CSCU9P6 Software Engineering II

Alloy Practical 1: Introduction to Alloy

# Aims

1. To gain experience in using the Alloy Analyzer to create simple specifications.
2. To learn more about using sets and relations in Alloy.

# Supporting Documentation

1. See <http://alloy.mit.edu/alloy/documentation.html> for an FAQ, language reference, quick guide, online tutorial, and other information.

**Note:** This worksheet contains **ONE** checkpoint.

# Introducing the Alloy Analyzer

The Alloy Analyzer is an interactive system for creating and analysing specifications written in the Alloy language. Specifications can be built up incrementally, using the Analyzer’s model-checking and visualization functions to check the correctness of the specification at each step.

In this first practical you will use the Analyzer to create and explore some simple specifications. The main purpose of the practical is to become comfortable using the tool, and to get a better understanding of how to work with sets and relations.

* Find Alloy in your Start menu and start it up now.

After reading the welcome message, close it and have a look at the screen. There are two main sections. On the left is a simple text editor, which is where you will create your specifications. On the right is a section for showing feedback from the Analyzer.

We will use an example seen in the lectures: sets of fruit.

* Type in the following into the text editor:

**module** setExamples

**sig** Fruit { }

This is a very simple specification. The first line is a header, giving the name of this specification module. It is not necessary to name your specification, but it can be useful when building large specifications which are split into several modules. Naming a module allows it to be imported into other modules where it can be reused.

The second line introduces a set called Fruit.

Now let’s introduce a command that will allow us to visualize models of our specification.

* Type in the following:

**pred** show { }

**run** show

The first line introduces a predicate, that is, a statement that expresses a logical property (similar to a Boolean expression in Java). A predicate has a *name*, some optional *parameters*, and a *body* enclosed in braces. This predicate is named show, it has no parameters, and its body is empty. A predicate with an empty body is always true.

The second line is a command: it instructs the Analyzer to run the predicate show. When this command is executed, the Analyzer will attempt to find a model instance of your specification in which the predicate show is true. Since show is always true, this amounts to asking the Analyzer simply to find a model instance of the specification.

* Execute the command. You can do this from the **Execute** menu, by choosing the command from the list displayed. Alternatively, click on the **Execute** button (this is a quick way to repeat the last command executed).

The analyser returns some feedback from executing the command. It has been able to find an instance. This means that the specification and predicate are *consistent*: it is possible to find models that meet the specification and also make the predicate true. Since the predicate show is always true anyway, this is really telling us something useful about the specification: it describes a world that makes sense and can actually exist. This is generally a good thing.

Let’s take a look at some of these possible worlds:

* Click on the word **Instance** highlighted in blue in the feedback panel.

This brings up the **Visualizer** tool which allows us to “see” the models that the Analyser has found. The first model may look rather disappointing with nothing to see apart from a message telling you that every atom is hidden. This model is the world in which the set Fruit is empty, so there are no atoms to show. This is a perfectly legal instance of the specification, but is rather boring. Let’s find a more interesting model.

* Click on the **Next** button in the Visualizer to bring up the next model.

You should see a more interesting model this time.

* Continue clicking the **Next** button to see all the models that the Analyzer can find.

The Analyzer finds models with 0, 1, 2, and 3 fruit. Why does it stop there? Because the Analyzer works within a limited *scope*, and the default scope is 3. To find larger models we need to use a bigger scope:

* Change the run command to run show for 5 and see how many instances are found.

Now let’s make the specification more interesting:

* Type in the following, then run show to see instances of your specification. Use a scope of about 5 or 6.

**sig** Banana, Apple, Pear **extends** Fruit { }

You should find models containing various combinations of Bananas, Apples, Pears, and other Fruit.

All the fruit look the same. Let’s make them look different.

* Click on the **Theme** button in the Visualizer window. The **Theme Editor** panel appears, displaying a tree showing the different sets present in your model (univ, Fruit, Apple, Banana, and so on). Click on Apple.

The Theme Editor now allows you to control how Apple atoms are displayed in the Visualizer. The default settings all say “Inherited”, meaning that Apples look just like Fruit (because Apple extends Fruit).

* Change the colour of Apples to red and change their shape to circles. Leave all the other settings untouched. Similarly, give appropriate colours and shapes to Bananas and Pears. When you are done, click **Apply** to apply the theme and **Close** to close the Theme Editor.

Your models should look a bit more interesting when you step through them now!

* Add the following to your specification:

**sig** Fresh, Expensive **in** Fruit { }

Can you remember the difference between **extends** and **in**? Here is a reminder:

**extends** is used to introduce subsets that are disjoint from each other (do not overlap). This means, for example, that a Banana cannot also be an Apple or a Pear. **in** is used to introduce subsets that may overlap with each other and with other existing subsets. This means that a Fresh fruit may also be Expensive (and may also be a Banana).

* Look at instances of the specification. Notice that the Visualizer adds labels to show which sets each Fruit belongs to. The labels are of the form this/Fresh, meaning that the atom belongs to the set Fresh defined in the local module (“this”). See if you can spot a model containing a Fresh Apple that is not Expensive (a bargain!).

Did you find one? In fact, if you are looking for a model with a specific property, there is a better way to find it than stepping though all possible models. You can run a predicate that describes the specific property you are looking for.

* Add the following to your specification and then execute the command you just added (**run** bargainApple, which you will find inside the **Execute** menu):

**pred** bargainApple {

**some** Apple & (Fresh - Expensive)

}

**run** bargainApple

Did you find that perfect apple? Now take a closer look at the predicate that helped you to find it. It uses some of the set operators we met in the lecture.

Fresh - Expensive uses the *difference* operator: this means that it is a set containing all atoms that belong to the set Fresh but not to the set Expensive.

Apple & **(**Fresh - Expensive) uses the *intersection* operator to find all atoms that belong to Apple and also to **(**Fresh - Expensive). So this term represents the set containing those affordable fresh apples. [Note that the brackets are unnecessary, but I find they aid understanding.]

Finally, the quantifier **some** placed before this set says that *there exists* *at least one* atom in this set. This turns the set expression into a logical sentence describing the kind of model we wish to find.

**Checkpoint task 1:** Complete the three predicates described below. **Copy your answers into the spaces provided.**

* + There are no expensive, non-fresh bananas. (Use the quantifier **no**).

**pred** noNastyBananas {

***no Banana & (Expensive - Fresh)***

}

* + All fresh fruit is expensive. (Recall S **in** T means that S is a *member* or *subset* of T.**)**

**pred** allFreshFruitIsExpensive {

***no Fruit & (Fresh - Expensive)***

***Fruit in Fresh &(Expensive)***

***Fresh in Fruit &(Expensive)***

}

* + Bananas and fresh apples are expensive. (You may need the *union* operator, +)

**pred** BananasAndFreshApplesAreExpensive {

***some (Apple & (Fresh & Expensive)) && some Banana & (Expensive)***

}

Now let’s take a look at one very useful part of the Visualizer that we haven’t met yet: the **Evaluator**.

* Run the show predicate again with a scope of about 7. Step through the instances until you find one that has a nice, varied selection of fruit. Now click on the **Evaluator** button.

The Evaluator panel appears. Read the instructions at the top. Now try evaluating an expression:

* + **univ**

This is a very useful first expression to evaluate, as it shows you the contents of **univ**, the “universal set”. In other words, it shows all the atoms present in the model instance you are looking at.

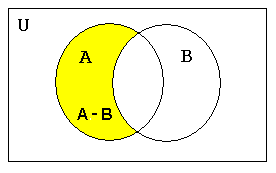
The universal set contains atoms corresponding to the various fruit in your model instance. Note that these are given “internal” names that look slightly different from those shown in the visualization: for example a fruit named Pear0 in the visualization has the internal name Pear$0.

* Try evaluating the following set expressions. What do you think they mean?

Fruit - Expensive

*Difference*

*Fruit that is not expensive*



Expensive – Fruit

*Difference*

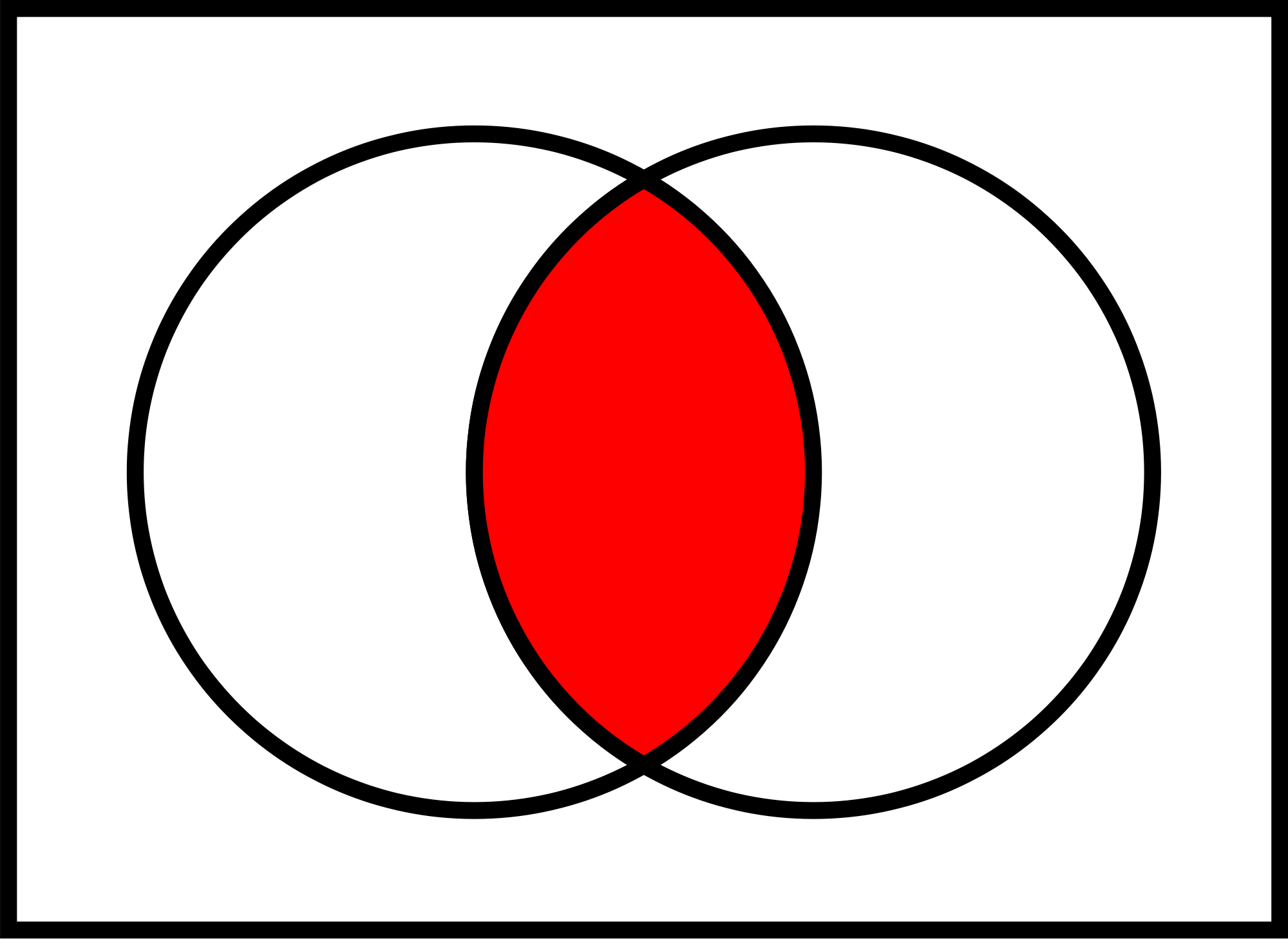
*Expensive that is not a fruit*

(Banana & Fresh) – Expensive

*Intersection, Difference*

*Fresh Banana that is not expensive*

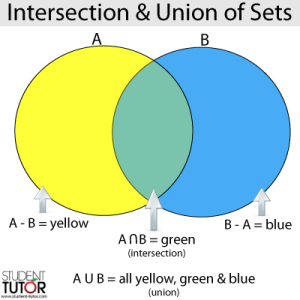
*A ∩ B.*



(Pear + Banana) & Fresh

*Union, Intersection*

*Fresh Pears and Fresh Bananas*



**Checkpoint task 2:** Write expressions to represent the two sets described below, and evaluate them in Alloy. **Copy your expressions into the spaces provided**

* All things (atoms) that are not fresh bananas

**pred notFreshBananas {**

**some Fruit no (Fresh & Banana)**

**}**

**run notFreshBananas**

* Fruit that is not fresh and is expensive

**pred notFreshAndExpensive {**

**some Fruit no(Fresh & Expensive)**

**}**

**run notFreshAndExpensive**

The evaluator can also be used to evaluate logical statements (expressions which have a true/false value).

* Try evaluating the following statements. Can you tell what they mean? Whether or not they are true will depend on the details of the model instance you have selected. If your model is missing some atom I have used (for example, Pear$0) substitute a different atom from your model.
  + Pear **in** (Expensive & Fresh)

Pear is expensive and fresh

* + **some** Banana – Fresh

Banana is never fresh

* + Pear$0 **not in** Fresh

Pear$0 cannot be found

Pear$0 is not Fresh

* + Pear$0 **in univ**

Pear$0 cannot be found

??

* + **no** (Pear + Apple) & Fresh

Pear and Apple are not fresh

**Checkpoint task 3:** Write statements to express the following two logical properties and evaluate them in Alloy. **Copy your statements into the spaces provided.**

* + There is something which is expensive and is not a fruit.

**some Expensive no(Fruit)**

* + Bananas and apples are fresh

**Banana + Apple in Fresh**

* Try coming up with some other statements of your own to evaluate.

When you have finished working with sets, save your specification in an appropriate place in your home folder. It will be saved as a file with the name you choose and the extension .als.

Checkpoint [Alloy 1: Sets]

Show the demonstrator your worksheet with completed answers to the three checkpoint tasks.

**Aside: Printing Alloy Specifications and models**

Alloy files (with the .als extension) are ordinary text files. They can be opened and printed from a text editor such as TextPad. The Alloy Analyser itself does not seem to have a Print function.

Alloy model visualizations can be exported as .png or .pdf files and inserted into other documents to be printed. To export a visualization, right click in the Visualizer window and choose the appropriate command from the resulting menu.