Alloy Specification Language

3EA3 Software Specifications and Correctness

Group 4:

Randa Mohsen

Paul Warnick

Instructor:

Musa Al-hassy

Introduction

Alloy is a popular specification language created in 1997 by Daniel Jackson that can be used for expressing complex constraints over a variety of systems. Alloy uses a declarative approach to modelling, essentially asking the question, “How would I recognize that X has happened?” vs an imperative approach of “How would I make X happen?”. Within this report, we plan to cover the basics of how Alloy can be applied by giving an overview of the core functionalities.

Modelling in Alloy

The idea behind writing models is to fully describe a small portion of a system and ensure that this section can be constrained to a point in which you’re certain there are no issues. By following this approach we are able to create better, more efficient programs, and eventually provide provable correctness to an entire system. In some situations, like safety critical systems, this could result in the difference between life and death.

Alloy has a number of key differences when compared to other modelling languages:

- Finite Scope Check:
    - In order to analyze a model you must give a specific scope to check within. By using this approach, Alloy can guarantee correctness up to the specified scope.
- Infinite Model:
    - The model itself in Alloy is infinite, as in you only describe how the components of a system interact, not how many there are.
- Automatic Analysis:
    - Alloy allows for automatic generation of solutions or counter examples to a specified system.

Core Components

Alloy's syntax has a number of base components that allow for model specification and constraining. These components are very intuitive to implement and understand.

Signatures

A signature is the simplest component in Alloy. They behave similarly to a class in object oriented programming. Each signature can extend another signature to act as its parent, essentially inheriting properties of the extended signature. Signatures often have internal fields to describe attributes about the object itself. The order in which signatures are defined in Alloy doesn't matter. An example of a signature structure would be:

```
/* A signature may refer to another signature not defined until
later in the model, by extending it. */

sig name1 extends name2{
    //most fields look like this:
    field-name-1: field-type-1,

    //multiple fields with the same type look like this:
    field-name-2a, field-name-2b: field-type-2
}
sig name2 extends name2{

    field-name: field-type
}
```

Facts

Facts are constraints placed upon a model in order to limit behaviour. They can be used to help describe what the model is constrained to. In Alloy *all* facts must *always* hold true . An example of a fact could be:

```
/* The name of a fact is not necessary, as it is overlooked by
the analyser, but may be used for better readability. */
fact [name] {
       // constraint1
       // constraint2
       // constraint3
}
```

Over Constraining

A serious problem in declarative modelling is to accidentally constrain your system in a way that eliminates model possibilities which you intended to allow. This can happen easily if you use a large number of fact statements. Fortunately, this can be avoided by wisely using Assert statements where possible.

Assertions

An assert is a statement that claims something *should* be true given a particular scope, instead of actually forcing it to always hold in the system like a fact would. Assertions can be verified with Check statements. An example of the structure of an assert could be:

```
/* The name of an assert statement is necessary, as it is
referenced later by Check statements. */
assert name-of-assertion {
    // constraint1
    // constraint2
    // constraint3
}
```

Fact Vs. Assert

Since assertions and facts are both statements which apply constraints to an alloy model, they are often used in place of each other. This can cause minor or major problems depending on the following scenarios:

- **An assertion which should have been a fact** is something which is not necessarily true, but you, as the designer, are assuming to be true. Sanity constraints fit under this category. If you accidently do this, your model is under constrained, as opposed to being over constrained as described earlier.
- **A true fact which should have been an assertion** is simply redundant and isn't very dangerous to your model. It is implied by other constraints in your model and yet you have explicitly forced it to hold.
- **A false fact which should have been an assertion** is very dangerous. It results in an overconstraint in your model, as described earlier, and you have explicitly forced it to always be true.

Checks

Check statements allow you to verify correctness in Alloy by confirming whether an assertion holds. In the event the assert does not hold, Alloy will automatically search for counterexamples that can be used to determine why the assertions was falsified. As mentioned

above, Alloy works within a finite scope and by using check statements we can specify the size of the scope to analyze. An example of a check statement could be:

```
check name-of-assert for integer
```

Quantifiers

Quantifiers are an essential part of any specification language and allow you to specify the quantity of objects. In Alloy the quantifiers can be utilized in the following format:

quantifier **a** : X { *formula* }

all	All instances of a of type X
some	One or more instance of a of type X
no	Exactly zero instances of a of type X
one	Exactly one instance of a of type X
lone	Either zero or one instance of a of type X

Predicates

A predicate is a statement that always evaluates to true or false. These can be used to further define the behaviour of a model. In Alloy, predicates take in inputs and contain a number of constraints, if the inputs satisfy the constraints within the body then the predicate evaluates to true. An example of a predicates structure could be:

```
pred name [parameter1:domain1, parameter2:domain2] {
    // constraint1
    // constraint2
    // constraint3
}
```

Run

The run command allows a user to generate a sample solution to a model. These can be used in unison with predicates and allow a user to essentially specify the end goal of the model.

```
/* The run statement may take one or more arguments, as seen in
the square brackets. */
run pred-name for integer sig-name [, integer sig-name, integer
sig-name, integer sig-name ]
```

Examples

File System Model

Overview

This problem is very simple and is designed to be understandable by any beginner to Alloy. Its is used as a demonstration of the syntax of the core components described in this report. The File System problem is widely known and has very little constraints:

- Only two types of objects must exist in the file system
    a. Files
    b. Directories
- Files must have one and only one parent object, which must be a directory
- Directories can have zero or one parent object, which must be a directory
- Directories can have zero or many objects as contents
- There must exist one directory that behaves as the root of the file system, which must no parent
- All objects in a file system must be contents of the root

Implementation with Alloy

```
// A file system object in the file system
sig FSObject { parent: lone Dir }

// A directory in the file system
sig Dir extends FSObject { contents: set FSObject }

// A file in the file system
sig File extends FSObject { }

// A directory is the parent of its contents
fact { all d: Dir, o: d.contents | o.parent = d }

// All file system objects are either files or directories
fact { File + Dir = FSObject }

// There exists a root
one sig Root extends Dir { } { no parent }

// File system is connected
fact { FSObject in Root.*contents }

// The contents path is acyclic
assert acyclic { no d: Dir | d in d.^contents }

// Now check it for a scope of 5
check acyclic for 5

// File system has one root
assert oneRoot { one d: Dir | no d.parent }

// Now check it for a scope of 5
check oneRoot for 5

// Every fs object is in at most one directory
assert oneLocation { all o: FSObject | lone d: Dir | o in d.contents }

// Now check it for a scope of 5
check oneLocation for 5
```

Solving the River Crossing Problem

Problem

The River Crossing Problem is a well know puzzle which involves are farmer transporting all of his possessions (a fox, chicken and a bag of grain) across a river to his farm. There are two challenges presented when trying to solve this problem. The first being that in order for the farmer to cross the river he must use a boat which can hold at most himself and one of his possessions. The second challenge is that if left unattended, certain possessions can be eaten, for example the fox will eat the chicken if the farmer is not around, and similarly the chicken will eat the grain. The goal of this puzzle is to ensure the farmer reaches the other side of the river (his farm) with all of his possessions in the minimum number of steps.

Solution

The solution is relatively straightforward and involves:

1. Bringing the chicken over
2. Returning empty handed
3. Bringing the grain over
4. Returning with the chicken
5. Bringing the fox over
6. Returning Empty handed
7. Brining the chicken over

For many people this process won't seem intuitive at first but fortunately by using Alloy to create a model of this problem (seen on the next page), a solution can be found easily. On Top of this, Alloy guarantees correctness so we need not worry about issues with the solution.

Implementation with Alloy

Here we have an implementation of the River Crossing problem as described above. By creating an appropriate specification for our model in Alloy, we're able to use the tool to find a solution in the smallest possible number of steps.

```
/* Impose an ordering on the State. */
open util/ordering[State]

/* Farmer and his possessions are objects. */
abstract sig Object { eats: set Object }
one sig Farmer, Fox, Chicken, Grain extends Object {}

/* Defines what eats what when the farmer is not around. */
fact { eats = Fox->Chicken + Chicken->Grain}

/* Stores the objects at starting and farm side of river. */
sig State { starting, farm: set Object }

/* In the initial state, all objects are on the starting side. */
fact { first.starting = Object && no first.farm }

/* At most one item to move from 'from' to 'to' */
pred crossRiver [from, from', to, to': set Object] {
  one x: from | {
    from' = from - x - Farmer - from'.eats
    to' = to + x + Farmer
  }
}

/* crossRiver transitions between states */
fact {
  all s: State, s': s.next {
    Farmer in s.starting =>
      crossRiver [s.starting, s'.starting, s.farm, s'.farm]
    else
      crossRiver [s.farm, s'.farm, s.starting, s'.starting]
  }
}

/* the farmer moves everything to the farm side of the river. */
run { last.farm = Farmer + Fox + Chicken + Grain } for exactly 8 State
```

An example of a solution generated using Alloy. Here we can see the River Crossing problem represented as a simple diagram with each object in its own shape having its current status in brackets beneath it. Note that the initial state represents the status of all the objects at the beginning of the problem while the final state represents the objects after reaching a solution.