CITS5501 Software Testing and Quality Assurance Specifications in Alloy

Unit coordinator: Arran Stewart

May 15, 2018

Sources

- Pressman, R., *Software Engineering: A Practitioner's Approach*, McGraw-Hill, 2005
- Huth and Ryan, *Logic in Computer Science*
- Pierce et al, *Software Foundations vol 1*
- Alloy tutorial at http://alloytools.org
- Jackson, *Software Abstractions*, 2006, MIT Press.

Using Alloy

- We've seen how you can express signatures in Alloy, which represent entities of some sort.
- We'll now explore some of the other constructs Alloy permits, and how they can be used to explore and check a model.

Language constructs

- *signatures*. We have seen these – they permit entities, and their relationships to other entities, to be declared.
- *functions*. Functions define a way of getting a set or relation given particular parameters.
- *predicates*. Predicates define formulas which evaluate to true or false, and can take parameters which are used in determining the result.
- *facts*. We have seen these – they act as *constraints*. They are deemed to be always true.
- *modules*. Similar to programming languages, we can organize Alloy specifications into modules.
- *assertions*. These also evaluate to true or false; but rather than being facts we declare to be true, they are things we would like Alloy to *check*.

“Running” models

- We can *check* assertions – we ask Alloy to create instance of our model (up to a certain size), and look for counterexamples to our assertions.
- We can also find instances that satisfy *predicates*.

Example

- Consider the following specification:

```
sig Name, Addr {}
sig Book {
  addr: Name -> lone Addr
}
```

- If we'd like Alloy to generate some instances of this, we do so by asking it to generate instances that satisfy a *predicate*.

Example – predicates

Here are some sample predicates:

- A predicate that takes no arguments, and is true if `2 < 3`:

```
pred myPred() {
  2 < 3
}
```

- A predicate that takes one argument, `a`, and is true if `a < 3`:

```
pred myPred(a : Int) {
  a < 3
}
```

Example – predicates

- If we have *no* formula inside a predicate, the body is assumed to just be "true":

```
pred myPred() {
}
```

Running predicates

- Let's ask Alloy to generate some instances. Initially, we won't put any constraints on the instances, so we'll just use an empty (always true) predicate:

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

- We run it like so:

```
run show for 3
```

  The "3" is a *scope* size – how large a state space Alloy should explore. In this case, at most 3 objects of each signature will be created.

Running predicates

- For our book example, let's limit the scope to just one Book, like this:

```
run show for 3 but 1 Book
```

  We'll create at most 3 objects, *except* for Book, which we'll only create 1 of.

Running predicates

- Often, we'll want to examine particular sorts of instance of our model.
- Alloy found us a basic instance that had a link from a single name to an address;
  let's try and find instance with more than one name.

```
pred show (b : Book) {
  #b.addr > 1
}
```

- This says we want more than one link, effectively. (# is the "cardinality" operator; it gives us the size of a set.)
- The Alloy GUI provides a range of options for visualizing the results.

Consistency

- Can we have one name linking to more than one address?

```
pred show (b: Book) {
  #b.addr > 1
  some n: Name | #n.(b.addr) > 1
  }
```

- We'll explore the second line a bit.

Function application

- Function application is where you apply a function to arguments: e.g.

```
Math.abs( -3.0 );
```

- Alloy doesn't exactly have function *application*.
- Rather, it uses *joins*, which are more like the way a database operates.

Function application

- Functions are a special sort of relation, in which something on the "left" uniquely maps to something on the "right".
- Imagine we have a relation between pets and owners (where pets can only have one owner):

```
(Fido,  Alice)
(Coco,  Bob)
(Rex,   Alice)
```

- This is a function, because no pet appears twice in the relation.

Function application

- If we also have a set of pets:

  ```
  pets = { Fido, Coco, Rex, Blackie }
  ```

  then we can construct the *join* of the set "pets", and our relation (call it "pet_owner"):

  ```
  pets.pet_owner = {
    (Fido,  Alice)
    (Coco,  Bob)
    (Rex,   Alice)
  }
  ```

- Some pets apparently don't have an owner, so this is a partial function; those pets don't end up in the joined relation.

Function application

- If we had multiple functions, we can join them one after another.
- Suppose we have a mapping from pets to owners, and from owners to mothers:

```
pets.pet_owner.mother
```

- If `s` is a set and `r` is a function, then `s.r` is the result of joining `r` to `s`; it's the *image* of the set `s` going forward through `r`

Consistency

- So, can we have one name linking to more than one address?

```
pred show (b: Book) {
  #b.addr > 1
  some n: Name | #n.(b.addr) > 1
}
```

- The second line asserts that there exist some (one or more) names, such that (in normal notation) the size of `b.addr(n)` is greater than 1.
- Alloy tells us that nothing satisfies this predicate (unsurprisingly, because of how we defined our signatures).

Consistency

- It's useful to periodically check to make sure that we haven't *over*-constrained our model . . .
  (i.e., made it impossible for consistent instances to ever exist)
- . . . and also to check that we have *enough* constraints.
  (i.e., the sorts of instances generated match up with our intentions.)

Consistency

- So let's check that we can have the result of "function application" result in a set larger than one – i.e., there is more than one address mapped to.

```
pred show (b: Book) {
  #b.addr > 1
  #Name.(b.addr) > 1
}


run show for 3 but 1 Book
```

- (This says to take the function `b.addr` for our book, and apply it to the set `Name`.)

Operations

- We can also write predicates that represent *operations* on things;
  typically, they'll refer to the "before" and "after" states of those things.

```
pred add (b, b': Book, n: Name, a: Addr) {
  b'.addr = b.addr + n -> a
}
```

- Our predicate add is a constraint, and says that `b'.addr` is the union of `b'.addr` and the tuple `(n,a)`.

Operations

- If we want to see if we can find instances that satisfy this predicate, we'll want to enlarge the scope:

```
pred showAdd (b, b': Book, n: Name, a: Addr) {
   add[b, b', n, a]
   #Name.(b'.addr) > 1
}

run showAdd for 3 but 2 Book
```

- Using the Alloy visualizer, we can see what the "before" and "after" books look like.
- In the predicate above, the "add" predicate is *invoked*.
  This is a bit more like traditional function application: we supply arguments to the predicate between square brackets.
  - (Earlier versions of Alloy used parentheses.)

Operations

- We can write similar code for other operations, like “delete”, and check that our expected constraints hold.

Advantages of using Alloy to check models

- Alloy allows us to build models incrementally.
- We can start with a small, simple model, and add features.
- Furthermore, it's much easier to see what our model *is* when it's not commingled with code.
    - Once an application becomes large, we can imagine that when written in Java (say), there is a great deal of implementation code that obscures the abstract model.

Comparison with other methods – “model checking”

- We refer to this as “checking our model”; but note that “model checking”, on its own, refers to a different sort of formal method.
- “Model checking” on its own normally refers to using various sorts of temporal logic to explore the evolution of finite state machines, and see whether particular constraints hold.

Comparison with other methods – proofs and verification

- Note that Alloy only generates model instances up to a certain size;
  - it doesn't *prove* that a model is consistent.
- However, often, if there is an inconsistency, it will show up in quite small models.

Exercise

- How could we model the *natural numbers* in Alloy?
- How could we model the process of transferring money between two bank accounts?