CITS5501 Software Testing and Quality Assurance
Formal methods – introduction

Unit coordinator: Arran Stewart

Overview

- What are formal methods?
- Why use them?
- How does formal verification work?
- What sorts of formal methods exist?

Sources

Some useful sources, for more information:

- Pressman, R., *Software Engineering: A Practitioner's Approach*, McGraw-Hill, 2005
- Huth and Ryan, *Logic in Computer Science*
- Pierce et al, *Software Foundations vol 1*

Overview

- When doing software engineering – specifying and developing software systems – the activities done can be done with varying levels of mathematical rigor.

Example

For instance, we could write a requirement

- informally, just using natural language, and perhaps tables and diagrams.
  - easy, but can be imprecise and ambiguous (and hard to spot when that has occurred)
- semi-formally, perhaps using occasional mathematical formulas or bits of pseudocode to express what's required
- mostly using mathematical notation, with a bit of English to clarify what the notation represents.
  - much more work, and harder to ensure the notation matches our intuitive idea of what the system should do
  - little or no vagueness or ambiguity

(Lightweight) formal methods

- Things towards the “more formal” side of this spectrum will tend to get called “lightweight formal methods” or “formal methods”.
- We’ll start with an example formal method (verification of programs using Hoare logic), then come back to the definition, and look at other sorts of formal methods.

Rationale

- Why use formal methods?
- Building reliable software is hard.
  - Software systems can be hugely complex, and knowing exactly what a system is doing at any point of time is likewise hard.
- So computer scientists and software engineers have come up with all sorts of techniques for improving reliability (many of which we've seen) – testing, risk management, quality controls, maths-based techniques for reasoning about the properties of software
  - And this last sort of technique is what we call formal methods.

Rationale

- By *reasoning* about the properties of software – i.e., proving things about it – we can get much greater certainty that our programs are reliable and error-free, than we can through testing
- Testing is a sort of *empirical investigation* – we go out and check whether we can find something (bugs, in this case)
- But if we don't find it, that doesn't mean that whatever we were looking for doesn't exist – we may not have looked hard enough or in the right places.
    - (People once thought it was an eternal and obvious truth that there weren't such things as black swans, but it turned out they weren't looking in the right places.)

Example

- Consider a bit of code, in some Python-like language, for multiplying `i` by `j`
  - (We will look at more complex examples later)

```
n := 0
while j > 0:
  n := n + i
  j := j - 1
return n
```

How can we show that this code is correct? (Assuming it is.)

Example (2)

- We could try it on a number of inputs. (We might try using 0, values near 0, and values not especially near zero.)
- We could inspect the code and see if it conforms to our idea of multiplication
- We could try to *prove* that, once the code has finished executing, for *any* integers `i` and `j`, `n` will equal `i * j`

Program verification

- *Proofs of correctness* use techniques from formal logic to prove that if the starting state (i.e., "input" variables) of a program satisfies particular properties, than the end state after executing a program (i.e., "output" variables) satisfies some other properties.
- The first lot of properties are called *preconditions* (assertions that hold prior to execution of a piece of code), and the second lot are *postconditions* (assertions that hold after execution)

Program verification (2)

For instance, if our program P is the snippet of code from before –

```
n := 0
while j > 0:
  n := n + i
  j := j - 1
return n
```

– then our input variables are `i` and `j`, our output variable is `n`, and our precondition might be

- `i` and `j` are any integers

and our postcondition might be

- `n` equals (the original value of) `i` times (the original value of) `j`

Assertions

Assertions are just statements we can make about variables in the program.

Examples:

- Bounds on elements of the data:

  $n \geq 0$

- Ordering properties of the data:

  for all $j : 0 \leq j < n - 1 : a_j \leq a_{j+1}$

- "Finding the maximum"

  e.g. Asserting that p is the position of the maximum element in some array $a[0..n-1]$

  $0 \leq p < n \vee (\text{for all}\, j : 0 \leq j < n : a_j \leq a_p)$

Assertions in a program - multiplication

We'll distinguish our input variables from our working variables, by giving the, different names. We'll make a and b input.

```
{ pre : ⊤ } // no precons -- "top" or "true"
i := a
j := b
n := 0
while j > 0:
  n = n + i
  j = j - 1
{ post: n = a * b }
```

Assertions in a program - old values

- The problem here is that we want to refer to the *old* values of `i` and `j` in our postcondition
- Systems and languages for program verification often will have a special syntax for this, to avoid having to introduce new variables
- e.g. In the Eiffel language, you can refer to the value of `i` in the pre-state as just `old i`
- In languages or notations that don't have such syntax, we may have to manually do the work ourselves.
  - Often, we'll use prime marks (`'`) to indicate a subsequent state of a variable
  - e.g. i' often means "the *next* value of i" (e.g. after a statement has executed, or a loop has executed, or we have transitioned to a new state)

Terminology

Hoare triples:

- Where we have a sequence [*preconditions*, *code fragment*, *postconditions*], we call this a **Hoare triple** (after logician and computer scientist Tony Hoare of Oxford, who also invented the Quicksort algorithm, amongst other things)

Loop invariants

- Where we have an assertion which should hold before and after *every* iteration of a loop, we call this a *loop invariant*

Proving things about loops

- It often turns out to be much easier to do a proof of correctness involving a loop if we tackle it in two steps:
    1. Prove that *if* the program terminates, *then* it produces the results we want
    2. Prove that the program terminates
- (Step 1 gives us something the formal methods people call **partial correctness**, and steps 1 and 2 together gives us **emphtotal correctness**.)

Proving things

- So how do we actually build up a proof?
  - We need rules about how to combine triples together.
  - We cover these rules now

Composition

- The composition rule says:

  If we have $\{\ a\ \}\ P_1\ \{\ b\ \}$ and $\{\ b\ \}\ P_2\ \{\ c\ \}$

  then we can derive $\{\ a\ \}\ P_1; P_2\ \{\ c\ \}$

Assignment

- The assignment rule states that, after an assignment, any predicate that was previously true for the right-hand side of the assignment now holds for the variable on the left-hand side.
- Formally:

  $$A[E/x] \;\; \mathrm{x} := \mathrm{E} \; \{ \; A \; \}$$

  Here, A represents some (probably and'ed together) assertion about variables;
  $A[E/x]$ means, "A, but wherever x appears, substitute for it the expression E instead"
- Which gives us the meaning we want: whatever *was* true of E is now true of x.

Conditional

- We represent “if” using the conditional rule
- The rule is:

  If we have

  $\{\ a \wedge C\ \}\ P_1\ \{\ b\ \}$, and we have $\{\ a \wedge \neg C\ \}\ P_2\ \{\ b\ \}$

  then we can derive

  $\{\ a\ \}$ if C then P_1 else $P2$ $\{\ b\ \}$

- (i.e., If we know that we end up with b holding regardless of whether C is true or not, then we know it must be true after the `if` statement)

"Partial while"

- The rule for partial correctness of while loops is:

  If we have a triple $\{\ A \land B\ \}\ P\ \{\ A\ \}$

  then we can derive:

  $\{\ A\ \}$ while B do P done $\{\ \neg B \land A\ \}$

- Here, A is what is called the **loop invariant** – it is something that should be *preserved by* (i.e. is true before and true after) the loop body.
- After the loop is done, A should still hold, no matter how many times the loop executed.
- *if* we got out of the loop, it must be the case afterwards that $\neg B$ (since otherwise, "while B" would've continued)

"Total while"

- If we can prove partial correctness for a while loop;
  and we can prove the loop terminates;
  then we've proved total correctness.
  - (There is syntax for this in Hoare logic, but we won't go into it.)

Worked example

- program fragment:

```
{ pre : b ≥ 0 }
i := a
j := b
n := 0
while j > 0:
  n = n + i
  j = j - 1
{ post: n = a * b }
```

- ... we try to work out a loop invariant ...
- invariant: $(j * i) + n = a * b$
  i.e., if we add our "result so far" (n), which is increasing, to the product of j and i (where j is decreasing), that should remain constant, and is equal to $a * b$.

Worked example

- let our loop invariant $\boldsymbol{L}$ be $(j * i) + n = a * b$

```
{ pre : b ≥ 0 }
i := a
j := b
n := 0
{ j > 0 ∧ L }
while j > 0:
  n = n + i
  j = j - 1
{ j ≤ 0 ∧ L }
{ post: n = a * b }
```

- The "partial while" rule lets us put assertions round the while loop.
- So one thing we know now is that *if* the loop finishes, then $\boldsymbol{L}$ will still hold.
- If j ended up being 0, then that would be handy, because
  $(j = 0) \wedge (j * i + n = a * b)$
  implies
  $n = a * b$
  which is what we would like to prove.

Worked example

- So if we can prove that
  - at the end of the loop, $j = 0$,
  - and that the loop terminates,
  - and that the initialization statements and the program preconditions satisfy the loop precondition

  ... then we've proved that the program does what we want. (Assuming $b \geq 0$, which is a precondition for the whole program.)

Worked example

- We won't work through the rest of the proof in detail, but hopefully you have an idea of how it will go.
- Roughly, to prove that the loop ends, and that j is 0 at the end of it:
  Since j is some finite number, and we are subtracting one off it each time, some math and logic tells us that we must end sometime, and that j will end up equaling zero.

Worked example

- Coming up with the loop invariant is usually the hard part; the other rules can be applied in a more automatic kind of way.
- Edsger Dijkstra said that to properly understand a while statement is to understand its invariant.

Sorts of formal methods

A typical approach

Often, we'll apply formal methods in the following way:

- We'll have something representing the system – this is called a *model*
  - This could be actual code, or it could be annotated code, or it could be some more abstract model of the system (like state machines, which we have seen earlier)
- We'll have some specification – some property that we want our system to have
  - e.g., that it calculates the factorial of a natural number; or never gets deadlocked; or has certain security properties.
- And we will try to show that the model meets the specification.

Back to formal methods

- So with our fragment of Python code, our *specifications* were assertions about variable values before and after the program executed, written as mathematical formulas.
- We used a method that was largely *manual* – putting assertions around fragments of code.
- Some bits of that could be partly automated – the rules for composition and assignment could be done by machine
- The loop invariant, however, requires ingenuity to come up with
- Our *model* of the system was, in fact, the code itself.
  - (The code is still just a *model*, a simplification, of the actual running binary. It isn't itself the binary.
    We also might ignore such things as limits on sizes of ints, if we are happy to accept that our proof only applies, if the ints are sufficiently small.)

Categorizing formal methods

- We can categorize formal methods in various ways . . .

Categorizing formal methods

Degree of formality:

- how formal are the specifications and the system description?
- in natural language (informal), or something more mathematical?

Categorizing formal methods

Degree of automation:

- the extremes are fully automatic and fully manual
- most computer-aided methods are somewhere in the middle

Categorizing formal methods

Full or partial verification of properties in the specification

- What is being verified about the system? Just one property? (e.g., that it does not deadlock, say – common for concurrent systems)
- Or many/all properties?
  - (This is usually very expensive, in terms of effort)

Categorizing formal methods

Intended domain of application:

- e.g. hardware vs software;
- reactive vs terminating;
    - reactive systems run a theoretically endless "loop" and aren't intended to terminate – they just keep *reacting* to an environment
    - e.g. operating systems, embedded hardware (modelled with state machines, often)
    - terminating systems terminate, usually with some sort of *result*
- sequential vs concurrent

Categorizing formal methods

pre- vs post-development:

- Is verification done early in development, vs later or afterwards?
- Earlier is obviously better, since things are much more expensive to fix if early, if it turns out our system *doesn't* meet the specs

Categorizing formal methods

- But sometimes the system comes first, then the verification
- Often true for programming languages . . .
  - e.g. Java was released in 1995, and in 1997, a machine-checked proof of "type soundness" of a subset of Java was proved.[1]
  - But: later versions of Java (from 5 onwards) turned out to have *unsound* type systems in various ways. Oops.
  - The interaction of sub-typing and inheritance turned out to make the early OO language Eiffel unsound. Also oops.[2]

[1] Syme. "Proving Java Type Soundness". 1997 [pdf]

[2] William R. Cook. A proposal for making Eiffel type-safe. The Computer Journal, 32(4):305–311, August 1989.

Categorizing formal methods

Are we trying to prove properties of an individual program? Or about *all* programs written in a particular language?

- An example of the first one is proving that a sorting function does what we want it to, or that a compiler implementation obeys some particular formal specification
- An example of the latter is proving results about the *type system* for a language, which lets us show that *all* programs in the language will have some sort of guarantees of good behaviour
  - e.g. Proving that well-typed Java programs cannot be subverted (assuming the JVM and compiler are implemented correctly) – it should be impossible to get a reference which doesn't point to a valid area of memory, for instance.

Aside – type systems

- We often don't think of type systems as being a "formal method", but some type systems are very expressive, and allow us to prove quite strong results about our programs
- We can use them to prove that (for instance) unsanitized user data never gets output to a web page

Type systems

- A type system many of us will have used in high school: consistency of SI units
- We can multiply and divide things which have different units (e.g. distance divided by time, or acceleration multiplied by time) ...
  ... but it makes no physical sense to *add* things with different units – we can't add seconds to metres – and the rules for consistency of SI units stop us from doing so, thus avoiding silly mistakes.
- In most programming languages: floating point numbers are used for all physical quantities – nothing to stop you adding a number representing seconds to one representing distance.
- Some languages (e.g. Fortress, F#) have dimensionality and unit checking built into the language –
  useful if coding something with a lot of physical quantities and want checks you haven't performed a physically nonsensical calculation.

Categorizing formal methods

Model-based vs proof-based approaches:

- We've seen one example of a *proof* based approach, Hoare logic.
  - Your specification is some formula in some suitable logic
  - In Hoare logic, our specification is what we want the program to *do* – it's expressed as assertions (postconditions which should hold after the program executes, if the preconditions held)
  - You try and *prove* that the system (or some abstraction of it) satisfies the specification.
- Usually requires guidance and expertise from the user

Categorizing formal methods

Model-based approaches:

- Again, our specification is some sort of formula
- This time, our system description is some mathematical structure, a **model**, $\mathcal{M}$
- We check whether the model $\mathcal{M}$ **satisfies** the specification (i.e. has the properties we want)
- In many cases, this can be done automatically.