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

- Formal methods are maths-based techniques for describing system properties
- When doing software engineering – specifying and developing software systems – the activities done can be done with varying levels of mathematical rigor.
- Things towards the "more formal" side of this spectrum will tend to get called "lightweight formal methods" or "formal methods".
- Once a technique is very widely accepted and used, people tend to stop thinking of it as a "formal method", and just call it "programming" or "specification".

Overview

- 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

Overview

- By *reasoning* about the properties of software – i.e., proving things about it – we can get *certainty* that our programs are reliable and error-free
- **Testing** is *empirical* – we go out and check whether we can find something (bugs, in this case)
  - But if we don't find a bug, that doesn't mean that no bugs exist – we may not have looked hard enough or in the right places.
- **Formal methods** are based on mathematical *deduction*.

Example

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

Example

If we wanted to specify

- exact commands and parameters accepted by a program, or
- the format of an HTTP request

we could do so in natural language. But this is very verbose, and often imprecise.

Or we could use a specification language we've already seen – BNF (or EBNF: extended BNF).

Example – HTTP

A version of EBNF is, in fact, what is used to define the format of HTTP requests, in RFC2616.

Example (source: RFC 2616)

```
DIGIT          = "0".."9"

HEX            = "A" | "B" | "C" | "D" | "E" | "F"
               | "a" | "b" | "c" | "d" | "e" | "f" | DIGIT

HTTP-Version   = "HTTP" "/" 1*DIGIT "." 1*DIGIT
```

Example

The advantages of BNF (over natural language) are that it is

- concise – much shorter than an equivalent natural language description would be
- precise and unambiguous – states exactly what is and isn't in the language being described
- capable of being processed and used programmatically – a computer can take your BNF and use it to create a parser or generator

Why use formal methods – problems in specifying systems

System specifications can suffer from a few potential problems.

- *Contradictions*. In a very large set of specifications, it can be difficult to tell whether there are requirements that contradict each other.
    - Can arise where e.g. specifications are obtained from multiple users/stakeholders
    - Example: one requirement says "all temperatures" in a chemical reactor must be monitored, another (obtained from another member of staff) says only temperatures in a specific range.

Problems in specifying systems

- *Ambiguities*. i.e., statements which can be interpreted in multiple different ways.
  "The operator identity consists of the operator name and password; the password consists of six digits. It should be displayed on the security screen and deposited in the login file when an operator logs into the system."

Problems in specifying systems

- *Ambiguities*. i.e., statements which can be interpreted in multiple different ways.
  "The operator identity consists of the operator name and password; the password consists of six digits. It should be displayed on the security screen and deposited in the login file when an operator logs into the system."
- ... Does "it" refer to the identity, or the password?

Problems in specifying systems

- *Ambiguities*. i.e., statements which can be interpreted in multiple different ways.
  "The operator identity consists of the operator name and password; the password consists of six digits. It should be displayed on the security screen and deposited in the login file when an operator logs into the system."
- . . . Does "it" refer to the identity, or the password?
- "Should" can be ambiguous – does "The system should do X" mean the system *must* do X, or that it is optional but desirable that the system do X?

Problems in specifying systems

- *Ambiguities*. i.e., statements which can be interpreted in multiple different ways.
  "The operator identity consists of the operator name and password; the password consists of six digits. It should be displayed on the security screen and deposited in the login file when an operator logs into the system."
- . . . Does "it" refer to the identity, or the password?
- "Should" can be ambiguous – does "The system should do X" mean the system *must* do X, or that it is optional but desirable that the system do X?
- Many terms have both technical and non-technical meanings (possibly multiple of each): for instance, "reliable", "robust", "composable", "category", "failure", "orthogonal", "back end", "kernel", "platform", "entropy" . . .

Problems in specifying systems

- *Vagueness*. Vagueness occurs when it's unclear what a concept covers, or which things belong to a category and which don't.

Problems in specifying systems

- *Vagueness*. Vagueness occurs when it's unclear what a concept covers, or which things belong to a category and which don't.
- "is tall" is vague: some people are definitely tall, and some are definitely short, but it can be difficult to tell when exactly someone meets the criterion of being tall.

Problems in specifying systems

- *Vagueness*. Vagueness occurs when it's unclear what a concept covers, or which things belong to a category and which don't.
- "is tall" is vague: some people are definitely tall, and some are definitely short, but it can be difficult to tell when exactly someone meets the criterion of being tall.
- Likewise "fast", "performant", "efficiently", "scalable", "flexible", "is user-friendly", "should be secure", "straightforward to understand" are all vague.

Problems in specifying systems

- *Incompleteness*. This covers specifications that, for instance, fail to specify what should happen in some case.

Problems in specifying systems

- *Incompleteness*. This covers specifications that, for instance, fail to specify what should happen in some case.
- e.g. An obviously incomplete requirement: "A user may specify normal or emergency mode when requesting a system shutdown. In normal mode, pending operations shall be logged and the system shut down within 2 minutes."

Problems in specifying systems

- *Incompleteness*. This covers specifications that, for instance, fail to specify what should happen in some case.
- e.g. An obviously incomplete requirement: "A user may specify normal or emergency mode when requesting a system shutdown. In normal mode, pending operations shall be logged and the system shut down within 2 minutes."
- ... So what happens in emergency mode?

Problems in specifying systems

- *Incompleteness*. This covers specifications that, for instance, fail to specify what should happen in some case.
- e.g. An obviously incomplete requirement: "A user may specify normal or emergency mode when requesting a system shutdown. In normal mode, pending operations shall be logged and the system shut down within 2 minutes."
- . . . So what happens in emergency mode?
- But other cases of incompleteness may be harder to spot.

Problems in specifying systems

- In addition to these, there are many other ways requirements can be written poorly –
  e.g. Overly long and complex sentences, mixed levels of abstraction (mixing high-level, abstract statements with very low-level ones $\rightarrow$ difficult to distinguish high-level architecture from low-level details), undefined jargon terms, specifying implementation rather than requirements (how vs what), over-specifying, don't satisfy business needs, etc.

Problems in specifying systems

- In addition to these, there are many other ways requirements can be written poorly –
  e.g. Overly long and complex sentences, mixed levels of abstraction (mixing high-level, abstract statements with very low-level ones → difficult to distinguish high-level architecture from low-level details), undefined jargon terms, specifying implementation rather than requirements (how vs what), over-specifying, don't satisfy business needs, etc.
- Formal specifications can potentially help avoid ambiguity, vagueness, contradiction and some gaps in completeness.

Problems in specifying systems

- In addition to these, there are many other ways requirements can be written poorly –
  e.g. Overly long and complex sentences, mixed levels of abstraction (mixing high-level, abstract statements with very low-level ones $\rightarrow$ difficult to distinguish high-level architecture from low-level details), undefined jargon terms, specifying implementation rather than requirements (how vs what), over-specifying, don't satisfy business needs, etc.
- Formal specifications can potentially help avoid ambiguity, vagueness, contradiction and some gaps in completeness.
- Other problems, not so much. Just as it's possible to write programs badly in any language, it's also possible to write formal specifications badly.
  There is still a need for *review* of specifications, as with any artifact.

Formal specifications

- Formal specifications can help with ameliorating these problems.
- Sometimes, just the process of attempting to formalize a requirement can reveal problems with it.
- Using a formal model can help reveal *ambiguity* and *vagueness* and allow them to be eliminated
- It may also be possible (depending on the mathematical model used) to detect inconsistencies
- Detecting whether a specification is *complete* is more difficult.
  - Some gaps may be able to be detected
  - But there are nearly always some details that are left undefined, or scenarios that may not have been considered.

Formal specifications

Formal specifications

- Meaning is defined in terms of mathematics
- Many sorts of formal specification languages and tools with different areas of application
- Small and specific specification languages:
  - *State charts* – define states and transitions
  - *BNF* – defines context-free languages.
  - *Regular expressions* – define *regular languages* (a subset of context-free languages)
    [NB: in practice, most programming languages use "extended regular expressions", which can define much more]
  - π-calculus – used to represent concurrent systems

Formal specification languages

Some *general-purpose* specification languages:

- **Z notation**
  - based on set theory and predicate logic
  - developed in the 1970s.
  - Now has an ISO standard, and variations (e.g. object-oriented versions)
- **TLA+**:
  - Stands for "Temporal Logic of Actions"
  - Especially well-suited for writing specifications of concurrent and distributed systems
  - For finite state systems, can check (up to some number of steps) that particular properties hold (e.g. safety, no deadlock)

Formal specification languages

- We'll be using the **Alloy** specification language
- Alloy is both a language for describing structures, and a tool (written in Java) for exploring and checking those structures.
- Influenced by Z notation, and modelling languages such as UML (the Unified Modelling Language).
- Website: http://alloy.mit.edu/ (The Alloy Analyzer tool can be downloaded from here.)

Aspects of a formal method

Any formal method usually includes:

- A domain of application: a topic or class of things to which the method can usefully be applied.

  example: BNF is used to specify grammars (languages or document formats).
- Some system property it can be used to specify or verify

  example: What commands and arguments are accepted by a program.

These properties could be

- functional requirements
- non-functional requirements (complexity, aspects of security)
- protocols
- etc.

Aspects of a formal method

A formal specification method usually includes:

- syntax: Rules for how the specification is written, and what constitutes a well-formed specification.
- semantics: How the specification is interpreted – what it means.
- rules of inference: Techniques for deriving useful information from the specification.

Categorizing formal methods

We can categorize formal methods in various ways . . .

Degree of formality
: how formal are the specifications and the system description?

Degree of automation
: full automatic through to fully manual. (Most computer-aided methods are somewhere in the middle.)

Properties verified
: What is being verified about the system? Just one property (e.g. does not deadlock) or many (usually v expensive)

Categorizing formal methods, cont'd

Intended domain of application e.g. hardware vs software; *reactive* systems (run in an endless loop) vs terminating; sequential vs concurrent

Life-cycle stage Verification done early in development, vs later (Earlier is obviously better – later is more expensive to fix)

Life-cycle stage

- 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.

Example – 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

Example – type systems

A common poor coding practice is "stringly typed" programs – programs representing information as string that could have been represented using types (e.g. enumerations)

- Stringly-typed: encode flight types as "return" or "oneway"
- Better: use an enum: `enum FlightType { RETURN, ONEWAY }`

Example – type systems

A common poor coding practice is "stringly typed" programs – programs representing information as string that could have been represented using types (e.g. enumerations)

- Stringly-typed: encode flight types as "return" or "oneway"
- Better: use an enum: `enum FlightType { RETURN, ONEWAY }`

Programmers who avoid "stringly typing" often still represent quantities as *numbers* when they represent completely incompatible things – e.g. using a plain `double` for both velocity and body mass index.

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.

Type systems

Other languages without a full unit system will still let you encapsulate numbers in some more specific type, that can't be freely added to normal numbers.

e.g. in Haskell

```
newtype Velocity = Velocity Double
  deriving (Read, Show, Num, Eq, Ord)
```