Alloy

Readings: Section 2.7.

In this module, we look at Alloy, an analyzer that uses a simple structural modelling language based on first-order logic. Alloy is a free download for all major platforms from `alloy.mit.edu`, and we encourage you to download it and experiment with it. The textbook uses the syntax from an earlier version (2.0), but we will be using 3.0 syntax in our examples.

Specification and modelling

We have seen that it is decidable whether $\mathcal{M} \models \phi$ for a finite model $\mathcal{M}$, and typically the structures we are working with in computer science are finite. However, if ϕ describes some property of a software design, then this may be too restrictive, as we are committing to a particular model.

A better option might be to describe the specification of our design by a set of formulas Γ, and then show that $\Gamma \models \phi$. However, we know that this is undecidable.

Alloy attempts to attain the advantages of both these approaches.

It does so by restricting the size of models under consideration to some small finite number. Semantic entailment thus becomes decidable.

The language of Alloy combines elements of predicate logic, mathematics, and computer science. It offers a fairly natural way to specify a system. That being done, there are two ways it can be used: we can make a logical assertion and ask Alloy to find a counterexample of bounded size, or we can ask Alloy to verify that the assertion holds in all models of bounded size. This facilitates interactive refinement of a specification.

Signatures

An Alloy file starts with a `module` statement. We will not be talking about situations where multiple modules are used, so this is just a bit of necessary naming syntax.

Next come a number of **signatures**, or structured types. These are reminiscent of structures in imperative languages, or class definitions in object-oriented languages. However, it is possible for a signature to have no internal structure.

As our first example, we will revisit the “None of Alma’s lovers’ lovers love her” situation from lecture module 07, which the text also reuses in section 2.7.

```
module AboutAlma
sig Person {}
sig SoapOpera {
  cast : set Person,
  alma : cast,
  loves : cast -> cast
}
```

Here, `Person` is a type with no internal structure, and `SoapOpera` contains what appear to be three fields. One has name `cast` and represents a set whose elements are of type `Person`; one has name `alma`, and is an element of `cast`; and one is clearly intended to represent the "loves" relation among the cast of characters of the soap opera.

Interpretations of Alloy signatures

The most obvious interpretation is the object-oriented one, though already the declaration `alma:cast` seems to violate the typing rules we know from programming languages. The OO interpretation can help when trying to understand code, but in order to write it and to understand error messages, we have to go deeper.

The next level of interpretation is set theory. If we had just written `cast:Person`, then `cast` would have referred to a single `Person`. It is better to think of this as a set of size one. That is, on this level, everything is a set, though the default is a singleton set. `set` is a **multiplicity keyword**; we will see others.

But how do we interpret `loves:cast->cast`?

`loves` is a **relation**, a set of ordered pairs. The notation `->` seems to imply a function, and in fact a function f can be represented as a set of ordered pairs $\{(x, f(x))\}$. However, we do not want `loves` to be a function in this case, but a relation, because one person may love several people.

If we had wanted `loves` to be a function, we could have written `loves:cast->one cast`, using the multiplicity keyword `one`. We can unify the idea of sets and relations by making everything a relation. A set is a unary relation (it consists of tuples of length one).

Now that we have introduced the syntax of signatures, we can demonstrate how to write an assertion such as "None of Alma's lovers' lovers love her."

```
assert OfLovers {
  all S : SoapOpera |
    all x : S.cast |
         not (S.alma in x.(S.loves) &&
              S.alma in x.(S.loves).(S.loves))
}
```

The $\forall$ quantification is expressed by `all`, with variables being typed, and the vertical bar | represents "such that". It is easy to understand `S.cast` and `S.alma`, using the object-oriented interpretation, but the rest is better done using relations.

The dot operator is best interpreted as relational join. The definition of relational join is that if we have relations `A` and `B`, then the join `A.B` consists of all tuples $(a_0, a_1, \ldots, a_{n-1}, b_1, b_2, \ldots b_m)$, where there are tuples $(a_0, a_1, \ldots a_n)$ and $(b_0, b_1, \ldots b_m)$ such that $a_n = b_0$. In other words, where we see a tuple from `A` whose last element matches the first element of a tuple from `B`, we take out the matching element and join the pieces together, and do this in all possible ways.

Join is one of the important operations in the relational model of databases, as discussed at length in CS 348.

So how does this view of the dot operator explain `x.(S.loves)`?

`S.loves` is a binary relation, and `x` is a set, which we said could be interpreted as a unary relation. So the join `x.(S.loves)` consists of all tuples in `S.loves` with the first element being `x`, and the join removes that element. In other words, `x.(S.loves)` is the unary relation of elements that `x` loves, or the set of cast members that `x` loves.

Similarly, `x.(S.loves).(S.loves)` is the set of cast members loved by someone that `x` loves. (The `with` keyword used in the text is not present in Alloy 3.0.)


```
assert OfLovers {
  all S : SoapOpera |
    all x : S.cast |
         not (S.alma in x.(S.loves) &&
              S.alma in x.(S.loves).(S.loves))
}
```

The subrelation operator `in` will work for both subset and "element of", since an element is viewed as a set of size one. We also see the logical operators `&&` (`and` will also work) and `not` (`!` will also work). Alloy also provides `or`, `||`, `implies`, `->`, `iff`, and `<->`.

Our translation of "None of Alma's lovers' lovers love her":

```
all x : S.cast |
     not (S.alma in x.(S.loves) &&
          S.alma in x.(S.loves).(S.loves))
```

thus is saying that for any cast member `x`, it is not the case that both of the following are true: Alma is loved by someone that `x` loves (that is, `x` is one of Alma's lovers' lovers) and that `x` loves Alma. Contrast this with our translation in predicate logic:

$$\forall x \forall y (L(x, a) \wedge L(y, x) \rightarrow \neg L(y, a)).$$

An assertion is something we are claiming about the model, so we can check it. We do this by the `check` command, which looks for small models. If we say

```
check OfLovers for 3 but 1 SoapOpera
```

then we are asking for models which contain up to three instances of all signatures (only `Person` in this case) but only one instance of `SoapOpera`. At this point, we have a complete example that we can run in the Alloy Analyzer. This will either report that all models up to the given size satisfy the assertion, or it will report a counterexample. Running it shows that Alma loving herself is a counterexample.

To take care of this situation, we add a clause that expresses the idea that "loves" is not reflexive.

```
fact NoSelfLove {
  all S : SoapOpera, p : S.cast |
     not p in p.(S.loves)
  }
```

This fact is required to be true in all models, as distinguished from an assertion, which is tested in all models. If we run the analyzer again, it will find a larger counterexample in which no one loves themselves.

Using facts requires some care. If we add two contradictory facts, then no model will satisfy both. In this case, our model is overconstrained, and a search for counterexamples will come up empty-handed. This looks good, but doesn't really say anything. Alloy also has a way of searching for satisfying models, as we will see.

If we don't add enough facts, on the other hand, our model is unconstrained, and Alloy may well find a silly or absurd example, such as the one where Alma is alone in the model loving herself. In this case, the absurdity of the model suggests constraints to add.

Alloy used the operator `in` for both "element of" and "subset of", because no expressiveness is lost by doing so. We know that with just "element of" and logical operators, we can simulate all familiar set-theoretic connectives. However, some of these simulations are not very expressive, so Alloy includes set operators.

The infix operators `&` and `+` express intersection and union, and `=` is set equality. There are also predicates for empty relation (`no`) and nonempty relation (`some`), as well as singleton relation (`one`) and at-most-singleton (`lone`, which can be read as "less than or equal to one"). These can also be used as quantifiers.

This suggests a simpler way of expressing our soap-opera assertion. As before, we quantify over `all S:SoapOpera`.

The set of Alma's lovers is `(S.loves).(S.alma)`.

The set of Alma's lovers' lovers is `(S.loves).(S.loves).(S.alma)`.

We wish to assert that the intersection of these two sets is empty.

```
assert OfLovers {
  all S : SoapOpera |
    no (S.loves).(S.alma) &
       (S.loves).(S.loves).(S.alma)
}
```

There are often many ways to express the same idea in Alloy.

How Alloy works

The basic idea behind Alloy is simple. If everything is a relation, then the variable involved in a quantification also represents a relation. Because the scope of the universe of elements is limited, we can create a Boolean variable $x_{i,j}$ for every pair of elements (i, j) that expresses whether or not (i, j) is in this relation (suppose it is binary). Then the formula being quantified over can be translated into a propositional formula on these variables.

In this fashion, the possibly higher-order statements of an Alloy module are translated into a single propositional formula (which in general is quite large) and then fed to a SAT solver. Alloy does a considerable amount of optimizing before this, and offers a choice of SAT solvers, including ones that can enumerate solutions.

Assignment 4 example

On assignment 4, question 3 gave you three formulas representing the reflexive, symmetric, and transitive properties of a binary relation, and asked you to come up with models in which two were true but the third was not.

$$
\begin{aligned}
\phi_1 &= \forall x P(x,x) \\
\phi_2 &= \forall x \forall y (P(x,y) \rightarrow P(y,x)) \\
\phi_3 &= \forall x \forall y \forall z ((P(x,y) \wedge P(y,z) \rightarrow P(x,z))
\end{aligned}
$$

Had you known about Alloy, you could have used it to find solutions.


```
module A4Q3a
sig Elts {}
sig RelEx {
  S: set Elts,
  P: S->S
} {all x: S | x in x.P
   all x,y: S | y in x.P implies x in y.P}
```

Here we see the use of a **signature fact**, which itself uses braces to imply logical `and`s between statements. The signature fact ensures reflexivity and symmetry. We can now add an assertion describing transitivity, and ask the Alloy analyzer to find a counterexample.

```
assert trans {
  all R: RelEx |
    let P = R.P |
      all x,y,z: R.S |
        (y in x.P and z in y.P) implies z in x.P
}
check trans for 5 but 1 RelEx
```

This finds the counterexample given in the model solutions.

Because there is no real need for types here – we are asserting something about abstract variables and predicates – we can clean up the code a bit by making use of the universal type, denoted `univ`. Every other type is a subtype of the universal type; it plays the same role as the root of an object-oriented class hierarchy (e.g. `Object` in Java).

We also introduce the notion of a predicate or `pred`. This has parameters, and when it is `run`, it looks for arguments which satisfy the assertions in the body of the predicate – in this case, two assertions describing reflexivity and symmetry, and one describing a counterexample to transitivity. The complete code is on the next slide.


```
module A4Q3a

pred RSnotT (P: univ->univ) {
  all x,y: univ  {
    x in x.P
    y in x.P implies x in y.P
    }
  some x,y,z: univ |
    not ((y in x.P and z in y.P) implies z in x.P)
  }

run RSnotT for 3
```


Group theory example

We can use Alloy in the context of group theory, which we examined in module 08 and on assignment 5.

```
module GroupTheory
sig Elt {}
sig Group {
  elts : some Elt,
  id : one elts,
  op : elts -> elts -> elts,
  inv: elts -> one elts
}
```

Here we have used a ternary relation to represent the group operation, because it has two arguments and one value.

But which value should go in which position? Our instinct is to place the arguments in positions one and two. We can use the keyword `one` to ensure that the operation is really a function.

```
op: elts -> elts -> one elts
```

Unfortunately, this leads to some awkwardness in applying the operation. To apply it to `x` and `y`, we would have to say `x.(y.(G.op))`. It gets even worse:

```
fact Associative {
  all G: Group, x,y,z: Elt |
    (x.(y.(G.op))).(z.(G.op))
     = x.((y.(z.(G.op))).(G.op))
}
```


Alloy provides the alternate syntax `a[b]` for `b.a`, which helps somewhat, but not enough.

```
fact RightIdentity {
  all G: Group, x: Elt |
    (G.op[x])[G.id] = x
  }
fact Associative {
  all G: Group, x,y,z: Elt |
    (G.op[(G.op[x])[y]])[z]
     = (G.op[x])[(G.op[y])[z]]
  }
```


A better solution is to define the ternary relation `op` so that the first and third elements in a tuple are x and y, and the middle element is $x \circ y$. This allows more natural-looking statements.

```
sig Group {
  elts : some Elt,
  id : one elts,
  op : elts -> elts -> elts,
  inv: elts -> one elts
}
fact OpIsFunction {
  all G: Group, x,y: Elt |
    one (x.(G.op).y)
  }
```



```
fact Associative {
  all G: Group, x,y,z: Elt |
    (x.(G.op).y).(G.op).z = x.(G.op).(y.(G.op).z)
  }

fact RightIdentity {
  all G: Group, x: Elt |
    x.(G.op).(G.id) = x
  }

fact RightInverse {
  all G: Group, x: Elt |
    x.(G.op).(G.inv[x]) = G.id
  }
```

Using these facts, we can code and check assertions about groups (both of these have counterexamples).

```
assert OwnInverse {
  all G: Group, x: Elt |
    x = G.inv[x]
  }
check OwnInverse for exactly 3 Elt, 1 Group
assert Commutative {
  all G: Group, x,y: G.elts |
    x.(G.op).y = y.(G.op).x
  }
check Commutative for 6 but 1 Group
```


File system example

We will examine an Alloy description of a file system as a means of introducing yet more syntax, and demonstrating ideas of refinement. (This example is adapted from the tutorial available on the Alloy web site.)

```
abstract sig FSObject {}

sig Dir, File extends FSObject {}
```

Here we see more OO syntax: the use of `abstract` to indicate a type that is the union of its extensions, and the use of `extends` to define two disjoint subtypes. Next, we define a file system signature.


```
sig FileSystem {
  objects: set FSObject,
  root: Dir & objects,
  contents: (Dir & objects) one-> (objects - root),
  parent:  (objects - root) ->one (Dir & objects)
}
```

Note the use of set-theoretic operations to constrain the defined fields. The root must be both a `Dir` and an object in the file system. Every directory has contents which cannot include the root, and the multiplicity keyword `one` ensures that everything except the root is in exactly one directory. We will add signature facts to further constrain `FileSystem`.


```
{ objects = root.*contents
  parent = ~contents }
```

The `*` operator performs reflexive-transitive closure on the relation which follows it, so `root.*contents` is the set of all elements reachable by applying the `contents` relation to `root` zero or more times. Thus the first formula says that every object is reachable from the root. Finally, the `~` operator is tuple reversal, so the last formula says that (x, y) is in `parent` if and only if (y, x) is in `contents`, or y is the parent of x iff x is contained in y.

Many of these “sanity checks” we could have discovered by generating examples using the analyzer.

A file system is not static, but dynamic. We might wish to model such dynamic behaviour, but it seems to go against the declarative nature of the language.

This is where defining something like `FileSystem` as a type pays off. So far, we've really only needed one of our biggest type (and in the group theory example, we did without types at all). Now we can talk about two instances of `FileSystem`, "before" and "after" a given operation. We expect these to differ only slightly, and we can describe that difference.

As an example, consider a move or `mv` command, which moves a given file to a given directory. The corresponding predicate will have four parameters, adding in the "before" and "after" filesystems.


```
pred mv (fs, fs': FileSystem,
         f: FSObject, d: Dir) {
...
}
```

Our task is to replace the `...` with real code which asserts the properties we want. The first property is that the file and directory involved in the move should actually be part of the "before" filesystem.

```
(f + d) in fs.objects
```

The precise description of the move is a little more complicated.

We want the `contents` relation of `fs'` to be a slight modification of that of `fs`. What has changed? `f` has been removed from its parent directory and added to the directory `d`. We must therefore remove the tuple `f.(fs.parent)->f` and insert the tuple `d->f`.

```
fs'.contents = fs.contents
               - f.(fs.parent)->f
               + d->f
```

These two formulas in the body of `pred mv` suffice to describe a move.

If we attempt to use the `run` command to generate an example, we get a rather trivial one, namely `f` added to the directory `d` in which it already is located. We can add another formula to `pred mv` stating `not d = fs.parent[f]`, and this gives us a suitable example.

Deletion is a little trickier. We will code two kinds of deletion: removing a file or empty directory, as in the Unix commands `rm` and `rmdir`, and recursively removing a directory and all subdirectories and files, as in the Unix command `rm -r`.

To delete a file, we have to make sure that it is actually in the "before" file system, and that the contents relation of the "after" file system does not contain the tuple consisting of the file's parent and the file.

```
pred rm (fs, fs': FileSystem, f: File) {
  f in fs.objects
  fs'.contents = fs.contents - fs.parent[f]->f
}
```


`rmdir` is similar to `rm`, but we must make sure that the directory is empty, and not the root.

```
pred rmdir(fs, fs': FileSystem, d: Dir) {
  d in fs.(objects - root)
  no d.(fs.contents)
  fs'.contents = fs.contents - d.(fs.parent)->d
}
```

Running this gives us examples, so we know the model is consistent. But does it reflect the behaviour of a file system that we expect? We need to write and check some assertions.

The first assertion we will write states that a move operation does not change the set of objects in the file system.

```
assert moveAddsRemovesNone {
  all fs, fs': FileSystem, f: FSObject, d:Dir |
    mv(fs, fs', f, d) => fs.objects = fs'.objects
}
check moveAddsRemovesNone for 5
```

This does not find any counterexamples. So far, so good.

Our next assertion states that `rm` removes exactly the specified file.

```
assert rmRemovesOneFile {
  all fs, fs': FileSystem, f: File |
    rm(fs, fs', f) => fs.objects - f = fs'.objects
}
```

This check fails with a scope of 3. The counterexample shows `fs` with a single root directory containing a single file. But `fs'` has a different root directory which is empty. How is this possible?

Looking at the formulas in `rm`:

```
pred rm (fs, fs': FileSystem, f: File) {
  f in fs.objects
  fs'.contents = fs.contents - fs.parent[f]->f
}
```

we see that the contents relation of `fs'` is a subset of `fs`, and that the general file system fact `objects = root.*contents` constrains any contents to be reachable from the root. But in the counterexample, the contents relation of `fs'` is empty, meaning its root can be anything. If we add to `rm` and `rmdir` the formula `fs'.root=fs.root`, then no counterexample is found up to scope 5.

Finally, we can write `rm_r` in a similar fashion, using the `*` operator to construct the set of descendants of the object `f` to be removed, and then taking out of the `contents` relation all parent-child tuples with a child in this set.

```
pred rm_r(fs, fs': FileSystem, f: FSObject) {
  f in fs.(objects - root)
  fs'.root = fs.root
  let subtree = f.*(fs.contents) |
    fs'.contents = fs.contents
                     - fs.parent[subtree]->subtree
}
```


Other features of Alloy

We will briefly sketch some other features. We could add a total ordering to any type by writing out the axioms of total ordering as facts or assertions constraining that type. Alloy allows us to put these into a separate module, and then invoke that module by providing it a type as a parameter. The ordering module is just one of several utility modules included in the Alloy distribution.

One use of ordering is to discuss the behaviour of a sequence of states, rather than just before and after. The Alloy tutorial mentioned earlier continues with a classic example of a river-crossing puzzle involving a fox, a chicken, and a sack of grain. Alloy can also be used to solve other logical puzzles encountered in recreational mathematics.

Goals of this module

Because we have not asked any assignment questions on Alloy, we will not be asking questions about it on the final exam. As with the deep background material discussed earlier, we believe that attention to this material will pay off in deeper understanding and integration of course concepts. An hour or two spent exploring the Alloy Analyzer with the examples we have used, or the ones included with the distribution, will be time well spent.

The School of Computer Science will likely mount an upper-year course in model checking and program verification in the near future, and this is good preparation, though such a course will not explicitly require knowledge of Alloy.