CITS5501 Software Testing and Quality Assurance
Specification languages – Alloy

Unit coordinator: Arran Stewart

Overview

- Alloy specification language and Alloy analyser

Alloy idea

The idea behind Alloy is that:

- It lets you capture the essence of a design at a high level
- It lect you identify risky aspects of a design
- It lets you develop a model incrementally
- It lets you simulate and analyze the model as you go

Alloy idea

In other words, *before* you start implementing a system, you can start specifying the entities that make it up, what constraints (i.e. invariants) hold for them, and how they hang together.

What is Alloy?

- A flexible *language* for describing structures (and how they interrelate)
- It can describe both
  - static structures
  - dynamic behaviours[1]

---

[1] An Alloy extension, Electrum, exists which is well-suited for modelling properties of systems over time using temporal logic. However, we will restrict ourselves to very simple dynamic behaviours using plain Alloy.

What is Alloy?

- A flexible *language* for describing structures (and how they interrelate)
- It can describe both
  - static structures
  - dynamic behaviours[1]
- Comes with a tool, the Alloy *Analyzer*
  - Generates counterexamples to theorems/statements

---

[1] An Alloy extension, Electrum, exists which is well-suited for modelling properties of systems over time using temporal logic. However, we will restrict ourselves to very simple dynamic behaviours using plain Alloy.

Alloy advantages

- Small and easy to use
- Has a simple and uniform semantics based on mathematical *relations*
- Can be easily analysed using automated tools

Comparison with UML

Alloy has some similarities with UML –

- It has a graphical notation
- It is somewhat similar to the *Objects Constraint Language* used by UML[2]

And several differences:

- Unlike UML, Alloy has precise semantics
- It is a far smaller and simpler formalism than UML
  - UML allows for many constructs (e.g. use cases, state charts) that don't have an equivalent in Alloy

---

[2] https://en.wikipedia.org/wiki/Object_Constraint_Language

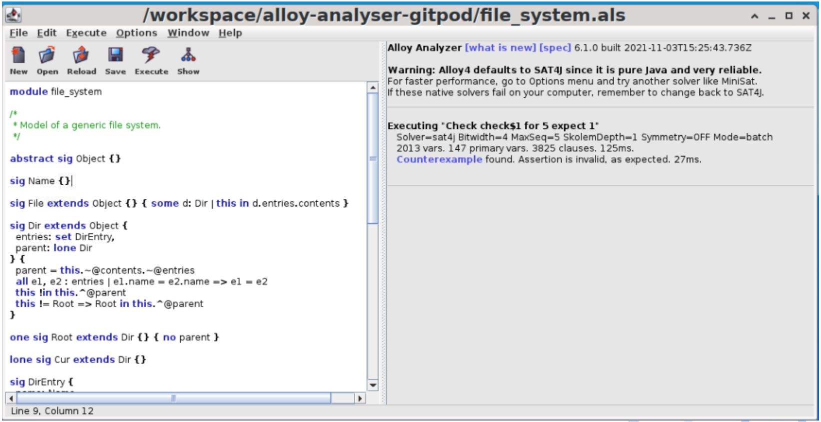
Using Alloy

- The Alloy analyser is distributed as a Java `.jar` file (or a `.dmg` file for Mac OS X) – see the Alloy 6.0 release page
  - The `.jar` file can be run like this:

```
java —jar org.alloytools.alloy.dist.jar
```

Using Alloy

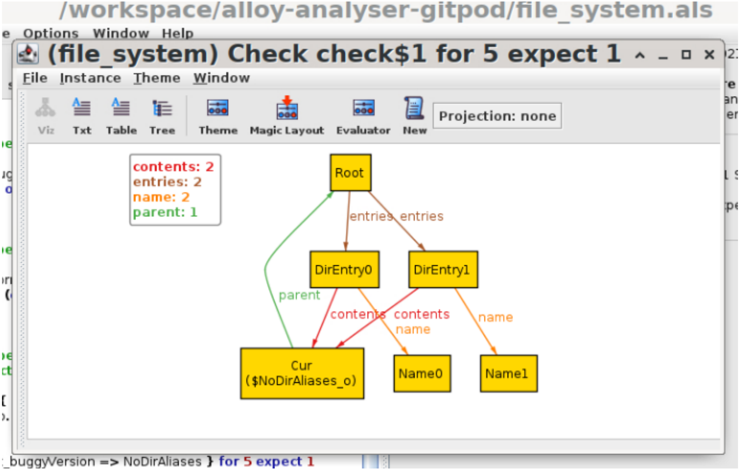
I have also set up a GitHub repository which lets you use the Alloy analyser from within an online IDE using Gitpod – visit https://github.com/arranstewart-dev/alloy-analyser-gitpod/



```
module file_system

/*
 * Model of a generic file system.
 */

abstract sig Object {}

sig Name {}

sig File extends Object {} { some d: Dir | this in d.entries.contents }

sig Dir extends Object {
  entries: set DirEntry,
  parent: lone Dir
} {
  parent = this.~@contents.~@entries
  all e1, e2 : entries | e1.name = e2.name => e1 = e2
  this !in this.^@parent
  this != Root => Root in this.^@parent
}

one sig Root extends Dir {} { no parent }

lone sig Cur extends Dir {}

sig DirEntry {
```

```
Alloy Analyzer [what is new] [spec] 6.1.0 built 2021-11-03T15:25:43.736Z

Warning: Alloy4 defaults to SAT4J since it is pure Java and very reliable.
For faster performance, go to Options menu and try another solver like MiniSat.
If these native solvers fail on your computer, remember to change back to SAT4J.

Executing "Check check$1 for 5 expect 1"
   Solver=sat4j Bitwidth=4 MaxSeq=5 SkolemDepth=1 Symmetry=OFF Mode=batch
   2013 vars. 147 primary vars. 3825 clauses. 125ms.
   Counterexample found. Assertion is invalid, as expected. 27ms.
```

Using Alloy



```
_buggyVersion => NoDirAliases } for 5 expect 1
```

Alloy analyser displaying a counterexample

Alloy – sigs

In Alloy, we declare rules about a mini-universe: things that exist, and properties that should be true of them.

We declare things that exist with `sigs` (short for “signatures”):

- “There are things called animals”

  ```
  sig Animal {}
  ```

- “A cat is a sort of animal”

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

The two declarations above declare kinds of “things” that exist.

Alloy – facts and assertions

We can also write:

- **facts**: These *force* something to be true of our model. They act as constraints on it. Alloy won't generate any instances of the model in which the facts don't hold.
    - A fact is *part* of our specification.
- **assertions**: An assertion *claims* something is true of our model (but it could be wrong).
    - You can think of these as similar to assertions in Dafny or in other languages
    - They're like a debugging tool such as `println` – they let you *ask* whether some fact is true or not.
    - The assertion isn't part of the specification; it's something we use to check what consequences flow from our specification.

Alloy

When modelling entities in Alloy – we normally include only the bare minimum of properties needed in order to show how the system "hangs together".

Alloy language

- For example – we'll look at a simple model of a file system (based on the Alloy tutorial at http://alloytools.org/tutorials/online/)
- An Alloy specification 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 primitives

- In Alloy, everything is built up from **atoms** and **relations**
- An *atom* in alloy is an indivisible, immutable value
  - We don't create these directly – they get automatically generated by the analyser
  - Example atoms: `A0`, `A1`, `B0`, `R0` ...
- A *relation* is a structure that relates atoms together –
  - It is a set of **tuples**

Alloy primitives

The easiest way to think of relations is probably to think of them as a sort of table – which show how columns of things are **related**.

e.g. "shares an office with":

Person A	Person B
Alice	Bob
Bob	Alice
Dan	Eve
Eve	Dan

Each row is called a *tuple*.

Alloy relations

In other languages, we might have scalar values (e.g. `int`s, `double`s), various types of containers (e.g. `array`, `List`), and ways of combining types together into a `class` (or `struct` in C).

In Alloy, these are all subsumed under *relations*.

We will see that sets, scalars, properties and so on, are all defined in terms of relations.

Alloy – relations

Alloy's semantics are defined in terms of relations.

Example relations:

- "Is less than". e.g. "$2 < 4$", "$10 < 9$".
- "Is the blood relative of". e.g. "Alice is the blood relative of Bob".
- "Shares an office with". e.g. "Bob shares an office with Carol".

These are all *binary relations*. Statements about two entities, which can be true or false.

Alloy – relations

Relations can also be *unary* (about one entity):

- "Is even". e.g. "*even*(2).
- "Is an employee". e.g. "Dan is an employee".

Alloy – relations

They can be ternary:

- "_ is delivered to _, by _". e.g. "The *blue book* was delivered to *Alice*, by *Bob*".

Or, in general, they can be n-ary – a statement about n things.

Alloy – relations

We can think of predicates as being a bit like functions – an *n*-ary predicate isn't true or false in itself, until we supply it with *n* arguments.

- "Is less than" isn't true or false, but "$2 < 4$" is.

Alloy – relations

Relations can be finite, or infinite.

An infinite relation: “is less than”

Number A	Number B
1	2
1	3
2	3
...	...

Alloy relations

- *Sets* are unary (1-column) relations. e.g.

```
Name = { N0,
         N1,
         N2 }
```

- *Scalars* are actually 1-element sets:

```
myName = N0
```

- Binary or ternary or higher relations are possible:

```
names = { (B0, N0),
          (B0, N1),
          (B1, N2)  }
```

Alloy – sigs

`sig Animal {}` says “There are things called animals”.

It defines a unary relation, “Animal”. Something thing can be-an-animal, or not.

Alloy – sigs

`sig Cat extends Animal {}` says "Cats are a sort of animal".

If something has the property "is-an-animal", *then* it might also have the property "is-a-cat".

We can read "extends" as also meaning "is a kind of", or "is a subtype of".

Alloy – subtypes

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

```
sig FSObject {}

sig Dir extends FSObject {}

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
- The above says "There are things called FSObjects. An FSObject might be a Dir or it might be a File, but not both".

Alloy – properties

We can specify *properties* of entities:

```
// 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 – properties

```
// 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 { }
```

These are usually written within the sig of an entity.

They actually represent *relations* between entities.

Alloy – properties

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

There are multiple ways of reading this:

- "There are such things as FSObjects. An FSObject has the property 'parent'. An FSObject can have zero or one parents."
  Or –
- "A relation 'parent' exists between FSObjects and Dirs. Whenever an FSObject appears in the relation, it can be association with at most one Dir."

These are exactly equivalent.

Alloy – properties

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

- The “`lone`” means “zero or one”. It is a *cardinality*.
- Other possible cardinalities are:
  - “some” (one or more)
  - “one” (exactly one)
  - “set” (zero or more)
- When we specify a property using a colon in this way, the default multiplicity is one.
- We can use cardinalities whenever we are specifying a set or relation: since sigs also represent sets (e.g. the set of Dirs), we can give them cardinalities, too.

Alloy – properties

```
one sig RootDir extends Dir { }
```

There exists a “RootDir”, but only one of them.

Exercise

Games:

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

Alloy language – comments

```
// 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 – comments

```
// 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 – comments

```
// 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 – 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`**.

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 – 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).

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`".
- Recall that **`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 `FSObject`s.
- 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

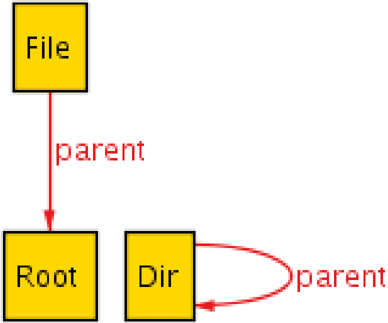
Back to the file system example

```
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.