Lecture 1: Introduction to Concurrency

Today, mostly Shared Memory

K. V. S. Prasad

TDA384/DIT391 Principles of Concurrent Programming
Chalmers Univ. and Univ. of Gothenburg

2 September 2019

Outline

Section 1

More practical stuff

From Robin Adams
to students of DIT991 Mathematical Modelling

We apologise for the mistake in timetabling. The recording of the Mathematical Modelling lecture will be available on the course website from this evening. Please watch it as soon as possible, and in particular make sure you understand the section "Structure of the Course". Please contact Robin Adams if anything is not clear.

Student Representatives

	email	Name
TKDAT	binde@student.chalmers.se	Johannes Binde
TIDAL	frodin@student.chalmers.se	Thomas Frdin Larsson
TKDAT	davhedg@student.chalmers.se	David Hedgren
TIDAL	ponjo@student.chalmers.se	Pontus Johansson
TKDAT	axeka@student.chalmers.se	Axel Karlsson
TKITE	markp@student.chalmers.se	Markus Pettersson
TKDAT	valterh@student.chalmers.se	Henrik Valter

Table: From CTH

Still waiting for GU admin to send us their lst of names. Volunteers?

Programming Languages

- Labs: Java for labs 1 and 3, Erlang for lab 2.
- Exam
  - pseudo-Java (full Java in Appendix if needed)
  - Erlang
- Lectures
  - pseudo-Java (fullish Java occasionally)
  - Erlang
  - Occasionally, we widen horizon beyond exam
    - These lectures or parts of lectures are optional, ignore if you wish.
    - We might use ad-hoc notation, or Promela. You only need to read these, not write.
    - Promela = concise modelling language for concurrency. Allows models beyond Java and Erlang. Runs on simulator with assertion checking.

Ben-Aris text book uses pseudo code, and supports Java and Promela.

What is Erlang?

- Java, C, Python, etc. are all *imperative*:
  - program = sequence of *commands*, which change the *state*.
  - *Assignment* is the only command. I/O is a special kind of assignment.
    - ⋆ *Control flow* (`if`, `case`, `loop`, etc.) says which assignment is next.
- Erlang
  - *has no assignment*
  - Each *process* can send *messages* to other processes, and receive messages from its inbox, where messages wait till they are read.
  - Messages from a process are typically functions of its state and inputs.
  - These functions are computed *functionally*
- Haskell is a *functional language* you may have seen
  - Erlang is one too (if you ignore the messages), but it has no static types, so expect more errors
  - **If you have never programmed functionally**
    - ⋆ **GET STARTED NOW WITH ERLANG TUTORIALS**

Tutorial on Erlang and functional programming in week 3.

Section 2

Specification, Validation, Behaviour, and Verification

How to design Concurrent (think Embedded) systems

Clear and efficient programs, good tools and libraries are fine, but we need *formal* (= machine checkable) *goals* and *tests* for the sysem to be built.

Why? The system (call it "*Sys*") might fly a plane, drive a car or train, or control a nuclear reactor or radiation therapy machine. Sys *can kill people*.

Two questions to clarify the goal and test of success:

- What should the system do? *Specification* ("`Spec`")
- What does it do? *(Operational) semantics*

and two slogans

- Build the right system—*Validate* `Spec`. Is it *consistent*? *complete*?
  - Do you really want Sys to *behave* as `Spec` says?
- Build the system right—*Verify* `Sys` against `Spec`

On Validation and Verification for Sequential programs

- What should a *sort* (sorting) or *sqrt* (square root) program do?
  - For input x of specific type (positive number to *sqrt*, a deck of cards only to *sort*), produce the desired output *Spec*(x).
- So *Spec* is a huge table of output for each input. Given as a *complete set of cases*, or a partial set of *use cases*, generalised sensibly.
- Formal *Spec* converts exhaustive check to *case analysis*.
  - As we prove theorems about every triangle, etc.
  - To Validate: exhaustively check that *Spec* is complete and consistent.
  - To Verify: exhaustively check that *Spec*=*Sys*
    - ⋆ (i.e., for every x, *Spec*(x)=*Sys*(x)).
- Sometimes, `Spec` is a simple, obviously correct but impractical program (called a *golden* or *reference* model in circuit design). Then again we want *Spec*=*Sys* as above, but rather more directly.
  - See *Shufflesort* below as an example reference program.

On specifications and assertions

Imperative programs are sequences of commands. The code snippet below has an implicit "evaluate 5*7" and then "put the result in x".

Listing 1: expression evaluation as implicit command

```
1 x = 5*7
```

In specifications, by contrast, *assertions* (boolean propositions) are central.

For example, given *x¿0*, a program to find the square root of *x* might have *Spec* "find the *y* such that $y^2 = x$". Note that *Spec* does not say how to find this *y*, though it implies that it exists.

Listing 2: `assert` to do run-time test that *Spec* is met

```
1 y=sqrt(x);      //function sqrt runs a sq. root
      algorithm
2 assert (y*y=x) //raises exception if sqrt fails test
```

Partial specifications: sorting a deck of cards

Suppose `sorted(xs)` returns `true` iff the deck `xs` is sorted.

Then the program below correctly sorts a given `deck` of cards, *if it ever gets to line 3*. The claim *Line 3*$\rightarrow$*sorted(deck)* is a *safety requirement*. Slogan: *nothing bad ever happens.* (Bad=*Line 3* $\wedge$ `not`*sorted(deck)*).

Listing 3: Shufflesort

```
1 while (not sorted(deck))
2     {shuffle deck}
3 assert (sorted(deck)) //exception if unsorted
```

But we also want the sort to *terminate*. This is an example of a *liveness* or *progress requirement*. Slogan: *something good eventually happens*.

That *Line 3* eventually holds cannot be shown by assertions.

E.g., to show that bubble sort terminates (when no elements out of place), show that each pass moves one element to its correct place, so the number of elements possibly out of place drops to 0.

Embedded Systems: what can go wrong with validation

https://spectrum.ieee.org/aerospace/aviation/how-the-boeing-737-max-disaster-looks-to-a-software-developer

I would appreciate hearing if you could find this using the Chalmers library.

So Boeing produced a dynamically unstable airframe, the 737 Max. That is big strike No. 1. Boeing then tried to mask the 737s dynamic instability with a software system. Big strike No. 2. Finally, the software relied on systems known for their propensity to fail (angle-of-attack indicators) and did not appear to include even rudimentary provisions to cross-check the outputs of the angle-of-attack sensor against other sensors, or even the other angle-of-attack sensor. Big strike No. 3.

None of the above should have passed muster. None of the above should have passed the OK pencil of the most junior engineering staff, much less a Designated Engineering Representative. Thats not a big strike. Thats a political, social, economic, and technical sin.

Embedded Systems: what else can go wrong and right

- Auto-pilots that dump the aircraft in the pilot's lap in crisis (do pilots now mostly monitor the auti-pilot, and scarcely fly?). Bad *Spec*?

  The Human Factor by William Langewiesche, Vanity Fair, Sep 2014. *www.vanityfair.com/news/business/2014/10/air-france-flight-447-crash*
- Bad implementation. The many things that can go wrong because of concurrency (coming soon).
  https://ocw.mit.edu/ans7870/6/6.005/s16/classes/19-concurrency/
  https://www.coursera.org/lecture/software-design-threats-mitigations/therac-25-case-study-VmQPa
  https://compas.cs.stonybrook.edu/ nhonarmand/courses/fa17/cse306/sl conc_bugs.pdf
- But all can go right. https://en.wikipedia.org/wiki/Shinkansen
  "Over the Shinkansen's 50-plus year history, carrying over 10 billion passengers, there have been no passenger fatalities due to train accidents such as derailments or collisions, despite frequent earthquakes and typhoons."

Parallelism around us

- The world is a parallel place. Most things exist, and some even act, all the time.
- In physics, chemistry, biology, economics, medicine, history, football, tennis, ...) the *agents* (or *processes*) act all the time—they don't cease to exist, or even go to sleep
- (Sequential) Programming is one of the few fields where only one thing happens at a time.
  - Was never really true (interrupts, etc.)

Concurrency around us

A concurrent process is an *abstraction* and actually runs only when *scheduled* on to a physical processor by the underlying *operating system* or *run-time system* that implements the process abstraction. Examples:

- A legal case once launched is *sub judice* and mostly just "stuck in the courts". It actually runs only when a hearing is scheduled.
- An application made to government is "under consideration", but your file is mostly waiting till an officer has the time for it.
- Film often has multiple stories sharing one screen. We return to pick up a story where we left it or where it has meanwhile got to (you can see time has passed).

Concurrency and non-determinism

Concurrent systems are *non-deterministic*. Why?

- We don't know if a process is actually running, or just *ready* to run.
- The underlying scheduler might be non-deterministic. We don't know who gets to access a shared resource first, or who speaks first.

Which means:

- We cannot assume any speed for the process
- Or how long it will take to do something
- *Synchronisation* with a process must be explicit

Consequences of non-determinism

- Re-running a concurrent program may not produce the same result.
- So standard debugging is impossible.
- So you must *reason* about program behaviour.
- This can get tedious and error-prone, so we use tools called *model-checkers*, *proof assistants* and *theorem provers*.

Tools: simulators

Humans are not good at intuiting the behaviour of systems with many agents. Simulation is an important tool to show us what might happen (not what will). Remember the systems are non-deterministic.

- We can now run 10 million agents to *simulate* the spread of infection. We don't use 10 million CPUs, far fewer. The scheduling mimics the non-determinism of reality. (If you are exposed to an infection, you will catch it with some probability).
- Many multi-agent systems show *emergent* properties. We don't have good theories of emergence, but simulations can help us visualise it.1

Validation and Verification for Concurrent programs

A sequential program is basically a function from input to output. The input can all be given at once, at the start.

A concurrent program *interacts* (*communicates*) with its *environment*. Its *behaviour* is not a function, but possible conversations. I know my first utterance, but my second depends on your reply to my first.

- What should the system do? *Specification* ("`Spec`")
  - Give a set of safety and liveness properties it must satisfy.
- What does it do? *(Operational) semantics*
  - The behaviour is a tree of states with branches labelled by I/O actions.

We can still use a reference design as *Spec*, but now we need relations like "any behaviour of *Sys* is also a behaviour of *Spec*".

Do you really want *Sys* to *behave* as *Spec* says? is now a hard question, needing exhaustive simulation and discussion.

First mention of Linear Temporal Logic (LTL)

LTL is formally a small part of the course, but it is good to at least understand what the fuss is all about.

An excellent reference (including a review of basic logic) is ftp://ftp.cs.bham.ac.uk/pub/authors/M.D.Ryan/tmp/Anongporn/Ch1+3.pd

You won't need all of it, but an overview will help calm nerves. Better definitions than in Ben-Ari.

Tools: model checkers

The model checker SPIN

- Checks Promela programs. It checks assertions and more general LTL formulas, showing where they fail to hold.
  - A single state is needed to disprove a safety property. You said "this bad thing won't happen". Well, here is where it does, and a path to get there.
  - A loop of states is needed to disprove a liveness property. You said "this good thing will eventually happen". Well, here is a loop where I can get stuck and where the good thing never happens.

Concurrent programming ca. 1955: unit record equipment

Listing 4: Read-process-print sequence

```
1 while ({"more cards to be read"})
2     {readinto(x);
3      y = f(x);
4      print(y)}
```

Since the reading and writing took about half a second, and the computation much less, the above ran at maybe 100 records a minute.

But the CDR and LPT ran independently off the buffers *x*, *y*. So you could *pipeline*: read the third while processing the second and priting the first. Ran at 200 records a minute. The synchronisation to prevent overwrites and re-use of old data was by making the CPU wait for half a second. There was no explicit synch, only estimates of time.

Concurrent programming proper got started in the 1960's, with the invention of the *semaphore*, an abstraction of how to deal with interrupts. Hardware versions were available as *test-and-set* instructions.

O-O and CP

This is a course on the *principles of Concurrent Programming* (CP, or more correctly, PCP). It is not a course on doing CP in Java, or even Erlang (a purpose designed language for telecom applications of CP).

In particular, the O-O aspects of Java are a needless distraction when discussing CP. For a more general critique of O-O, see

Objects Never? Well, Hardly Ever! By Ben-Ari, Mordechai. in *Communications of the ACM*. Sep 2010, Vol. 53 Issue 9, p32-35.

It is at least plausible that embedded software has a much bigger code base than O-O. Certainly much more safety-critical code.