CITS5501 Software Testing and Quality Assurance
Formal methods

Unit coordinator: Arran Stewart

May 8, 2018

Sources

- 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*
- Alloy tutorial at http://alloytools.org

Specification languages

Some 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.

Some 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."

Some 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?

Some 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.

Some 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.

Some 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 "is user-friendly", "should be secure", "straightforward to understand" are all vague.

Some problems in specifying systems

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

Some 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."

Some 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?

Some 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.

Some problems in specifying systems

- *Mixed levels of abstraction*. Mixing very high-level, abstract statements with very low-level ones. This makes it difficult to distinguish the high-level architecture from low-level details.

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 scenarious that may not have been considered.

Formal specification languages

Many *specification languages* exist, which allow aspects of a system to be modellled mathematically.

Some examples:

- **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"
  - A general-purpose specification language
  - 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/

Alloy language

- We'll look at a simple model of a file system (based on the Alloy tutorial at http://alloytools.org/tutorials/)
- To a first approximation, Alloy looks a little like Java:

```
// A file system object in the file system
sig FSObject { parent: lone Dir }

// A directory in the file system
sig Dir extends FSObject { contents: set FSObject }

// A file in the file system
sig File extends FSObject { }
```

Alloy language

```
// A file system object in the file system
sig FSObject { parent: lone Dir }

// A directory in the file system
sig Dir extends FSObject { contents: set FSObject }

// A file in the file system
sig File extends FSObject { }
```

- Comments can be written in multiple ways

Alloy language

```
// A file system object in the file system
sig FSObject { parent: lone Dir }

// A directory in the file system
sig Dir extends FSObject { contents: set FSObject }

// A file in the file system
sig File extends FSObject { }
```

- Comments can be written in multiple ways
  - single-line comments with "//" or "--"

Alloy language

```
// A file system object in the file system
sig FSObject { parent: lone Dir }

// A directory in the file system
sig Dir extends FSObject { contents: set FSObject }

// A file in the file system
sig File extends FSObject { }
```

- Comments can be written in multiple ways
  - single-line comments with "//" or "--"
  - multile comments with "/* ... */"

Alloy language

```
// A file system object in the file system
sig FSObject { parent: lone Dir }

// A directory in the file system
sig Dir extends FSObject { contents: set FSObject }

// A file in the file system
sig File extends FSObject { }
```

- Comments can be written in multiple ways
  - single-line comments with "//" or "--"
  - multile comments with "/* ... */"
- **sig** is used to write a *signature declaration*;
  you can think of a signature as defining an "entity".
  It says what kinds of things there are.

Alloy language

```
// A file system object in the file system
sig FSObject { parent: lone Dir }

// A directory in the file system
sig Dir extends FSObject { contents: set FSObject }

// A file in the file system
sig File extends FSObject { }
```

- Comments can be written in multiple ways
  - single-line comments with "`//`" or "`--`"
  - multile comments with "`/* ... */`"
- **`sig`** is used to write a *signature declaration*; you can think of a signature as defining an "entity". It says what kinds of things there are.
- To be precise, it defines a *set* of things. **`sig FSObject`** defines a set named **`FSObject`**. Which in this case represents the set of all *file system objects* (both files and directories)

Alloy – subtypes

- We use **`extends`** to indicate subtypes (similar to Java).

Alloy – subtypes

- We use **`extends`** to indicate subtypes (similar to Java).
- Here, **`Dir`** and **`File`** are both subtypes of **`FSObject`**:

```
// A file system object in the file system
sig FSObject { parent: lone Dir }

// A directory in the file system
sig Dir extends FSObject { contents: set FSObject }

// A file in the file system
sig File extends FSObject { }
```

Alloy – subtypes

- We use **extends** to indicate subtypes (similar to Java).
- Here, **Dir** and **File** are both subtypes of **FSObject**:

```
// A file system object in the file system
sig FSObject { parent: lone Dir }

// A directory in the file system
sig Dir extends FSObject { contents: set FSObject }

// A file in the file system
sig File extends FSObject { }
```

- When we declare **Dir** or a **File** to be sub-types of **FSObject**, they are considered to be *mutually disjoint* sets

Alloy – subtypes

- We use **extends** to indicate subtypes (similar to Java).
- Here, **Dir** and **File** are both subtypes of **FSObject**:

```
// A file system object in the file system
sig FSObject { parent: lone Dir }

// A directory in the file system
sig Dir extends FSObject { contents: set FSObject }

// A file in the file system
sig File extends FSObject { }
```

- When we declare **Dir** or a **File** to be sub-types of **FSObject**, they are considered to be *mutually disjoint* sets
  - i.e., nothing is both a **Dir** or a **File**

Alloy – facts

- How can we express that any **`FSObject`** is either a **`Dir`** or a **`File`**? (i.e., there are no other sorts of **`FSObject`**)

Alloy – facts

- How can we express that any **FSObject** is either a **Dir** or a **File**? (i.e., there are no other sorts of **FSObject**)
- Alloy also allows us to specify *constraints*. These are introduced with the keyword **fact**.

```
sig FSObject { parent: lone Dir }
sig Dir extends FSObject { contents: set FSObject }
sig File extends FSObject { }

// All file system objects are either files or directories
fact { File + Dir = FSObject }
```

Alloy – facts

- The general syntax for a fact is

  `fact` *name* { *formulas* }

- *formulas* are Boolean expressions, and by putting them in a fact, we're constraining them to be true.

Alloy – abstract signatures

- (An alternative way to say that all FSObjects must be Dirs or Files would be to declare FSObject **`abstract`**)

Alloy – abstract signatures

- (An alternative way to say that all FSObjects must be Dirs or Files would be to declare FSObject **abstract**)
- (This is similar to the use of the **abstract** keyword in Java; it means there are no objects that are *directly* of type FSObject; they must be members of some subtype, instead.)

Alloy – operators

Operators are available to construct Boolean expressions.

- subset: **`in`**
  - *set1* **`in`** *set2* — *set1* is a subset of *set2*
  - informally: "some *set2* are *set1*", or "a *set2* may be *set1*"; but the set-theoretic meaning is more precise.
- set equality: **`=`**
  - *set1* **`=`** *set2* — *set1* equals *set2*
- scalar equality: **`=`**
  - `scalar` **`=`** *value* — *scalar* equals *value*

Alloy – example signatures and facts

- “There are things called animals”

```
sig Animal {}
```

- “A cat is a sort of animal”

```
sig Cat extends Animal {}
```

Exercise

Games:

- There are things called games.
- Games can be board games, or field games.
- There may be other sorts of games.

Alloy – subsets

- We saw that subtypes are disjoint.
- We can also declare sub*sets*:

  sig *signame* **in** *supername* { ... }

- Subsets are *not* necessarilly disjoint, and may have multiple parents

Alloy – subsets

```
sig Animal {}
sig Cat extends Animal {}
sig Dog extends Animal {}
sig FurryPet in Cat + Dog {}
```

- "`FurryPet`" is a subset of the union of `Cat` and `Dog`.
- Some dogs and cats may not be furry (hairless breeds).
- We could *make* them all furry as follows:

```
fact { Cat + Dog = FurryPet }
```

- Are there animals other than cats and dogs?
  Can they be furry?

More operators

- We can use Boolean connectives **`and`**, **`or`**, **`implies`**, **`iff`**, **`not`** to join Boolean expressions.
- e.g.

```
fact { A + B = C and X + Y = Z }
```

Relations

- In our file-system example, we also saw things in the *body* of signatures (i.e., between the braces).
- ```
  // A file system object in the file system
  sig FSObject { parent: lone Dir }

  // A directory in the file system
  sig Dir extends FSObject { contents: set FSObject }

  // A file in the file system
  sig File extends FSObject { }
  ```
  ```

Relations

- ```
  // A file system object in the file system
  sig FSObject { parent: lone Dir }

  // A directory in the file system
  sig Dir extends FSObject { contents: set FSObject }

  // A file in the file system
  sig File extends FSObject { }
  ```
- To a first approximation, we can think of relations as behaving like *fields* in an OO language.
- `sig FSObject { parent: lone Dir }` can be read as "Things of type `FSObject` *have a parent*, which is of type `Dir`".
- **`lone`** means "at most one" – i.e., you can have zero or one parents. (We need this because the root directory has no parent.)
  ```

Relations

- ```
  // A file system object in the file system
  sig FSObject { parent: lone Dir }

  // A directory in the file system
  sig Dir extends FSObject { contents: set FSObject }

  // A file in the file system
  sig File extends FSObject { }
  ```
- More precisely, `parent` is a relation between `FSObject` and `Dir`.
  ```

Relations – multiplicities

- **`lone`** is a type of multiplicity – it says how many of something there are.
- Other multiplicities:
  - **`one`** - one
  - **`some`** - at least one; one or more
  - **`set`** - zero or more
  - **`no`** - zero
- The default multiplicity is one.

Relations – multiplicities

- In set theory terms . . .
- **`one`** means the relation is a total function –
  `sig Student { name : one String }` –
  for every Student, we can map to a string which is their name.
- **`lone`** means the relation is a *partial* function –
  `sig Student { driverLicenseNum : lone String }` – \
  for every Student, we *may* be able to map to a diver's license number.
  (Here, it's assumed you can't have more than one license.)

Relations

- So, signature declarations will look like:

```
sig SomeName {
  field1 : FieldType,
  field2a, field2b : OtherFieldType
}
```

- The order of declarations doesn't matter – `SomeName`, `FieldType` and `OtherFieldType` could be declared in any order in a file.

Relations

- ```
  // A directory in the file system
  sig Dir extends FSObject { contents: set FSObject }
  ```
- Here, we say that a `Dir` has a field `contents`, which is a *set* of `FSObjects`.
- The could contain one item, many items, or no items.
  ```

Examples

- "A car has one engine"

  `sig Car { engine: one Engine }`, or
  `sig Car { engine: Engine }`

- "People have zero or more hobbies"

  `sig Person { hobbies: set Activity }`

Exercises

- Classes have at least one lecturer, and zero or more students.

Exercises

- Classes have at least one lecturer, and zero or more students.
- Animals have zero or more legs

Exercises

- Classes have at least one lecturer, and zero or more students.
- Animals have zero or more legs
- Some animals are carnivores

Exercises

- Classes have at least one lecturer, and zero or more students.
- Animals have zero or more legs
- Some animals are carnivores
- Textbooks have one or more pages

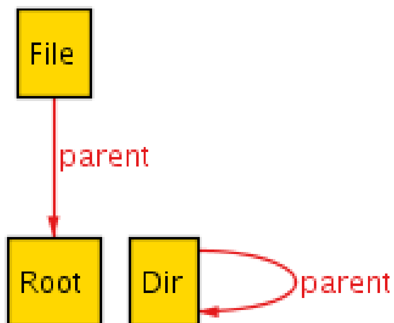
Back to the file system example

```
sig FSObject { parent: lone Dir }

sig Dir extends FSObject { contents: set FSObject }

sig File extends FSObject { }

// There exists a root
one sig Root extends Dir { } { no parent }
```

- FSObjects have parents, and directories have contents, and we have constrained the multiplicities . . .
- but there's currently no connection between them.

File system

- So we could have this situation:



File system

- We will need to constrain things more, so we'll use a *fact*.

```
// A directory is the parent of its contents
fact { all d: Dir, o: d.contents | o.parent = d }
```

- This says: "for any thing (let's call it *d* for the moment) of type `Dir`, and for any thing (let's call it *o* for the moment) which is in the set `d.contents`:
  o's parent is *d*.
- It uses a *quantifier* ("all") – we'll look at these more later.