Introduction

Specification and Verification I

- **This course is on the specification and verification of software**
- **It is a prerequisite for the course entitled:**
  - *Specification and Verification II*
  - **which concerns hardware**
- **The two courses are really a single course**

Program Specification and Verification

- **This course is about *formal* ways of specifying and validating software**
- **This contrasts with *informal* methods:**
  - **natural language specifications**
  - **testing**
- **Formal methods are *not* a panacea**
  - **formally verified designs may still not work**
  - **can give a false sense of security**
- **Assurance versus debugging**
  - **formal verification (FV) can reveal hard-to-find bugs**
  - **can also be used for assurance e.g. "proof of correctness"**
  - **Microsoft use FV for debugging, NSA use FV for assurance**
- **Goals of course:**
  - **enable you to understand and criticise formal methods**
  - **provide a stepping stone to current research**

Testing

- **Testing can quickly find obvious bugs**
  - **only trivial programs can be tested exhaustively**
  - **the cases you do not test can still hide bugs**
  - **coverage tools can help**
- **How do you know what the correct test results should be?**
- **Many industries' standards specify maximum failure rates**
  - **e.g. fewer than 10^{-6} failures per second**
  - **assurance that such rates have been achieved cannot be obtained by testing**

Formal Methods

- *Formal Specification* **- using mathematical notation to give a precise description of what a program should do**

- *Formal Verification* **- using precise rules to mathematically prove that a program satisfies a formal specification**

- *Formal Development (Refinement)* **- developing programs in a way that ensures mathematically they meet their formal specifications**

- **Formal Methods should be used in conjunction with testing,** *not* **as a replacement**

Should we always use formal methods?

- They can be expensive
  - though can be applied in varying degrees of effort
- There is a trade-off between expense and the need for correctness
- It may be better to have something that works most of the time than nothing at all
- For some applications, correctness is especially important
  - nuclear reactor controllers
  - car braking systems
  - fly-by-wire aircraft
  - software controlled medical equipment
  - voting machines
  - cryptographic code
- Formal proof of correctness provides a way of establishing the absence of bugs when exhaustive testing is impossible

Floyd-Hoare Logic

- **This course is concerned with Floyd-Hoare Logic**
  - **also known just as Hoare Logic**
- **Floyd-Hoare Logic is a method of reasoning mathematically about** *imperative* **programs**
- **It is the basis of mechanized program verification systems**
  - **the architecture of these will be described later**
- **Industrial program development methods like SPARK use ideas from Floyd-Hoare Logic to obtain high assurance**
- **Developments to the logic still under active development**
  - **e.g. separation logic (reasoning about pointers) – google if curious**

Syllabus

- **Program specification:**
  - **partial and total correctness**
- **Hoare notation**
- **Axioms and rules of Floyd-Hoare Logic**
  - **discussion of soundness and completeness**
- **Mechanised program verification**
  - **verification conditions**
- **Program refinement**
- **Higher order logic**
- **Semantic embedding in higher order logic**
- **Related notions**
  - **weakest preconditions and strongest postconditions**
  - **VDM**
  - **dynamic logic**
  - **. . .**

A Little Programming Language

Expressions:

$E ::= \;\; N \;\mid\; V \;\mid\; E_1 + E_2 \;\mid\; E_1 - E_2 \;\mid\; E_1 \times\; E_2 \;\mid\; \ldots$

Boolean expressions:

$B ::= \;\; \mathtt{T} \;\mid\; \mathtt{F} \;\mid\; E_1 {=} E_2 \;\mid\; E_1 \leq E_2 \;\mid\; \ldots$

Commands:

$C ::=$ `SKIP`
| V `:=` E
| V`(`E_1`)` `:=` E_2
| C_1 `;` C_2
| `IF` B `THEN` C **(redundant)**
| `IF` B `THEN` C_1 `ELSE` C_2
| `BEGIN VAR` V_1 `; .. VAR` V_n `;` C `END`
| `WHILE` B `DO` C
| `FOR` V `:=` E_1 `UNTIL` E_2 `DO` C

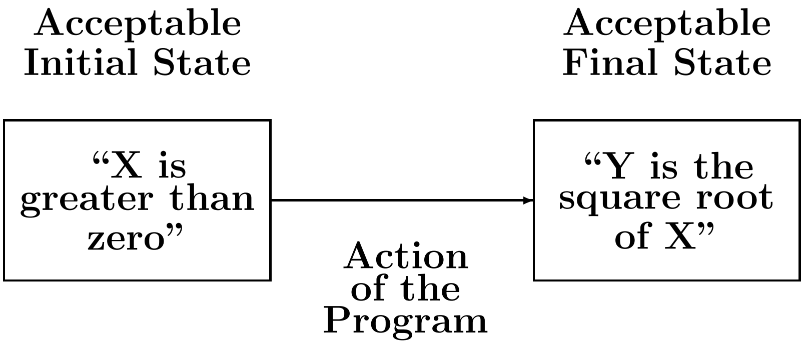
Some Notation

- **Programs are built out of *commands* like assignments, conditionals, while-loops etc**

- **The terms 'program' and 'command' are synonymous**
  - **the former generally used for commands representing complete algorithms**

- **The term 'statement' is used for conditions on program variables that occur in correctness specifications**
  - **potential for confusion as some writers use this word for commands**
  - **e.g. '`FOR`-statement' wrong here, use '`FOR`-command'**

Specification of Imperative Programs



Hoare's notation

- **C.A.R. Hoare introduced the following notation called a** *partial correctness specification* **for specifying what a program does:**

$$\{P\}\ C\ \{Q\}$$

  where:

  - C **is a command**
  - P **and** Q **are conditions on the program variables used in** C

- **Conditions on program variables will be written using standard mathematical notations together with** *logical operators* **like:**

  - $\wedge$ **('and'),** $\vee$ **('or'),** $\neg$ **('not'),** $\Rightarrow$ **('implies')**

- **Hoare's original notation was** $P\ \{C\}\ Q$ **not** $\{P\}\ C\ \{Q\}$**, but the latter form is now more widely used**

Meaning of Hoare's Notation

- $\{P\}\ C\ \{Q\}$ **is true if**
  - whenever C is executed in a state satisfying P
  - and *if* the execution of C terminates
  - then the state in which C terminates satisfies Q

- **Example:** $\{\mathtt{X} = 1\}$ `X:=X+1` $\{\mathtt{X} = 2\}$
  - P is the condition that the value of `X` is `1`
  - Q is the condition that the value of `X` is `2`
  - C is the assignment command `X:=X+1`
    - i.e. '`X` becomes `X+1`'

- $\{\mathtt{X} = 1\}$ `X:=X+1` $\{\mathtt{X} = 2\}$ **is true**

- $\{\mathtt{X} = 1\}$ `X:=X+1` $\{\mathtt{X} = 3\}$ **is false**

Specifications as the Basis of Contracts

- **A formal specification can be used as the basis for requirements analysis and procurement**
  - **is this what the customer intended?**
  - **see link** *Design by Contract* **on course web page**
- **"I want a program that swaps the values in X and Y"**
- $\{\texttt{X=x} \wedge \texttt{Y=y}\}\ C\ \{\texttt{X=y} \wedge \texttt{Y=x}\}$
- **The command**

  `BEGIN R:=X; X:=Y; Y:=R END`

  would fulfil the specification and so the contract
- **The command** `BEGIN X:=Y; Y:=X END` **would not**
- **How do we determine whether a program fulfils the contract?**

Formal versus Informal Proof

- **Mathematics text books give** *informal proofs*
- **English arguments are used**
  - **proof of** $(\mathtt{X}+\mathtt{1})^2 = \mathtt{X}^2 + 2 \times \mathtt{X} + \mathtt{1}$

    "follows by the definition of squaring and distributivity laws"
- **Formal verification uses** *formal proof*
  - **the rules used are described and followed very precisely**
  - **formal proof has been used to discover errors in published informal ones**
- **Here is an example formal proof**

1.	$(\mathtt{X}+\mathtt{1})^2$	$= (\mathtt{X}+\mathtt{1}) \times (\mathtt{X}+\mathtt{1})$	Definition of $()^2$.
2.	$(\mathtt{X}+\mathtt{1}) \times (\mathtt{X}+\mathtt{1})$	$= (\mathtt{X}+\mathtt{1}) \times \mathtt{X} + (\mathtt{X}+\mathtt{1}) \times \mathtt{1}$	Left distributive law of $\times$ over $+$.
3.	$(\mathtt{X}+\mathtt{1})^2$	$= (\mathtt{X}+\mathtt{1}) \times \mathtt{X} + (\mathtt{X}+\mathtt{1}) \times \mathtt{1}$	Substituting line 2 into line 1.
4.	$(\mathtt{X}+\mathtt{1}) \times \mathtt{1}$	$= \mathtt{X}+\mathtt{1}$	Identity law for $\mathtt{1}$.
5.	$(\mathtt{X}+\mathtt{1}) \times \mathtt{X}$	$= \mathtt{X} \times \mathtt{X} + \mathtt{1} \times \mathtt{X}$	Right distributive law of $\times$ over $+$.
6.	$(\mathtt{X}+\mathtt{1})^2$	$= \mathtt{X} \times \mathtt{X} + \mathtt{1} \times \mathtt{X} + \mathtt{X} + \mathtt{1}$	Substituting lines 4 and 5 into line 3.
7.	$\mathtt{1} \times \mathtt{X}$	$= \mathtt{X}$	Identity law for $\mathtt{1}$.
8.	$(\mathtt{X}+\mathtt{1})^2$	$= \mathtt{X} \times \mathtt{X} + \mathtt{X} + \mathtt{X} + \mathtt{1}$	Substituting line 7 into line 6.
9.	$\mathtt{X} \times \mathtt{X}$	$= \mathtt{X}^2$	Definition of $()^2$.
10.	$\mathtt{X} + \mathtt{X}$	$= 2 \times \mathtt{X}$	2=1+1, distributive law.
11.	$(\mathtt{X}+\mathtt{1})^2$	$= \mathtt{X}^2 + 2 \times \mathtt{X} + \mathtt{1}$	Substituting lines 9 and 10 into line 8.

The Structure of Proofs

- **A proof consists of a sequence of lines**

- **Each line is an instance of an** *axiom*
  - **like the definition of** $()^2$

- **or follows from previous lines by a** *rule of inference*
  - **like the substitution of equals for equals**

- **The statement occurring on the last line of a proof is the statement** *proved* **by it**
  - **thus** $(\mathtt{X}+\mathtt{1})^2 = \mathtt{X}^2 + \mathtt{2} \times \mathtt{X} + \mathtt{1}$ **is proved by the proof on the previous slide**

- **These are 'Hilbert style' formal proofs**
  - **can use a tree structure rather than a linear one**
  - **choice is a matter of convenience**

Formal proof is syntactic ‘symbol pushing’

- Formal Systems reduce verification and proof to symbol pushing
- The rules say...
  - if you have a string of characters of this form
  - you can obtain a new string of characters of this other form
- Even if you don’t know what the strings are intended to mean, provided the rules are designed properly and you apply them correctly, you will get correct results
  - though not necessarily the desired result
- Thus computers can do formal verification
- Formal verification by hand generally not feasible
  - maybe hand verify high-level design, but not code
- Famous paper that’s worth reading:
  - “Social processes and the proofs of theorems and programs”. R. A. DeMillo, R. J. Lipton, and A. J. Perlis. CACM, May 1979
- Also see the book “Mechanizing Proof” by Donald MacKenzie

Hoare's Verification Grand Challenge

- **Bill Gates, keynote address at WinHec 2002**

  ```
  ``... software verification ... has been the Holy Grail of computer
  science for many decades but now in some very key areas, for example,
  driver verification we are building tools that can do actual proof
  about the software and how it works in order to guarantee the reliability.''
  ```

- **Hoare has posed a challenge**

  ```
  The verification challenge is to achieve a significant body of
  verified programs that have precise external specifications,
  complete internal specifications, machine-checked proofs of
  correctness with respect to a sound theory of programming.

  The Deliverables

    A comprehensive theory of programming that covers the
    features needed to build practical and reliable programs.

    A coherent toolset that automates the theory and scales up
    to the analysis of large codes.

    A collection of verified programs that replace existing
    unverified ones, and continue to evolve in a verified state.
  ```

- **"You can't say anymore it can't be done! Here, we have done it."**

Exercise: debug the proof of "1=2" below

1.	$1/1 = 1/1$	Reflexivity of equality
2.	$-(1/1) = -(1/1)$	Negate both sides
3.	$-1/1 = 1/-1$	Use $-(n/m) = (-n)/m = n/(-m)$
4.	$\sqrt{-1/1} = \sqrt{1/-1}$	Square root of both sides
5.	$\sqrt{-1}/\sqrt{1} = \sqrt{1}/\sqrt{-1}$	Use $\sqrt{m/n} = \sqrt{m}/\sqrt{n}$ (since $(m/n)^2 = m^2/n^2$)
6.	$i/1 = 1/i$	As $i = \sqrt{-1}$ and $1 = \sqrt{1}$
7.	$i/2 = 1/2i$	Halving both sides
8.	$i/2 + 3/2i = 1/2i + 3/2i$	Adding same thing to both sides
9.	$i^2/2 + 3i/2i = i/2i + 3i/2i$	Multiplying both sides by i
10.	$-1/2 + 3/2 = 1/2 + 3/2$	Using $i^2 = -1$ and cancelling
11.	$1 = 2$	Adding the fractions