Languages for Global Computing

Paolo Baldan

Corso di Laurea Magistrale in Informatica
Università di Padova

2017/2018

What are we talking about?

What are we talking about?

Global computing

- Widespread diffusion of computing devices
- interconnected by a network infrastructure
- abstractly seen as global computer

What are we talking about?

Global computing

- Widespread diffusion of computing devices
- interconnected by a network infrastructure
- abstractly seen as global computer

What are we talking about?

Global computing

- Widespread diffusion of computing devices
- interconnected by a network infrastructure
- abstractly seen as global computer

The world of software is getting more and more complex

- concurrency
- distribution
- mobility
- dynamicity
- open endedness
- ...

The world of software is getting more and more complex

- concurrency
- distribution
- mobility
- dynamicity
- open endedness
- ...

The world of software is getting more and more complex

- concurrency
- distribution
- mobility
- dynamicity
- open endedness
- ...

The world of software is getting more and more complex

- concurrency
- distribution
- mobility
- dynamicity
- open endedness
- ...

The world of software is getting more and more complex

- concurrency
- distribution
- mobility
- dynamicity
- open endedness
- ...

The world of software is getting more and more complex

- concurrency
- distribution
- mobility
- dynamicity
- open endedness
- ...

Old and new problems

Good old concurrency problems . . .

- deadlock
- starvation
- fairness
- . . .

. . . and many others

- connectivity
- remote failures
- security
- resource control
- . . .

Old and new problems

Good old concurrency problems . . .

- deadlock
- starvation
- fairness
- . . .

. . . and many others

- connectivity
- remote failures
- security
- resource control
- . . .

An old story . . .

Concurrency is a theme since the early years of CS . . .

- multitasking/multiuser operating systems (processes sharing cpu/memory/devices . . .)
- multiuser databases (with concurrent transaction on common data)

. . . with (continuously) renewed interest

- with the advent of the internet
- with enormous growth of interconnected computing devices (laptops, smartphones, embedded devices, . . .)
- with multicore CPUs
- . . .

An old story . . .

Concurrency is a theme since the early years of CS . . .

- multitasking/multiuser operating systems (processes sharing cpu/memory/devices . . .)
- multiuser databases (with concurrent transaction on common data)

. . . with (continuously) renewed interest

- with the advent of the internet
- with enormous growth of interconnected computing devices (laptops, smartphones, embedded devices, . . .)
- with multicore CPUs
- . . .

The world is concurrent

Computing is pervasive in the world

Computing needs to be concurrent

[Aristotle (almost)]

The world is concurrent

Computing is pervasive in the world

Computing needs to be concurrent

[Aristotle (almost)]

The world is concurrent

Computing is pervasive in the world

Computing needs to be concurrent

[Aristotle (almost)]

Concurrency is everywhere

it is very useful

but complex!

We need help!

Concurrency is everywhere

it is very useful

but complex!

We need help!

Complexity: technological advances

The technological progress is tumultuous . . .

Each second day we have new

- programming languages
- tools
- paradigms
- architectural solutions
- . . .

We need conceptual tools to be guided in the "technological forest"!

Complexity: technological advances

The technological progress is tumultuous . . .

Each second day we have new

- programming languages
- tools
- paradigms
- architectural solutions
- . . .

We need conceptual tools to be guided in the "technological forest"!

Complexity: too many details!

Example: Producer-Consumer

A process producing data and one using such data (asynchronously)

Complexity: too many details!

Example: Producer-Consumer

A process producing data and one using such data (asynchronously)


```
public class ProducerConsumerTest {
   public static void main(String[] args) {
      CubbyHole c = new CubbyHole();
      Producer p1 = new Producer(c, 1);
      Consumer c1 = new Consumer(c, 1);
      p1.start();
      c1.start();
   }
}
class CubbyHole {
   private int contents;
   private boolean available = false;

   public synchronized int get() {
      while (available == false) {
         try {
            wait();
         } catch (InterruptedException e) {}
      }
      available = false;
      notifyAll();
      return contents;
   }
   public synchronized void put(int value) {
      while (available == true) {
         try {
            wait();
         } catch (InterruptedException e) { }
      }
      contents = value;
      available = true;
      notifyAll();
   }
```

```
class Consumer extends Thread {
   private CubbyHole cubbyhole;
   private int number;

   public Consumer(CubbyHole c, int number) {
      cubbyhole = c;
      this.number = number;
   }
   public void run() {
      int value = 0;
      for (int i = 0; i < 10; i++) {
         value = cubbyhole.get();
         System.out.println("Consumer #" + this.number + " got: " + value);
      }
   }
}
class Producer extends Thread {
   private CubbyHole cubbyhole;
   private int number;
   public Producer(CubbyHole c, int number) {
      cubbyhole = c;
      this.number = number;
   }
   public void run() {
      for (int i = 0; i < 10; i++) {
         cubbyhole.put(i);
         System.out.println("Producer #" + this.number + " put: " + i);
         try {
            sleep((int)(Math.random() * 100));
         } catch (InterruptedException e) { }
      }
```

Complexity: conceptual!

A problem

- We want to buy a new laptop, with a budget of 1000 euros.
- We collect a (long) list of e-shops where to look for

Our goal

Develop a program for querying each e-shop until one offering a laptop with the right price is found.

A brilliant idea: Let's go concurrent!

To speed up the program we split the list in two pieces and we run two programs in parallel, each taking care of one sublist.

Complexity: conceptual!

A problem

- We want to buy a new laptop, with a budget of 1000 euros.
- We collect a (long) list of e-shops where to look for

Our goal

Develop a program for querying each e-shop until one offering a laptop with the right price is found.

A brilliant idea: Let's go concurrent!

To speed up the program we split the list in two pieces and we run two programs in parallel, each taking care of one sublist.

Complexity: conceptual!

A problem

- We want to buy a new laptop, with a budget of 1000 euros.
- We collect a (long) list of e-shops where to look for

Our goal

Develop a program for querying each e-shop until one offering a laptop with the right price is found.

A brilliant idea: Let's go concurrent!

To speed up the program we split the list in two pieces and we run two programs in parallel, each taking care of one sublist.

Complexity: conceptual!

A problem

- We want to buy a new laptop, with a budget of 1000 euros.
- We collect a (long) list of e-shops where to look for

Our goal

Develop a program for querying each e-shop until one offering a laptop with the right price is found.

A brilliant idea: Let's go concurrent!

To speed up the program we split the list in two pieces and we run two programs in parallel, each taking care of one sublist.

Concurrency is complex: conceptually!

The same problem, abstractly

Let f be a (computationally expensive) function from integers to integers

Our goal

Develop a program that terminates iff function f has a non-null zero, i.e., there is $x \neq 0$ such that $f(x) = 0$, and proceeds indefinitely otherwise.

A brilliant idea: Let's go concurrent!

Define

- A positive zero an integer $n > 0$ such that $f(n) = 0$
- A negative zero an integer $z < 0$ such that $f(z) = 0$

To speed up we run in parallel two programs, one looking for a positive zero and the other for a negative zero

Concurrency is complex: conceptually!

The same problem, abstractly

Let f be a (computationally expensive) function from integers to integers

Our goal

Develop a program that terminates iff function f has a non-null zero, i.e., there is $x \neq 0$ such that $f(x) = 0$, and proceeds indefinitely otherwise.

A brilliant idea: Let's go concurrent!

Define

- A positive zero an integer $n > 0$ such that $f(n) = 0$
- A negative zero an integer $z < 0$ such that $f(z) = 0$

To speed up we run in parallel two programs, one looking for a positive zero and the other for a negative zero

Concurrency is complex: conceptually!

The same problem, abstractly

Let f be a (computationally expensive) function from integers to integers

Our goal

Develop a program that terminates iff function f has a non-null zero, i.e., there is $x \neq 0$ such that $f(x) = 0$, and proceeds indefinitely otherwise.

A brilliant idea: Let's go concurrent!

Define

- A positive zero an integer $n > 0$ such that $f(n) = 0$
- A negative zero an integer $z < 0$ such that $f(z) = 0$

To speed up we run in parallel two programs, one looking for a positive zero and the other for a negative zero

Attempt 1

A program T1 that looks for a positive zero

```
found=false
n = 0
while (not found)                    T1
   n++
   found = (f(n) == 0)
```

. . . and T2 that looks for a negative zero, by cut-and-paste

```
found=false
z = 0
while (not found)                    T2
   z--
   found = (f(z) == 0)
```

And run T1 and T2 in parallel:

```
T1 | T2
```

Attempt 1

A program T1 that looks for a positive zero

```
found=false
n = 0
while (not found)                    T1
   n++
   found = (f(n) == 0)
```

. . . and T2 that looks for a negative zero, by cut-and-paste

```
found=false
z = 0
while (not found)                    T2
   z--
   found = (f(z) == 0)
```

And run T1 and T2 in parallel:

```
T1 | T2
```

Attempt 1

A program T1 that looks for a positive zero

```
found=false
n = 0
while (not found)                    T1
   n++
   found = (f(n) == 0)
```

. . . and T2 that looks for a negative zero, by cut-and-paste

```
found=false
z = 0
while (not found)                    T2
   z--
   found = (f(z) == 0)
```

And run T1 and T2 in parallel:

```
T1 | T2
```

Attempt 1, contd.

T1

```
found=false
n = 0
while (not found)
   n++
   found = (f(n) == 0)
```

T2

```
found=false
z = 0
while (not found)
   z--
   found = (f(z) == 0)
```

Attempt 1, contd.

T1

```
found=false
n = 0
while (not found)
   n++
   found = (f(n) == 0)
```

T2

```
found=false
z = 0
while (not found)
   z--
   found = (f(z) == 0)
```

Wrong!

If f has only a positive zero and T1 terminates before T2 starts, the latter sets `found` to `false` and looks indefinitely for the nonexisting negative zero.

Attempt 1, contd.

T1

```
found=false
n = 0
while (not found)
   n++
   found = (f(n) == 0)
```

T2

```
found=false
z = 0
while (not found)
   z--
   found = (f(z) == 0)
```

Wrong!

If f has only a positive zero and T1 terminates before T2 starts, the latter sets `found` to `false` and looks indefinitely for the nonexisting negative zero.

Idea

The problem is the fact that `found` is initialized to `false` twice.

Attempt 2

A solution that initializes found only once

```
found=false; (T1 | T2)
```

where

T1

```
n = 0
while (not found)
   n++
   found = (f(n) == 0)
```

T2

```
z = 0
while (not found)
   z--
   found = (f(z) == 0)
```

Attempt 2

A solution that initializes found only once

```
found=false; (T1 | T2)
```

where

T1

```
n = 0
while (not found)
   n++
   found = (f(n) == 0)
```

T2

```
z = 0
while (not found)
   z--
   found = (f(z) == 0)
```

Wrong!

If f has (again) only a positive zero assume that:

1. T2 just enters the while body and is preempted
2. T1 computes till it finds the positive zero
3. T2 gets the CPU back, set `found` to `false` and loop forever
4. The program does not terminate!!!!

Attempt 2, ctd.

Idea

The problem is the fact that `found` is set to `false` after it has been already set to `true`.

Attempt 3

Avoid assigning found to false in T1 and T2

```
found=false; (T1 | T2)
```

where

T1

```
n = 0
while (not found)
   n++
   if (f(n) == 0)
      found=true
```

T2

```
z = 0
while (not found)
   z--
   if (f(z) == 0)
      found=true
```

Attempt 3

Avoid assigning found to false in T1 and T2

```
found=false; (T1 | T2)
```

where

T1

```
n = 0
while (not found)
   n++
   if (f(n) == 0)
      found=true
```

T2

```
z = 0
while (not found)
   z--
   if (f(z) == 0)
      found=true
```

Wrong!

If f has (again) only a positive zero, it can happen that

1. T2 gets the CPU to keep it forever
2. T1 will never have the chance of finding the positive zero
3. The program does not terminate!!!!

Attempt 3 - ctd

Idea:

This problem is due to non fair scheduling policies.

Attempt 4

Fairness with token passing

```
turn=1; found=false; (T1 | T2)
```

where

T1

```
n = 0
while (not found)
  wait (turn == 1)
  turn=2
  n++
  if (f(n) == 0)
     found=true
```

T2

```
z = 0
while (not found)
   wait (turn == 2)
   turn=1
   z--
   if (f(z) == 0)
      found=true
```

Attempt 4

Fairness with token passing

```
turn=1; found=false; (T1 | T2)
```

where

T1

```
n = 0
while (not found)
  wait (turn == 1)
  turn=2
  n++
  if (f(n) == 0)
     found=true
```

T2

```
z = 0
while (not found)
   wait (turn == 2)
   turn=1
   z--
   if (f(z) == 0)
      found=true
```

Wrong!

If T1 finds a zero and stops when T2 has already set turn to 1, then T2 would be blocked by the wait command because the value of turn cannot be changed.

Attempt 4 - ctd.

Idea

The program does not terminate because of the waiting of an impossible event: on termination care is needed for other processes.

Attempt 5

On termination, enable the other process

```
turn=1; found=false; (T1; turn=2 | T2; turn=1)
```

where

T1

```
n = 0
while (not found)
  wait (turn == 1)
  turn=2
  n++
  if (f(n) == 0)
     found=true
```

T2

```
z = 0
while (not found)
   wait (turn == 2)
   turn=1
   z--
   if (f(z) == 0)
      found=true
```

Is this correct?

Looks like, but we are unsure!!!

Attempt 5

On termination, enable the other process

```
turn=1; found=false; (T1; turn=2 | T2; turn=1)
```

where

T1

```
n = 0
while (not found)
  wait (turn == 1)
  turn=2
  n++
  if (f(n) == 0)
     found=true
```

T2

```
z = 0
while (not found)
   wait (turn == 2)
   turn=1
   z--
   if (f(z) == 0)
      found=true
```

Is this correct?

Looks like, but we are unsure!!!

We need help!

We need somebody's help!

The proposal in this course

Adopt a rigorous, solidly grounded approach to the study of concurrency as (one of your) tools for understanding, designing and programming systems.

Program

- Start from foundations
- and study how they reflect on languages and programming

Benefits at two levels . . .

We need somebody's help!

The proposal in this course

Adopt a rigorous, solidly grounded approach to the study of concurrency as (one of your) tools for understanding, designing and programming systems.

Program

- Start from foundations
- and study how they reflect on languages and programming

Benefits at two levels ...

We need somebody's help!

The proposal in this course

Adopt a rigorous, solidly grounded approach to the study of concurrency as (one of your) tools for understanding, designing and programming systems.

Program

- Start from foundations
- and study how they reflect on languages and programming

Benefits at two levels ...

We need somebody's help!

The proposal in this course

Adopt a rigorous, solidly grounded approach to the study of concurrency as (one of your) tools for understanding, designing and programming systems.

Program

- Start from foundations
- and study how they reflect on languages and programming

Benefits at two levels ...

Conceptual level

- A foundational approach that identifies the basic operators and constructs of concurrency
- helps in understanding the multiplicity of languages, architectures, paradigms
- which are reduced to a bunch of fundamental principles.

Think of what you've done for

- imperative
- functional
- object-oriented

Conceptual level

- A foundational approach that identifies the basic operators and constructs of concurrency
- helps in understanding the multiplicity of languages, architectures, paradigms
- which are reduced to a bunch of fundamental principles.

Think of what you've done for

- imperative
- functional
- object-oriented

Practical level

Errors, errors and errors

Software is error-prone and even small concurrent programs can be hard to understand and analyse

A formal framework comes along with techniques for

- design of systems
- specification of the desired properties
- verification, assisted or automatic.

- Not for free: syntax and semantics have to be defined rigorously
- But rewarding!

Practical level

Errors, errors and errors

Software is error-prone and even small concurrent programs can be hard to understand and analyse

A formal framework comes along with techniques for

- design of systems
- specification of the desired properties
- verification, assisted or automatic.

- Not for free: syntax and semantics have to be defined rigorously
- But rewarding!

Practical level

Errors, errors and errors

Software is error-prone and even small concurrent programs can be hard to understand and analyse

A formal framework comes along with techniques for

- design of systems
- specification of the desired properties
- verification, assisted or automatic.

- Not for free: syntax and semantics have to be defined rigorously
- But rewarding!

We are not alone!

- No longer confined to the academia
- Formal techniques, like model checking and abstract interpretation, are commonly used (and investigated) by the software giants (**Microsoft**, **Apple**, **Facebook**, **Google**)

New foundations?

Why new foundations?

We already know that in any (imperative) language we can find constructs

- assignements (`x = expr`)
- control (conditionals: `if`, `case`, iterations: `while`, `for`, ...)
- structuring, encapsulation

Sequential behaviour

A sequential program `P` implements a function:

$$[[P]] : \text{inputs} \rightarrow \text{outputs}$$

Example: Factorial

```
fact(n):
   res=1
   while (n > 0)
      res = res * n
      n--
   return res
```

Why new foundations?

We already know that in any (imperative) language we can find constructs

- assignements (`x = expr`)
- control (conditionals: `if`, `case`, iterations: `while`, `for`, ...)
- structuring, encapsulation

Sequential behaviour

A sequential program `P` implements a function:

$$[[P]] : \text{inputs} \rightarrow \text{outputs}$$

Example: Factorial

```
fact(n):
   res=1
   while (n > 0)
      res = res * n
      n--
   return res
```

Why new foundations?

We already know that in any (imperative) language we can find constructs

- assignements (`x = expr`)
- control (conditionals: `if`, `case`, iterations: `while`, `for`, ...)
- structuring, encapsulation

Sequential behaviour

A sequential program `P` implements a function:

$$[[P]] : \text{inputs} \rightarrow \text{outputs}$$

Example: Factorial

```
fact(n):
   res=1
   while (n > 0)
      res = res * n
      n--
   return res
```

Why new foundations?

In the Seventies they were wondering the same ...

Robin Milner

Tony Hoare

Turing award winners

Sequential programs

Example: Factorial

```
fact(n):
   res=1
   while (n > 0)
      res = res * n
      n--
   return res
```

- Non termination is BAD!
- Output is UNIQUELY determined by input!
  Actually, each step is uniquely determined by the memory

Sequential programs

Example: Factorial

```
fact(n):
   res=1
   while (n > 0)
      res = res * n
      n--
   return res
```

- Non termination is BAD!
- Output is UNIQUELY determined by input!
  Actually, each step is uniquely determined by the memory

Concurrent programs: Output not unique

The situation changes radically for concurrent programs!

Example: A strange program ...

```
   strange (x):
     set2(x) | set2(x)
     return x
where
   set2 (x):
     x=2
```

What does `strange(x)` compute?

Concurrent programs: Output not unique

Example: An even stranger program ...

```
   strangenew (x):
     set2(x) | set2new(x)
     return x
where
   set2new (x):
     x=0
     x=x+2
```

What does `strange-new(x)` compute?

Concurrent programs: Output not unique, contd.

An execution of `strangenew(x)`

`set2`	`set2new`
	`x=0`
`x=2`	
	`x=x+2`

We can get 2 but also 4 . . .

Concurrent programs: Output not unique, contd.

An execution of `strangenew(x)`

`set2`	`set2new`
	`x=0`
`x=2`	
	`x=x+2`

We can get 2 but also 4 ...

Concurrent programs: Non termination

Non termination possible, often desirable

Hence the concept of input-output behaviour can cease to be meaningful.

Example: Printer Daemon

- receive a job to be printed
- send the job to the printer
- send an ack when done

- Rather than on I/O behaviour the interest is shifted to **interactivity** (communication capabilities)
- Internal behaviour (calculation) is inessential

We are interested at how a system react to external stimuli

We will use the term reactive systems.

Concurrent programs: Non termination

Non termination possible, often desirable

Hence the concept of input-output behaviour can cease to be meaningful.

Example: Printer Daemon

- receive a job to be printed
- send the job to the printer
- send an ack when done

- Rather than on I/O behaviour the interest is shifted to **interactivity** (communication capabilities)
- Internal behaviour (calculation) is inessential

We are interested at how a system react to external stimuli

We will use the term reactive systems.

Concurrent programs: Non termination

Non termination possible, often desirable

Hence the concept of input-output behaviour can cease to be meaningful.

Example: Printer Daemon

- receive a job to be printed
- send the job to the printer
- send an ack when done

- Rather than on I/O behaviour the interest is shifted to **interactivity** (communication capabilities)
- Internal behaviour (calculation) is inessential

We are interested at how a system react to external stimuli

We will use the term reactive systems.

Concurrent programs: Non termination

Non termination possible, often desirable

Hence the concept of input-output behaviour can cease to be meaningful.

Example: Printer Daemon

- receive a job to be printed
- send the job to the printer
- send an ack when done

- Rather than on I/O behaviour the interest is shifted to **interactivity** (communication capabilities)
- Internal behaviour (calculation) is inessential

We are interested at how a system react to external stimuli

We will use the term reactive systems.

Concurrent programs: Non termination

Non termination possible, often desirable

Hence the concept of input-output behaviour can cease to be meaningful.

Example: Printer Daemon

- receive a job to be printed
- send the job to the printer
- send an ack when done

- Rather than on I/O behaviour the interest is shifted to **interactivity** (communication capabilities)
- Internal behaviour (calculation) is inessential

We are interested at how a system react to external stimuli

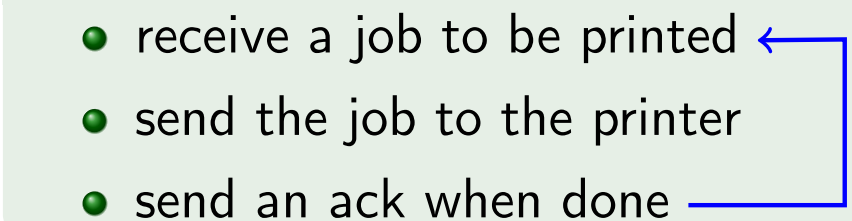
We will use the term reactive systems.

Concurrent programs: Non termination

Non termination possible, often desirable

Hence the concept of input-output behaviour can cease to be meaningful.

Example: Printer Daemon



- Rather than on I/O behaviour the interest is shifted to **interactivity** (communication capabilities)
- Internal behaviour (calculation) is inessential

We are interested at how a system react to external stimuli

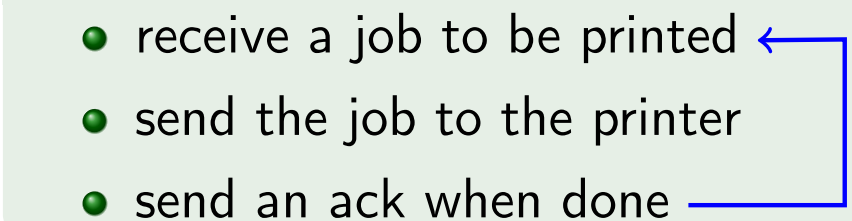
We will use the term reactive systems.

Concurrent programs: Non termination

Non termination possible, often desirable

Hence the concept of input-output behaviour can cease to be meaningful.

Example: Printer Daemon



- Rather than on I/O behaviour the interest is shifted to **interactivity** (communication capabilities)
- Internal behaviour (calculation) is inessential

We are interested at how a system react to external stimuli

We will use the term reactive systems.

Programs, behaviour and correctness

Programs, behaviour and correctness

We need three ingredients:

- **Syntax**: Language for writing programs
- **Semantics**: Behaviour (for saying what a program does) and program equivalence
- **Verification**: for saying that a program does the right things

Calculus of communicating systems

Calculus of Communicating Systems [Milner, 80's]

It describes a concurrent system by highlighting

- structure (parallel components)
- possible interactions (communications)

abstracting from the internal computation

A system represented as

A set of processes in parallel interacting through ports

Calculus of communicating systems

Calculus of Communicating Systems [Milner, 80's]

It describes a concurrent system by highlighting

- structure (parallel components)
- possible interactions (communications)

abstracting from the internal computation

A system represented as

A set of processes in parallel interacting through ports

Calculus of communicating systems

Calculus of Communicating Systems [Milner, 80's]

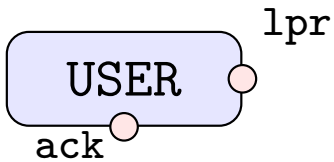
It describes a concurrent system by highlighting

- structure (parallel components)
- possible interactions (communications)

abstracting from the internal computation

A system represented as

A set of processes in parallel interacting through ports



Calculus of communicating systems

Calculus of Communicating Systems [Milner, 80's]

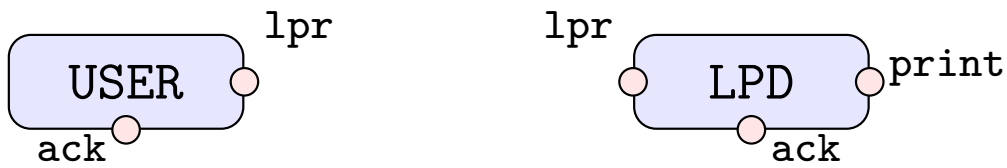
It describes a concurrent system by highlighting

- structure (parallel components)
- possible interactions (communications)

abstracting from the internal computation

A system represented as

A set of processes in parallel interacting through ports



Calculus of communicating systems

Calculus of Communicating Systems [Milner, 80's]

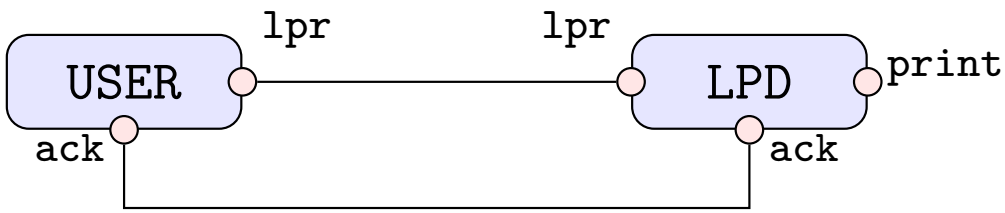
It describes a concurrent system by highlighting

- structure (parallel components)
- possible interactions (communications)

abstracting from the internal computation

A system represented as

A set of processes in parallel interacting through ports



CCS: Behaviour

Communication

output (send)	input (receive)
$\overline{\texttt{lpr}}(\text{file})$	$\texttt{lpr}(x)$

Parallel execution

Given processes P and Q

$$P \mid Q$$

Nondeterministic composition

Given process P and Q

$$P + Q$$

Restriction of a channel

Given process P and channel $\texttt{lpr}$

$$P \setminus lpr$$

CCS: Behaviour

Communication

output (send)	input (receive)
$\overline{\texttt{lpr}}(\text{file})$	$\texttt{lpr}(x)$

Parallel execution

Given processes P and Q

$$P \mid Q$$

Nondeterministic composition

Given process P and Q

$$P + Q$$

Restriction of a channel

Given process P and channel $\texttt{lpr}$

$$P \setminus lpr$$

CCS: Behaviour

Communication

output (send)	input (receive)
$\overline{\texttt{lpr}}(\text{file})$	$\texttt{lpr}(x)$

Parallel execution

Given processes P and Q

$$P \mid Q$$

Nondeterministic composition

Given process P and Q

$$P + Q$$

Restriction of a channel

Given process P and channel $\texttt{lpr}$

$$P \setminus lpr$$

CCS: Behaviour

Communication

output (send)	input (receive)
$\overline{\texttt{lpr}}(\text{file})$	$\texttt{lpr}(x)$

Parallel execution

Given processes P and Q

$$P \mid Q$$

Nondeterministic composition

Given process P and Q

$$P + Q$$

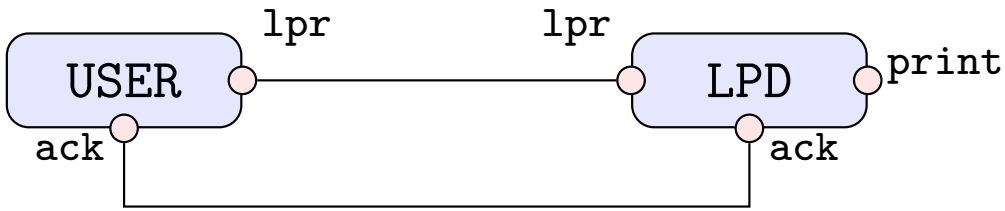
Restriction of a channel

Given process P and channel $\texttt{lpr}$

$$P \setminus lpr$$

...

Back to the example

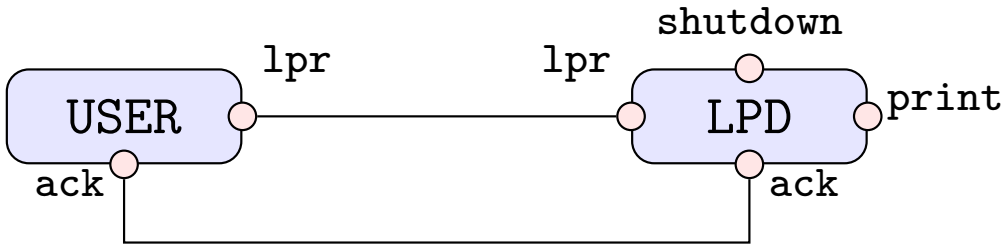


$$\begin{aligned}
\mathtt{USER} &= \\
&\quad \overline{\mathtt{lpr}}(\mathtt{file}). \\
&\quad \mathtt{ack}()
\end{aligned}
\qquad
\begin{aligned}
\mathtt{LPD} &= \\
&\quad \mathtt{lpr}(\mathtt{x}). \qquad + \ \mathtt{shutdown}() \\
&\quad \overline{\mathtt{print}}(\mathtt{x}). \\
&\quad \overline{\mathtt{ack}}(). \\
&\quad \mathtt{LPD}
\end{aligned}$$

System

$$\mathtt{System} = (\mathtt{USER} \mid \mathtt{LPD}) \setminus \{\ \mathtt{lpr},\ \mathtt{print},\ \mathtt{ack}\ \}$$

Back to the example



$$\begin{array}{ll} \texttt{USER} = & \texttt{LPD} = \\ \quad \overline{\texttt{lpr}}(\texttt{file}).\ \texttt{ack}() & \quad \texttt{lpr}(\texttt{x}).\ \overline{\texttt{print}}(\texttt{x}).\ \overline{\texttt{ack}}().\ \texttt{LPD} \quad + \ \texttt{shutdown}() \end{array}$$

System

$$\texttt{System} = (\texttt{USER} \mid \texttt{LPD}) \setminus \{\ \texttt{lpr}, \texttt{print}, \texttt{ack}\ \}$$

Program equivalence?

- Certainly not the same I/O behaviour
- Same interactivity: Two processes are equivalent if interacting with them we cannot observe any difference

Example

For instance, for `LPD` we are happy if

- it is willing to receive a `file`, after that, eventually, the file get `printed` and we get an ack
- independently of any internal computation

Observe communications

Idea of observational semantics, intuitive, not easy to formalize . . .

Program equivalence?

- Certainly not the same I/O behaviour
- Same interactivity: Two processes are equivalent if interacting with them we cannot observe any difference

Example

For instance, for LPD we are happy if

- it is willing to receive a `file`, after that, eventually, the file get `printed` and we get an ack
- independently of any internal computation

Observe communications

Idea of observational semantics, intuitive, not easy to formalize ...

Program equivalence: Idea

Observe communications

$P \xrightarrow{com} P'$ if process P can perform communication com and become P'

Bisimulation, intuitively

Given processes P and Q we define $P \sim Q$ if

- for any transition $P \xrightarrow{com} P'$ there exists a transition $Q \xrightarrow{com} Q'$ for some Q' such that $P' \sim Q'$
- for any transition $Q \xrightarrow{com} Q'$ there exists a transition $P \xrightarrow{com} P'$ for some P' such that $P' \sim Q'$

P simulates each interaction of Q and vice versa, and after they remain equivalent

Not a definition!!

. . . but we can work it out . . .

. . . and get a well-defined notion of program equivalence which is compositional

Program equivalence: Idea

Observe communications

$P \xrightarrow{com} P'$ if process P can perform communication com and become P'

Bisimulation, intuitively

Given processes P and Q we define $P \sim Q$ if

- for any transition $P \xrightarrow{com} P'$ there exists a transition $Q \xrightarrow{com} Q'$ for some Q' such that $P' \sim Q'$
- for any transition $Q \xrightarrow{com} Q'$ there exists a transition $P \xrightarrow{com} P'$ for some P' such that $P' \sim Q'$

P simulates each interaction of Q and vice versa, and after they remain equivalent

Not a definition!!

. . . but we can work it out . . .

. . . and get a well-defined notion of program equivalence which is compositional

Program equivalence: Idea

Observe communications

$P \xrightarrow{com} P'$ if process P can perform communication com and become P'

Bisimulation, intuitively

Given processes P and Q we define $P \sim Q$ if

- for any transition $P \xrightarrow{com} P'$ there exists a transition $Q \xrightarrow{com} Q'$ for some Q' such that $P' \sim Q'$
- for any transition $Q \xrightarrow{com} Q'$ there exists a transition $P \xrightarrow{com} P'$ for some P' such that $P' \sim Q'$

P simulates each interaction of Q and vice versa, and after they remain equivalent

Not a definition!!

. . . but we can work it out . . .

. . . and get a well-defined notion of program equivalence which is compositional

Verification

Single-language approach

Write the system specification `Spec` as an abstract process and then prove correctnes of the implementation `Impl` by showing

$$\texttt{Spec} \sim \texttt{Impl}.$$

A language for specifying behavioural properties

Temporal properties of the kind:

- If I send a file it will be eventually printed
- The system will never reach a deadlock
- ...

with tools for (automatic) verification that a program enjoy the property.

Course overview

What will we do?

Foundations

- **Calculus of Communicating Systems**
  A foundational (specification) language for concurrent systems
- **Behaviour and correctness**
  Does my program have the desired behaviour? What is it?
- **Specification and verification**
  An assertion language for specifying the properties desired and automatic verification tools

From specification to programming

Languages with (modern) design choices consistent with the studied theory:

- **Google Go**, message passing concurrency
- **Erlang** (Elixir), and the actor model
- **Clojure**, functional concurrency (or, data concurrency for free)
- Jolie, for service oriented computing
- and others?

What will we do?

Foundations

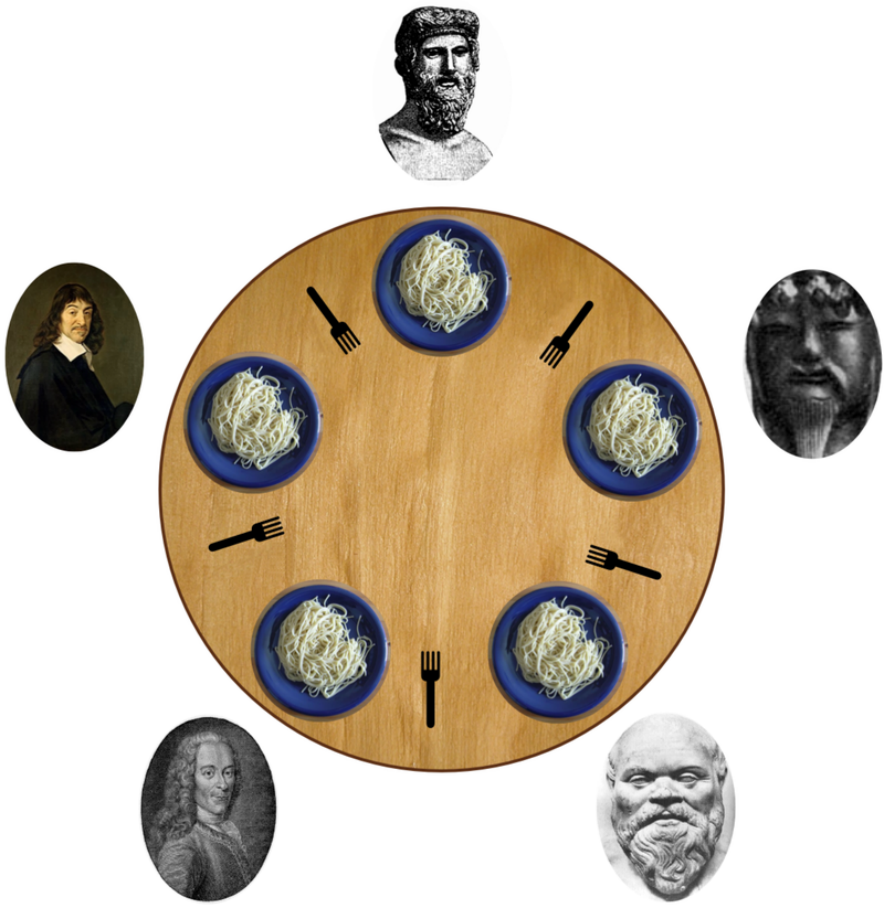
- **Calculus of Communicating Systems**
  A foundational (specification) language for concurrent systems
- **Behaviour and correctness**
  Does my program have the desired behaviour? What is it?
- **Specification and verification**
  An assertion language for specifying the properties desired and automatic verification tools

From specification to programming

Languages with (modern) design choices consistent with the studied theory:

- **Google Go**, message passing concurrency
- **Erlang** (Elixir), and the actor model
- **Clojure**, functional concurrency (or, data concurrency for free)
- Jolie, for service oriented computing
- and others?

Example: Dining philosophers



- Philosophers think and eat
- In order to eat they need the left and right fork
- Hence their standard behaviour consists of thinking, taking the forks, eating, releasing the forks and so on ...

Dining philosophers in CCS

Forks

Forks are active entities

$$F_i \stackrel{\text{def}}{=} \texttt{take}_i.\texttt{leave}_i.\ \mathit{Fork}$$

Philosophers

$$P_i \stackrel{\text{def}}{=} \texttt{think}.\overline{\texttt{take}}_i.\overline{\texttt{take}}_{i+1}.\texttt{eat}.\overline{\texttt{leave}}_i.\overline{\texttt{leave}}_{i+1}.P_i$$

System

$$\mathit{Sys} \stackrel{\text{def}}{=} (P_1 \mid F_1 \mid P_2 \mid F_2 \ldots \mid P_5) \setminus L$$

where $L = \{\texttt{take}_i, \texttt{leave}_i \mid i = 1, \ldots,\}$

Dining philosophers in CCS

Forks

Forks are active entities

$$F_i \stackrel{\text{def}}{=} \texttt{take}_i.\,\texttt{leave}_i.\,\mathit{Fork}$$

Philosophers

$$P_i \stackrel{\text{def}}{=} \texttt{think}.\,\overline{\texttt{take}}_i.\,\overline{\texttt{take}}_{i+1}.\,\texttt{eat}.\,\overline{\texttt{leave}}_i.\,\overline{\texttt{leave}}_{i+1}.\,P_i$$

System

$$\mathit{Sys} \stackrel{\text{def}}{=} (P_1 \mid F_1 \mid P_2 \mid F_2 \ldots \mid P_5) \setminus L$$

where $L = \{\texttt{take}_i, \texttt{leave}_i \mid i = 1, \ldots, \}$

Dining philosophers in CCS

Forks

Forks are active entities

$$F_i \stackrel{\text{def}}{=} \texttt{take}_i.\,\texttt{leave}_i.\; \mathit{Fork}$$

Philosophers

$$P_i \stackrel{\text{def}}{=} \texttt{think}.\,\overline{\texttt{take}}_i.\,\overline{\texttt{take}}_{i+1}.\,\texttt{eat}.\,\overline{\texttt{leave}}_i.\,\overline{\texttt{leave}}_{i+1}.\,P_i$$

System

$\mathit{Sys} \stackrel{\text{def}}{=} (P_1 \mid F_1 \mid P_2 \mid F_2 \ldots \mid P_5) \setminus L$
where $L = \{\texttt{take}_i, \texttt{leave}_i \mid i = 1, \ldots, \}$

Dining philosophers in CCS

Forks

Forks are active entities

$$F_i \stackrel{\text{def}}{=} \texttt{take}_i.\,\texttt{leave}_i.\; \mathit{Fork}$$

Philosophers

$$P_i \stackrel{\text{def}}{=} \texttt{think}.\,\overline{\texttt{take}}_i.\,\overline{\texttt{take}}_{i+1}.\,\texttt{eat}.\,\overline{\texttt{leave}}_i.\,\overline{\texttt{leave}}_{i+1}.\,P_i$$

System

$\mathit{Sys} \stackrel{\text{def}}{=} (P_1 \mid F_1 \mid P_2 \mid F_2 \ldots \mid P_5) \setminus L$
where $L = \{\texttt{take}_i, \texttt{leave}_i \mid i = 1, \ldots, \}$

Dining philosophers in Google Go

see file

Material and exam

Material

- First part: We will use the book
  L. Aceto, A. Ingolfsdottir, K.G. Larsen, J. Srba
  Reactive systems
  Cambridge University Press, 2007 (Chap. 1-7, except 6.4)
- Second part: Electronic resources linked at the course page (Slide decks will be made available).

Exam

Two parts ...

- Two exercises on the foundational part chosen from a list, available at the course page
- Mini-project or deepening