COMP3151/9154

Foundations of Concurrency

Course Introduction, Concurrent Semantics

Johannes Åman Pohjola
CSE, UNSW
Term 2 2022

Who are we?

I am Johannes Åman Pohjola. I will be the lecturer and course convenor.

Who are we?

I am Johannes Åman Pohjola. I will be the lecturer and course convenor.

Raphael Douglas Giles is the tutor. He 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. Questions about course content should typically be asked there. You can ask 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, but experience shows this is not always so. 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.

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, but experience shows this is not always so. 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

Lectures are on Wednesdays at 4PM in Law Theatre G23 (K-F8-G23), and Fridays at 11AM in Law Theatre G02 (K-F8-G02).
You can also participate remotely via Zoom.

Lecture recordings should pop up on Echo360.

Textbook

While we draw on a number of other sources. The one we draw the most from is Mordechai Ben-Ari's Principles of Concurrent and Distributed Programming. This book can be ordered from the campus bookshop at a ludicrous price. Other vendors are not much better.

Textbook

While we draw on a number of other sources. The one we draw the most from is Mordechai Ben-Ari's Principles of Concurrent and Distributed Programming. This book can be ordered from the campus bookshop at a ludicrous price. Other vendors are not much better.

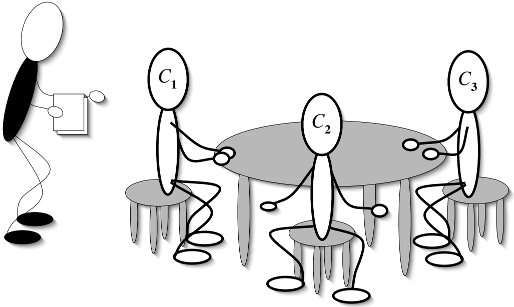
Copyright Infringement

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.

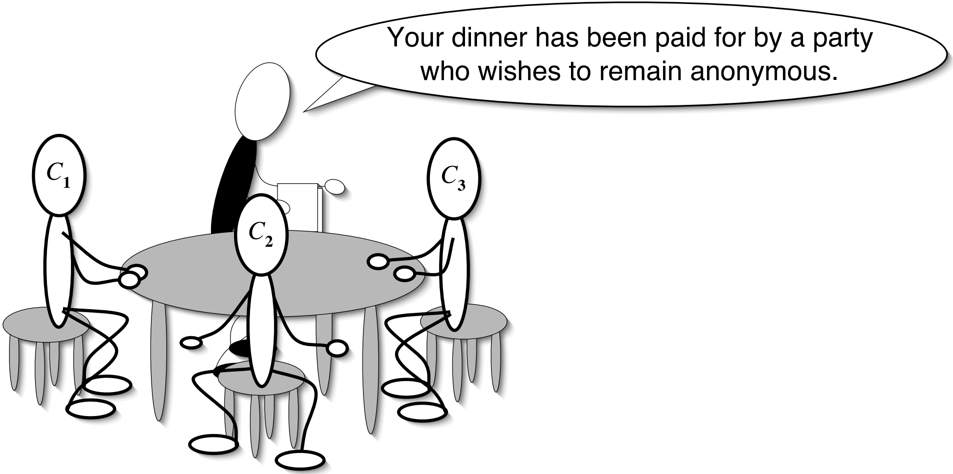
Dining Cryptographers Problem

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

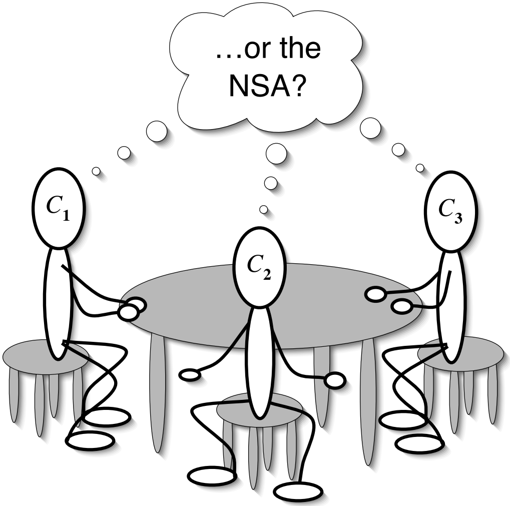


Dining Cryptographers

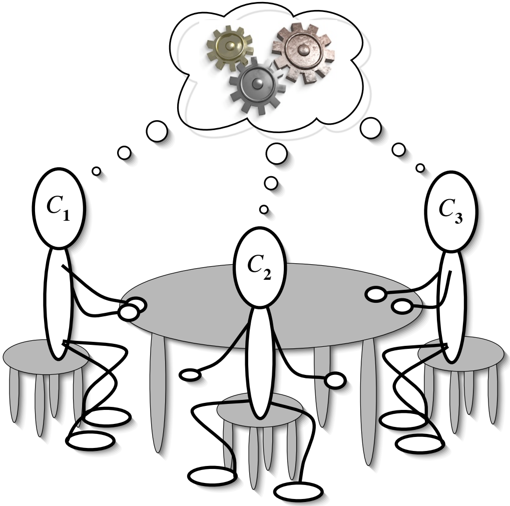


Dining Cryptographers

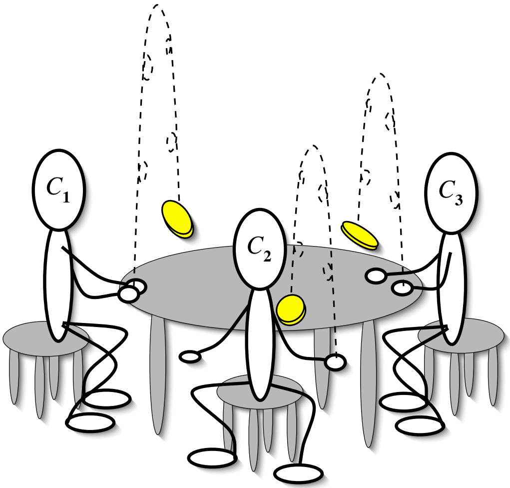
Dining Cryptographers



Dining Cryptographers



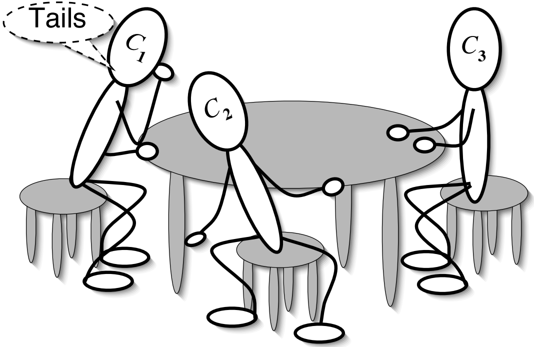
Dining Cryptographers



Procedure

1. Each C_i flips a coin.

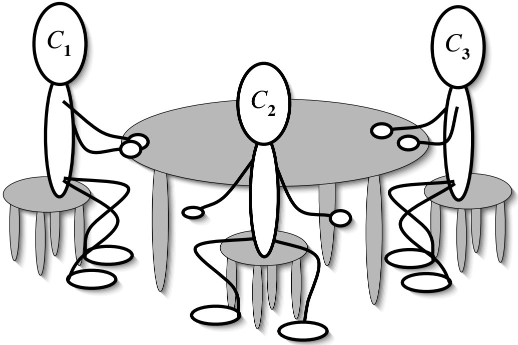
Dining Cryptographers



Procedure

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

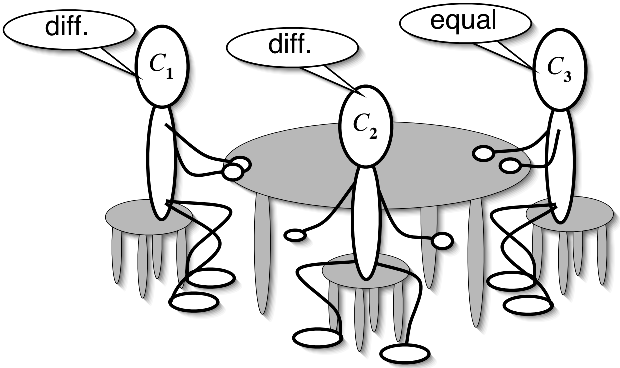
Dining Cryptographers



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.

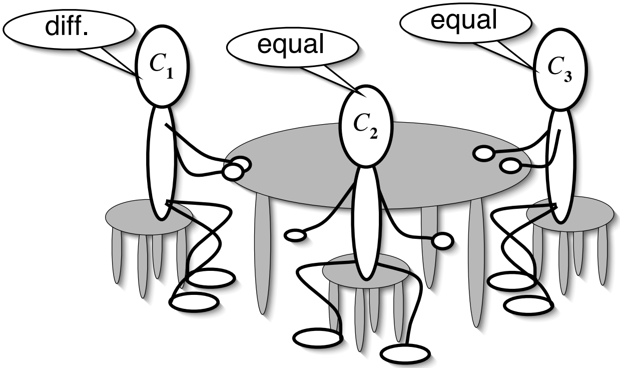
Dining Cryptographers



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.

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

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.

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*:

$$\bullet \longrightarrow \bullet \longrightarrow \bullet \longrightarrow \bullet \longrightarrow \bullet \longrightarrow \bullet \rightarrow \cdots$$

Sequential vs Concurrent

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

$$\bullet \longrightarrow \bullet \longrightarrow \bullet \longrightarrow \bullet \longrightarrow \bullet \longrightarrow \bullet \rightarrow \cdots$$

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

$$\texttt{LD R0,X} \longrightarrow \texttt{LDI R1,5} \longrightarrow \texttt{ADD R0,R1} \longrightarrow \texttt{ST X,R0}$$

Sequential vs Concurrent

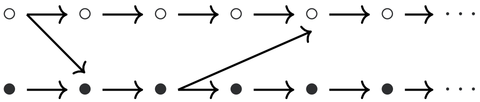
We could consider a *sequential* program (a *process* or *thread*) as a sequence (or *total order*) of *actions*:

$$\bullet \longrightarrow \bullet \longrightarrow \bullet \longrightarrow \bullet \longrightarrow \bullet \longrightarrow \bullet \longrightarrow \cdots$$

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

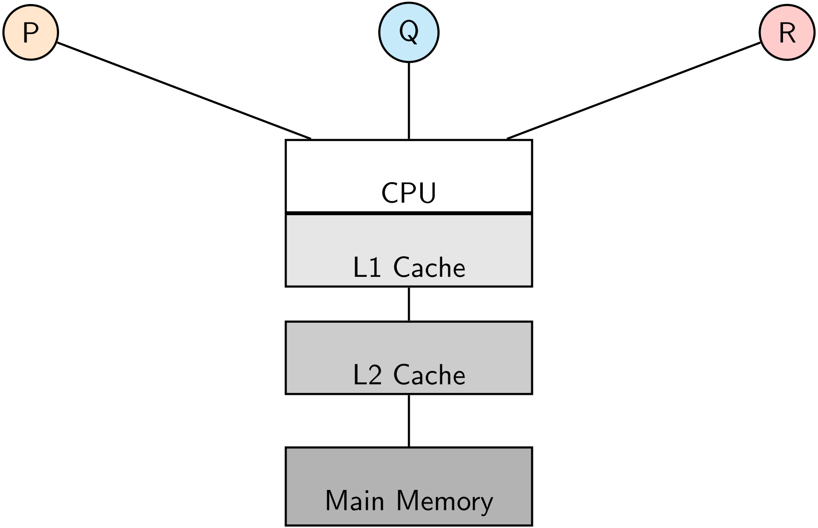
$$\texttt{LD R0,X} \longrightarrow \texttt{LDI R1,5} \longrightarrow \texttt{ADD R0,R1} \longrightarrow \texttt{ST X,R0}$$

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

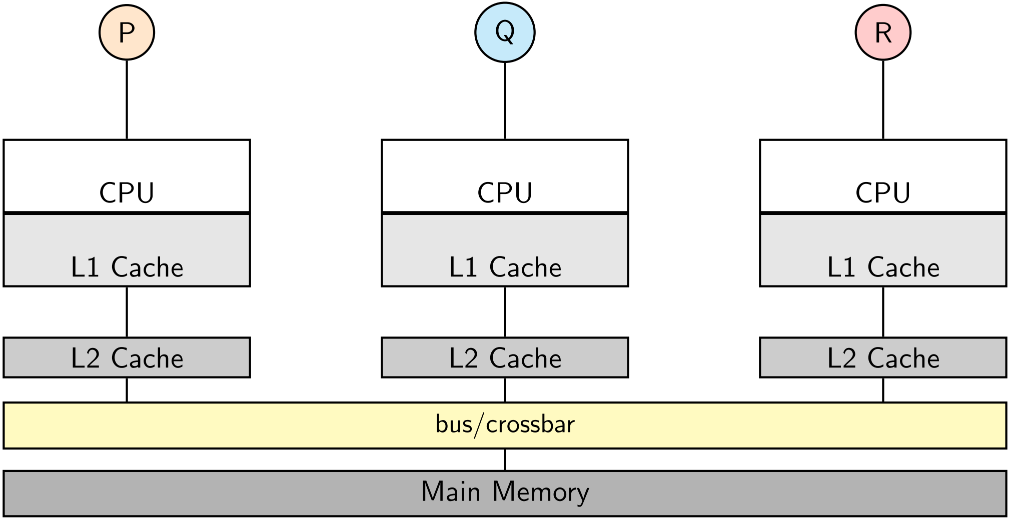


This means that there are now multiple possible *interleavings* of these actions — our program is non-deterministic where the interleaving is left to the *execution model*.

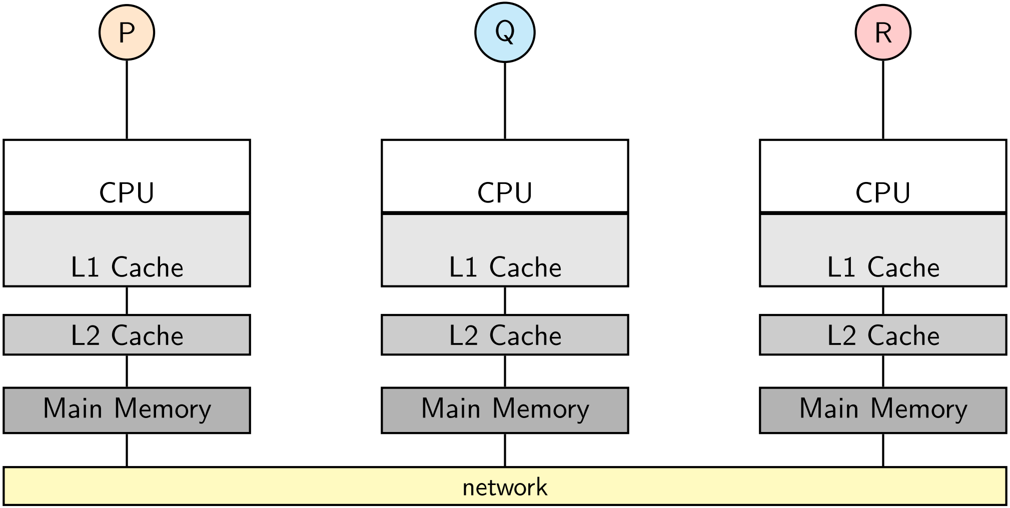
Multithreaded Execution



Parallel Multiprocessor Execution



Parallel Distributed Execution



Synchronisation

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

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.

Synchronisation

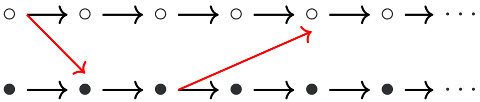
Regardless of the execution model, processes need to communicate to organise and co-ordinate their actions.

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:



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.

In a nutshell

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

1. **R**eading concurrent code and programming idioms in a variety of execution contexts.
2. w**R**iting concurrent software using various abstractions for synchronisation.

In a nutshell

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

1. **R**eading concurrent code and programming idioms in a variety of execution contexts.
2. w**R**iting concurrent software using various abstractions for synchronisation.
3. **R**easoning about concurrent systems with formal proof and automatic analysis tools.

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. w**R**iting 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.

$$\{\varphi\}\ P\ \{\psi\}$$

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* φ and it terminates, it will end in a state satisfying the *post-condition* ψ.

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* φ and it terminates, it will end in a state satisfying the *post-condition* ψ.

Semantics

Consider each action as a function from state to state $\Sigma \to \Sigma$.

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* φ and it terminates, it will end in a state satisfying the *post-condition* ψ.

Semantics

Consider each action as a function from state to state $\Sigma \to \Sigma$. Then the *semantics* or meaning of a sequential program $[\![P]\!]$ is the composition of all the functions in the sequence. Then the above Hoare triple actually means:

$$\forall s \in \Sigma.\ \varphi(s) \Rightarrow \psi([\![P]\!](s))$$

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* φ and it terminates, it will end in a state satisfying the *post-condition* ψ.

Semantics

Consider each action as a function from state to state $\Sigma \to \Sigma$. Then the *semantics* or meaning of a sequential program $[\![P]\!]$ is the composition of all the functions in the sequence. Then the above Hoare triple actually means:

$$\forall s \in \Sigma.\ \varphi(s) \Rightarrow \psi([\![P]\!](s))$$

Note that we only care about the initial and final states here.

Concurrent Programs

Consider the following concurrent processes, sharing a variable n.

var $n := 0$		
p_1: **var** $x := n$;	q_1: **var** $y := n$;	r_1: **var** $z := n$;
p_2: $n := x + 1$;	q_2: $n := y - 1$;	r_2: $n := z + 1$;

Question

What are the possible final values of n?

Concurrent Programs

Consider the following concurrent processes, sharing a variable n.

var $n := 0$		
p_1: **var** $x := n$;	q_1: **var** $y := n$;	r_1: **var** $z := n$;
p_2: $n := x + 1$;	q_2: $n := y - 1$;	r_2: $n := z + 1$;

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.

Semantics for Concurrency

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

Many concurrent systems never terminate, but instead run forever waiting for new requests (e.g. a server). So there may not be any final state!

Semantics for Concurrency

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

Many concurrent systems never terminate, but instead run forever waiting for new requests (e.g. a server). So there may not be any final state!

Behaviours

A *behaviour* is an infinite sequence of states, i.e. Σ^ω.

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.

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:

$$[\![P]\!] \subseteq S$$

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.

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.

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.

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).

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.

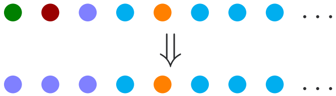
Internal vs. External State

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.



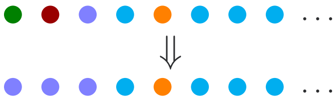
Internal vs. External State

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.

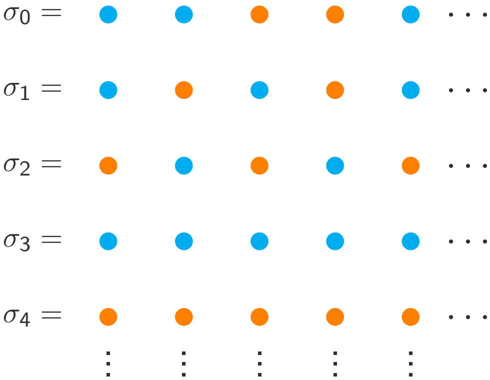


This kind of (finite) repetition of the same state is called *stuttering*. We generally don't want properties to distinguish behaviours that are equivalent modulo stuttering.

Cantor's Uncountability Argument

Result

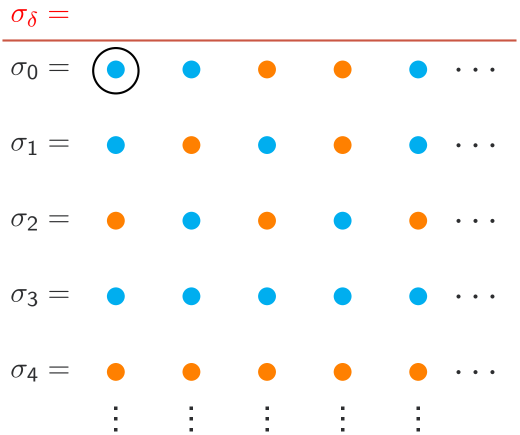
It is impossible in general to enumerate the space of all behaviours.



Cantor's Uncountability Argument

Result

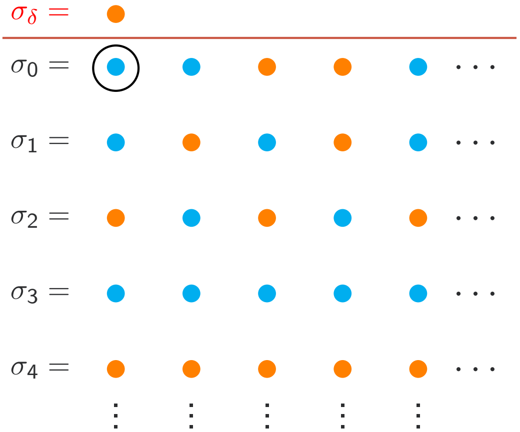
It is impossible in general to enumerate the space of all behaviours.



Cantor's Uncountability Argument

Result

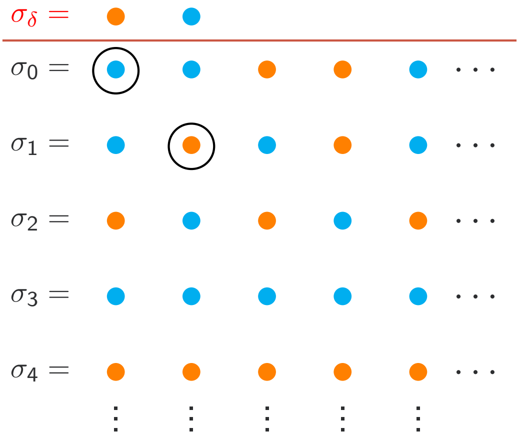
It is impossible in general to enumerate the space of all behaviours.



Cantor's Uncountability Argument

Result

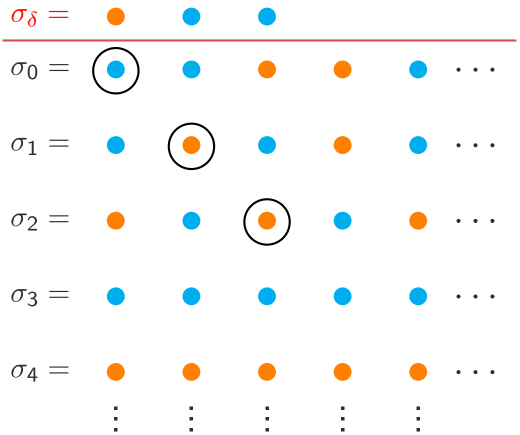
It is impossible in general to enumerate the space of all behaviours.



Cantor's Uncountability Argument

Result

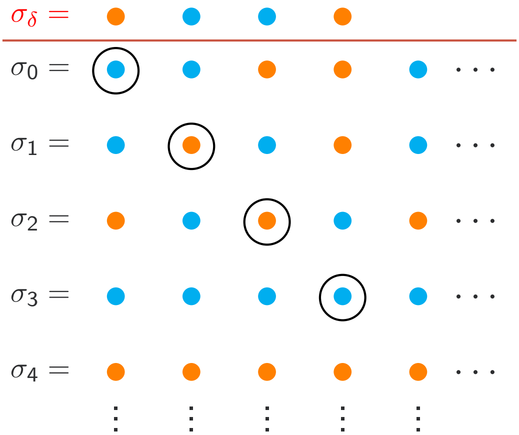
It is impossible in general to enumerate the space of all behaviours.



Cantor's Uncountability Argument

Result

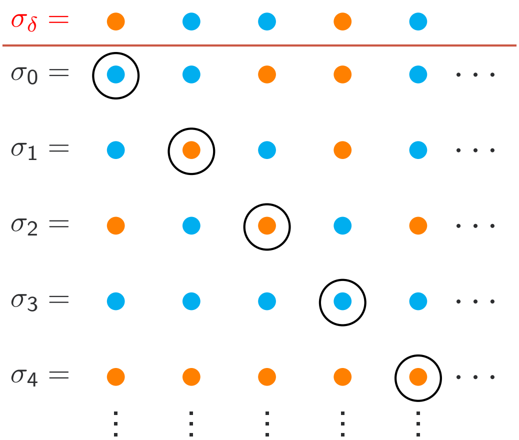
It is impossible in general to enumerate the space of all behaviours.



Cantor’s Uncountability Argument

Result

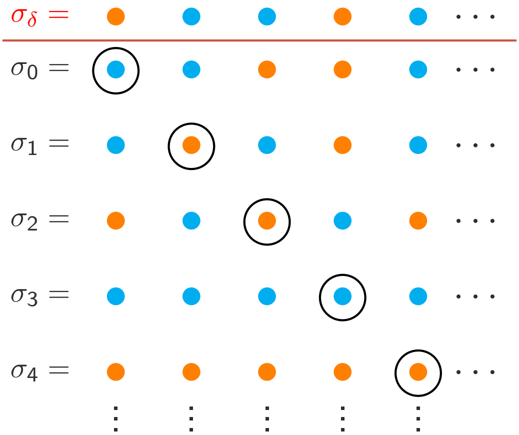
It is impossible in general to enumerate the space of all behaviours.



Cantor's Uncountability Argument

Result

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



Proof

Suppose there exists a set of behaviours $\sigma_0, \sigma_1, \sigma_2, \ldots$ that enumerates all behaviours.

Then we can construct a delightfully devilish behaviour σ_δ that differs from any σ_i at the ith position, and thus is not in our sequence.

Contradiction!

Properties

Recall

A linear temporal property is a set of behaviours.

Properties

Recall

A linear temporal property is a set of behaviours.

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

   I will never run out of money.

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

Properties

Recall

A linear temporal property is a set of behaviours.

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

   I 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 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*

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*

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*

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*

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*

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*

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.

Limits

If σ is a behaviour, we write $\sigma|_k$ to denote the prefix of σ comprising its first k states.

Definition (Limit closure)

The *limit closure* of a set $A \subseteq \Sigma^\omega$, denoted $\overline{A}$, is defined as follows:

$$\overline{A} = \{\sigma \in \Sigma^\omega \mid \forall n \in \mathbb{N}.\ \exists \sigma' \in A.\ \sigma|_n = \sigma'|_n\}$$

In words: a behaviour σ is in $\overline{A}$ if every finite prefix of σ is also a prefix of some behaviour in A.
Intuitively: $\overline{A}$ is all behaviours that cannot be distinguished from behaviours in A by making finite observations.

Limits

Example

What is $\overline{\emptyset}$?

Limits

Example

What is $\overline{\emptyset}$?

Example

Let $\Sigma = \{0, 1\}$, and let A be the set of all behaviours that start with a finite number of 0:s, followed by infinitely many 1:s. What is $\overline{A}$?

Limits

Definition (Limit closed sets)

A set A of behaviours is *limit closed* if $\overline{A} = A$.

Definition (Dense sets)

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.

Safety Properties are Limit Closed

Let P be a safety property.

- Assume that there exists a behaviour $\sigma_\omega \in \overline{P}$ such that $\sigma_\omega \notin P$.

Safety Properties are Limit Closed

Let P be a safety property.

- Assume that there exists a behaviour $\sigma_\omega \in \overline{P}$ such that $\sigma_\omega \notin P$.
- For σ_ω to violate the safety property P, there must be a specific state in σ_ω where shit hit the fan.

Safety Properties are Limit Closed

Let P be a safety property.

- Assume that there exists a behaviour $\sigma_\omega \in \overline{P}$ such that $\sigma_\omega \notin P$.
- For σ_ω to violate the safety property P, there must be a specific state in σ_ω 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.

Safety Properties are Limit Closed

Let P be a safety property.

- Assume that there exists a behaviour $\sigma_\omega \in \overline{P}$ such that $\sigma_\omega \notin P$.
- For σ_ω to violate the safety property P, there must be a specific state in σ_ω 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.
- Since $\sigma_\omega \in \overline{P}$, there must be a behaviour $\sigma \in P$ such that $\sigma_\omega|_k = \sigma|_k$.

Safety Properties are Limit Closed

Let P be a safety property.

- Assume that there exists a behaviour $\sigma_\omega \in \overline{P}$ such that $\sigma_\omega \notin P$.
- For σ_ω to violate the safety property P, there must be a specific state in σ_ω 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.
- Since $\sigma_\omega \in \overline{P}$, there must be a behaviour $\sigma \in P$ such that $\sigma_\omega|_k = \sigma|_k$.
- Thus, σ both violates and satisfies the property P.

Safety Properties are Limit Closed

Let P be a safety property.

- Assume that there exists a behaviour $\sigma_\omega \in \overline{P}$ such that $\sigma_\omega \notin P$.
- For σ_ω to violate the safety property P, there must be a specific state in σ_ω 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.
- Since $\sigma_\omega \in \overline{P}$, there must be a behaviour $\sigma \in P$ such that $\sigma_\omega|_k = \sigma|_k$.
- Thus, σ both violates and satisfies the property P.

Contradiction.

Liveness Properties are Dense

Let P be a liveness property. We want to show that $\overline{P}$ contains all behaviours. Let σ be a behaviour.

Liveness Properties are Dense

Let P be a liveness property. We want to show that $\overline{P}$ contains all behaviours. Let σ be a behaviour.

- If $\sigma \in P$,

Liveness Properties are Dense

Let P be a liveness property. We want to show that $\overline{P}$ contains all behaviours. Let σ be a behaviour.

- If $\sigma \in P$, then $\sigma \in \overline{P}$, because $P \subseteq \overline{P}$.
- If $\sigma \notin P$:
  - σ must not "do the right thing eventually", i.e. no finite prefix of σ ever fulfills the promise of the liveness property.

Liveness Properties are Dense

Let P be a liveness property. We want to show that $\overline{P}$ contains all behaviours. Let σ be a behaviour.

- If $\sigma \in P$, then $\sigma \in \overline{P}$, because $P \subseteq \overline{P}$.
- If $\sigma \notin P$:
  - σ must not "do the right thing eventually", i.e. no finite prefix of σ ever fulfills the promise of the liveness property.
  - However, every finite prefix $\sigma|_i$ of σ could be extended differently with some ρ_i such that $\sigma|_i\rho_i$ is in P again.

Liveness Properties are Dense

Let P be a liveness property. We want to show that $\overline{P}$ contains all behaviours. Let σ be a behaviour.

- If $\sigma \in P$, then $\sigma \in \overline{P}$, because $P \subseteq \overline{P}$.
- If $\sigma \notin P$:
  - σ must not "do the right thing eventually", i.e. no finite prefix of σ ever fulfills the promise of the liveness property.
  - However, every finite prefix $\sigma|_i$ of σ could be extended differently with some ρ_i such that $\sigma|_i\rho_i$ is in P again.
  - In other words, every finite prefix of σ is a prefix of some behaviour in P.

Liveness Properties are Dense

Let P be a liveness property. We want to show that $\overline{P}$ contains all behaviours. Let σ be a behaviour.

- If $\sigma \in P$, then $\sigma \in \overline{P}$, because $P \subseteq \overline{P}$.
- If $\sigma \notin P$:
  - σ must not "do the right thing eventually", i.e. no finite prefix of σ ever fulfills the promise of the liveness property.
  - However, every finite prefix $\sigma|_i$ of σ could be extended differently with some ρ_i such that $\sigma|_i\rho_i$ is in P again.
  - In other words, every finite prefix of σ is a prefix of some behaviour in P.
  - Thus, by definition, $\sigma \in \overline{P}$.

The Big Result

Alpern and Schneider's Theorem

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

The Big Result

Alpern and Schneider's Theorem

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

$$P = \underbrace{\overline{P}}_{\text{closed}} \cap \underbrace{\Sigma^\omega \setminus (\overline{P} \setminus P)}_{\text{dense}}$$

The Big Result

Alpern and Schneider's Theorem

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

$$P = \underbrace{\overline{P}}_{\text{closed}} \cap \underbrace{\Sigma^\omega \setminus (\overline{P} \setminus P)}_{\text{dense}}$$

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

The Big Result

Alpern and Schneider's Theorem

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

$$P \quad = \quad \underbrace{\overline{P}}_{\text{closed}} \quad \cap \quad \underbrace{\Sigma^\omega \setminus (\overline{P} \setminus P)}_{\text{dense}}$$

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.

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.

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.

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 $n := 0$	
p_0: **do** 10 times:	q_0: **do** 10 times:
p_1: **var** $x := n$;	q_1: **var** $y := n$;
p_2: $x := x + 1$;	q_2: $y := y + 1$;
p_3: $n := x$;	q_3: $n := y$;
p_4: **od**	q_5: **od**

Question

What are the possible final values of n?

Bonus Topological Detour

The following slides are non-examinable bonus material.

It's an alternative way of defining limit closures by drawing on a topological characterisation of Σ^ω. We won't need this for the course, so feel free to skip, and don't worry if you find it challenging.

Metric for Behaviours

We define the *distance* $d(\sigma, \rho) \in \mathbb{R}_{\geq 0}$ between two behaviours σ and ρ 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 σ and $2^{-\infty} = 0$.
Intuitively, we consider two behaviours to be close if there is a long prefix for which they agree.

Observations

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

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 σ_ω is called a *limit* of this sequence if the sequence *converges* to σ_ω, i.e. for any positive ε:

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

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.