CITS5501 Software Testing and Quality Assurance
Semester 1, 2020
Week 11 Workshop – Formal methods

Reading

It is strongly suggested you complete the recommended readings for weeks 1-10 *before* attempting this lab/workshop.

Installing/accessing the Alloy Analyzer

The Alloy Analyzer is a tool used for checking Alloy models.

A web interface is available at:

- http://alloy4fun.inesctec.pt

You can also download the analyzer to run on your own computer (or a lab computer) – it should run on (at least) Windows, MacOS X and Linux, as long as you have a Java runtime installed.

You can download the analyzer from the web page here:

- https://github.com/AlloyTools/org.alloytools.alloy/releases/tag/v5.1.0

Use the `alloy.dmg` file if you are on a Mac, and the `org.alloytools.alloy.dist.jar` file if you are on Windows or Linux.

On many systems, to run the analyzer, you simply need to double-click on the `.jar` file. If this doesn't work, then it can be run from the command line:

```
1 $ java -jar org.alloytools.alloy.dist.jar
```

Alloy syntax

- Note that an "Alloy syntax cheat sheet" is available at https://esb-dev.github.io/mat/alloy-cheatsheet.pdf, but we will be covering only a small fraction of the Alloy syntax.

Using the Alloy analyzer

A tutorial for using the Alloy Analyzer is available here:

- https://alloytools.org/tutorials/online/index.html

Work through chapters 0 and 1, which show how to use the Analyzer.

Alloy sigs and properties

How would you translate the following into Alloy syntax? All of these can be done by declaring *sigs* and *properties*.

a. There exist such things as chessboards.
b. There is one, and only one, tortoise in the world.
c. There exists at least one policeman.
d. Files have exactly one parent directory.
e. Directories have at most one parent directory.
f. Configuration files have at least one section.

Sample solutions

a. There exist such things as chessboards:

```
sig Chessboard { }
```

b. There is one, and only one, tortoise in the world:

```
one sig Tortoise { }

// Note that any time we want to constrain the number of
// members of a sig, we can do it in the shorthand way
// above; but it can also be constrained using a *fact*
// (syntax below).
// But putting it in the sig is usually easiest
// to read.

sig Tortoise {}

fact oneTortoise {
  #Tortoise = 1
}
```

1. There exists at least one policeman:

```
some sig Policeman { }
```

2. Files have exactly one parent directory.

   Note that we *only model what we are asked to model*; we don't add in any sigs or properties, besides the ones needed.

   The question doesn't say (for instance) that files and directories are both types of "file system object" – so we shouldn't model it.

```
sig Directory { }

// note that we could leave the 'one' off if
// we wanted -- it is the default multiplicity --
// but it's often clearest to leave it in.
sig File {  parent : one Directory }
```

1. Directories have at most one parent directory.

```
sig Directory { parent : lone Directory }
```

   There is no problem with having "recursive" sigs that "refer to themselves". Recall that a property is really just expressing a *relation* between entitities. The above code just says that there are such things as directories, and that any directory can be in a relationship with zero or one other directories.

   (Extra exercise: does the sig allow a directory to be its own parent? You should be able to work this out from the lecture slides and the tutorial – or by using the Alloy Analyzer.)

2. Configuration files have at least one section.

```
sig ConfigFile { sections: some Section }

sig Section {}
```

Alloy facts

Recall that in Alloy, *facts* are additional constraints about the world, that aren't expressed in the sigs, and can be used to "tighten" the meaning of your model. (Some constraints could be expressed either in the sig, or as a fact.) For instance, using the example `File` and `Dir` and `FSObject` sigs from the lecture:

```
fact {
  File + Dir = FSObject
}
```

means, "the set of files, plus the set of directories, is the same as the set of all file-system objects".

Or, if we give our fact a name:

```
fact noOtherFSObjects {
  File + Dir = FSObject
}
```

Take a look at the Alloy quick reference, which gives other operators you can use besides "`+`" and "`=`". For instance, "`-`" (set subtraction), "`#`" (set cardinality, or "size"), "`&`" (set intersection), "`in`" (set membership), typical comparison operators ("`<`", "`>`", "`=<`", "`=>`"), and typical logical operators ("`&&`", "`||`", "`!`"), and see if you can give Alloy facts which express the following.

a. Assume we have a sig `LectureTheatre{}` and a sig `Venue{}`.

   Give a fact which constrains every lecture theatre to also be a venue. (Note that we could do this using "extends" in the sig, also. But sometimes it's more convenient to express things using facts, or the constraint we want is too complicated for just "extends".)

b. Assume we have the sigs `DomesticatedAnimal{}`, `Canine{}`, `Dog{}`. Write a fact constraining `Dog` to be the intersection of `DomesticatedAnimal` and `Canine`.

Sample solutions:

a. Every lecture theatre is a venue

```
sig Venue {}
sig Lecture {}

// All lecture theaters are venues
fact { LectureTheatre in Venue }

// However note that you will get a warning in Alloy if you
// try this: by default, each sig is a distinct type.

// the following will run without warnings:
//
// sig Venue {}
// sig Lecture in Venue {}
```

b. `Dog` is the intersection of `DomesticatedAnimal` and `Canine`.

```
sig DomesticatedAnimal { }
sig Canine { }
sig Dog {}

fact { Dog = DomesticatedAnimal & Canine }

// As before -- the above will give warnings.
//
// Code that runs without warnings:
//
// sig Animal {}
// sig DomesticatedAnimal in Animal {}
// sig Canine in Animal {}
// sig Dog in Canine {}
//
// fact { Dog = DomesticatedAnimal & Canine }
```

Facts with quantifiers

The facts in the previous section constrain sets (e.g. the set of lecture theatres, or the set of omnivores).

We can also write constraints that apply to every entity *in* some set.

For example, suppose we have the following sigs:

```
sig Activity {}
sig Person { hobbies: set Activity }
sig ComputerScientist extends Person {}
```

We can apply the following constraint: "Computer scientists have no hobbies:"

```
fact {
  all cs : ComputerScientist | #cs.hobbies = 0
}
```

In other words: people can have zero or more hobbies; but for all people who are computer scientists, if we look at their hobbies, the cardinality will be 0.

It's also possible to write this using "no" (another sort of "multiplicty", like `lone` or `set`):

```
fact {
  all cs : ComputerScientist | no cs.hobbies
}
```

Challenge exercise: try extending this model to say:

a. Economists are also people.
b. Economists have at most one hobby.
c. Students are people.
d. Students have at least one hobby.
e. Bots are not people.

Sample solutions:

a. Economists are also people:
```
// if we assume Economist never overlaps with ComputerScientist sig
↪ Economist extends Person {}
```
b. Economists have at most one hobby.
```
fact { all econ : Economist | lone econ.hobbies }
```
c. Students are people.
```
sig Student extends Person {} // Alternative to this: plausibly, we
↪ might instead model // students as a subset of Person, so you can
↪  // be a student economist.
```
d. Students have at least one hobby.
```
fact { all s : Student | some s.hobbies }
```
e. Bots are not people.
```
// We can just declare bots as a separate sig -- // by default, they
↪  won't overlap with Person sig Bot {}
```

Further challenge exercises

Model the following systems:

1. An *alarm clock* has two sorts of time it can keep track of: the *current* time, and an *alarm* time.

   It *always* has a current time, and *may* have an alarm time.

2. A person can have up to two parents (who are also people). No person is their own parent. There exists exactly one person – let's call them Bob – who has no parents. (Poor Bob.)

3. Entities called *nodes* exist (we will use them to model linked lists). A node may have a successor node, called its "*next*" node.

4. A *contact list* may contains multiple *entries*. Each entry must have a "personName", and may have one or more telephone, street address, or emails.