A Guide To Alloy

Second Year Group Project

Edward Yue Shung Wong, Michael Herrmann, Omar Tayeb
Supervisor: Dr. Krysia Broda

Abstract

Specification languages are a large part of Software Engineering. Learning how to write a rigorous model is a fundamental part of designing good code. This report aims to explore the Alloy specification language, its application through the Alloy Analyzer tool and its potential in teaching. By writing a website [1] aimed at conveying Alloy to our peers, we were able to experience first-hand the intricacies of Alloy. We found that Alloy is a highly intuitive language that lends itself well as an introduction to formal specification languages. The simplistic synthax and the well developed Alloy Analyzer tool makes Alloy highly condusive to quick learning. As a result of this report we hope Alloy will see more widespread use in education as well as encouraging further development of the language itself.

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

Contents

1 A Guide To Alloy **1**
- 1.1 Introduction . . . 1
  - 1.1.1 Philosophy . . . 1
  - 1.1.2 History . . . 2
- 1.2 Tutorial . . . 2
  - 1.2.1 Getting Started . . . 2
  - 1.2.2 Statics I . . . 3
  - 1.2.3 Statics II . . . 5
  - 1.2.4 Dynamics I . . . 7
  - 1.2.5 Dynamics II . . . 10
- 1.3 Alternatives to Alloy . . . 12
  - 1.3.1 Z & Object Z . . . 12
  - 1.3.2 Object Constraint Language in UML (OCL) . . . 12
  - 1.3.3 Vienna Development Method (VDM) . . . 13
- 1.4 Test Yourself! . . . 13
  - 1.4.1 Questions . . . 13
- 1.5 Final Thoughts . . . 17

2 Further Worked Examples **18**
- 2.1 The Seven Bridges of Königsberg . . . 18
  - 2.1.1 The Problem . . . 18
  - 2.1.2 Using Alloy To Show No Solution Exists . . . 19
- 2.2 Modelling Memory in Object-Z and Alloy . . . 20
  - 2.2.1 The Problem . . . 20
  - 2.2.2 The Solution In Object-Z . . . 20
  - 2.2.3 The Solution In Alloy . . . 21

3 Design, Development & Content of Website **22**
- 3.1 Contents . . . 22
  - 3.1.1 Java Web Start Issues and the Flash Demo . . . 23
- 3.2 Website Layout . . . 23
- 3.3 Choice of Colours . . . 24
- 3.4 Using PHP . . . 24

4 Conclusion & Evaluation **25**
- 4.1 Conclusion . . . 25
- 4.2 Personal Experience of the Technologies used . . . 25
  - 4.2.1 HTML . . . 25

4.2.2 JavaScript . . . 25
4.2.3 PHP . . . 25
4.2.4 LaTeX . . . 26
4.2.5 Demo Builder . . . 26
4.2.6 Alloy . . . 26
4.2.7 Alloy as a Teaching Tool in Comparison to Object-Z . . . 26
4.2.8 Outlook . . . 27

A Glossary **28**

B Website File List **29**

Bibliography **30**

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

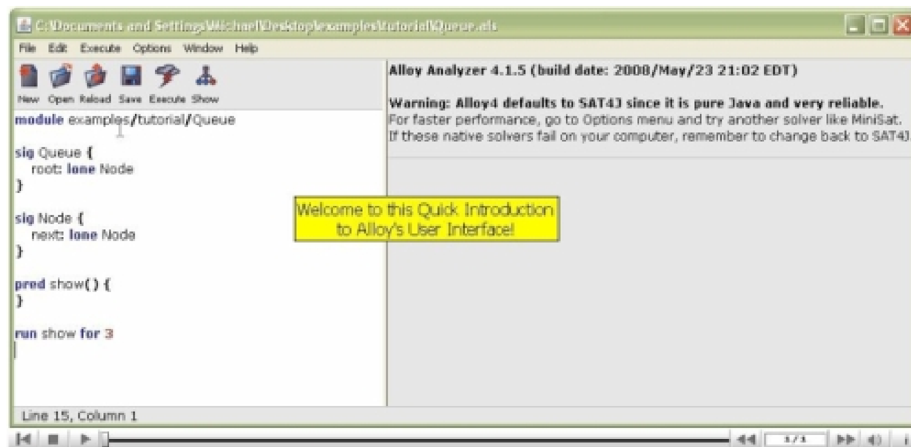


Figure 1.2: Click to launch video!

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() {}
```

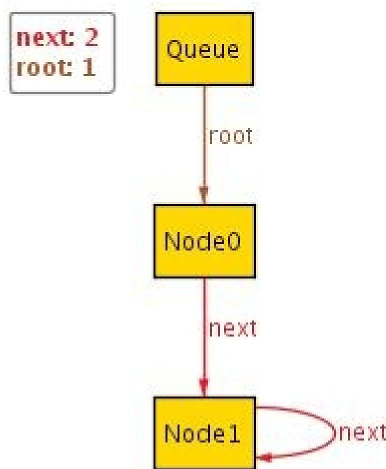
To get sample instances of the predicate we use the command **run**:

```
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
```



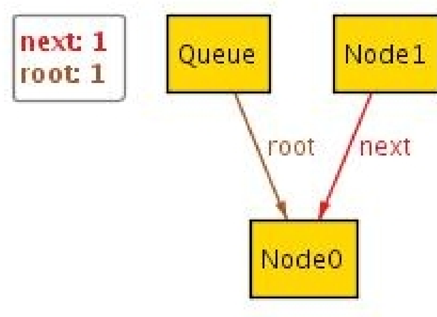
In the above image, we see that Node1 is its own successor. As this is not what we usually expect from a linked list, we add a **fact** to constrain our model:

```
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:

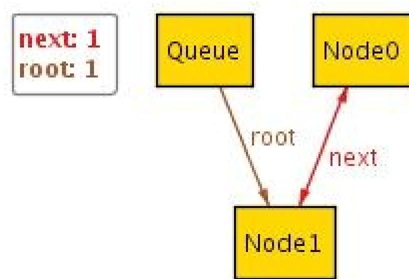
```
q.root,
q.root.next,
q.root.next.next,
...
```

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
```

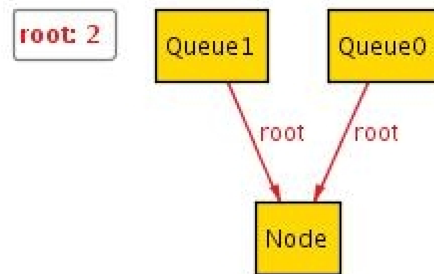
In contrast to the .* operator, .ˆ represents the *non-reflexive transitive closure* which returns the set consisting of all elements:

```
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*:

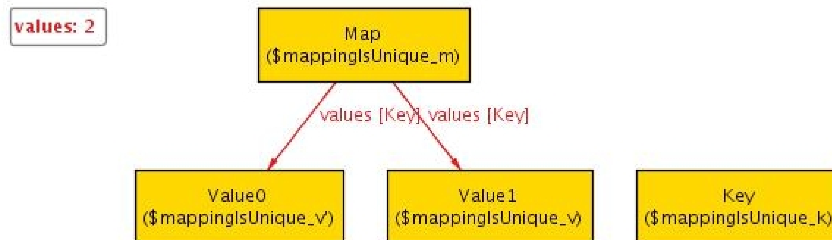
```
assert mappingIsUnique {
        all m:Map,  k:Key, v, v':Value |
                    k -> v in m.values and
                    k -> v' in m.values
                    implies v = v'
}
```

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).

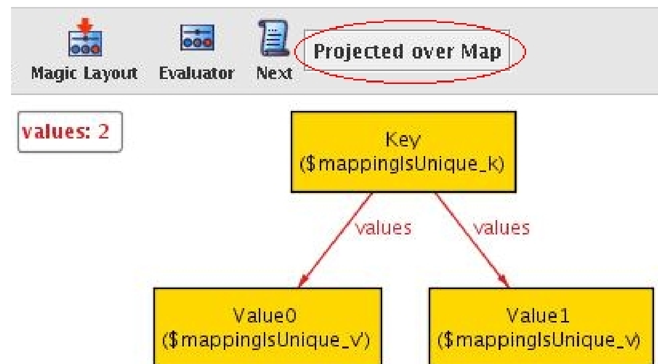
To check an assertion, we can use the command **check**. As for **run**, we have to specify the scope:

```
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.

This concludes the tutorial! If you want to see further examples of Alloy in action, look at section: Further Worked Examples. 2

1.3 Alternatives to Alloy

1.3.1 Z & Object Z

As Alloy was heavily influenced by Z, it is only natural to compare the two.

Z was first developed in 1977[6] by Jean-Raymond Abrial at Oxford University. Being both based on logic and set theory, Alloy closely resembles Z. In fact, Alloy can be viewed as a subset of the Z language.

One of the advantages of Z is that it has a rich mathematical notation making it more expressive than Alloy. However, Alloy is first order making it automatically analyzable, theorem provers in Z are rather more limited. They are automated only up to a point, complex proofs often require guidance from an experienced user.

The distinct style and notation of schema calculus gives it the ability to support many different idioms. Alloy on the other hand has a pure ASCII notation and requires no special typesetting tools. Writing out schemas can be tedious in comparison to typing a signature on Alloy!

Overall, Z has had much more widespread use in education and research. Z has already been applied to several large projects including that on the CICS system by IBM and Oxford University[7]. With continuing development and research being done, perhaps Alloy will one day emulate and even surpass Z.

1.3.2 Object Constraint Language in UML (OCL)

The Object Constraint Language(OCL) is the constraint language of UML. It was developed at IBM and ObjecTime Limited and was added to the UML in 1997. Because it was initially designed to be an annotation language for UML class diagrams, it does not include a textual notation for declarations[8]. Variants of OCL such as USE[9] overcome this limitation.

OCL is based on first-order predicate logic but uses a syntax similar to programming languages and closely related to the syntax of UML. Alloy is similar to OCL, but its creators claim that Alloy has a more conventional syntax and simpler semantics. Alloy is fully declarative, whereas OCL allows mixing declarative and operational elements. The creators of Alloy argue[10] that OCL is too implementation-oriented and therefore not well-suited for conceptual modelling though others believe[11] that experience can help keep the specification process

[6] cf. [1]
[7] cf. [2]
[8] cf. [2] p. 312ff.
[9] http://www.db.informatik.uni-bremen.de/projects/USE/
[10] cf. http://sdg.csail.mit.edu/pubs/1999/omg.pdf
[11] cf. http://www.cs.colostate.edu/ bieman/Pubs/GeorgBiemanFranceUMLWrkshp.PDF

with OCL at an abstract level.

Many tools are available supporting OCL such as Octopus and the Eclipse Model Development Tools. Typical features include the interpretation of OCL constraints over test cases and code generation. Some, such as the USE tool mentioned above, support design-time analysis and exhaustive search over a finite space of cases similar to Alloy.

1.3.3 Vienna Development Method (VDM)

Vienna Development Method (VDM) is a set of techniques for developing computer systems. It originated from IBM's Vienna Laboratory in the mid-1970s and was developed by Cliff Jones and Dines Bjorner. VDM includes a specification language called VDM-SL (VDM specification language).

After the development of VDM in the 1970s there were a few tools for checking VDM specifications, but these were mostly informal projects[12]. In 1988 Peter Froome developed a tool called SpecBox which was the first industrialised tool for checking VDM specification. SpecBox is used in civil nuclear, railway and security applications.

There are many other tools for checking VDM specification such as IFAD VDM-SL and Centaur-VDM environment[13] which is an interactive and graphical tool for VDM. However unlike Alloy these do not provide fully automatic analysis in the style of a model checker. Both Alloy and VDM support object-orientation and concurrency.

Although it is one of the first formal methods in development, it has been refined, standardized and is still widely used in industry by such organisations as British Aerospace Systems & Equipment, Rolls Royce and Dutch Department of Defence[14]. Alloy is not used as much in industry as VDM.

1.4 Test Yourself!

You can use the following four questions to test your knowledge of what has been explained so far. More exercises and answers can be found in the Test Yourself! section of the website[15].

1.4.1 Questions

Sample Question 1 of 4

Look at the following code:

```
abstract sig Fruit {}
sig Apple extends Fruit {}
```

[12] cf. [7] p. 2.
[13] http://www.vienna.cc/evdm.htm
[14] http://www2.imm.dtu.dk/courses/02263/F08/Files/DTU%20VDM%20Technology%20in%20Industry%202008.ppt
[15] http://www.doc.ic.ac.uk/project/2007/271j/g06271j01/Web/test_yourself.php

```
sig Orange extends Fruit {}
sig Pear extends Fruit {}
fact { lone Apple }
fact { one Pear }
pred show() { #Fruit > 0 }
// RUN COMMAND
```

Figure 1.3: Fruit

Which of the following commands produce the above instance?

```
a. run show for 4 but exactly 3 Orange

b. run show for 2 but exactly 0 Apple

c. run show for 3 but 2 Orange

d. run show for 2 but exactly 2 Orange

e. none of the above!
```

Sample Question 2 of 4

Look at the following schema, taken from [3]:

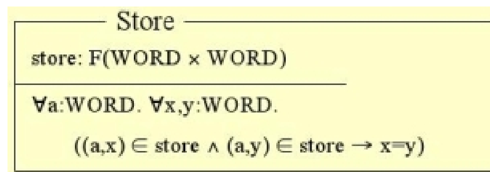


Figure 1.4: Store - Software Engineering I - 2008

Which one of the following is the correct Alloy translation?

a)

```
sig Store  store: Word -> Word
sig Word {}
fact unique {
     all a: Word | all x,y: Word | some s:Store |
     x -> a in s.store and y -> a in s.store
          implies x = y
}
fact singleStore { one Store }
```

b)

```
sig Store { store: Word -> Word }
sig Word {}
fact unique {
     all a,x,y:Word | all s:Store | a -> x in s.store
         and a -> y in s.store
             implies x = y
}
```

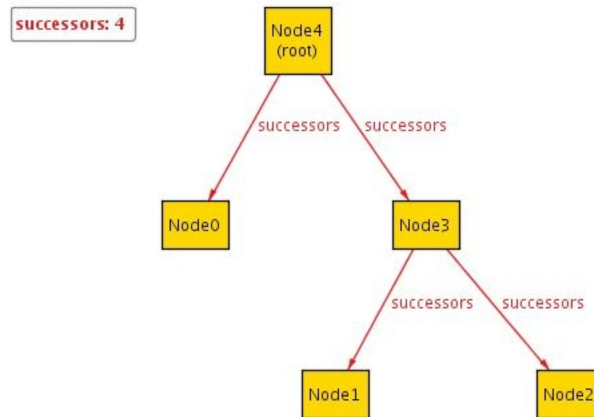
c)

```
sig Store { store: Word -> lone Word }
sig Word {}
fact singleStore { one Store }
```

Sample Question 3 of 4

A tree can be implemented in Alloy as below:

```
sig Tree {
    root: lone Node,
    successors: Node -> Node
}

sig Node {}

fact notCyclic {
    no n:Node | n in n.^(Tree.successors)
}

fact binaryTree {
    all n:Node |
        #n.(Tree.successors) = 2
        or #n.(Tree.successors) = 0
}

fact allNodesBelongToOneTree {
    // ???
}

fact singleMapping {
    all n,x,y:Node |
            x -> n in Tree.successors
            and y -> n in Tree.successors
                     implies x = y
}

pred show() {}
run show for exactly 5 Node, 1 Tree
```



a. Alloy is more expressive than Object-Z

b. Theorem provers in Alloy are not fully automatic and often require guidance from the user

c. In contrast to Alloy, Object-Z allows you to prove that a model is correct.

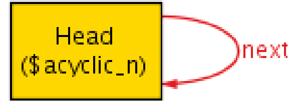
Sample Question 4 of 4

Which of the following is a counterexample to the following assertion[16]?

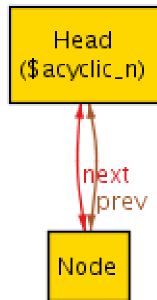
```
sig Node {
    next: lone Node,
    prev: lone Node
}

sig Head extends Node {}

fact exactlyOneHead {
     one Head
}

fact headHasNoPrev {
     no Head.prev
}

fact prevIsInverseOfNext {
     all n:Node |
           n.next.prev = n
}

assert acyclic { no n:Node | n in n.^next }
check acyclic for 5
```

[16]This is an abridged version of an example taken from [4]

a)



b)



c) `None. The assertion is valid`

1.5 Final Thoughts

Building our knowledge of Alloy on top of our existing experience of Object-Z was intuitive. With hindsight, we believe not having prior knowledge of Object Z would not have made learning Alloy much more difficult. The ASCII based synthax makes for easy reading and unambigious models.

Using the Alloy Analyzer was straight-forward and the visualisation feature really helped our understanding. Nothing about the GUI is over-complex and compatibility across Linux, Mac OS and Windows is very good. The fully automated analysis feature is really helpful, instant feedback on your model makes learning more efficient.

The main disadvantage is that Alloy is not used very widely (yet) which can make it hard to find answers to very specific questions. Solutions to most issues are addressed in [2] and addressed by the official (and our!) website! Also, there is a discussion group[17] which is unfortunately closed: you have to ask for moderator approval to be able to access it!

In our opinion, Alloy is a great starting point for beginners to specification languages because of its interactive tool support. We think that we would have preferred learning Alloy over Object-Z.

[17] http://tech.groups.yahoo.com/group/alloy-discuss/

Chapter 2

Further Worked Examples

2.1 The Seven Bridges of Königsberg

In this section, we will show a special case where Alloy can be used to *prove* that a puzzle has no solution.

2.1.1 The Problem

The city of Königsberg, Prussia (now Kaliningrad, Russia) is set on the river Pregel (now Pregoyla). The river divides, creating two large islands which in the eighteenth century were connected to each other and the mainland by seven bridges[9]:

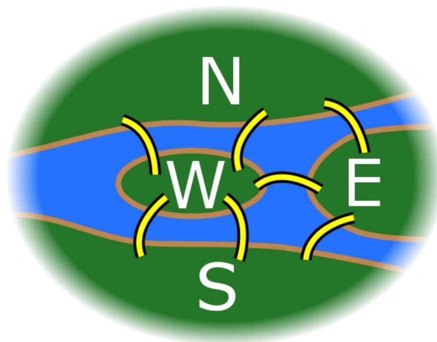


Figure 2.1: The Seven Bridges of Königsberg

It had become a challenge amongst the citizens to see if anyone could walk around the town, crossing each bridge once and only once. Nobody seemed to be able to find such a path and in 1735, Euler proved that the task was impossible. His proof is now considered to be the first theorem of graph theory[10].

[9] Picture taken from http://en.wikipedia.org/wiki/Image:7_bridges.svg
[10] cf. [5] p. 43, [6] p. 2f., http://en.wikipedia.org/wiki/Seven_Bridges_of_Koenigsberg

2.1.2 Using Alloy To Show No Solution Exists

To model the Königsberg bridge problem in Alloy, we first introduce signatures representing the four landmasses as in the picture above.

```
abstract sig Landmass {}
one sig N, E, S, W extends Landmass {}
```

Here, **one** specifies that the signatures N, E, S and W are *singleton*, i.e. that there is exactly one 'instance' of each of them.

For encoding the bridges, we introduce `Bridge`:

```
abstract sig Bridge { connects: set Landmass } { #connects = 2 }
```

The second pair of curly brackets after the field declarations of a signature above is an example of an *appended fact*. This is equivalent to:

```
abstract sig Bridge { connects: set Landmass }
fact { all b:Bridge | #connects = 2 }
```

Note that we have chosen to model the two landmasses that a bridge connects by a set rather than by a relation or two fields of type `Landmass`. This makes it easier to reflect the fact that bridges can be traversed in both directions.

Modelling the actual bridges is straightforward:

```
one sig Bridge1 extends Bridge {} { connects = N + W }
one sig Bridge2 extends Bridge {} { connects = N + W }
one sig Bridge3 extends Bridge {} { connects = N + E }
one sig Bridge4 extends Bridge {} { connects = E + W }
one sig Bridge5 extends Bridge {} { connects = E + S }
one sig Bridge6 extends Bridge {} { connects = S + W }
one sig Bridge7 extends Bridge {} { connects = S + W }
```

Next, we need to be able to represent a `Path` across bridges:

```
sig Path { firstStep: Step }
sig Step {
   from, to: Landmass,
   via: Bridge,
   nextStep: lone Step
} { via.connects = from + to }
fact {
   all curr:Step, next:curr.nextStep |
         next.from = curr.to
}
```

(This might remind you of the `Queue` model in section 1.2.2!) There are two constraints on a path and its steps: First, each step over a bridge has to start at one side of the bridge and end on its other side. Second, the destination of the current step is the start of the next.

Finally, we want Alloy to find a path that crosses all bridges:

```
fun steps (p:Path): set Step {
   p.firstStep.*nextStep
}
pred path() {
   some p:Path | steps[p].via = Bridge
}
```

But how do we ensure that the path crosses each bridge *exactly once*? Well, if each of the seven bridges is crossed exactly once, then the path will consist of seven steps provided each step actually crosses a bridge. As this is implicit from our model (you are encouraged to check this by using an assertion and finding out why!), it is enough to constrain the scope to 7:

```
run path for 7 but exactly 1 Path
```

This shows that the Koenigsberg Bridge Problem has no solution! (Mathematically, the problem can be translated to graph theory by viewing the land masses as nodes and the bridges as edges of a graph. Euler proved the general result that for there to be a path that visits each edge exactly once either all, or all but two vertices must have an even number of vertices connected to them (such a path is now called an *Eulerian path*). As this is not the case for the landmasses and bridges of Königsberg, there cannot be such a path.)

2.2 Modelling Memory in Object-Z and Alloy

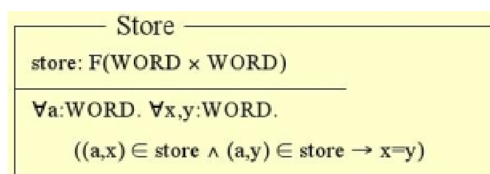
In this section, we will show how Object-Z and Alloy can be used to model the same example[11].

2.2.1 The Problem

We will be modelling a computer store that associates addresses with the values at the respective addresses in the store. To be as general as possible, both addresses and values will be represented as *words*. We will also define a function *lookup* which returns the value associated with a particular address in the store.

2.2.2 The Solution In Object-Z

We have already seen the solution to the first part in section 1.4.1 (p. 14):

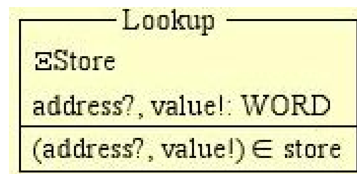


`store` is defined as a finite subset of the cartesian product $WORD \times WORD$ and can thus be seen as a binary relation. The axiom of the schema then further

[11] This example was taken from [3].

constrains it to be a partial function, i.e. it ensures that each address maps to at most one value.

Having defined the state schema for the store, we now define the operation schema for `lookup`:



The variable `address` is followed by `?` to indicate it is an input variable. Similarly, `value` is followed by `!` to indicate it is an output variable. $\Xi Store$ indicates that the state of the store is not affected by the operation. Finally, the single axiom ensures that our function returns the correct value associated with the given address in the store.

2.2.3 The Solution In Alloy

As before, if you have already answered question 2 in section 1.4.1, this might seem familiar:

```
one sig Store { store: Word -> lone Word }
sig Word {}
```

Just as in the state schema above, `store` is a relation mapping each `Word` to at most one other `Word`. Thanks to the simpler syntax of Alloy, the Alloy model is much more concise than the corresponding schemas. There is just one thing that is implicit in the schemas which we have to specify explicitly in the Alloy model: that there is exactly **one** `Store`.

Now on to the lookup function!

```
fun lookup(address:Word) : Word { address.(Store.store) }
```

Here, we specify that `lookup` takes `address` as an argument and returns the value it is associated with in the store.

Notice that Alloy will quietly swallow the error if we give the function a bad input. In Object-Z, we would have to allow for bad inputs by use of preconditions or by handling exceptions.

Chapter 3

Design, Development & Content of Website

As the website was the main presentation tool for our research into Alloy it was important to take design into account. For quick efficient formatting of the website there was a high degree of separation between content and presentation. To ensure this a Cascading Style Sheet (CSS) and PHP were used alongside the HTML. CSS is used to describe the presentation of the HTML. It describes the fonts, colours and layout of certain elements of the website by giving them "class names" which are defined in the CSS of the page. The website ensures universal access by taking into account issues such as colour blindness (by using opposite colours where appropriate) and reduced vision (increasing text size affects the layout minimally).

3.1 Contents

Most of the contents of the website is in the tutorial which goes through a couple of models with the user, encouraging him/her to try them out in Alloy. The tutorial also includes a Flash demo of the Alloy GUI.

There are seven main sections in the website:

1. **Introduction**. This section introduces the user to the history and philosophy of Alloy.
2. **Tutorial**. The tutorial is split into five subsections and includes incrementally developing two models in Alloy. The "Getting Started" subsection includes the demo of the Alloy GUI mentioned above.
3. **Alternatives to Alloy**. This section provides some of the advantages and disadvantages of Alloy in relation to other formal specification languages.
4. **Test Yourself!**. Once the users have finished reading the above sections, they have an opportunity to test their knowledge in this section.
5. **Glossary**. The "Glossary" provides a short reference to some of the Alloy keywords.

6. **FAQs**. This is the frequently asked questions that addresses common problems that users may have.

7. **Final Thoughts**. We have given our opinions on Alloy in this section.

3.1.1 Java Web Start Issues and the Flash Demo

Ideally, we would have wanted to have Alloy running in a frame on the website next to the tutorial. Unfortunately, this would have required embedding Alloy in an Applet and, as there is no official Applet version, digging into and modifying its source code. The next best solution would have been to provide a Java Web Start version so that the user can download and launch Alloy with a single click from the website.

Alloy used to support Java Web Start, however its homepage [12] states that this is no longer the case for the current version (4.1.5). Naively, we tried repackaging the original distribution of Alloy ourself so it could be distributed with Java Web Start. We managed to produce a runnable version that however produced errors whose cause we were unable to determine when trying to execute a model.

Therefore, we finally ended up with what can now be seen on the website: A short explanation of how to obtain Alloy and a video that gives an introduction to its user interface. This way, the reader can also the follow the rest of the tutorial on the website if he does not download Alloy.

3.2 Website Layout

Every page on the website has the main components:

- The title of the website, A Guide to Alloy, which is the same on every page.
- The menu of the left, which is the same on every page except the menu item of the current page is in bold. It shows a clear visual hierarchy of the website and gives users the option to jump to (almost) any page on the website from where ever they are.
- The main content, which includes the main text with images. There are "Next" and "Back" links at the top and bottom of the pages so that the website can be read in order, like a book. The reason for putting these links at the top and bottom is to stop the inconvenience of having to scroll to the top of the page to press "Next" on long pages. This is also convenient for someone wanting to quickly browse through the pages, not having to scroll down. The "Next" and "Back" links were not included in the "Test Youself!" section as this may confuse users as to whether it will take them to the next section of the website (Glossary) or to the next question.

[12] http://alloy.mit.edu/alloy4

3.3 Choice of Colours

For the design of the website we chose a four-colour scheme: Blue, Gold, Grey, White. For the main content we used black and white to ease reading. We used blue and gold for the menu as they are opposite colours so they contrast each other and items will be easily read by colour blind people. In order for the title "A Guide to Alloy" stands out from the menu, white was used for the text. To give an indication to the user where in the website he/she is, the selected menu item is bold.

The default hyperlink colour of blue was kept as this is recognised by most users as text that can be clicked.

For the "Test Yourself!" section it was decided that the explanation for a correct or wrong answer has a green or red background respectively. The reason for this is so that users will get instant feedback about whether or not they have answered correctly and if they want to find out more they would read the explanation of the answer.

3.4 Using PHP

Initially we were going to use frames, splitting the content of the website. This was so that we have one file as a template and have a content frame. Although this approach would have saved us from the problem of long pages (scrolling), it may have caused a few problems for someone browsing the website:

- If someone was to visit the website through a search engine, it may take them to the contents of the frame rather than the whole frame-set, causing navigation through the website difficult.
- When the user wants to bookmark a specific page most browsers will always bookmark the first page (i.e. the homepage).
- Layout may change across different browsers.
- Changing content in more than one frame will change the behavior of the back button.
- Someone wanting to print the page directly from the website will not be able to do so, they would have to copy and paste the text into another file.

We had a few options to get around this problem. One was to use Server Side Includes (SSI) another was to use PHP script, we used the latter as it offered us more flexibility than SSIs.

We created one PHP file which contained the template of the website. It consisted of a layout for the website and the ordering of the pages (used for the menu and the "Next/Back" buttons). Every page included this template file. For the "Test Yourself!" section there is an extra PHP file that is included, this is the questions template. The reason for this is to order the questions and have a consistent layout.

Chapter 4

Conclusion & Evaluation

4.1 Conclusion

We have developed a website that introduces undergraduate students familiar with first-order logic and Object-Z to the structural modeling language Alloy and its Analyzer. In this report, we have given the contents of the website, an additional larger example showing how Alloy can be used to solve puzzles and an overview of the technologies used and decisions made. To conclude our project, we will now give our personal experience of the technologies used throughout the project, an account of Alloy as a teaching tool and an outlook on how the website could be used and expanded upon in the future.

4.2 Personal Experience of the Technologies used

4.2.1 HTML

This was the first language we looked at for writing our webpage in. Whilst it is simple and easy to learn, we found formatting in HTML can be quite difficult. Using CSS improved formatting significantly and made for uniform style across all our pages.

4.2.2 JavaScript

We made limited use of JavaScript to increase accessibility, as some browsers have it disabled. We did find it useful in certain applications such as disabling the Submit button in the "Test Youself" section and to open a new window for the Flash demo. We found JavaScript straight-forward to learn and use however spotting errors is hard because some browsers do not give you feedback when there is a JavaScript error.

4.2.3 PHP

Our knowledge of object oriented programming gave us a good foundation for learning PHP. There is a very large and active PHP developer community online which was helpful when we ran into problems. Learning and using PHP was simple and provided us with a quick and efficient way of organising the website.

4.2.4 LaTeX

LaTeX offers an efficient way of formatting the contents of a document. Having worked with HTML and CSS the principles of Latex were easy to grasp. We found the seperation between content and design enabled us to write the report in a formal fashion seamlessly.

4.2.5 Demo Builder

Having already worked with Adobe Flash, Demo Builder was extremely simple to learn and intuitive to use. It enabled us to create the introductory video to Alloy's user interface on the website in a few hours (that is, from downloading and learning Demo Builder to the completion of the final version ov the video). We believe that creating such a movie cannot be made much easier and highly recommend Demo Builder (its website is http://www.demo-builder.com).

4.2.6 Alloy

As already stated in section 1.5, learning Alloy on top of our existing experience with first-order logic, Object-Z and programming was very easy. This was helped very much by the Alloy Analyzer which is itself intuitive and simple to use. In addition, the Official Alloy tutorial (http://alloy.mit.edu/alloy4/tutorial4) was of great help, as was Daniel Jackson's book Software Abstractions.

4.2.7 Alloy as a Teaching Tool in Comparison to Object-Z

We have seen in section 1.3 that while Alloy may look like it is less mathematical than Object-Z, the two are actually very similar. Like Alloy, Z (and thus Object-Z since it is an extension of Z) is at heart just a logic, augmented with some syntactic constructs to make it easy to describe software abstractions. In Z, schemas are used to encapsulate both declarations and constraints.

In Alloy, declarations are packaged into signatures while constraints are encapsulated in functions, facts and predicates. Schemas can be refined and extended using schema inclusion in Z. The same can be achieved in Alloy using the signature extension mechanism. Class Schemas in Object-Z are a bit more powerful than signatures in Alloy because they provide constructors.

Therefore, even though an Alloy model might look less mathematical than an Object-Z model, in reality the differences between the two languages are minor. For teaching purposes, we thus believe that the distinguishing factor does not lie within their syntax or semantics but in the tool support available.

The most notable tool supporting Z/Object-Z seems to be the Community Z Tools project CZT (http://czt.sourceforge.net/) that can be used standalone on the command line or as a plugin to jEdit/Eclipse where it provides automatic code completion, type checking etc. It comes with a command line, *textual* tool called ZLive that provides an animator for evaluating Z expressions, predicates and schemas and is intended to be used for interactively testing a model. CZT seems to be well suited for people experienced in using Z/Object-Z; however it

is by far not as user friendly and intuitive as the Alloy Analyzer.

Also, in our opinion, the one feature that can help someone learning a modelling language the most, being able to visualize instances of a model, is implemented in the Alloy Analyzer far better than in CZT. Because the underlying theoretical concepts are nearly identical, that is why, from our limited perspective as students, we think that Alloy would be a good alternative to Object-Z for introducing undergraduate students to formal specifications.

4.2.8 Outlook

Alloy offers a lightweight approach to formal specifications which fits well with the ongoing move away from heavyweight methods and the increasing popularity of Agile methodologies in software engineering. It is hard to tell whether it will become used widely; it might not seem rigorous enough to purists and not worth the effort to software engineers. However, we believe that Alloy, even if it does not become an industrial standard, is very suitable for introducing undergraduate students to formal specifications.

In its current state, the website we have developed can be used to introduce students who have a background in Z/Object-Z to Alloy. For example, it might be as a starting point for a student project that uses Alloy. In the future, the website might be enhanced in the following ways:

More General Audience The website currently requires some knowledge of formal specifications. It could be expanded to address a more general audience.

Browser Compatibility The website was only designed for Mozilla's Firefox web browser and can look quite different on other browsers (eg in Microsoft Internet Explorer). This could be fixed.

Conformance with Web Standards Many minor aspects of the website such as specifying "alt"-attributes for images, making links underlined so they stick out to people with color vision deficiencies etc. could be improved(cf. http://www.maxdesign.com.au/presentation/checklist.htm).

Appendix A

Glossary

(+) Set-union operator.

(-) Difference of two set.

(.*) Transitive closure (reflexive).

(.^) Transitive closure (non-reflexive).

all Corresponds to forall.

assertion These are assumptions about the model that you can ask the analyzer to find counter-examples of.

check Command given to Alloy to attempt to find counter examples of an assertion.

fact These are constraints of the model that are assumed to always hold.

function An expression that returns a result.

lone Multiplicity symbol indicating zero or one.

module Similar to package in Java and corresponding to the path to the .als file from the project directory.

one Multiplicity symbol indicating exactly one.

predicate consists of one or more constraints and can be used to represent operations.

run Command to ask Alloy to find instances that satisfy a predicate.

signature A signature in Alloy is similar to the signature of a schema (in Z) in that it defines the vocabulary for the model.

some Multiplicity symbol indicating more than zero (corresponds to there exists).

Appendix B

Website File List

Below is a list of the most important files of the website. A full file list can be found at http://www.doc.ic.ac.uk/project/2007/271j/g06271j01/Web/file_list.php.

Alloy Demo.html HTML for Flash demo of Alloy GUI.

Alloy Demo.swf Flash demo of Alloy GUI.

main.css CSS for website.

aboutus.php Final Thoughts and About us.

alternatives.php Alternatives to Alloy.

faqs.php Frequently asked questions.

glossary.php Glossary.

index.php Homepage.

intro.php Introduction.

question1.php ... question10.php Test Yourself questions.

question_results.php Page displaying the results with comments about score.

questions_template.php Template for the Test Yourself questions.

template.php Main website template.

test_yourself.php Test Yourself section.

tutorial0.php, tutorial1_1.php, tutorial1_2.php... Tutorial pages.

template.php Main website template.

Bibliography

[1] Alloy MIT website, FAQ section: http://alloy.mit.edu/faq.php

[2] Jackson, D. *Software Abstractions: Logic, Language and Analysis*, MIT Press, 2006.

[3] Russo, A. *Lecture Slides of Undergraduate Course Software Engineering I taught at Imperial College London 2007*, http://www.doc.ic.ac.uk/ ar3/lectures/Sed/NewCourseStructure.html

[4] Alloy MIT Online Tutorial http://alloy.mit.edu/tutorial3/maintext-FS-I.html

[5] Du Sautoy, M. *The Music of the Primes: Why an Unsolved Problem in Mathematics Matters*, Harper Perennial, 2004.

[6] Agnarsson, G., Greenlaw, R. *Graph Theory: Modeling, Applications, and Algorithms*, Pearson Education, 2006.

[7] Froome, P. K. D., Monahan, B.Q. & Bloomfield, R. E., *Specbox - a checker for VDM specifications*, In *Proc. Second Int. Conf. on Software Engineering for Real Time Systems*, 1989.