *Journal of Systems and Software*, 120, 133-153.

DE MONTFORT UNIVERSITY LEICESTER