Alloy Primer

Sam Coy

Last Revision: June 13, 2017

Introduction

Alloy is a language and tool for relational modelling, and can be found at `http://alloy.mit.edu`. This document aims to give a brief and accessible overview of Alloy's syntax and semantics.

Relations

A *relation* is a subset of the Cartesian product of a series of sets. For example, given the sets $A = \{1, 2, 3\}$ and $B = \{4, 5\}$: $\emptyset$, $\{(1, 4), (3, 5)\}$, and $\{(1, 4), (2, 5), (3, 4), (3, 5)\}$ are all relations between A and B. The individual elements of a relation, such as $(1, 4)$, are called *tuples*. In Alloy's syntax, a relation between two sets is notated using an arrow (->), for example a relation between A and B is denoted A−>B. A tuple is denoted a−>b. For a single set, a relation is any subset of that set (see below). For more than two sets, the notation just chains, giving A−>B−>C for three sets, and so on.

Alloy represents everything as a relation of some order, which allows for a consistent syntactic approach. As a stylistic point, for this guide, types will start with capital letters, and instances of types will start with a lower case letter. Here is the syntax for relation declarations in Alloy.:

```
boss: Employee -> Employee
empDesk: Employee -> Office -> Desk
empDesk1 = e1 -> o1 -> d1 // specific tuples of a type can be
                          // put in a relation as well, using =
```

In the above examples, *boss* and *empDesk* are relations between the specified sets, however the tuples in the relation not specified (it could be **any** valid set of tuples between those two sets). *empDesk1* is a specific relation (of the same type as *empDesk*), containing one tuple, $(e1, o1, d1)$ in mathematical notation, or e1−>o1−>d1 in Alloy notation. We can specify relations with more than one tuple, using a + to concatenate. For example: rel =a1−>b1 + a2−>b2. Note that **=** is used to define a relation specifically, in terms of tuples, and **:** is used to define the type of a relation.

As unary relations, sets are notated differently - by default instead of a relation of arbitrary size, e:Employee is of size one. The cardinality can be specified as follows:

```
e: set Employee // any number of Employees
e: one Employee // one Employee (note that "one" can be omitted)
e: some Employee // one or more Employees
e: lone Employee // one or zero Employees
e: Employee // one Employee
```

In each of the above examples, the variable *e* represents a certain set of Employee objects.

Also note that similar cardinality restrictions can be applied to relation declaration as well, using A m−>n B syntax, where A and B are types and m and n are cardinality specifiers. If no specifier is given, it defaults to "set". For example:

```
boss: Employee -> lone Employee // each Employee on the left is mapped
                                // to one or zero Employees on the right
mainOffice: Office one -> Desk // only one Office appears on the left
                               // it is mapped to each Desk in a set of Desks
```

Dot Joins

A dot join is a binary operation on two relations in Alloy. It takes two relations where the last type of the first is the same as the first type of the second (the "middle types"), and joins them together, removing the middle types, for instances where the middle terms are equal. For example, a−>b.a−>c evaluates to a−>c, but a−>b1.b2−>c evaluates to none, as the middle terms differ.

Formally, given two relations of type R1:A−>...−>B−>C and R2:C−>D−>...−>E (in that order), a dot join R1.R2 returns a new relation RDOT:A−>...−>B−>D−>...−>E, where a tuple a−>...−>b−>d−>...−>e is in RDOT if r1=a−>...−>b−>c1 is in R1, r2=c2−>d−>...−>e is in R2, and c1=c2. It's important to note that if either R1 or R2 is a unary relation (a set), this is fine; if R2:C for example, then RDOT: A−>B, where a tuple a−>b is in RDOT provided the c's are equal for r1 and r2. For example:

```
rel1 = n1->a1 + n2->a2 + n1->a2
rel2 = a2
rel3 = a1->a3

// Using the above relations:
rel1.rel2 = n2 + n1 // from the second and third tuples of rel1
rel1.rel3 = n1->a3 // from the first tuple of rel1
rel2.rel3 = none // alloy's notation for the empty set
                 // no tuple in rel3 begins with a2
```

It is possible in Alloy to "trim" a relation using dot joins. For example, in the *empDesk* relation from earlier (empDesk: Employee−>Office−>Desk), it is possible to get the relation mapping an Employee to their office by dot joining *empDesk* with Desk. This is because the Desk type is actually a set (unary relation) of all instances of Desk.

From the definition of the dot join earlier, this results in a relation containing all Employee−>Office relations, since the rightmost element for all relations in *empDesk* is **guaranteed** to match an element in Desk, because that is simply all Desks.

Transitive Closure

It is possible to apply a dot join an arbitrary number of times.

```
emp.*boss // the result of emp + emp.boss + emp.boss.boss + ...
emp.^boss // the result of emp.boss + emp.boss.boss + ...
```

The .^ operator represents **transitive closure**, and the .∗ operator represents **reflexive transitive closure**, each of these signifying a dot join being applied any number of times (regular transitive closure excludes zero dot joins however).

Relation Comprehension

Alloy supports relation comprehensions, a way of specifiying a relation based on a predicate.

```
{e:Employee, o:Office  | <pred> } // produces relation of type Employee->Office
```

The syntax for such a comprehension is shown. The comprehension must be enclosed in braces, the type of the relation is declared (with local bound variables), followed by a pipe character, followed by a predicate (which can use the bound variables declared before the pipe).

Logic and Quantifiers

Logic

Alloy has a standard set of logical operators.

- **not/!**
- **and/&&**
- **or/||**
- **implies/=>**
- **iff/<=>**

Both forms (text and symbol) can be used interchangeably.

Set Operations and Cardinality

Alloy supports a range of operations on sets. The binary set operations are as follows and only work if the two sets (or relations) are of the same type. The top three all output a relation, the last two output a boolean value.

- **+** (Union)
- - (Difference)
- **&** (Intersection)
- **in** (Subset)
- **=** (Equality)

The cardinality of a set (or relation) can be found using the **#** operator, and can then be compared to integer literals or the cardinality of other sets. As an aside, any such comparison uses standard mathematical notation, except for $\leq$, where Alloy uses $=<$. An example:

```
rel = n1->a0 + n0->a2 + n0->a0 + n2->a1
rel2 = a0 + a2
rel3 = a0 + a1

#rel = 4 // true
#rel > 5 // false
#rel =< 3 // false
#(rel2 + rel3) = 3 // true
#(rel2 & rel3) = 1 // true
```

Quantifiers

Alloy supports five quanitifers, examples of their use are given below:

```
no e:Employee | <pred> // no Employees satisfy <pred>
lone e:Employee | <pred> // at most one Employee satisfies <pred>
one e:Emplyee | <pred> // exactly one Employee satisfies <pred>
some e:Employee | <pred> // at least one Employee satisfies <pred>
all e:Employee | <pred> // all Employees satisfy <pred>
```

The predicate can be anything that has a boolean value, for example a cardinality check, a set operation that returns a boolean value, or a **pred** (see below).

Language Features of Alloy

There are several important linguistic structures in Alloy to consider. For the purposes of this section, the following type signatures will be used as an example:

```
sig Salary, Office {}
sig Employee {
  boss: lone Employee,
  workplace: Office
}
sig Company {
  emps: set Employee,
  payroll: Employee -> Salary
}
```

Signatures

Signatures are Alloy's way of defining a type. Invoked using the **sig** keyword, the above example contains four of them. Signatures are very similar to structs. They can contain any number of fields, each one being a set or relation in terms of other types.

Looking at the example above, the first signature declaration is as follows:

```
sig Salary, Office {}
```

This is an empty signature declaration declaring both the *Salary* and *Office* types, neither of which contain any fields. Note that multiple types can be declared in the same signature in this way if their fields are exactly the same, and for empty signatures like these, that is common practice.

```
sig Employee {
  boss: lone Employee,
  workplace: Office
}
```

The Employee signature contains two fields: one is the *boss* field, which contains either one or zero Employees and refers to the Employee's boss, and the other is the *workplace* field, referring to exactly one Office. Field declarations must be separated by a comma.

```
sig Company {
  emps: set Employee,
  payroll: Employee -> Salary
}
```

The Office signature also contains two fields: *emps* is a set of Employee instances, and *payroll* is a relation between Employee and Salary.

It is important to note that all fields are stored as relations. For example, the *payroll* field is actually stored as Company—>Employee—>Salary. This means that for a specific Company *c*, its payroll can be accessed using c. payroll , and this is actually a dot join. Similarly, Company.payroll joins all of the payrolls for all Company instances.

Signature Constraints

While a signature sets the fields of a type, it does not by default impose any sensible set of restrictions on what can happen. There is no reason why an employee has to be part of the company to be on the payroll, for example. Some of these problems can be solved using Signature Constraints. For example, to make sure the Employees of the company exactly match the Employees on the payroll, the signature for Company can be modified as follows:

```
sig Company {
  emps: set Employee,
  payroll: Employee -> Salary
}{
  emps = payroll.Salary
}
```

A second pair of braces has been added, inside which is an expression that returns a boolean value. Multiple expressions can be in a signature constraint (although they do not need to be separated by anything other than whitespace), and all of them must be true for any instance of the given type.

Predicates

Predicates, invoked using the **pred** keyword, are best desribed as a boolean function.

```
pred sameOfficeSameComp [e, e':Employee]{
  e.workplace = e'.workplace
  some c:Company | e in c.emps and e' in c.emps
}
```

The predicate *sameOfficeSameComp* returns true if the two Employees passed in as arguments both work in the same office, and share a company. The second expression in the predicate is a line of predicate logic as described earlier, and returns true if there is some company that has both *e* and *e′* on staff.

As with signature constraints, expressions only need to be separated by whitespace, and the predicate only evaluates to true if all expressions evaluate as true. All variables not used in a predicate quantifier must be parameters of the predicate, as *e* and *e′* are in this case.

Running predicates

The primary purpose of Alloy is to execute predicates and find examples of situations where they are true. To do this, the keyword **run** is used.

```
    run sameOfficeSameComp
    run sameOfficeSameComp for 4
    run sameOfficeSameComp for 4 but 1 Salary
    run {}
    run {all c:Company | #c.emps > 3}
```

Above are examples of the run command in use. This tells Alloy to find a situation (a setup of instances, also called a world) where the specified predicate (*sameOfficeSameComp*) is true, and all facts (see below) are true. It is possible to replace a predicate name with an inline predicate (which can be left blank and evaluates as true if it is), as in the fourth and fifth examples. If multiple **run** statements are in a file, Alloy will only execute the first.

Also note the **for** and **but** keywords. These specify the size (scope) of the possible situations that Alloy can show you. for 2, means that there are at most 2 of every type in any world that Alloy shows you. for 4 but 1 Company specifies that at most 4 of every type is in any world, except for Company, where at most 1 is allowed. Some predicates may only be satisfiable with a certain number of instances of a type, and so the scope may need to be increased to find a valid example. However, increasing the scope too much leads to both a very hard to read visualiser, and increased calculation time. By default Alloy runs everything for 3.

While this guide will not focus on the GUI of the Alloy Analyser a lot, it is worth noting that to execute the instruction, simply press the "Execute" button at the top of the UI, followed by the "Show" button to open the visualiser. Any errors will appear in the log on the right hand side.

Facts

Facts, invoked with the **fact** keyword, are zero-paramter predicates that must be true for any world to be valid. They are very similar to signature constraints, and follow the same syntax

```
fact {all e:Employee | e not in e.^boss}
```

This fact specifies that there can be no circularity in the boss relation. Facts can have names, and these are placed in between the **fact** keyword and the open brackets, as with a predicate.

Assertions

Assertions, invoked with the **assert** keyword, are zero-paramter predicates that can be Alloy can check for counter-examples to.

```
assert allEmpsHaveOneJob {
  all e:Employee | lone c:Company | e in c.emps
}
check allEmpsHaveOneJob
```

This assertion checks that all Employees can have at most one job. To check the predicate, the **check** keyword is used as shown, in an identical way to **run** statements.

Functions

Functions, invoked with the **fun** keyword, are best described as a function that returns a relation.

```
fun subordinates[e:Employee]: set Employee{
  {e':Employee | e in e'.^boss}
}
```

This function returns the set of all Employees that are the subordinates of *e* to some degree.

Functions must have names, and must declare a return type for the relation, in this case it returns a set containing an arbitrary number of Employees. Oddly, the function above uses double braces, one is for the function body, the other is for the relation comprehension.

It is best to check whether a function definition has worked by using a predicate to check tuples in it and running the predicate.

Applications

The flexibility of the language allows many different constructs to be modelled with Alloy. In this section

State Changes

Sometimes it is desirable to model an object changing, and it is possible to model such a state changes using predicates. Using the example type definition from earlier:

```
pred addEmployee[c,c':Company, e:Employee]{
  c'.emps = c.emps + e
}
```

This predicate "adds" an Employee *e* to a Company *c*. Except it doesn't quite do this. The *addEmployee* predicate specifies a **before** and **after** state (by convention, named the same as the before state with an appended apostrophe) of a Company. This is because when Alloy executes a command, it simply provides a snapshot of a world that obeys the constraints. As such, assignment does not really exist, as there is no timeframe for it to happen in. The goal of state change predicates is to describe the result of an operation on an object in terms of a seperate object. It is important not to think of objects in a world that Alloy generates as existing simultaneously.

The above predicate has a flaw, in that the *payroll* relation is not specified between the two. This means that the payroll for the two Copmanys could be radically different, despite the fact that that should not change if all that is being added is an Employee. It's also worth noting that the added Employee also needs to be on the left hand side of the payroll relation.

This last problem can be fixed in one of two ways, either the new Employee's Salary is specified in the predicate, or it is not. The first implementation is referred to as deterministic, the latter is nondeterministic.

```
pred addEmployee[c,c':Company, e:Employee]{ // non-deterministic
  c'.emps = c.emps + e
  all emp:Employee, sal:Salary | emp->sal in c.payroll implies emp->sal in c.payroll
  all emp:Employee, sal:Salary | emp->sal in c'.payroll implies (emp->sal in c.payroll or emp=e)
}
pred addEmployee[c,c':Company, e:Employee, s:Salary]{ // deterministic
  c'.emps = c.emps + e
  c'.payroll = c.payroll + e->s
}
```

It is entirely possible that Alloy presents an example where *e* is in *c*, in which case, *c* and *c'* will be the same. It is up to the user to decide whether to explicitly disallow this, and thinking about edge cases such as this is an important skill to develop when using Alloy.

It is also very possible that *c* is an invalid Company in some way. This is not possible if the validity of a Company is specified in **facts** and signature constraints, but if any validity specification is in predicates, then *c* may not be a valid Company. It is good practice to write predicates in Alloy as **honest operations**, that is writing the predicate on the assumption that the before state isvalid, and given that information, be guaranteed to specify an output state that is also valid.

Time Modelling

It is possible to model the passage of time in Alloy in several ways. Being a flexible tool, Alloy does not explicitly provide a way to do this, and given the resources provided, there are many possible iplementations of time. In this section, the preferred method will be considering modelling an object as several distinct objects, one for each time "point", and using the util/ordering module to order them.

The util/ordering module can be imported and applied as follows:

```
open util/ordering[Company]
```

This opens the ordering module and applies it to Company. In order to "traverse" the ordering, the module provides several functions:

- **first[]** - returns the first item in the ordering
- **last[]** - returns the last item in the ordering
- **next[x]** - returns the item after x in the ordering
- **prev[x]** - returns the item before x in the ordering
- **nexts[x]** - returns the set of items after x in the ordering
- **prevs[x]** - returns the set of items before x in the ordering

Now the Company instances are ordered, it is possible to specify (in a fact) that transitions between a Company at one time to the next must be valid.

```
fact {
  valid[first[]]
  all c:Company - last[]
    | let c' = next[c]
    | one e:Employee
    | addEmployee[c,c',e]
}
```

A few things need explaining here. The *valid[]* predicate should be some user defined predicate that returns true iff the Company is a valid instance. This does not necessarily have to exist, the Company could need to be valid through **facts** anyway.

The **let** keyword has not been shown elsewhere in this primer, as it is only useful in this context. **let** allows you to alias a variable within a certain scope. In this case, *next[c]* has been aliased to *c'*. The fact as a whole states that each Company except the last (because there is no next Company), must have a valid transition between itself and the next Company. A reminder that while these are different objects, they are being used to represent the same object at different times.

Acknowledgments

Thanks go to Russell Boyatt, the author of an Alloy syntax guide (available internally to Warwick students), and Jane Sinclair, who wrote the materials for and lectured the Logic and Verification module at Warwick (CS262).