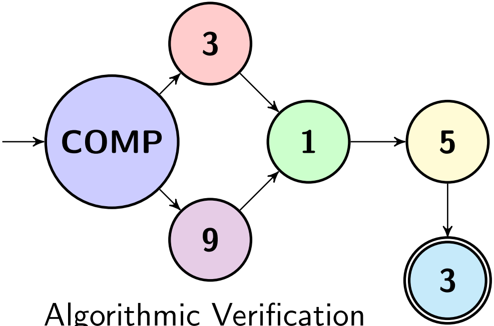


Algorithmic Verification

Course Introduction, Logics and Automata

Dr. Liam O'Connor
CSE, UNSW (for now)
Term 1 2020

Who are we?

I am Dr. Liam O'Connor. I do research work on formal methods and programming languages, and casual teaching at UNSW.

Who are we?

I am Dr. Liam O'Connor. I do research work on formal methods and programming languages, and casual teaching at UNSW.

Dr. Miki Tanaka is a senior research engineer at CSIRO/Data61 who works on, among other things, formal verification of mixed-criticality real-time systems.

Prof. Rob van Glabbeek is a leading expert on the theory of concurrent computation, with numerous seminal contributions to the field.

Who are we?

I am Dr. Liam O'Connor. I do research work on formal methods and programming languages, and casual teaching at UNSW.

Dr. Miki Tanaka is a senior research engineer at CSIRO/Data61 who works on, among other things, formal verification of mixed-criticality real-time systems.

Prof. Rob van Glabbeek is a leading expert on the theory of concurrent computation, with numerous seminal contributions to the field.

A/Prof. Peter Höfner, who now works at ANU, is the former lecturer of this course. Hopefully we can maintain the high standard he set.

Contacting Us

http://www.cse.unsw.edu.au/~cs3153

Forum

There is a Piazza 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.

I highly recommend disabling the Piazza Careers rubbish.

Administrative questions should be sent to liamoc@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 *induction*. MATH1081 ought to be sufficient for aptitude in these skills, but experience has shown this is not always true.

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 *induction*. MATH1081 ought to be sufficient for aptitude in these skills, but experience has shown this is not always true.

Programming

We expect you to be familiar with imperative programming languages like C. Course assignments may require some programming in modelling languages. Some self-study may be needed for these tools.

Assessment

There are **five** homework assignments for this course.

The final assessment is made up of your assignments plus the final exam, weighted 60/40 in favour of the exam.

Resources

Lecture Recordings

In previous years, no recordings were made available for this course. I will make them available this year, **however**:

Resources

Lecture Recordings

In previous years, no recordings were made available for this course. I will make them available this year, **however**:

Lecture recordings are **only** guaranteed to be usable up until week 3, due to students affected by coronavirus quarantines.

Resources

Lecture Recordings

In previous years, no recordings were made available for this course. I will make them available this year, **however**:

Lecture recordings are **only** guaranteed to be usable up until week 3, due to students affected by coronavirus quarantines.

After week 3, **no effort** will be made to make lecture recordings usable as substitutes for attendance.

Resources

Lecture Recordings

In previous years, no recordings were made available for this course. I will make them available this year, **however**:

Lecture recordings are **only** guaranteed to be usable up until week 3, due to students affected by coronavirus quarantines.

After week 3, **no effort** will be made to make lecture recordings usable as substitutes for attendance.

Textbooks

This course follows more than one textbook. Each week's slides will include a bibliography. A list of books is given in the course outline, all of the books listed are available from the library.

Hardware Bugs: 1994 FDIV Bug

$$\frac{4195835}{3145727} =$$

Hardware Bugs: 1994 FDIV Bug

$$\frac{4195835}{3145727} = 1.33370$$

Missing entries in a hardware lookup table lead to 3-5 million defective floating point units.

Consequences:

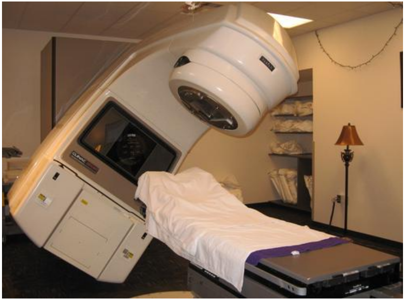
- Intel image badly damaged
- $450 million to replace FPUs.

Software Bugs: Asiana 777 Crash in 2014

Airline Blames Bad Software in San Francisco Crash

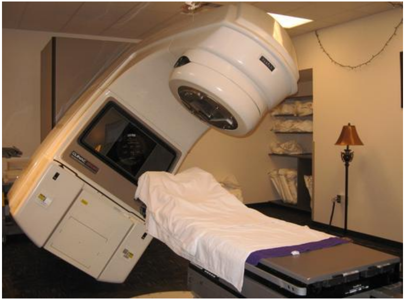
The New York Times

Software Bugs: Therac-25 (1980s)



- Radiation therapy machine.
- Two operation modes: high and low energy.
- Only supposed to use high energy mode with a shield.

Software Bugs: Therac-25 (1980s)



- Radiation therapy machine.
- Two operation modes: high and low energy.
- Only supposed to use high energy mode with a shield.
- Bug caused high energy mode to be used without shield.
- At least five patients died and many more exposed to high levels of radiation.

Software Bugs: Toyota Prius (2005)

- Sudden stalling at highway speeds.
- Bug triggered ”fail-safe” mode (heh).

Software Bugs: Toyota Prius (2005)

- Sudden stalling at highway speeds.
- Bug triggered "fail-safe" mode (heh).

Consequences:

- 75000 cars recalled.
- Cost unknown... but high.

Software Bugs: Ariane 5, Flight 501 (1996)

- Reuse of software from Ariane 4
- Overflow converting from 64 bit to 16 bit unsigned integers.

Software Bugs: Ariane 5, Flight 501 (1996)

- Reuse of software from Ariane 4
- Overflow converting from 64 bit to 16 bit unsigned integers.

Consequences:

- Rocket exploded after 37 seconds.
- US$370 million cost

Northeast Blackout (2003)

- Alarm went unnoticed.
- Bug in alarm system, probably due to a race condition.

Northeast Blackout (2003)

- Alarm went unnoticed.
- Bug in alarm system, probably due to a race condition.

Consequences:

- Total power failure for 7 hours, some areas up to 2 days.
- 55 million people affected
- More than US$6 billion cost

Verification

Ensuring that software or hardware satisfies requirements.

Verification

Ensuring that software or hardware satisfies requirements.

Requirements are:

- That it does what it's supposed to (morally, liveness)

Verification

Ensuring that software or hardware satisfies requirements.

Requirements are:

- That it does what it's supposed to (morally, liveness)
- That it doesn't do what it's not supposed to (morally, safety)

Verification

Ensuring that software or hardware satisfies requirements.

Requirements are:

- That it does what it's supposed to (morally, liveness)
- That it doesn't do what it's not supposed to (morally, safety)

We'll get to more precise definitions later.

Does a program satisfy requirements?

We could try testing, but it's not exhaustive.

Does a program satisfy requirements?

We could try testing, but it's not exhaustive.

Program testing can be used to show the presence of bugs, but never to show their absence!

Edsger W. Dijkstra (1970) "Notes On Structured Programming" (EWD249)

Does a program satisfy requirements?

We could try testing, but it's not exhaustive.

Program testing can be used to show the presence of bugs, but never to show their absence!

Edsger W. Dijkstra (1970) "Notes On Structured Programming" (EWD249)

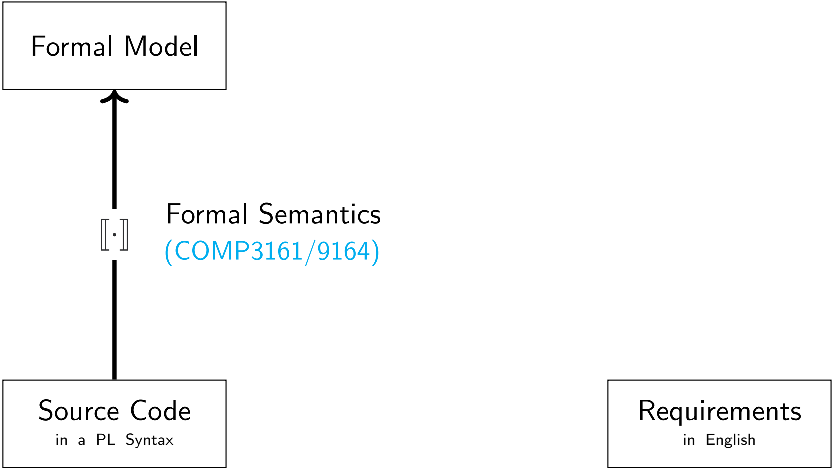
We want a rigorous and exhaustive method of verification.

Formal Verification

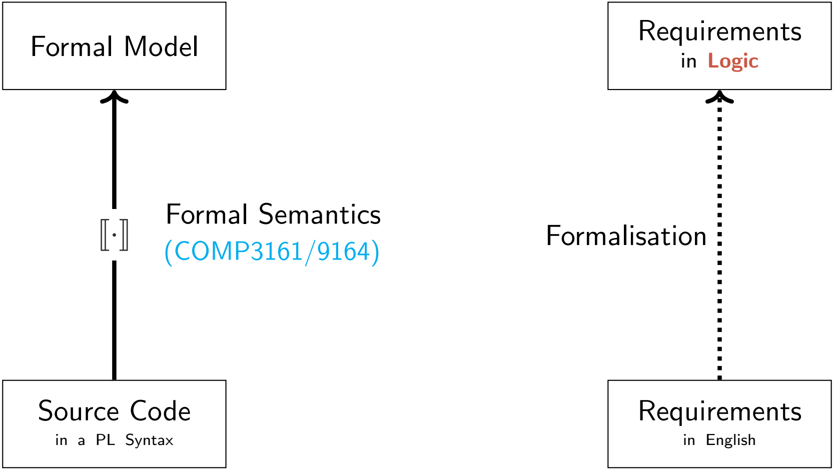
Source Code
in a PL Syntax

Requirements
in English

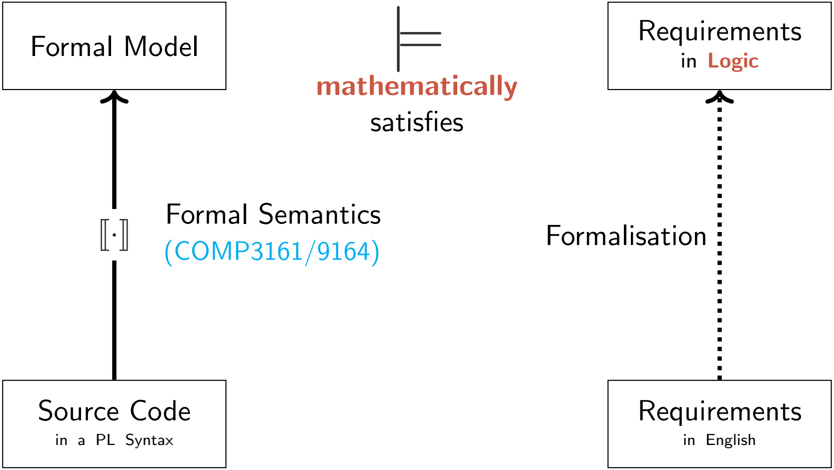
Formal Verification



Formal Verification



Formal Verification



Methods of Formal Verification

Method	Automation	Speed	Expressivity	Courses
Pen/Paper Proof	None	Slow	Unbounded	COMP6721, COMP2111
Proof Assistant	Some	Medium	Unbounded	COMP4161
Model Checking	Full	Fast	Limited	This course!
Static Analysis	Full	Fast	Limited	This course!

Methods of Formal Verification

Method	Automation	Speed	Expressivity	Courses
Pen/Paper Proof	None	Slow	Unbounded	COMP6721, COMP2111
Proof Assistant	Some	Medium	Unbounded	COMP4161
Model Checking	Full	Fast	Limited	This course!
Static Analysis	Full	Fast	Limited	This course!

The twin foci of this course:
Model Checking and **Static Analysis**.

Model Checking

Introduced intependently by Clarke, Emerson and Sistla (1980) and Queille and Sifakis (1980). **Turing Award 2007**

Formal Model

Some kind of finite automata.

Model Checking

Introduced intependently by Clarke, Emerson and Sistla (1980) and Queille and Sifakis (1980). **Turing Award 2007**

Formal Model

Some kind of finite automata.

Requirements

Specify dynamic requirements with a temporal logic (Pnueli 1977 - **Turing Award 1996**).

By dynamic we mean a property of the program's executions.

Model Checking

Introduced intependently by Clarke, Emerson and Sistla (1980) and Queille and Sifakis (1980). **Turing Award 2007**

Formal Model

Some kind of finite automata.

Requirements

Specify dynamic requirements with a temporal logic (Pnueli 1977 - **Turing Award 1996**).

By dynamic we mean a property of the program's executions.

Model checkers work by exhaustively checking the state space of the program against requirements.

Any forseeable problems with that?

State space explosion

Imagine a program with a 100 integer variables $\in [0, 10]$.

State space explosion

Imagine a program with a 100 integer variables $\in [0, 10]$.

- 10^{100} possible states.

State space explosion

Imagine a program with a 100 integer variables $\in [0, 10]$.

- 10^{100} possible states.
- Number of atoms in the universe: 10^{78}.

State space explosion

Imagine a program with a 100 integer variables $\in [0, 10]$.

- 10^{100} possible states.
- Number of atoms in the universe: 10^{78}.

Concurrency/nondeterminism also exhibits this problem. How many states are there for a program with n processes consisting of m steps each?

State space explosion

Imagine a program with a 100 integer variables $\in [0, 10]$.

- 10^{100} possible states.
- Number of atoms in the universe: 10^{78}.

Concurrency/nondeterminism also exhibits this problem. How many states are there for a program with n processes consisting of m steps each?

	$n = \mathbf{2}$	**3**	**4**	**5**	**6**
$m = \mathbf{2}$	6	90	2520	113400	$2^{22.8}$
3	20	1680	$2^{18.4}$	$2^{27.3}$	$2^{36.9}$
4	70	34650	$2^{25.9}$	$2^{38.1}$	$2^{51.5}$
5	252	$2^{19.5}$	$2^{33.4}$	$2^{49.1}$	$2^{66.2}$
6	924	$2^{24.0}$	$2^{41.0}$	$2^{60.2}$	$2^{81.1}$

State space explosion

Imagine a program with a 100 integer variables $\in [0, 10]$.

- 10^{100} possible states.
- Number of atoms in the universe: 10^{78}.

Concurrency/nondeterminism also exhibits this problem. How many states are there for a program with n processes consisting of m steps each?

	$n = \mathbf{2}$	**3**	**4**	**5**	**6**
$m = \mathbf{2}$	6	90	2520	113400	$2^{22.8}$
3	20	1680	$2^{18.4}$	$2^{27.3}$	$2^{36.9}$
4	70	34650	$2^{25.9}$	$2^{38.1}$	$2^{51.5}$
5	252	$2^{19.5}$	$2^{33.4}$	$2^{49.1}$	$2^{66.2}$
6	924	$2^{24.0}$	$2^{41.0}$	$2^{60.2}$	$2^{81.1}$

$$\frac{(nm)!}{m!^n}$$

State Space Explosion

There are many techniques to make model checking a more tractable problem, such as symbolic and bounded model checking, SAT-based techniques, and abstraction/refinement. We will examine these techniques throughout the course.

Tools

- SPIN, an explicit LTL model checker used for protocols, which uses heuristics to control state space.
- nuSMV, a symbolic model checker using binary decision diagrams.
- SLAM and CBMC, which are SAT-based tools using bounded model checking.

Static Analysis

Check static invariants about programs, about data or control flow.

Static Analysis

Check static invariants about programs, about data or control flow.

Example (Static Invariants)

No `NULL`-pointer dereferences, no array out-of-bound accesses.

Static Analysis

Check static invariants about programs, about data or control flow.

Example (Static Invariants)

No `NULL`-pointer dereferences, no array out-of-bound accesses.

Based on the abstract interpretation technique of Cousot and Cousot (1977). We'll look at this around Week 6, but:

Key Idea

Abstract from *specific values* to *classes of values*, increasing the non-determinism of the program but making it easier to analyse possible effects of the program.

Tools: ASTREE, Absint, Coverity, Grammatech, Polyspace, PVS-Studio, Goanna etc. etc.

Logic

We typically state our requirements with a logic.

Logic

We typically state our requirements with a logic.

Definition

A logic is a formal language designed to express logical reasoning.
Like any formal language, logics have a syntax and semantics.

Logic

We typically state our requirements with a logic.

Definition

A logic is a formal language designed to express logical reasoning. Like any formal language, logics have a syntax and semantics.

Example (Propositional Logic Syntax)

- A set of atomic propositions $\mathcal{P} = \{a, b, c, \ldots\}$
- An inductively defined set of formulae:
  - Each $p \in \mathcal{P}$ is a formula.
  - If P and Q are formulae, then $P \wedge Q$ is a formula.
  - If P is a formula, then $\neg P$ is a formula.

(Other connectives are just sugar for these, so we omit them)

Semantics

Semantics

Semantics are a mathematical representation of the meaning of a piece of syntax. There are many ways of giving a logic semantics, but we will use models.

Semantics

Semantics are a mathematical representation of the meaning of a piece of syntax. There are many ways of giving a logic semantics, but we will use models.

Example (Propositional Logic Semantics)

A model for propositional logic is a valuation $\mathcal{V} \subseteq \mathcal{P}$, a set of "true" atomic propositions. We can extend a valuation over an entire formula, giving us a satisfaction relation:

$$
\begin{array}{lcl}
\mathcal{V} \models p & \Leftrightarrow & p \in \mathcal{V} \\
\mathcal{V} \models \varphi \wedge \psi & \Leftrightarrow & \mathcal{V} \models \varphi \text{ and } \mathcal{V} \models \psi \\
\mathcal{V} \models \neg\varphi & \Leftrightarrow & \mathcal{V} \not\models \varphi
\end{array}
$$

We read $\mathcal{V} \models \varphi$ as $\mathcal{V}$ "satisfies" φ.

Automata

We will model our computations using finite automata.

Automata

We will model our computations using finite automata.

Definition

A finite automata (FA) is a quintuple $(Q, q_0, \Sigma, \delta, F)$ where:

- Q is a finite set of states.
- $q_0 \in Q$ is the initial state.
- Σ is a finite set of actions called an alphabet.
- δ is a transition relation $Q \times \Sigma \to 2^Q$.
- $F \subseteq Q$ is a set of final states.

A FA is called deterministic iff δ is a function, i.e.

$$\forall (s, a) \in Q \times \Sigma.\ |\delta(s, a)| \leq 1$$

Example: binary strings ending with double zero

Automata

A run from an automata A is a sequence of transitions:

$$q_0 \xrightarrow{a_1} q_1 \xrightarrow{a_2} \cdots \xrightarrow{a_{n-1}} q_{n-1} \xrightarrow{a_n} q_n$$

This run can also be written $q_0 \xrightarrow{a_1 a_2 \ldots a_n} q_n$ or, if we don't care about the actions $q_0 \xrightarrow{\star} q_n$.

Automata

A run from an automata A is a sequence of transitions:

$$q_0 \xrightarrow{a_1} q_1 \xrightarrow{a_2} \cdots \xrightarrow{a_{n-1}} q_{n-1} \xrightarrow{a_n} q_n$$

This run can also be written $q_0 \xrightarrow{a_1 a_2 \ldots a_n} q_n$ or, if we don't care about the actions $q_0 \xrightarrow{\star} q_n$.

The language $\mathcal{L}(A)$ of an automata A is all sequences of actions (words) whose runs end in the set of final states F:

$$\mathcal{L}(A) = \{w \in \Sigma^* \mid q_0 \xrightarrow{w} q, q \in F\}$$

Non-determinism

Non-deterministic finite automata can be converted to deterministic finite automata, by using sets of NFA states as the set of states for the DFA (the subset construction).

Liam: Example on board

Non-determinism

Non-deterministic finite automata can be converted to deterministic finite automata, by using sets of NFA states as the set of states for the DFA (the subset construction).

Liam: Example on board

ε-transitions

We can enrich NFAs with transitions that do not have actions (or equivalently, transitions with the empty word ε as their action) without affecting expressiveness. Subset construction still works.

Non-determinism

Non-deterministic finite automata can be converted to deterministic finite automata, by using sets of NFA states as the set of states for the DFA (the subset construction).

Liam: Example on board

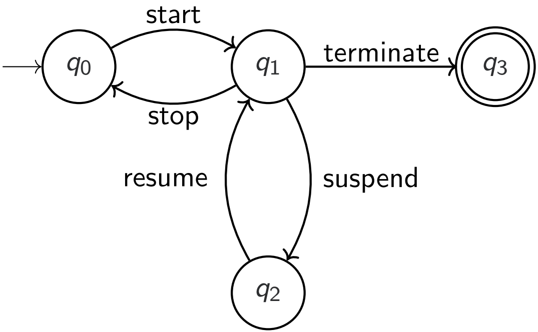
ε-transitions

We can enrich NFAs with transitions that do not have actions (or equivalently, transitions with the empty word ε as their action) without affecting expressiveness. Subset construction still works.

Thus,

$$\text{DFA} = \text{NFA} = \text{NFA}^{\varepsilon}$$

Modelling with Automata



What sort of runs can this automata produce?

Intersection of Languages

Problem

Let A be a FA such that $\mathcal{L}(A)$ is the set of strings with an even number of as.

Intersection of Languages

Problem

Let A be a FA such that $\mathcal{L}(A)$ is the set of strings with an even number of $\mathtt{a}$s.
Let B be a FA such that $\mathcal{L}(B)$ is the set of strings with an odd number of $\mathtt{b}$s.

Intersection of Languages

Problem

Let A be a FA such that $\mathcal{L}(A)$ is the set of strings with an even number of $\mathtt{a}$s.
Let B be a FA such that $\mathcal{L}(B)$ is the set of strings with an odd number of $\mathtt{b}$s.
How can we combine A and B into a new automata C such that $\mathcal{L}(C) = \mathcal{L}(A) \cap \mathcal{L}(B)$?

(try to come up with a general technique for any automata)

Intersection of Languages

Problem

Let A be a FA such that $\mathcal{L}(A)$ is the set of strings with an even number of $\mathtt{a}$s.
Let B be a FA such that $\mathcal{L}(B)$ is the set of strings with an odd number of $\mathtt{b}$s.
How can we combine A and B into a new automata C such that $\mathcal{L}(C) = \mathcal{L}(A) \cap \mathcal{L}(B)$?
(try to come up with a general technique for any automata)

We need to create the product of two automata.

Automata Product

Definition

The product of two automata
$A_1 = (Q_1, q_0^1, \Sigma_1, \delta_1, F_1)$ and
$A_2 = (Q_2, q_0^2, \Sigma_2, \delta_2, F_2)$
is defined as: $(Q, q_0, \Sigma, \delta, F)$ where:

Automata Product

Definition

The product of two automata
$A_1 = (Q_1, q_0^1, \Sigma_1, \delta_1, F_1)$ and
$A_2 = (Q_2, q_0^2, \Sigma_2, \delta_2, F_2)$
is defined as: $(Q, q_0, \Sigma, \delta, F)$ where:

- $Q = Q_1 \times Q_2$

Automata Product

Definition

The product of two automata
$A_1 = (Q_1, q_0^1, \Sigma_1, \delta_1, F_1)$ and
$A_2 = (Q_2, q_0^2, \Sigma_2, \delta_2, F_2)$
is defined as: $(Q, q_0, \Sigma, \delta, F)$ where:

- $Q = Q_1 \times Q_2$
- $q_0 = (q_0^1, q_0^2)$

Automata Product

Definition

The product of two automata
$A_1 = (Q_1, q_0^1, \Sigma_1, \delta_1, F_1)$ and
$A_2 = (Q_2, q_0^2, \Sigma_2, \delta_2, F_2)$
is defined as: $(Q, q_0, \Sigma, \delta, F)$ where:

- $Q = Q_1 \times Q_2$
- $q_0 = (q_0^1, q_0^2)$
- $\Sigma = \Sigma_1 \cup \Sigma_2$

Automata Product

Definition

The product of two automata
$A_1 = (Q_1, q_0^1, \Sigma_1, \delta_1, F_1)$ and
$A_2 = (Q_2, q_0^2, \Sigma_2, \delta_2, F_2)$
is defined as: $(Q, q_0, \Sigma, \delta, F)$ where:

- $Q = Q_1 \times Q_2$
- $q_0 = (q_0^1, q_0^2)$
- $\Sigma = \Sigma_1 \cup \Sigma_2$
- $\delta(\ (q_1, q_2)\ , a) =$
$$\begin{cases} \{(q_1', q_2') \mid q_1' \in \delta_1(q_1, a), q_2' \in \delta_2(q_2, a)\} & \text{if } a \in \Sigma_1 \cap \Sigma_2 \\ \{(q_1', q_2) \mid q_1' \in \delta_1(q_1, a)\} & \text{if } a \in \Sigma_1 \setminus \Sigma_2 \\ \{(q_1, q_2') \mid q_2' \in \delta_2(q_2, a)\} & \text{if } a \in \Sigma_2 \setminus \Sigma_1 \end{cases}$$

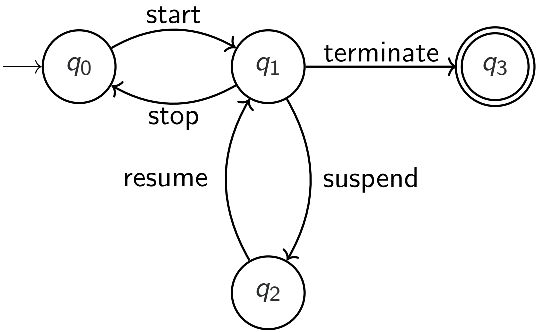
Automata Product

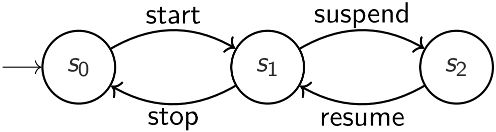
Definition

The product of two automata
$A_1 = (Q_1, q_0^1, \Sigma_1, \delta_1, F_1)$ and
$A_2 = (Q_2, q_0^2, \Sigma_2, \delta_2, F_2)$
is defined as: $(Q, q_0, \Sigma, \delta, F)$ where:

- $Q = Q_1 \times Q_2$
- $q_0 = (q_0^1, q_0^2)$
- $\Sigma = \Sigma_1 \cup \Sigma_2$
- $\delta(\ (q_1, q_2)\ , a) =$
  $$\begin{cases} \{(q_1', q_2') \mid q_1' \in \delta_1(q_1, a), q_2' \in \delta_2(q_2, a)\} & \text{if } a \in \Sigma_1 \cap \Sigma_2 \\ \{(q_1', q_2) \mid q_1' \in \delta_1(q_1, a)\} & \text{if } a \in \Sigma_1 \setminus \Sigma_2 \\ \{(q_1, q_2') \mid q_2' \in \delta_2(q_2, a)\} & \text{if } a \in \Sigma_2 \setminus \Sigma_1 \end{cases}$$
- $F = F_1 \times F_2$

Task and Scheduler





Products can encode communication. Compute the product of these two processes.

Integer Variables

Problem

Imagine we extended our notion of actions to allow automata to read or write from a finite set of bounded integer variables.
Does this affect the expressivity of automata?

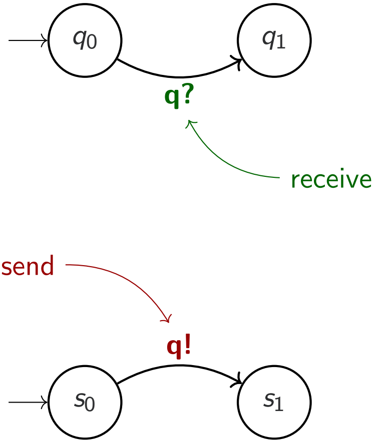
Integer Variables

Problem

Imagine we extended our notion of actions to allow automata to read or write from a finite set of bounded integer variables. Does this affect the expressivity of automata?

No. We can encode the integers as automata and use synchronisation. (demonstrate on whiteboard)

Message passing



Different tools offer broadcast or unicast communication. **Check the manual!**

Bibliography

Propositional Logic:

- Huth/Ryan: Logic in Computer Science, Section 1
- Bayer/Katoen: Principles of Model Checking, Appendix A3

Automata:

- Sipser: Introduction to the Theory of Computation, sections 1.1 and 1.2
- Kozen: Automata and Computability, Sections 3-5