CITS5501 Software Testing and Quality Assurance
Semester 1, 2018
Workshop 9 – Formal specifications

This workshop contains some revision notes on Alloy syntax, and four exercises.

Alloy notes

Expressions

- Two sorts of expressions: *boolean* expressions (result in a true/false) and *relational* expressions (results in a set or relation).

  Boolean expressions are used in constraints.

Operators

- Logical operators (there are English-word synonyms for the symbols):
  - "not": `!`, `not`
  - "and": `&&`, `and`
  - "if .. then", "implies": `=>` , `implies`
  - "if and only if": `<=>`, `iff`
  - "or": `||`, `or`
- Relational operators: these operate on relations/sets.

  e.g. "set union" (`+`) and "set difference" (`-`). These operate on sets and return sets.

  Cardinality (`#`): what is the size of a set?
- Quantifiers: `all`, `no`, `lone`, `some`, `one`.

  Quantifiers can be used to introduce names of things you can then do boolean tests on.

Writing constraints/facts

- Constraints will often look something like `all x: X | F`, where `x` gives us a name we can refer to things by (it introduces a *variable name*);

"`X`" is a set or relation;
amd "`F`" is a boolean expression which we're asserting is always true.
- You can think of the bar symbol (`|`) as being pronounced "it is the case that" or "is true of". So `all x : X | F` can be read as "for all `x` in the set `X`, it is the case that `F`".
- The `sigs` let us declare what in the universe exists; facts let us declare immutable laws which are true in our universe.

Example constraints:

- Suppose we have the following types:

  ```
  sig Activity {}
  sig Person { hobbies: set Activity }
  sig ComputerScientist extends Person {}
  ```

  We can apply the following constraint: "Computer scientists have no hobbies:"

  ```
  fact {
    all cs : ComputerScientist | no cs.hobbies
  }
  ```

  Another way of phrasing this: "The cardinality of a computer scientist's hobbies is 0"

  ```
  fact {
    all cs : ComputerScientist | #cs.hobbies = 0
  }
  ```

- "For any directory *d*, and for any object *o* in its contents,
  the parent of *o* must be *d*."

  ```
  fact {
    all d: Dir | all c : d.contents | c.parent = d
  }
  ```

Prohibiting self-loops

- It's common to insist that some relation is *not reflexive*.

  e.g. "A directory cannot be its own parent."

  ```
  fact parentNotReflexive {
    all d : Dir | no c : d.contents | c = d
  }
  ```

We can also specify things that are true of not just the contents, but anything that is the contents of the contents, or the contents of the contents of the contents, and so on.

- We can use the `^` operator to get the *transitive closure* of a relation.
- Suppose we have

  ```
  sig Person { father : Person, mother : Person }
  ```

  If p is a person, then the expression

  ```
  p.^father
  ```

  means, "`p.father`, *and* `p.father.father`, *and* `p.father.father.father`, and so on ...". This is called the "non-reflexive transitive closure of father on p".

  Suppose we want to include p in the set as well: we can use the `*` operator, which gives the "*reflexive* transitive closure":

  ```
  p.*father
  ```

Exercises

Model the following systems.

1. An *alarm clock* has two sorts of time it can keep track of: the *current* time, and an *alarm* time.

   It *always* has a current time, and *may* have an alarm time.

2. A person can have up to two parents (who are also people). No person is their own ancestor. There is one person – call them bob – who has no parents.
3. Entities called *nodes* exist, with which we can make linked lists. A node may have a successor node, called its "*next*" node. Cyclic lists do not exist.
4. A *contact list* may contains many *entries.* Each entry must have a "personName" property, and may have telephone, street address, and email properties.