COMP2111

System Modelling and Design

COMP2111 19t1 Staff

Lecturer:	Paul Hunter (W12)
Email:	paul.hunter@unsw.edu.au
Research:	Theoretical CS: Algorithms, Formal verification
Tutors:	Charles Bradford (H12, H13), Daniel Brownlow (T14, W13), Tsz (Edward) Lu (W14, H10), Harrison Scott (W16, W17)

What is this course?

Slightly different from previous years (not as intense!)

What is this course?

Bridge between MATH1081 and SENG2011
(and COMP3151, COMP3153, COMP3161, COMP4181, COMP4141, COMP4418, COMP6752, COMP4161)

- Reinforce concepts from Discrete Mathematics
- Emphasise the connection between Discrete Mathematics and Computer Science
- Use mathematical concepts to **reason about programs**

Why do we want to reason about programs?

- Next step in programming to meet requirements
- Provable behaviour
- Provable security
  - seL4
- Identify errors
  - Pentium floating point error
- Identify optimizations
  - `if true then S else T` simplifying to S

Why do we want to reason about programs?

- Next step in programming to meet requirements
- Provable behaviour
- Provable security
  - seL4
- Identify errors
  - Pentium floating point error
- Identify optimizations
  - `if true then S else T` simplifying to S

Why do we want to reason about programs?

- Next step in programming to meet requirements
- Provable behaviour
- Provable security
  - seL4
- Identify errors
  - Pentium floating point error
- Identify optimizations
  - `if true then S else T` simplifying to S

Why do we want to reason about programs?

- Next step in programming to meet requirements
- Provable behaviour
- Provable security
  - seL4
- Identify errors
  - Pentium floating point error
- Identify optimizations
  - `if true then S else T` simplifying to `S`

How?

- Acquire (and understand) languages to **formally specify** systems
- Acquire (and understand) structures to **formally model** systems
- Learn how to prove that a program satisfies its specification

Why all the formality?

- Avoid ambiguity
- Automate the procedure

Code

```
...
slowCarDown()

...
```

Specification

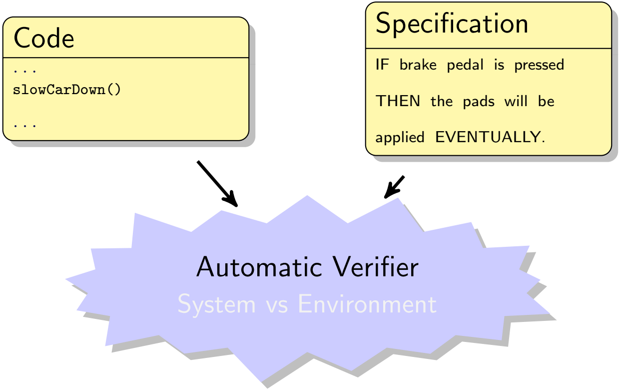
IF brake pedal is pressed

THEN the pads will be

applied EVENTUALLY.

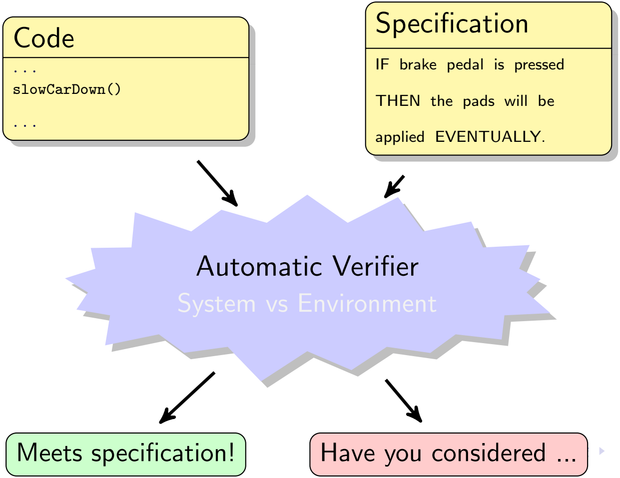
Why all the formality?

- Avoid ambiguity
- Automate the procedure



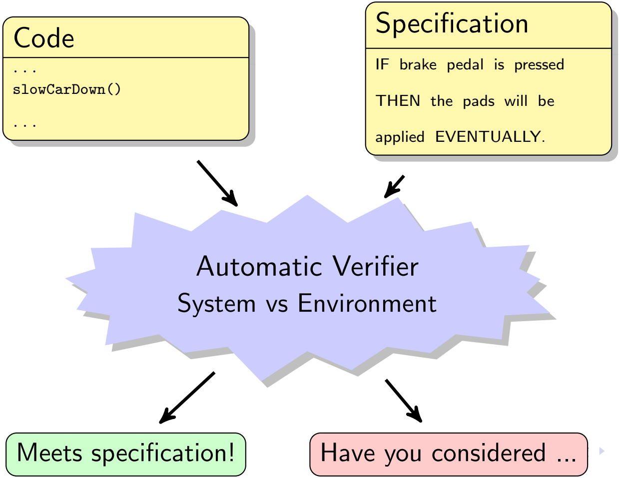
Why all the formality?

- Avoid ambiguity
- Automate the procedure



Why all the formality?

- Avoid ambiguity
- Automate the procedure



An example: Factorial (definition)

The factorial function $! : \mathbb{N} \to \mathbb{N}$ can be defined as:

- $0! = 1$
- $(n+1)! = (n+1) \cdot n!$

The first line tells us how to compute $0!$, whereas the second line tells us how to compute the factorial of a positive number if we know the factorial of its predecssor.

Together they are known as an *inductive definition* of the (mathematical) factorial function.

An example: Factorial (specification to implementation)

Task: Given a number $n \in \mathbb{N}$ compute its factorial $n!$ without changing n in the process.

Plan:

1. Compute $0!$
2. Repeatedly use the second property to compute factorials of larger numbers

Simple? Any problems?

An example: Factorial (correctness)

Depends on the language.

In Haskell:

```
fact :: Integer → Integer
fact 0 = 1
fact n = n * (fact (n-1))
```

In C:

```
unsigned int fact(unsigned int n){
  return (n==0)?1:n*fact(n-1);
}
```

An example: Factorial (specification to code II)

Recursion is good, but what about an iterative version?

Idea: Use a variable f to save the last factorial we have computed, and an additional variable k to keep track of the number such that $f = k!$. So the plan becomes:

1. Achieve $f = k!$ by setting $f = 1$ and $k = 0$.
2. As long as $k \neq n$, increase k and change f in a way that preserves $f = k!$

NB

This is an example of a **Dynamic Programming** *solution.*

An example: Factorial (correctness)

The property that $f = k!$ is a **loop invariant**. Loop bodies will generally change the state, but loop invariants express properties that are preserved when executing the loop body. At the completion of the loop, we have that $k = n$ so the loop invariant tells us that $f = n!$ as required. So the code will be correct.

To argue that the program (or loop) terminates, we use **variants**: functions that map program states to $\mathbb{N}$ (or any well-founded domain). To show that a loop terminates one proves that every iteration of the loop strictly decreases the value of the variant. A suitable variant here would be $n - k$ because "increase k and ..." decreases the value of $n - k$.

An example: Factorial (summary)

We haven't accomplished anything we couldn't do before, but that wasn't really the point.
We have alluded to concepts such as

- induction
- specification
- implementation
- correctness
- variants and invariants

In this course you will learn what they really mean.

Course Structure

Course aims:

- Reinforce concepts from Discrete Mathematics
- Emphasise the connection between Discrete Mathematics and Computer Science
- Use mathematical concepts to **reason about programs**

Course Structure

The course content will be as follows (subject to change):

Week 1: Course introduction/motivation; Recap of relevant Discrete Mathematics content

Week 2: Recursion and induction

Week 3: Propositional Logic

Week 4: Predicate Logic. Assignment 1 due

Week 5: Introduction to program semantics

Week 6: Set-based semantics

Week 7: Operational semantics

Week 8: State machine models. Assignment 2 due

Week 9: Invariants and their proofs

Week 10*: Course recap. Assignment 3 due

*Monday Week 10 is a public holiday and the lecture will be held on Monday in Week 11.

Assessment

Three assignments:

- Assignment 1 (due 17 March): worth 20%
- Assignment 2 (due 7 April): worth 15%
- Assignment 3 (due 28 April): worth 15%

Lateness penalty: 10% (of raw mark) per 12 hour period.

Final exam: worth 50%
You **must** achieve a score of 40% or higher on your final exam in order to pass the course.

Resources

Course website (WebCMS)

Short post by Liam O'Connor

Old course website

- E Lehman, FT Leighton, A Meyer: Mathematics for Computer Science
- C Morgan: Programming from Specifications
- KA Ross and CR Wright: Discrete Mathematics