Course Introduction, Concurrent Semantics

Johannes Åman Pohjola
CSIRO’s Data61 (and UNSW)
Term 2 2021
Who are we?

I am Johannes Åman Pohjola, a research scientist at CSIRO’s Data61 and conjoint lecturer at UNSW. I will be the lecturer and course convenor.

Vincent Jackson, a PhD student at UNSW researching formal methods for concurrency, will be grading your homework.

Most of the material for this course was developed by its previous lecturers: Liam O’Connor, Vladimir Tosic, and Kai Engelhardt. Mistakes are mine :)
Contacting Us

http://www.cse.unsw.edu.au/~cs3151

Forum

There is an Ed forum available on the website. Questions about course content should typically be made there. You can ask us private questions to avoid spoiling solutions to other students.

Administrative questions should be sent to cs3151@cse.unsw.edu.au.
What do we expect?

Maths
This course uses a significant amount of discrete mathematics. You will need to be reasonably comfortable with logic, set theory and proof. MATH1081 ought to be sufficient for aptitude in these skills, but experience has shown this is not always true. There is a math resources subsection of the website if you feel yourself falling behind in this area. We will do our best to support you.

Programming
We expect you to be familiar with imperative programming languages like Java or C. Course assignments may require some programming in modelling languages, as well as Java.
Assessment

**Homework (10%)** One for every week of teaching (except Week 10). Either theoretical (requiring answers on a page) or practical (requiring programming or modelling).

**Assignments (40%)** One smaller warmup assignment, and two major assignments. Major assignments are supposed to be done in pairs. Please try to organise this as soon as you can.

**Exam (50% + pass hurdle)** Online exam.

The full assessment breakdown is on the course website.
Lectures are on Wednesdays at 2pm and Thursdays at 4pm, in Matthews 101. Technology willing, you can also participate remotely via Zoom (link on Ed).

Lecture recordings should pop up on Echo360.
Textbook

While we draw on a number of other sources, we do have a textbook for this course. Mordechai Ben-Ari’s Principles of Concurrent and Distributed Programming. This book can be ordered from the campus bookshop at a ludicrous price, and other vendors are not much better.

Copyright Infringement

I have been told that copyright infringement has occurred and that the textbook is being freely made available on a website called Library Genesis, a site accessible via a mere Google search.

I do not condone copyright infringement.
Three cryptographers are sitting down to dinner at their favorite three-star restaurant. Their waiter informs them that arrangements have been made with the maître d’hôtel for the bill to be paid anonymously. One of the cryptographers might be paying for the dinner, or it might have been NSA (U.S. National Security Agency). The three cryptographers respect each other’s right to make an anonymous payment, but they wonder if NSA is paying.
Dining Cryptographers

1. Each $C_i$ flips a coin.
2. Each $C_i$ tells what they tossed only to their right.
3. Each $C_i$ announces if the two coin tosses are equal unless they paid.
4. An even number of "diff." means the NSA paid.
5. An odd number of "diff." means one of the $C_i$ paid.
Dining Cryptographers

1. Each $C_i$ flips a coin.
2. Each $C_i$ tells what they tossed only to their right.
3. Each $C_i$ announces if the two coin tosses are equal unless they paid.
4. An even number of “diff.” means the NSA paid.
5. An odd number of “diff.” means one of the $C_i$ paid.

Your dinner has been paid for by a party who wishes to remain anonymous.
Dining Cryptographers

1. Each $C_i$ flips a coin.
2. Each $C_i$ tells what they tossed only to their right.
3. Each $C_i$ announces if the two coin tosses are equal unless they paid.
4. An even number of “diff.” means the NSA paid.
5. An odd number of “diff.” means one of the $C_i$ paid.

Was it one of us?

…or the NSA?
Dining Cryptographers

1. Each $C_i$ flips a coin.

2. Each $C_i$ tells what they tossed only to their right.

3. Each $C_i$ announces if the two coin tosses are equal unless they paid.

4. An even number of "diff." means the NSA paid.

5. An odd number of "diff." means one of the $C_i$ paid.

Was it one of us? …or the NSA?
Dining Cryptographers

1. Each $C_i$ flips a coin.
2. Each $C_i$ tells what they tossed only to their right.
3. Each $C_i$ announces if the two coin tosses are equal unless they paid.
4. An even number of "diff." means the NSA paid.
5. An odd number of "diff." means one of the $C_i$ paid.

---

Your dinner has been paid for by a party who wishes to remain anonymous.

Was it one of us? ...or the NSA?
Dining Cryptographers

Procedure

1. Each $C_i$ flips a coin.
Dining Cryptographers

**Procedure**

1. Each $C_i$ flips a coin.
2. Each $C_i$ tells what they tossed **only** to their right.

---

Was it one of us? ...or the NSA?
Dining Cryptographers

**Procedure**

1. Each $C_i$ flips a coin.
2. Each $C_i$ tells what they tossed only to their right.
3. Each $C_i$ announces if the two coin tosses are equal unless they paid.
Dining Cryptographers

Procedure

1. Each $C_i$ flips a coin.
2. Each $C_i$ tells what they tossed only to their right.
3. Each $C_i$ announces if the two coin tosses are equal unless they paid.
4. An even number of “diff.” means the NSA paid.

Was it one of us? …or the NSA?
Dining Cryptographers

**Procedure**

1. Each $C_i$ flips a coin.
2. Each $C_i$ tells what they tossed only to their right.
3. Each $C_i$ announces if the two coin tosses are equal unless they paid.
4. An even number of “diff.” means the NSA paid.
5. An odd number of “diff.” means one of the $C_i$ paid.
Questions

- Does it work?
- Why does it work?
- Is it useful?
Definitions

Definition

Concurrency is an abstraction for the programmer, allowing programs to be structured as multiple threads of control, called processes. These processes may communicate in various ways.

Example Applications: Servers, OS Kernels, GUI applications.

Anti-definition

Concurrency is not parallelism, which is a means to exploit multiprocessing hardware in order to improve performance. However, parallel hardware can be used to support concurrent applications.
Sequential vs Concurrent

We could consider a *sequential* program (a *process* or *thread*) as a sequence (or *total order*) of *actions*:

- • → • → • → • → • → • → • → ...

The ordering here is “happens before”. For example, processor instructions:

```
LD  R0, X  →  LDI  R1, 5  →  ADD  R0, R1  →  ST  X, R0
```

A concurrent program is not a total order but a *partial order*.

```
○ → ○ → ○ → ○ → ○ → ○ → ○ → ...
```

```
• → • → • → • → • → • → • → ...
```

This means that there are now multiple possible *interleavings* of these actions — our program is non-deterministic where the interleaving is left to the *execution model*. 
Multithreaded Execution

- P
- Q
- R

CPU

L1 Cache

L2 Cache

Main Memory
Parallel Multiprocessor Execution

- **P**: CPU, L1 Cache, L2 Cache, bus/crossbar, Main Memory
- **Q**: CPU, L1 Cache, L2 Cache, bus/crossbar, Main Memory
- **R**: CPU, L1 Cache, L2 Cache, bus/crossbar, Main Memory
Parallel Distributed Execution

P

CPU
L1 Cache
L2 Cache
Main Memory

Q

CPU
L1 Cache
L2 Cache
Main Memory

R

CPU
L1 Cache
L2 Cache
Main Memory

network
Synchronisation

Regardless of the execution model, processes need to communicate to organise and co-ordinate their actions.

**Types of Communication**

**Shared Variables**  Typically on single-computer execution models.

**Message-Passing**  Typically on distributed execution models.

This communication introduces new constraints on the possible interleavings:

\[
\begin{align*}
\circ & \rightarrow \circ \rightarrow \circ \rightarrow \circ \rightarrow \circ \rightarrow \circ \rightarrow \cdots \\
\bullet & \rightarrow \bullet \rightarrow \bullet \rightarrow \bullet \rightarrow \bullet \rightarrow \bullet \rightarrow \cdots
\end{align*}
\]

The red arrows are called *synchronisations*. 
**In a nutshell**

This course is about the three R’s of concurrent programming:

1. **Reading** concurrent code and programming idioms in a variety of execution contexts.
2. **Writing** concurrent software using various abstractions for synchronisation.
3. **Reasoning** about concurrent systems with formal proof and automatic analysis tools.

**Why Reasoning?**

a.k.a. why all the maths?

It’s simply not feasible to test concurrent systems with standard methods. We need a way to rigorously analyse our software when running it no longer provides a reasonable indication of correctness.

We will learn more about this next lecture.
Reasoning

Sequential program reasoning (COMP2111, COMP6721) is usually done with a proof calculus like Hoare Logic.

\[ \{ \varphi \} \ P \ \{ \psi \} \]

This notation means that if the program \( P \) starts in a state satisfying the pre-condition \( \varphi \) and it terminates, it will end in a state satisfying the post-condition \( \psi \).

Semantics

Consider each action as a function from state to state \( \Sigma \rightarrow \Sigma \). Then the semantics or meaning of a sequential program \( \llbracket P \rrbracket \) is the composition of all the functions in the sequence. Then the above Hoare triple actually means:

\[ \forall s \in \Sigma. \varphi(s) \Rightarrow \psi(\llbracket P \rrbracket(s)) \]

Note that we only care about the initial and final states here.
Consider the following concurrent processes, sharing a variable $n$.

<table>
<thead>
<tr>
<th></th>
<th>var $n := 0$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$p_1$:</td>
<td>var $x := n$;</td>
</tr>
<tr>
<td>$p_2$:</td>
<td>$n := x + 1$;</td>
</tr>
<tr>
<td>$q_1$:</td>
<td>var $y := n$;</td>
</tr>
<tr>
<td>$q_2$:</td>
<td>$n := y - 1$;</td>
</tr>
<tr>
<td>$r_1$:</td>
<td>var $z := n$;</td>
</tr>
<tr>
<td>$r_2$:</td>
<td>$n := z + 1$;</td>
</tr>
</tbody>
</table>

**Question**

What are the possible final values of $n$?

We can’t just look at the initial and final states from each process!
Semantics for Concurrency

For concurrency, just initial and final states aren’t enough. We have to worry about all intermediate states as well.

It’s also very common for concurrent systems never to terminate, but instead to run forever, waiting for new requests (e.g. a server). So there is not necessarily even a final state!

Behaviours

A *behaviour* is an infinite sequence of states, i.e. $\Sigma^\omega$. Note we don’t record what *actions* have taken place, only the effects they have on the state (variables, program counters etc.).

If a process terminates, we consider the final state to repeat infinitely.
Semantics and Specifications

A better semantics for a concurrent program $[P]$ is the set of all possible behaviours from all the different available interleavings of actions.

**Specs**

Preconditions and postconditions don’t work for behaviours – there’s no final state!

We want to specify systems with (linear) temporal properties like

”Two processes never access the same shared resource simultaneously”

Or:

”If a server accepts a request, it will eventually respond”

These are examples of safety and liveness properties respectively.
Semantics and Specifications

If we consider a property to be a set of behaviours, then a program $P$ meets a specification property $S$ iff:

$$[P] \subseteq S$$

This works for correctness properties like the ones we’ve seen, but not for security properties or real-time properties.

Example (Security Properties)

In the Dining Cryptographers, we desire confidentiality of who paid. If all coins were known to always land heads-up, then this property is violated. However, this variant of a problem has a subset of the behaviours of the original one. Therefore, we cannot construct a specification $S$ that is satisfied by the original scenario, but not by our non-confidential one.
Internal vs. External State

We often wish to distinguish between state that is observable from outside (e.g. shared variables) and state that is not (e.g. local variables).

Example (Dining Cryptographers)

In the dining cryptographers problem, the internal state might be the value of the coins, and the external state might be what the cryptographers say.

If we abstract away from all internal state, actions that only affect internal state will appear not to change the state at all.

\[
\begin{array}{c}
\bullet \bullet \bullet \bullet \bullet \bullet \bullet \bullet \\
\downarrow \\
\bullet \bullet \bullet \bullet \bullet \bullet \bullet \bullet \\
\end{array}
\]

This kind of (finite) repetition of the same state is called *stuttering*. We generally don’t want properties to distinguish behaviours that are equivalent modulo stuttering.
Cantor’s Uncountability Argument

Result
It is impossible in general to enumerate the space of all behaviours.

\[
\sigma_\delta = \bullet \bullet \bullet \bullet \bullet \ldots
\]

\[
\sigma_0 = \bigcirc \bullet \bullet \bullet \bullet \bullet \ldots
\]

\[
\sigma_1 = \bullet \bigcirc \bullet \bullet \bullet \bullet \bullet \ldots
\]

\[
\sigma_2 = \bullet \bullet \bigcirc \bullet \bullet \bullet \bullet \ldots
\]

\[
\sigma_3 = \bullet \bullet \bullet \bigcirc \bullet \bullet \bullet \bullet \ldots
\]

\[
\sigma_4 = \bullet \bullet \bullet \bullet \bigcirc \bullet \bullet \bullet \bullet \ldots
\]

\[
\vdots \quad \vdots \quad \vdots \quad \vdots \quad \vdots \quad \vdots
\]

Proof
Suppose there exists a sequence of behaviours \( \sigma_0 \sigma_1 \sigma_2 \ldots \) that enumerates all behaviours.

Then we can construct a delightfully devilish behaviour \( \sigma_\delta \) that differs from any \( \sigma_i \) at the \( i \)th position, and thus is not in our sequence.

Contradiction!
A linear temporal property is a set of behaviours.

1. A **safety** property states that something **bad** does not happen. For example:

   \[ I \text{ will never run out of money.} \]

   These are properties that may be violated by a **finite prefix** of a behaviour.

2. A **liveness** property states that something **good** will happen. For example:

   \[ If \text{ I start drinking now, eventually I will be smashed.} \]

   These are properties that can always be satisfied **eventually**.
Properties Examples

Are they safety or liveness?

- *When I come home, there must be beer in the fridge* – **Safety**
- *When I come home, I’ll drop on the couch and drink a beer* – **Liveness**
- *I’ll be home later* – **Liveness**
- *The program never allocates more than 100MB of memory* — **Safety**
- *The program will allocate at least 100MB of memory* – **Liveness**
- *No two processes are simultaneously in their critical section* — **Safety**
- *If a process wishes to enter its critical section, it will eventually be allowed to do so* – **Liveness**

Now let’s try to mathematically formalise what it means for a property to be safety or liveness.
Metric for Behaviours

We define the *distance* $d(\sigma, \rho) \in \mathbb{R}_{\geq 0}$ between two behaviours $\sigma$ and $\rho$ as follows:

$$d(\sigma, \rho) = 2^{-\sup \{ i \in \mathbb{N} \mid \sigma|_i = \rho|_i \}}$$

Where $\sigma|_i$ is the first $i$ states of $\sigma$ and $2^{-\infty} = 0$. Intuitively, we consider two behaviours to be close if there is a long prefix for which they agree.

**Observations**

- $d(x, y) = 0 \iff x = y$
- $d(x, y) = d(y, x)$
- $d(x, z) \leq d(x, y) + d(y, z)$

This forms a *metric space* and thus a *topology* on behaviours.
Topology

Definition
A set $S$ of subsets of $U$ is called a topology if it contains $\emptyset$ and $U$, and is closed under union and finite intersection. Elements of $S$ are called open and complements of open sets are called closed.

Example (Sierpiński Space)
Let $U = \{0, 1\}$ and $S = \{\emptyset, \{1\}, U\}$.

Questions
- What are the closed sets of the Sierpiński space?
- Can a set be clopen i.e. both open and closed?
Topology for Metric Spaces

Our metric space can be viewed as a topology by defining our open sets as (unions of) open balls:

$$B(\sigma, r) = \{ \rho \mid d(\sigma, \rho) < r \}$$

This is analogous to open and closed ranges of numbers.

**Why do we care?**

Viewing behaviours as part of a metric space gives us notions of limits, convergence, density and many other mathematical tools.
Limits and Boundaries

Consider a sequence of behaviours $\sigma_0\sigma_1\sigma_2\ldots$. The behaviour $\sigma_\omega$ is called a *limit* of this sequence if the sequence *converges* to $\sigma_\omega$, i.e. for any positive $\varepsilon$:

$$\exists n. \forall i \geq n. \ d(\sigma_i, \sigma_\omega) < \varepsilon$$

The *limit-closure* or *closure* of a set $A$, written $\overline{A}$, is the set of all the limits of sequences in $A$.

**Question**

Is $A \subseteq \overline{A}$?

A set $A$ is called *limit-closed* if $\overline{A} = A$. It is easy (but not relevant) to prove that limit-closed sets and closed sets are the same.

A set $A$ is called *dense* if $\overline{A} = \Sigma^\omega$ i.e. the closure is the space of all behaviours.
Safety Properties are Limit Closed

Let $P$ be a safety property.

- Assume that there exists a sequence of behaviours $\sigma_0\sigma_1\sigma_2 \ldots$ such that every $\sigma_i \in P$ but their limit $\sigma_\omega \notin P$.

- For $\sigma_\omega$ to violate the safety property $P$, there must be a specific state in $\sigma_\omega$ where shit hit the fan. That is, there must be a specific $k$ such that any behaviour with the prefix $\sigma_\omega|_k$ is not in $P$.

- For $\sigma_\omega$ to be the limit of our sequence, however, that means there is a particular point in our sequence $i$ after which all $\sigma_j$ for $j \geq i$ agree with $\sigma_\omega$ for the first $k + 1$ states. According to the above point, however, those $\sigma_j$ cannot be in $P$.

Contradiction.
Liveness Properties are Dense

Let $P$ be a liveness property. We want to show that $\overline{P}$ contains all behaviours, that is, that any behaviour $\sigma$ is the limit of some sequence of behaviours in $P$.

- If $\sigma \in P$, then just pick the sequence $\sigma\sigma\sigma\ldots$ which trivially converges to $\sigma$.
- If $\sigma \notin P$:
  - It must not “do the right thing eventually”, i.e. no finite prefix of $\sigma$ ever fulfills the promise of the liveness property.
  - However, every finite prefix $\sigma|_i$ of $\sigma$ could be extended differently with some $\rho_i$ such that $\sigma|_i\rho_i$ is in $P$ again.
  - Then, $\lim_{i \to \infty}(\sigma|_i\rho_i) = \sigma$ and thus $\sigma$ is the limit of a sequence in $P$. 
The Big Result

Alpern and Schneider’s Theorem

*Every property is the intersection of a safety and a liveness property*

\[ P = \overline{P} \cap \Sigma^\omega \setminus (\overline{P} \setminus P) \]

Why are these two components closed and dense? Also, let’s do the set theory reasoning to show this equality holds.

This is very significant, it gives us a separation of concerns: a concurrent program suggests correct actions (safety) and a scheduler chooses which actions to take (liveness).

Also, safety and liveness require different proof techniques.
Decomposing Safety and Liveness

Let’s break these up into their safety and liveness components.

- The program will stay in state $s_1$ for a while, then go to state $s_2$ and stay there forever.
- The program will allocate exactly 100MB of memory.
- If given an invalid input, the program will return the value -1.
- The program will sort the input list.
Something to think about.

| var n := 0 |
|------------------|------------------|
| p0: do 10 times: | q0: do 10 times: |
| p1: var x := n;  | q1: var y := n;  |
| p2: x := x + 1;  | q2: y := y + 1;  |
| p3: n := x;      | q3: n := y;      |
| p4: od           | q5: od           |

**Question**

What are the possible final values of $n$?