CITS5501 Software Testing and Quality Assurance
Specification languages

Unit coordinator: Arran Stewart

Specification languages

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.

Review – Alloy signatures

We've seen that Alloy lets us declare that there are particular kinds of things, using signatures – for example

```
sig FSObject {}  // there are file system objects

sig Animal   {}  // there are animals

sig Node {}      // there are nodes in a data structure
```

Alloy signatures

Alloy also has some signatures built in – for instance `Int` – and others are available in standard library modules (for instance there is a module `util/sequence` with useful signatures for modelling sequences (list-like objects).

Relations

We've seen that Alloy lets us declare that there are *relations* between things.

```
sig Person { friends : Person }
// People can have friends
```

Relations

```
sig Person { friends : Person }
// People can have friends
```

We can use relations to model things like

- containment – one sort of entity *contains* others
- labelling – for instance, we might state that computers have an IP address, which acts as a sort of "name"
- grouping – we might want to single out objects which have some common property (e.g. carnivores, which are animals, and all have the property that they eat meat)
- linking – there is a link between objects in which they are "peers" (rather than one "containing" the other)

... in fact, any sort of relationship between entities we want.

Multiplicities

We can declare *multiplicities* for relations:

```
sig Node { next : lone Node }
  // The node can have one 'next' Node

sig Dir  { contents : set FSObject }
  // directories have 0 or more objects they contain

one Phoenix extends Animal {}
  // There is one Phoenix in the world
```

Facts

We can declare additional constraints which must be true of any possible “world”.

```
sig Employee {}

fact atLeastTwoEmployees {
  #Employee >= 2
}

sig Manager {}

fact moreManagersThanEmployees {
  #Manager >= #Employee
}
```

Running the Alloy Analyzer

There are two main ways of using the Alloy Analyzer.

The run command asks the analyzer to *construct* examples of our model – up to some maximum size – and try to find one which satisfies a condition we're interested in. conditions we specify.

For the check command, we specify some assertion which we think *should* be true, and ask the analyzer if it can find any counterexamples.

(They are a bit like opposites – run is asking for a case where our condition *is* true, check for one where it is not.)

The run command

The run command uses *predicates*, statements which can be true or false, to filter the “worlds” we’re interested in.

Alloy predicates

An example predicate:

```
pred hasSuccessor(n : Node) {
 #(n.next) = 1
}
```

Alloy predicates

```
pred hasSuccessor(n : Node) {
 #(n.next) = 1
}
```

This says “this predicate is true if the Node we apply it to has exactly one ‘next’ node”.

Predicates take zero or more arguments, and can be re-used in multiple places in our model.

Predicates always evaluate to either “true” or “false” – you can think of them as always having return type `bool`.

Alloy predicates

Predicates contain *constraints*.

```
pred oneBeforeLast(n : Node) {
 #(n.next) = 1
 #(n.next.next) = 0
}
```

Alloy predicates

We could rewrite the previous examples as follows:

```
pred hasSuccessor(n : Node) {
 one n.next
}

pred oneBeforeLast(n : Node) {
 one n.next
 no n.next.next
}
```

`one` just means "has cardinality one", and `no` just means "has cardinality zero".

Alloy predicates

If our predicate has *no* constraints in it, then it is always true:

```
pred alwaysTrue(n : Node) {
}

pred alsoAlwaysTrue() { // preds can have no arguments
}
```

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
}
```

Predicates operating on sets

The arguments to predicates can be sets, not just “individuals”:

```
sig Card {suit: Suit}
sig Suit {}
pred ThreeOfAKind (hand: set Card) {
  #hand.suit = 1 and #hand = 3
  }
```

run command

We “run” an Alloy model by asking the analyzer to look for a sample “world” for us which satisfies some predicate (up to a particular “size” of the world).

By convention, if we want to put no constraints on what we see, we call our predicate “show”.

```
sig Node { next : lone Next }
pred show() {}
run show for 3
```

run command

```
sig Node { next : lone Node }
pred show() {}
run show for 3
```

- the `show` means we want the analyzer to find a world in which `show` is true. (Which is any world – `show` is *always* true.)
- `for 3` means the analyzer will consider worlds in which there are up to 3 objects for any signature we specified.
  (It needs to know this "scope" so it can decide when to give up if it can't find an example.)

Example of run

```
sig Node { next : lone Node }
pred show() {}

pred oneBeforeLast(n : Node) {
  one n.next
  no n.next.next
}
run oneBeforeLast for 3
```

This asks Alloy to find a universe in which the predicate `oneBeforeLast` is true of some `Node`.

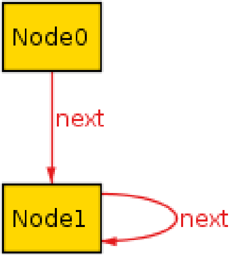
Example of run

```
sig Node { next : lone Node }

pred allHaveSuccessors() {
  all n : Node | one n.next
}

run allHaveSuccessors for 3
```

This asks Alloy to find a universe in which *all Nodes* have a ‘next’ Node – what sort of example might it come up with?

Example of run

```
sig Node { next : lone Node }

pred allHaveSuccessors() {
  all n : Node | one n.next
}

run allHaveSuccessors for 3
```



Example of run

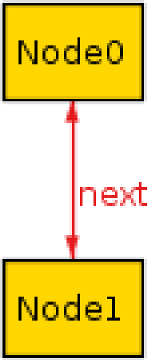
Oops. If we were intending to model non-cyclic linked lists, this probably isn't what we had in mind – you can never reach the "end" of this list.

We need to constrain our world a bit more.

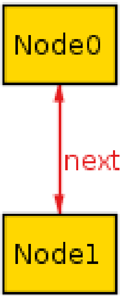
```
sig Node { next : lone Node }

fact noSelfSuccessors {
  all n : Node | n.next != n
}

pred allHaveSuccessors() {
  all n : Node | one n.next
}

run allHaveSuccessors for 3
```

Example of run

```
sig Node { next : lone Node }

fact noSelfSuccessors {
  all n : Node | n.next != n
}

pred allHaveSuccessors() {
  all n : Node | one n.next
  #Node > 0
}

run allHaveSuccessors for 3
```



Example of run



By viewing examples which satisfy particular predicates, we can refine our model until it matches what we want.

check

Alternatively, we might think there's some predicate we think should never be violated, and ask Alloy to double-check this – can it find a counter-example?

We'll see examples of check commands in the workshop.

File system example

Let's revisit the file system example from last lecture.

```
sig FSObject { parent: lone Dir }

sig Dir extends FSObject { contents: set FSObject }

sig File extends FSObject { }

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

- FSObjects have parents, and directories have contents, and we have constrained the multiplicities

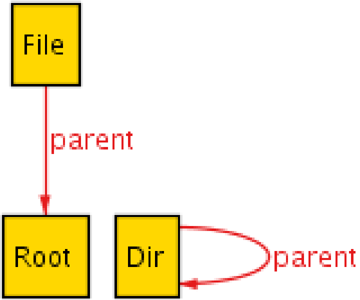
File system example

We can run this to see examples of file systems which match our specifications.

File system example

We can run this to see examples of file systems which match our specifications.

```
sig FSObject {
  parent: lone Dir
}

sig Dir extends FSObject {
  contents: set FSObject
}

sig File extends FSObject { }

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

pred show() {}
run for 3
```



File system

- We need to constrain things more, so we'll use a *fact*.

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

- This says: "for any thing (let's call it *d* for the moment) of type `Dir`, and for any thing (let's call it *o* for the moment) which is in the set `d.contents`:
  o's parent is *d*.

Address book example

- Consider the following specification for an address book:

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

  Let's limit the scope to just one Book, like this:

  ```
  pred show() {}
  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

- Alloy will find us a basic instance with 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 address in our Book

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
}
```

- 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

- 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
}
```

Operations

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

- Alloy allows apostrophes ("`'`") in names, so "`b'`" is just another parameter name.
- Alloy doesn't make any connection between a variable called (say) "x" and one called "`x'`" (pronounced "x prime").
- But in modelling, the intended meaning when we write a variable like "`x'`" is usually "x, but at the next step in time", or "x, but after the completion of this operation".
- 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 if people refer to “model checking”, on its own, that 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.
- In the workshop, we'll see additional examples of Alloy models.