CITS5501 Software Testing and Quality Assurance
Formal methods

Unit coordinator: Arran Stewart

Formal methods

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.
- For instance, we could write a requirement
  - informally, just using natural language, and perhaps tables and diagrams. This is 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. This is typically a lot more work, and it can be harder to ensure the notation matches our intuitive idea of whhat the system should do, but has little or no vagueness or ambiguity.

Overview (2)

- Things towards the "more formal" side of this spectrum will tend to get called "lightweight formal methods" or "formal methods".

Definitions

> Formal methods used in developing computer systems are mathematically based techniques for describing system properties. Such formal methods provide frameworks within which people can specify, develop, and verify systems in a systematic, rather than ad hoc manner.
>
> — Encyclopedia of Software Engineering [Mar01]

Problems with conventional specs:

- contradictions
- ambiguities
- vagueness
- incompleteness
- mixed levels of abstraction

A typical approach

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

- 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'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)
- And we will try to show that the model meets the specification.

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.)

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)

Example

By way of example, we'll use fragments of code from the Dafny programming language.

It is somewhat similar in style to Java or C#, but includes built-in features for program verification.

To write a method `Abs()` which calculates the absolute value of an integer, we woud write code something like this:

```
method Abs(x: int) returns (y: int) {
   if x < 0
      { return -x; }
   else
      { return x; }
}
```

Dafny code

One difference from Java is that the return value is given its own name, “y”.

```
method Abs(x: int) returns (y: int) {
   if x < 0
      { return -x; }
   else
      { return x; }
}
```

Dafny postconditions

Why is this? It's because we can add *postconditions* to Dafny code, which refer to the return value (or to input parameters, as well), so it's convenient to give it a name.

```
method Abs(x: int) returns (y: int)
   ensures 0 <= y
{
   ...
}
```

- Multiple "ensures" specifications can be added
- "ensures" specifications can make use of the ususal logical connectives (e.g. "&&", "||")
- The suggested style is for distinct "properties" to be given their own "ensures" specification

Dafny preconditions

Preconditions can be specified with keyword "`requires`"

```
method AddOne(x: int) returns (y: int)
   requires x > 0
   ensures  y > 0
{
   return x + 1;
}
```

Dafny verification

- Dafny will actually *reject* programs with postconditions it can't prove are correct.
- i.e., It attempts to prove that, if the preconditions are correct, then the postconditions will be also, and if it can't do that, reports a verification error
- A method with no "ensures" specifications has no preconditions, so will always verify.

Dafny verification

- A programmer *calling* a method must ensure the preconditions are met
  (else Dafny reports an error)
- A programmer *writing* a method may ensure the preconditions are already true, but must ensure the postconditions are met
  (else Dafny reports an error)

Dafny live coding on the web

You can experiment with the Dafny language on the web – https://rise4fun.com/Dafny/tutorial

Dafny assertions

In addition to preconditions and postconditions, Dafny lets you write *assertions* – these are found somewhere in the body of a method.

They assert that something is true at that point in the code (and if Dafny can't prove it is so, it will report an error).

Dafny assertions

```
method MyMethod()
{
  assert 2 < 3;
}
```

Assertions don't *have* to mention any of the variables or return values of a method (though obviously they are going to be more useful if they do).

Dafny assertions

You can think of assertions as a way of "asking" the Dafny verifier what it knows to be true at any point in the program.

```
method Abs(x: int) returns (y: int)
   ensures 0 <= y
{
   if x < 0
      { return -x; }
   else
      { return x; }
}
method MyMethod()
{
  var v := Abs(-3);
  assert v >= 0;
}
```

Dafny verification errors

- There are two main reasons you might get a verification error:
  - Firstly, there might be something actually incorrect with your code.
  - Secondly, it might be correct, but the Dafny verifier isn't "clever" enough to prove that the required properties hold.
- In the latter there are two main causes for Dafny verification errors: specifications that are inconsistent with the code, and situations where it is not "clever" enough to prove the required properties.

Proving loops correct

Loops pose a problem for Dafny.

To prove that the postconditions are true (assuming the preconditions are), it needs to consider *all* the possible paths through a method.

But for a loop, the verifier doesn't know in advance how many times the loop will be executed. There are potentially infinite paths through the program.

Loop invariants

The solution is to make use of *loop invariants*.

These are expressions that hold true

- upon entering the loop
- after every execution of the loop body

Loop invariant example

Loop invariants are put just before the body of a loop:

```
var i := 0;
while i < n
   invariant 0 <= i
{
   i := i + 1;
}
```

Loop invariant example

```
var i := 0;
while i < n
   invariant 0 <= i
{
   i := i + 1;
}
```

The verifier reasons as follows:

- Is `0 <= i` true before the loop starts?
  - Yes, since `i` is 0, and `0 <= 0` is true.
- *If* the invariant was true at the start of the loop, will it also be true at the end of the loop?
  - Yes, it will.
    If `0 <= i` at the start of the loop, all we do in the body is increment `i` by 1; so `0 <= i` will *still* be true at the end of the loop.
- From this, Dafny concludes that if the invariant was true *before* entering the loop, it will also be true *after* the loop (since there's no place it could have been made false)

Loop invariant applications

The example above is very simple, but we can work our way up to more complex loops.

For instance, here is a loop that calculates $m \times n$ (though in any modern programming language, we already have integer multiplication):

```
// assume m and n are parameters, say
var tot := 0;
while m > 0
{
  tot := tot + n;
  m := m - 1;
}
```

Could we *prove* that, after the loop ends, $tot = m \times n$?

Loop invariant applications

It makes things easier if, rather than altering `m` and `n`, we leave them as is and copy their values into other variables. Let's write this as a method in Dafny.

(In fact, Dafny will not *let* us mutate parameters.)

```
method MyMethod(m : int, n : int) {
var tot := 0;
var a := m; var b := n;
while a > 0
  {
    tot := tot + b;
    a := a - 1;
  }
}
```

Loop invariant applications

Now we can write a postcondition in terms of m and n:

```
method MyMethod(m : int, n : int) returns (r: int)
  ensures r == m * n
{
  var tot := 0;
  var a := m; var b := n;
  while a > 0
    {
      tot := tot + b;
      a := a - 1;
    }
  return tot;
}
```

This will fail, as Dafny cannot prove it is true.

Loop invariant applications

One thing that is always true about the loop:

- `tot` is the "total so far"
- If we add the bits "still to go" (`a * b`) to the total, we should get `m * n`.

So an invariant is `a * b + tot == m * n`.

```
method MyMethod(m : int, n : int) returns (r: int)
  ensures r == m * n
{
  var tot := 0;
  var a := m; var b := n;
  while a > 0
    invariant a * b + tot == m * n
    {
      tot := tot + b;
      a := a - 1;
```

Loop invariant applications

Because we have forgotten to deal with the possibility that `m` might be negative.
If it were, we'd end up with an endless loop.

So let's make sure `m` and `n` are non-negative.

```
method MyMethod(m : int, n : int) returns (r: int)
  requires m >= 0 && n >= 0
  ensures r == m * n
{
  var tot := 0;
  var a := m; var b := n;
  while a > 0
    invariant a * b + tot == m * n
    {
      tot := tot + b;
      a := a - 1;
    }
  assert tot == m * n;
  return tot;
}
```

Loop invariant applications

```
method MyMethod(m : int, n : int) returns (r: int)
  requires m >= 0 && n >= 0
  ensures r == m * n
{
  var tot := 0;
  var a := m; var b := n;
  while a > 0
    invariant a * b + tot == m * n
    {
      tot := tot + b;
      a := a - 1;
    }
  assert tot == m * n;
  return tot;
}
```

Dafny will confirm that this method is correct – it understands enough basic arithmetic to work out that the loop invariant holds before and after each loop iteration.

Loop invariant applications

```
  // ...
  while a > 0
    invariant a * b + tot == m * n
    {
      tot := tot + b;
      a := a - 1;
    }
  assert tot == m * n;
}
```

And if the loop invariant holds in those cases, it also holds after; and since `a == 0` after the loop,

$$
\begin{aligned}
& \texttt{a * b + tot == m * n} \\
\rightarrow\ & \texttt{0 * b + tot == m * n} \\
\rightarrow\ & \texttt{0 + tot == m * n} \\
\rightarrow\ & \texttt{tot == m * n}
\end{aligned}
$$

Power of specifications

We will not examine the Dafny language in detail, but hopefully you can see that this technique is quite powerful.

If we can prove that small portions of code are correct (i.e., meet their specification), and we can chain them together, then we will be able to prove correctness of large programs.

Example assertions

We can use postconditions, preconditions, assertions and invariants to express:

- Bounds on elements of the data:

  $$n \geq 0$$

- Ordering properties of the data:

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

- "Finding the maximum"

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

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

Theory

- 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)

How verification works

It's often handy to tackle a proof of correctness in two stages:

1. Prove that *if* the program terminates, *then* it produces the results we want
2. Prove that the program terminates

Step 1 gives us what's called "partial correctness"; and if we can prove step 2, we have what's called *total* correctness.

Hoare logic

Hoare logic has small rules that say things like "if we have one Hoare triple we know is correct, with precons *a* and postcons *b*, we can combine it with another with precons *b*".

Composition rule:

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

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

By putting these together (which is what the Dafny verifier does), we can prove that larger and larger fragments of code are correct.

Worked example

- Coming up with the loop invariant is usually the hard part!
- Other rules can be applied in a more automatic kind of way, which is why once we'd supplied a loop invariant, Dafny could prove our "multiplication" example was correct.
- Edsger Dijkstra said that to properly understand a while statement is to understand its invariant.

Sorts of formal methods

Back to formal methods

- So here, our *specifications* were assertions about variable values before and after the program executed, written as mathematical formulas.
- We used a method that was partly *manual* – putting assertions around fragments of code – and partly *automated* (the Dafny verifier could prove many properties of code for us)
- 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.