CITS5501 Software Testing and Quality Assurance
Semester 1, 2022
Workshop 10 (week 11) – Formal methods – solutions

Reading

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

Acessing the Alloy Analyser

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

You can 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.

Running on your own laptop/PC

You can download the analyzer from the web page here:

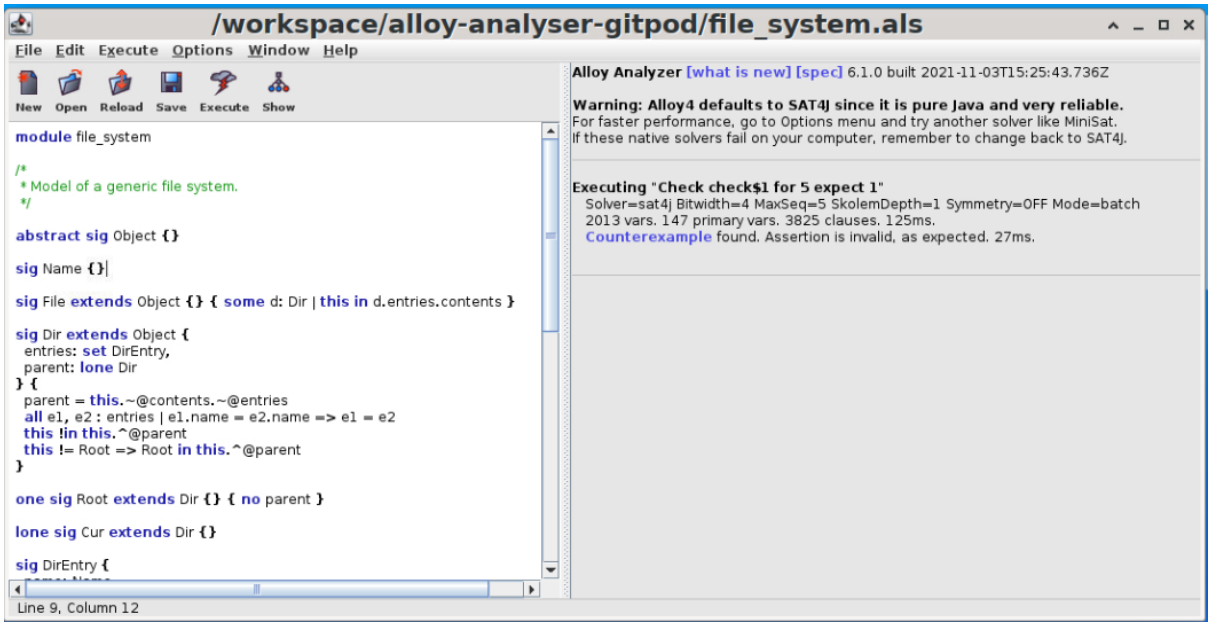
- https://github.com/AlloyTools/org.alloytools.alloy/releases/tag/v6.0.0

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:

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

The running analyser should look something like this:



If the *fonts* displayed look odd – from the "Options" / "Font" menu, select "Courier" or "Courier New", and a reasonable font size (12pt or 14pt), then restart the analyser.

Accessing via Gitpod

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/ and follow the instructions there.

A limitation of Gitpod is that you can't copy and paste between the *Alloy analyser* window and the clipboard of the computer you're using. However, you *can* copy and paste to the tab in which Microsoft VS Code is open. So if you want to paste code into an Alloy `.als` file, the way to do it:

- Create the `.als` file using VS Code, using "File" / "New file" from the menus.
- *Also* open the `.als` file in the Alloy Analyser.
  (Note that your files will be available under `/workspace/alloy-analyser-gitpod`.)
- Edit and/or pase into the VS Code tab
- In the Alloy Analyser tab, after you've made a change, select "File" / "Reload all" to load the current version of the code.

Further information

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.

For your reference, a tutorial for using the Alloy Analyzer is available here:

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

An online book that shows in more detail how to create and analyse software specifications with Alloy is available here:

- https://haslab.github.io/formal-software-design/

A. Sigs and properties

We will start with Alloy by investigating how to model entities ("sigs", in Alloy) and their 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
}
```

c. There exists at least one policeman:

```
some sig Policeman { }
```

d. 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 }
```

e. 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.)

f. Configuration files have at least one section.

```
sig ConfigFile { sections: some Section }

sig Section {}
```

B. Viewing "possible universes"

Alloy has two *commands*, "`run`" and "`check`". We'll consider "`run`" first. We can ask Alloy to generate sample "universes" which meet the constraints set out by our model, by using the "`run`" command.

Let's suppose we want to model a situation in which "Houses have doors, and houses can have more than one door".

Try entering the following into a fresh `.als` file:

```
sig Door {}
sig House { doors: set Door }

```

```
4 pred example() {
5 }
6
7 run example for exactly 2 House, 2 Door
```

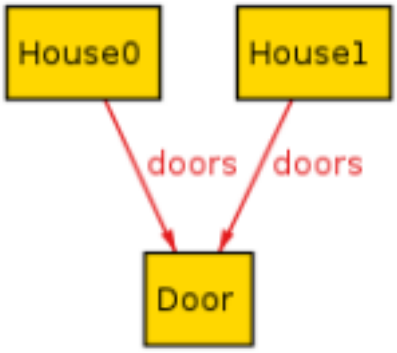
Save the file, and under the "Execute" menu, you should now see a menu itme "Run example for exactly 2 House, 2 Door" – select that menu item. (Or, press the "Execute" button.)

In the right-hand alloy pane, you should see the message "Instance found", and a clickable link – click the instance, and you should see one "possible universe" that the Alloy analyser has created for us.

Some points to note:

- The `run` command takes one mandatory argument, a predicate. The predicate acts as a sort of "filter", limiting the possible "universes" Alloy will show us. An empty predicate, like `example`, means "no limits".
- The "`for exactly 2 House, 2 Door`" part specifies a *scope*, and is optional. Try deleting it and execute the `run` command again. By default, the analyser assumes there may up to three of each top-level signature, and any number of relations. You can read more about scopes in the alloy documentation.
- We can have more than one `run` command in a file: by default, the analyser will execute the first one it encounters.

Reinstate the scope we originally had and re-execute the `run` command, and click on the link to show the visualisation. You should see one "possible universe" that the Alloy analyser has created for us:



Assuming we think that doors should only ever belong to a single house, then the universe we're being shown here is *underconstrained* – our specification is too loose, and allows in "silly" possibilities we don't actually want.

There are multiple ways we could fix this.

Option 1: We could decide that doors aren't really important enough to be an entity, and model the number of doors a house has as just an `Int` (capital "I"):

```
sig House { numDoors: one Int }

pred example() {}

run example for exactly 2 House
```

Option 2: We could add in additional *facts* to constrain our "universes". The easiest way to do this is to do the following:

- Give each `Door` a property `house: one House`
- Constrain the model so that for all houses and doors: if d is the door of some house h, then $d.house = h$. (In other words – the property "house" is like the inverse, or opposite direction, of "door".)

We end up with

```
sig Door  { house: one House }
sig House { doors: set Door }

fact HouseDoorInverse {
  all h: House, d: Door | d in h.doors implies d.house = h
}

pred example() {
}

run example for exactly 2 House, 2 Door
```

Try running this again. You should end up with the following visualisation:

If we think every house should have at least one door, then our model is still underconstrained, because the analyser is showing us houses with no doors; we should have use "some" instead of "set" when writing `sig House { doors: set Door }`. Try changing "set" to "some", and run again.

Experiment with different scopes for the "run" command.

C. 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 }
```

D. 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 {}
```