# Languages for Concurrency and Distribution

Paolo Baldan

Master's Degree in Computer Science
University of Padua

# What are we talking about?

# What are we talking about?

An approach for (just outlined)

understanding

designing

verifying

concurrent and distributed systems

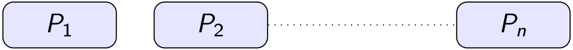
with a view on the corresponding

programming languages

# What is concurrency?

## Concurrent System

A system where different activities (processes) [programs, processes, transactions] are performed at the same time [At least conceptually]




Activities can proceed

- independently
- interacting
  - cooperation, for completing some task
  - competition, for accessing resources

# 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 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 of software is getting more and more complex

- concurrency
- distribution with connectivity problem
- mobility data and code mobility
- dynamicity communication topology is not fixed
- open endedness the environment it works in is not always completely known
- ...

# Old and new problems

## Good old concurrency problems ...

- deadlock
- starvation
- fairness
- ...

## ... and many others

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

Concurrency is everywhere

it is very important and 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
  Exercise: Take a random three letter strong and check whether it is a CS acronym
- ...

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)


```java
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();
   }
```

```java
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 processes in parallel, each taking care of one sublist.

# Complexity: conceptual!

## 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 processes, 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, 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)
```

## 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

Let us see what happens if we do not perform "unnecessary" assignments and only assign true when we find a x or a y such that f(x) = 0 or f(y) = 0.

## 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

To avoid assumptions on the scheduler, we can think of forcing context switches by adding a turn variable with processes passing the turn one to the other.

## 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!!!

We need help!

# We need somebody's help!

## The proposal in this course

Adopt a rigorous, solidly mathematically 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

# 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

For proving that a program is correct (it does what we want), we have to define what it does (its semantics) and since we want the checks to be automatized, this must be formal!

- But rewarding!

# We are not alone!

While until 15 years ago formal methods were confined to the academy, nowadays . . .

- 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:

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

## 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

# 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 . . .

```
strange-new (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 `strange-new(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

Operating systems, communication protocols, embedded systems

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. instead of concurrent

# 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

The above considerations motivates the design of the CCS.

## Calculus of Communicating Systems [Milner, 80's]

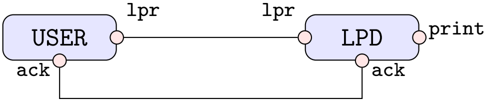
It describes a concurrent system by highlighting

- 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

The behaviour is described by simple constructs:

## 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 $Q$

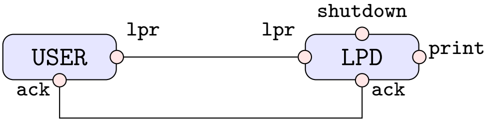
$$P + Q$$

## Restriction of a channel

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

$$P \setminus lpr$$

# Back to the example




```
USER =
      lpr(file).
      ack()
```

USER =

$\overline{\texttt{lpr}}$(file).
ack()

LPD =

lpr(x). + shutdown()
$\overline{\texttt{print}}$(x).
$\overline{\texttt{ack}}$().
LPD

## System

System = (USER | LPD) $\setminus$ { lpr, 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: 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 correctness 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)
- **Rust** based on ownership;
- and others? (Jolie / Ballerina for service oriented computing)

# 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 available).

## Exam

Two parts ...

1. Two exercises on the foundational part chosen from a list, available at the course page
2. Mini-project / deepening (/ programming exercises)