Chapter 1

A Guide To Alloy

1.1 Introduction

In this report we will discuss the subject of our group project, the Alloy specification language and its application through the Alloy Analyzer. The website we produced as a part of this project can be foundhere [2]. This chapter is an adapted version of it.

1.1.1 Philosophy

Software development is difficult; choosing correct abstractions to base your design around is a tricky process! Quite often apparently simple and robust designs can turn out to be incoherent and even inconsistent when you come to implement them.

One way of avoiding this problem of "wishful thinking" as well as making software development more straightforward is through formal specification. This method approaches software abstraction head-on using precise and unambiguous notation. An example of this would be the declarative language Z.

Naturally this is not fool proof either though; proving your specification is correct/sound requires a substantial amount of effort. Whilst conventional theorem based analysis tools have improved, their level of automation is still relatively poor compared to model checkers.

Hence, Alloy was developed in the hope of adapting a declarative language like Z to bring in fully automatic analysis. To do this Alloy sacrifices the ability totally prove a system's correctness. Rather, it attempts to find counterexamples within a limited scope that violate the constraints of the system. Thus by combining the best of both worlds Alloy produces a rigorous model that gives the user immediate feedback.

[2] http://www.doc.ic.ac.uk/project/2007/271j/g06271j01/Web/

1.1.2 History

The design was done by the Software Design Group at MIT lead by Professor Daniel Jackson. In 1997 they produced the first Alloy prototype, which was then a rather limited object modelling language. Over the years Alloy has developed into a powerful modelling language capable of expressing complex structural constraints and behaviour.

Figure 1.1: Professor Daniel Jackson

By employing first order logic, Alloy is able to translate specifications into large Boolean expression that can be automatically analyzed by SAT solvers. Thus when given a model (in Alloy), the Alloy Analyzer can attempt to find an instance which satisfies it.

One of the great benefits this lends to Alloy is the ability of incremental analysis. A programmer may explore design ideas starting from a tiny model which is then scaled up, with Alloy able to analyse it at every step.

Alloy has gradually improved in performance and scalability and has been applied to many fields including scheduling, cryptography and instant messaging.

1.2 Tutorial

1.2.1 Getting Started

This is an adapted version of our online tutorial which is available here [3].

We will be incrementally developing examples and introducing the features and concepts as we go. Keywords will be **highlighted** and their definitions can be found in the glossary.

What will be included in this tutorial:

- A guide to Alloy's interface and visualiser
- Writing a model in Alloy
- Troubleshooting & Tips on Alloy

[3] http://www.doc.ic.ac.uk/project/2007/271j/g06271j01/Web/Tutorial_Intro.php

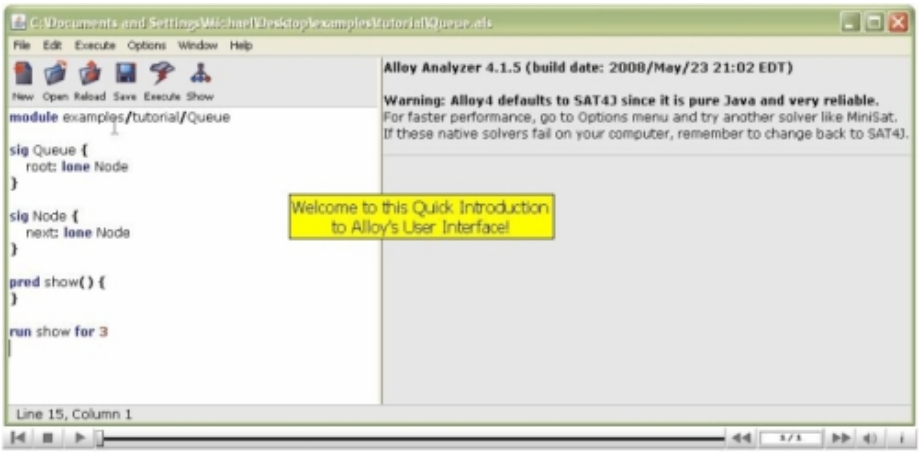


Figure 1.2: Click to launch video!

A video tutorial on using the Alloy Analyzer is available on the website[4].

1.2.2 Statics I

In this section, we will be using Alloy to model a linked list implementation of a Queue: Every queue has a root node and each node has a reference to the next node in the queue.

Each Alloy model starts with the **module** declaration:

```
module examples/tutorial/Queue
```

This is similar to declaring a class `Queue` in the package `examples.tutorial` in Java. Similar to Java, the file should be named 'Queue.als' and be located in the subfolder examples/tutorial of the project folder.

The first step is to declare the **signature** for the queue and its nodes:

```
sig Queue { root: lone Node }
sig Node  { next: lone Node }
```

You can think of `root` and `next` as being fields of type `Node`. **lone** is a multiplicity keyword which indicates that each `Queue` has less than or equal to one root (similarly for `next`).

A **signature** in Alloy is similar to the signature of a schema in that it defines the vocabulary for the model. We can already visualize what we have written so far. The common way of doing this is to write a predicate and then make Alloy produce instances that satisfy this predicate. Asking Alloy to find instances is similar to finding models of a given schema.

[4] http://www.doc.ic.ac.uk/project/2007/271j/g06271j01/Web/Alloy%20Demo.html

Because we want any instance of the Queue so far, the predicate has no constraints:

```
pred show() {}
```

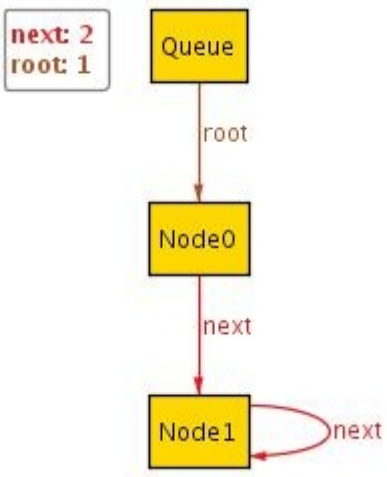
To get sample instances of the predicate we use the command **run**:

```
run show for 2
```

A very important fact about Alloy is that it is only designed to search for instances within a finite scope. In the above case this scope is set to at most 2 objects in each signature(i.e. at most two `Queues` and at most two `Nodes`). Note that this is always an upper bound; Alloy may return a smaller instance.

In order to produce and visualize an instance, we first execute the **run** command by clicking on the *Execute* button. After the execution has finished, Alloy will tell us that it has found an instance which we can visualize by clicking on the *Show* button.

```
module examples/tutorial/Queue
sig Queue { root: Node }
sig Node { next: lone Node }
pred show() {}
run show for 2
```



In the above image, we see that Node1 is its own successor. As this is not what we usually expect from a linked list, we add a **fact** to constrain our model:

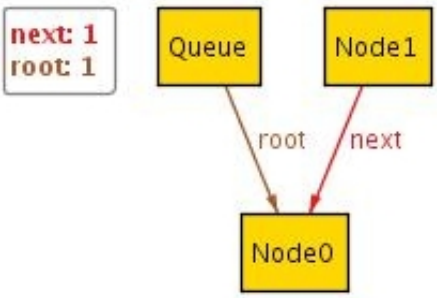
```
fact nextNotReflexive { no n:Node | n = n.next }
```

Facts in Alloy are 'global' in that they always apply. They correspond to the axioms of a schema. When Alloy is searching for instances it will discard any that violate the facts of the model. The constraint above can be read just like in first order logic: there is no `Node n` that is equal to its successor.

Executing **run** `show` now produces the following instance:

```
module examples/tutorial/Queue

sig Queue { root: Node }
sig Node { next: lone Node }
fact nextNotReflexive { no n:Node | n = n.next }
pred show() {}
run show for 2
```



We no longer have a Node that is its own successor, but now notice another problem of our model: There are Nodes that do not belong to any Queue.

1.2.3 Statics II

To address the issue of nodes that do not belong to any queues, we add another **fact**:

```
fact allNodesBelongToSomeQueue {
      all n:Node | some q:Queue | n in q.root.*next
}
```

all and **some** behave exactly like the forall and exists quantifiers in predicate logic. The .* operator represents the *reflexive transitive closure*: It returns the set consisting of all elements:

```
q.root,
q.root.next,
q.root.next.next,
...
```

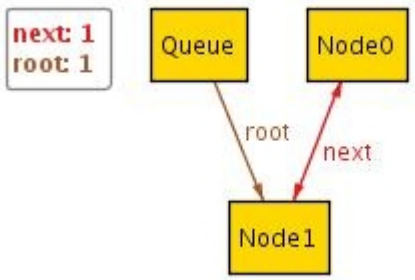
If we press Execute, Alloy tells us that it found an instance. However, when we ask it to visualize the instance, it tells us that every atom is hidden. In order to get the next solution, we click Next at the top. Browsing through the instances, we find that one of them contains a cycle:

```
module examples/tutorial/Queue

sig Queue { root: Node }
sig Node { next: lone Node }
fact nextNotReflexive { no n:Node | n = n.next }
```

```
fact allNodesBelongToSomeQueue {
      all n:Node | some q:Queue | n in q.root.next
}


pred show() {}
run show for 2
```



In order to fix our model we add another **fact**:

```
fact nextNotCyclic  no n:Node | n in n.^next
```

In contrast to the .* operator, .^ represents the *non-reflexive transitive closure* which returns the set consisting of all elements:

```
n.next,
n.next.next,
n.next.next.next,
...
```

(Note that this set does not include `n` itself).

Executing, visualizing and browsing through a few instances, we spot another error:

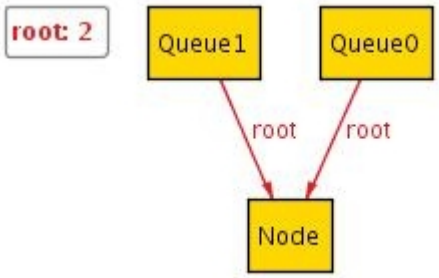
```
module examples/tutorial/Queue

sig Queue { root: Node }
sig Node { next: lone Node }
fact nextNotReflexive { no n:Node | n = n.next }

fact  allNodesBelongToSomeQueue {
      all n:Node | some q:Queue | n in q.root.next
}

fact nextNotCyclic { no n:Node | n in n.^next }

pred show() {}
run show for 2
```



The problem here is that Node belongs to two different queues. This is because `allNodesBelongToSomeQueue` constrains a Node to belong to some Queue while it should actually belong to exactly one. To fix this we modify the **fact**:

```
fact allNodesBelongToOneQueue {
      all n:Node | one q:Queue | n in q.root.*next
}
```

Browsing through the instances for this version of the model, Alloy soon tells us that 'there are no more satisfying instances'. All the solutions found within the scope provided seem to be correct. To gain more confidence, we increase the scope further:

```
run show for 3
```

Still, we cannot find any instances that clash with our definition of a linked list and conclude that this seems to be a good model for a linked list implementation of a queue. However, note that this does not *prove* that our model is correct! We can only guarantee that it corresponds with our definition for at most three Queues and three Nodes. As Alloy only ever searches a finite scope, it never allows you to prove that your model is correct. However, you can gain a fair amount of confidence that the main errors have been eliminated.

Now that we have seen how to model the static aspects of a system, we will move on to modelling dynamic behaviour and adding operations. For this, we will consider a different example.

1.2.4 Dynamics I

In this second part of the tutorial, we will be modelling a Map that associates unique keys with values (such as eg. `Map` [5] in Java) and show how Alloy can be used to explore dynamic behaviour.

The static aspects are easy to specify:

```
module examples/tutorial/Map

abstract sig Object {}
```

[5] http://java.sun.com/j2se/1.4.2/docs/api/java/util/Map.html

```
sig Key, Value extends Object {}
sig Map { values: Key -> Value }
```

As in Java, **abstract** ensures that there cannot be any direct 'instances' of Object. Similarly, **extends** means that Objects can either be Keys or Values, but not both.

values above is a *relation* that represents the mapping of Keys to Values. This would be written in a schema in Object-Z as `values` $\subseteq K \times V$ where `K` is the set of all Keys and `V` is the set of all Values.
We can visualize this model as before:

```
pred show() {}
run show for 2
```

The first instance produced by Alloy only consists of a Key and a Value but no Map. We could browse through the various solutions until we find one that actually includes a Map, however there is a better way.

As mentioned before, Alloy produces instances that satisfy the predicate passed to **run**. We can add a constraint that specifies that there is at least one Map:

```
pred show() { #Map > 0 }
run show for 2
```

We now get 2 Maps and one Value. To get exactly one Map, we can either change the constraint to:

```
#Map = 1
```

or we can modify the **run** command as follows:

```
run show for 2 but exactly 1 Map
```

Next, let us check that the mapping of keys to values is unique. This can be done in Alloy via *assertions*:

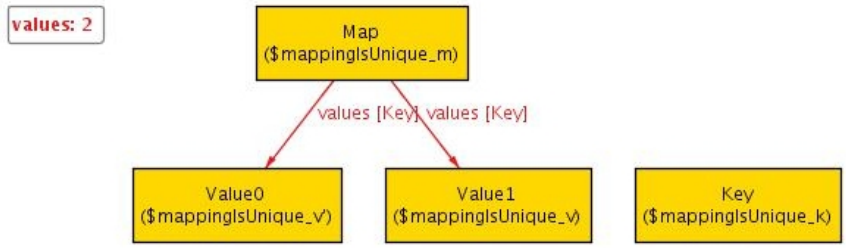
```
assert mappingIsUnique {
        all m:Map,  k:Key, v, v':Value |
                    k -> v in m.values and
                    k -> v' in m.values
                    implies v = v'
}
```

This says that if a Map m maps a key k to two values v and v' then they must be the same. Note how the relational product operator ("->") in k -> v is used to represent the tuple (k, v) and how m.values is treated as a set of tuples (Key, Value).

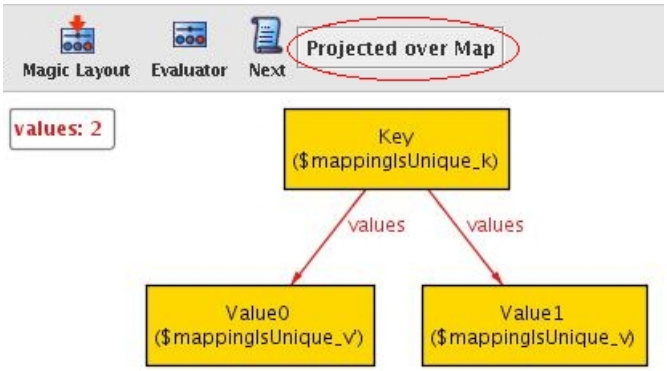
To check an assertion, we can use the command **check**. As for **run**, we have to specify the scope:

```
check mappingIsUnique for 2
```

If we execute this, Alloy searches all possible combinations of at most two objects of each signature for a counterexample that violates the assertion. Since it cannot find one, it tells us that the assertion may be valid. However, if we increase the scope to three, it produces the following:



To make this even more obvious, we can ask Alloy to project over Map using the Projection button.



We see that a key refers to two values. To fix this, we use the **lone** keyword that we have already seen:

```
sig Map { values: Key -> lone Value }
```

Now, the assertion holds even if we increase the scope to 15 (say). This produces a much larger coverage than any testcase ever could and gives us a high level of confidence that the assertion may be valid.

Finally, to make the instances of later steps less verbose, we include the constraint that all keys (and values) belong to some Map:

```
fact {
      all k:Key | some v:Value, m:Map | k -> v in m.values
      all v:Value | some k:Key, m:Map | k -> v in m.values
}
```

Note that, unlike in the Queue example, it makes sense for a key/value to belong to more than one Map. Also observe that the fact above is anonymous; Alloy does not require you to provide a name.

We have now developed the static aspects of the Map. In the next section we will continue to develop the dynamics of our example.

1.2.5 Dynamics II

We now proceed to adding dynamic behaviour to our model. In particular, we define the operation of adding entries to the Map:

```
pred put(m, m':Map, k:Key, v:Value) { m'.values = m.values + k -> v }
```

This is very similar to an operation schema in Object-Z: the instances of `put` consist of the state before(`m`), inputs(`v`), state after(`m'`) and outputs of the operation. As in Object-Z, the convention is to use ' to decorate the state after the operation.

+ is the set union operator. The above constraint therefore reads as:

"The set of values after is equal to the union of the set of values before with the extra mapping from k to v"

(Note how `m.values` is again treated as a set of tuples). We execute this predicate with the usual command:

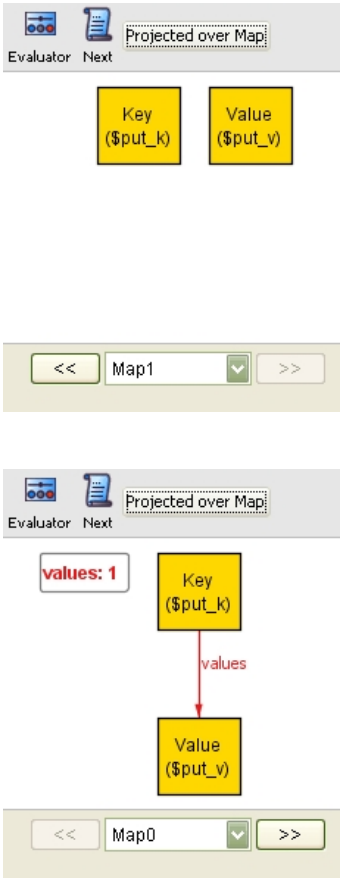
```
run put for 3 but exactly 2 Map, exactly 1 Key, exactly 1 Value
```

Alloy produces the following instance which illustrates the functionality of the put operation:

```
module examples/tutorial/Map
abstract sig Object {}
sig Key, Value extends Object {}

sig  Map {
     values: Key -> lone Value
}
assert mappingIsUnique {
     all m:Map, k:Key, v, v':Value |
          k -> v in m.values
          and k -> v' in m.values
                      implies v = v'
}
pred put(m, m':Map, k:Key, v:Value) {
     m'.values = m.values + k -> v
}

run put for 3 but exactly 2 Map, exactly 1 Key, exactly 1 Value
```



The assertion `putLocal` checks that `put` does not change the existing mappings in the set:

```
assert putLocal { all m, m': Map, k, k': Key, v: Value |
        put[m,m',k,v] and k != k' implies
                lookup[m,k'] = lookup[m',k']
}
fun lookup(m: Map, k: Key): set Value  k.(m.values)
```

`lookup` is an example of a *function* in Alloy. It uses the . operator to return all values that are associated with the Key `k` in the Map `m`, i.e. the set $\{v|(k,v)$ `in m.values`$\}$. Note that the arguments to a function are enclosed in square brackets [ ]. This is new in version 4; in older versions of Alloy, they were enclosed in parentheses (). We check that the assertion holds as before:

```
check putLocal
```

Alloy informs us that no counter-examples were found. This is also the case if we increase the scope (to 15, say). Confident that our assertion may be valid, we stop here.