Applied Concurrency Theory
Lecture 1: Introduction

Hubert Garavel
Alexander Graf-Brill
About us

- **Hubert Garavel**
  - Directeur de recherche Inria (Grenoble, France)
  - One week per month in Saarbrücken (Humboldt foundation)
  - E-mail: garavel@mx.uni-saarland.de

- **Alexander Graf-Brill**
  - Master student (Saarland U., Depend group)
  - Block Course Assistant
  - E-mail: agrafbrill@depend.cs.uni-saarland.de
What is concurrency?
What is concurrency?

- A branch of computer science
- Several actors (or subsystems, machines, computers, processors, components, processes, threads...)
  - Each actor behaves individually
  - A common task to accomplish by all actors
  - (often:) Shared resources between actors
  - Co-operation between actors (accomplish the common task)
  - Competition between actors (access the shared resources)
- Specific problems
- Corpus of mathematical results (‘Concurrency theory’)

Lecture 1
Concurrency is everywhere

- In computer hardware:
  - in processors, fast memories, buses, embedded devices, etc.
  - from the lowest levels (gates, netlists)
  - to the highest levels (supercomputers)

- In computer software:
  - multi-user, multi-task operating systems
  - parallel programming (threads, processes)

- In networking and distributed systems:
  - computer networks, Internet, GSM
  - aerospace, trains, power grids, etc.
Concurrency is difficult

- Faster but more difficult than sequential computing

- Frequent errors
  - Deadlocks
  - Race conditions
  - Loss of global consistency

- Additional reasons for complexity
  - Communication may fail (e.g., unreliable network)
  - Some actors may fail (e.g., node crash)
Strategies to handle concurrency

1. Don’t use it
   Avoid concurrency as much as possible

2. Only use ‘easiest’ forms of concurrency
   - Pipelining (actors organized along a simple flow of data)
   - Synchronous computing (actors scheduled by a central clock)

3. When concurrency is absolutely needed:
   Learn how to master it
Concurrency in computing: since the 60s

- hardware design
- software and system design

Before: concurrency studied in other contexts

- coordination of humans acting together (work, dance, music)
- coordination of machines (e.g., trains)

In computing, concurrency has no linear history

- no continuous progress
- past knowledge is often forgotten
- major scientific/technical regressions
Initially, asynchronous logics
- the first hardware designs were asynchronous (in the 60s)
- but too difficult at that time

Then, advent of synchronous logics
- all parts of the circuit scheduled by a clock
- a proper methodology for designing reliable complex circuits
- today: most ASICs and CAD tools are synchronous

Today, synchronous logics faces limitations
- problems scaling up to high frequencies and complex VLSI
- energy (clocks waste energy), secrecy (EM radiations)

Asynchronous logics is back!
Concurrency in hardware design (2/3)

- In the first computers, a single CPU did everything
- Then, advent of multiprocessing (60s and 70s)
  - asymmetric: dedicated processors (I/O, arithmetic, graphics, crypto)
  - symmetric: multiple identical CPUs
  - shared memories, caches
  - parallel computing
- Progressive merge with telecommunications/networking
  - client/server applications
  - distributed systems
  - networks of workstations (NoW)
  - clusters, grids
  - Web services
  - supercomputing, high-performance computing
Concurrency in hardware design (3/3)

Lot of concurrency inside CPUs:
- Pipelining
- Multi-level caching
- Branch prediction

Moore law coming to an end:
- Clock frequency cannot increase any more
- Sequential processors reached performance peak
- Next step: multi-core ('many-core') processors
Concurrent in software design (1/3)

Goal: How to program parallel computers?

Low-level (hardware-oriented) approaches
- shared memory / shared variables
- study of problems: e.g., race conditions, deadlocks

Higher-level (language-oriented) approaches
- Petri nets (1962)
- Simula (1967): multiple actors and coroutines
- Algol 68 (1968): begin A, B end
- PL/1 (1973): multitasking
- Unix Bourne shell (1977): operators & (concurrent) and | (pipeline)
- (concurrency much less easier in today’s mainstream languages!)
In the 70s
- deep studies to understand concurrency issues
- new language features for safer concurrent programming (semaphores, critical sections, monitors, rendezvous, etc.)

In the 80s
- Pascal and C take off: no support for concurrency
- yet, Ada and Erlang have built-in concurrency
- automated verification techniques for concurrent problems (protocol engineering, state exploration, model checking)
- theoretical advances (process calculi, process algebra)
Concurrent in software design (3/3)

- In the 90s
  - C++: no support at all for concurrency
  - Java: a major regression to low-level programming ignores all lessons in designing better concurrent languages
  - strong criticisms: Per Brinch Hansen, William Pugh
  - UML: an imprecise model of concurrency
  - silent progress in parallel compilers

- In the 2000s
  - significant progress in analyzing concurrent systems with:
    - probabilistic behaviours
    - (hard or soft) real-time aspects
Concurrency today
Concurrent machines at hand

For long, concurrent machines were rare:
- Reserved to big military or civil projects
- Sometimes available in research labs

Now, they are available to the masses:
- Your laptop is probably dual-core or quad-core
- Machines with 24 cores already exist
- Clusters and grids accessible from the desktop

Concurrency is now a major concern in industry
Most existing software
- was designed for sequential machines (e.g., Wintel)
- is not ready for concurrency

Major revisions will be needed for:
- exploiting multi-core machines
- exploiting cloud computing resources
- developing reliable concurrent systems and programs
Mainstream programming languages are not ready:
  - C and C++: nothing for concurrency
  - Java: a catastrophe
  - Ada and Erlang: barely used

New software must be developed to help designing and verifying
  - asynchronous circuits / architectures
  - concurrent software programs
Goals of the block course
Three goals

- Get acquainted with concurrency
  - Recognize concurrent problems where they are
  - Learn vocabulary and key concepts

- Learn various languages for concurrency
  - Process calculi
  - Automata-based languages
  - Semantic concepts: SOS, LTS, etc.

- Experiment with state-of-the-art tools
  - a ‘Matlab reflex’ for concurrency
  - Tools from Grenoble, Oxford, and Saarland
Key concepts of concurrency
Interleaving

- Several actors have to execute actions independently
- A global observer sees ‘diamonds’ of actions
State explosion - combinatorial explosion

A consequence of interleaving

The number of states is exponential in the number of concurrent actors:
- two actors: planary diamonds
- three actors: cubes
- N actors: hypercube with N dimensions

State explosion is a major problem for verification techniques based on exhaustive state explorations
Processes vs Threads

Two main approaches to communication between actors
- shared memory (e.g., blackboards)
- message passing (e.g., e-mail)

**Shared memory → actors are called ‘threads’**
- Close to hardware and usually efficient
- Multiple incompatible semantics (Posix, etc.)
- Often dependent on hardware ⇒ portability problems
- Low-level ⇒ makes proofs and automated reasoning difficult

**Message passing → actors are called ‘processes’**
- Higher abstraction level, more suitable for formal analysis
- Can model hardware, software, and networking problems
- Perhaps less efficient to implement (?)
A concept borrowed from particle physics

The future evolution of a concurrent program cannot be predicted, even if one fully knows its past history and its current state

- Each actor evolves at its own speed
- Some algorithms are intrinsically nondeterministic

A major difference wrt sequential programming

Nondeterminism makes life much harder:
- each state may have several possible futures
- execution runs / tests are not reproducible
Race conditions

- Nondeterministic behaviour arising from threads accessing a common resource (shared variable)

- Example: 2 threads and 1 shared variable X
  Initially: $X = 0$
  - thread 1: $X := X + 1$
  - thread 2: if $X = 0$ then $X := 2 \times X + 1$
    (hypothesis: testing $X$ and assigning $X$ are two different steps)
  Finally: $X = 1$, 2, or 3 depending on relative execution speeds

- Race condition also exists with electronic signals
Critical sections

- Approaches proposed to avoid race conditions:
  - while an actor is accessing shared resources, block other actors
  - other actors have to wait until the first actor has finished

- Test-and-set instructions
  - simplest form, implemented as microprocessor instructions
    example: if X = 0 then X := 1 (single, atomic instruction)

- Locks
  - one thread becomes ‘owner’ for a limited time (aquire/release)
  - examples: semaphores, object locks in Java

- Critical sections
  - piece of code to be executed atomically
    example: critical_begin if X = 0 then X := 2 * X + 1 critical_end
  - examples: monitors, conditional critical sections, etc.
Deadlocks

- Improper use of critical sections / locks / etc.
- Each actor is waiting to access shared resources blocked by other
- Example: the dining philosophers problem
  - rule: each philosopher needs two forks
  - if each philosophers starts by taking the left fork, then everyone is blocked
  - various solutions exist (see Wikipedia)
Local deadlocks and livelocks

- A deadlock is a global problem: everyone is blocked
  - there are similar related issues

- Local deadlocks:
  - starvation: one or several actors are blocked
  - coalitions: certain actors join forces to prevent others from accessing shared resources

- Livelocks:
  - similar to deadlocks, except that actors are not blocked but are constantly active without being productive
Rendezvous

- High-level alternative to shared variables and locks

**Principle:**
- two (or more) actors decide to meet at a given point RV
- the first actor arrived at RV waits for the others (and so on)
- when all actors are ready, they can exchange data
- after the rendezvous, each actor restarts independently

**Combines in a single mechanism**
- Synchronization between actors
- Communication by messages

**Clean semantics preserving modularity**
Rendezvous is ‘synchronous’:
- all actors have to be there simultaneously
- not to be confused with synchronous computing (clocks)

Alternative approach:
- an actor $S$ sends a message $M$ to another actor $R$
- $M$ is put in a message queue (e.g., FIFO queue)
- $S$ is not blocked and continues its execution after sending $M$
- some time later, $R$ checks the queue and reads $M$

Popular model, but theoretical problems
- queue is finite: overflow issues ($M$ discarded or $S$ blocked)
- queue is infinite: $S$ can continuously fill in the queue
Structure of the block course
Six lectures

September

1. Introduction
2. Process calculi (LOTOS)
3. Next-generation formal methods (LOTOS NT)
4. Pi-calculus and mobility

October

5. Probabilistic systems (PRISM)
6. Stochastic and timed systems (MODEST)
Four projects (lab exercises)

September:
- Project #1. LOTOS and LOTOS NT
- Project #2. PIC (pi-calculus)
deadline is October 1st (12:00)

October:
- Project #3. PRISM
- Project #4. MODEST
deadline is October 12 (12:00)
Some challenges

- Challenges are small exercises (< 1 hour) to be done after each lecture before the next one.

- ‘Without such exercises, your students will attend the lectures and wait until the end of September to undertake their projects; suddenly, they will realize that they have to produce something, that they are late, and they will start panicking.’

  (a respected German professor)
Today’s challenge
Starting up

- Get from the CMS the document entitled: *How to install the software tools needed for the course?*

- The ‘official’ solution is strongly advised
  - Install Virtual Box 4.1.22 on your machine
  - Install the AppliedConcurrencyTheory virtual machine
  - Request your CADP license to register your software

- Test if the tools are properly installed:
  - Type the shell command: `bcg_edit $CADP/demos/demo_13/A1.bcg`
  - Save the drawing as a PostScript file
  - Email this file to Alexander (agrafbrill@depend.cs.uni-saarland.de)
A few references

■ Wikipedia:
  ◦ Usually informative and well-done
  ◦ Read more about the terms mentioned in this lecture: asynchronous circuit, nondeterminism, semaphore, deadlock, etc.

■ Critical assessment of concurrency in C/C++

■ Critical assessment of concurrency in Java