CITS5501 Software Testing and Quality Assurance
Specifications languages

Unit coordinator: Arran Stewart

Re-cap of formal methods

We've divided up formal methods into three rough categories (though the boundaries can be fuzzy):

- advanced type systems
- program verification
- model-based systems

Formal methods

How can we tell what techniques fall into what category?

Formal methods

How can we tell what techniques fall into what category?

advanced type systems:

- I assume a basic familiarity with what a type system *does* – it enforces rules such as (in Java), "You can't assign a `String` object to (say) a variable of type `boolean`."
- If you want a definition of a type system, here is one from Pierce (2002):

  "a tractable syntactic method for proving the absence of certain program behaviors by classifying phrases according to the kinds of values they compute"
- I use "advanced" just to mean "Not in widespread use in the most popular statically type-checked languages" (which would be Java, C# and C++)
  - not a high bar, they are fairly simple type systems
  - languages with more complex type systems: Haskell, Rust, ML, Ocaml, PureScript, ATS

Formal methods

program verification:

- Does the process involve using the source code as the model, and proving it meets preconditions and postconditions?
  - Then it's program verification

Model-based systems

- Does the process involve using a model which is fairly *different* to the source code, and checking or proving properties?
    - Then it's a model-based system.
- Terminology you may encounter:
    - One particular class of model-based systems are called "model checkers" – but we won't look at them in detail
- "Specification language":
    - Model-based systems are often used to make more precise the *specification* for a system or some component – in which case they may be called *specification languages*

Specification languages

Some examples of general-purpose specification languages:

- **Z notation**
  - based on set theory and predicate logic
  - developed in the 1970s.
  - Now has an ISO standard, and variations (e.g. object-oriented versions)
- **TLA+**:
  - Stands for "Temporal Logic of Actions"
  - A general-purpose specification language
  - Especially well-suited for writing specifications of concurrent and distributed systems
  - For finite state systems, can check (up to some number of steps) that particular properties hold (e.g. safety, no deadlock)

TLA+

Using TLA+, code for Peterson's mutual exclusion algorithm:

```
--algorithm Peterson {
     variables flag = [i \in {0, 1} |-> FALSE],  turn = 0;
        \* Declares the global variables flag and turn and their initial values;
        \* flag is a 2-element array with initially flag[0] = flag[1] = FALSE.
     fair process (proc \in {0,1}) {
       \* Declares two processes with identifier self equal to 0 and 1.
       \* The keyword fair means that no process can stop forever if it can
       \* always take a step.
       a1: while (TRUE) {
            skip ;  \* the noncritical section
       a2:  flag[self] := TRUE ;
       a3:  turn := 1 - self ;
       a4:  await (flag[1-self] = FALSE) \/ (turn = self);
             \* \/ is written || in C.
       cs:  skip ;  \* the critical section
       a5:  flag[self] := FALSE
```

TLA+

- The TLA+ tools turn this algorithm (written in a language called PlusCal), into a TLA+ specification, which can then be checked.
- The TLC model checker can verify that the algorithm satisfies two important properties:
  - mutual exclusion, meaning that two processes are never executing their critical section at the same time
  - starvation freedom, meaning that each process keeps executing its critical section.

Alloy

- We'll be using the **Alloy** specification language
- Alloy is both a language for describing structures, and a tool (written in Java) for exploring and checking those structures.
- Influenced by Z notation, and modelling languages such as UML (the Unified Modelling Language).
- Website: http://alloy.mit.edu/ (The Alloy Analyzer tool can be downloaded from here.)

Alloy language

- We'll look at a simple model of a file system (based on the Alloy tutorial at http://alloytools.org/tutorials/online/)
- To a first approximation, Alloy looks a little like Java:

```
// A file system object in the file system
sig FSObject { parent: lone Dir }

// A directory in the file system
sig Dir extends FSObject { contents: set FSObject }

// A file in the file system
sig File extends FSObject { }
```

Alloy

In Alloy, we declare rules about a mini-universe: things that exist, and properties that should be true of them.

- “There are things called animals”

  ```
  sig Animal {}
  ```

- “A cat is a sort of animal”

  ```
  sig Cat extends Animal {}
  ```

Alloy – relations

Alloy's semantics are defined in terms of mathematical *relations*.

Example relations:

- "Is less than". e.g. "$2 < 4$", "$10 < 9$".
- "Is the blood relative of". e.g. "Alice is the blood relative of Bob".
- "Shares an office with". e.g. "Bob shares an office with Carol".

These are all *binary relations*. Statements about two entities, which can be true or false.

Alloy – relations

Relations can also be *unary* (about one entity):

- "Is even". e.g. *even*(2).
- "Is an employee". e.g. "Dan is an employee".

Alloy – relations

They can be ternary:

- "_ is delivered to _, by _". e.g. "The *blue book* was delivered to *Alice*, by *Bob*".
- "_ was made by _, programming in _". e.g. "The *timetabling system* was made by *Ralph*, programming in *Java*".

Or, in general, they can be n-ary – a statement about n things.

Alloy – relations

We can think of predicates as being not-yet-complete functions – an *n*-ary predicate isn't true or false in itself, until we supply it with *n* arguments.

- "Is less than" isn't true or false, but "$2 < 4$" is.

Alloy – relations

Another way of viewing relations is as being a sort of table – containing all the things of which the predicate is true.

e.g. "shares an office with":

Person A	Person B
Alice	Bob
Bob	Alice
Dan	Eve
Eve	Dan

Alloy – relations

Relations can be finite, or infinite.

An infinite relation: "is less than"

Number A	Number B
1	2
1	3
2	3
...	...

Alloy – sigs

`sig Animal {}` says "There are things called animals".

It defines a unary relation, "Animal". Something thing can be-an-animal, or not.

Alloy – sigs

`sig Cat extends Animal {}` says "Cats are a sort of animal".

If something has the property "is-an-animal", *then* it might also have the property "is-a-cat".

We can read "extends" as also meaning "is a kind of", or "is a subtype of".

Alloy – subtypes

- So, **extends** indicates subtypes (similar to Java).
- Here, **Dir** and **File** are both subtypes of **FSObject**:

```
sig FSObject {}

sig Dir extends FSObject {}

sig File extends FSObject {}
```

- When we declare **Dir** or a **File** to be sub-types of **FSObject**, they are considered to be *mutually disjoint* sets
- The above says "There are things called FSObjects. An FSObject might be a Dir or it might be a File (or neither), but not both".

Alloy – properties

We can specify *properties* of entities (which look a bit like instance variables in OO languages):

```
// A file system object in the file system
sig FSObject { parent: lone Dir }

// A directory in the file system
sig Dir extends FSObject { contents: set FSObject }

// A file in the file system
sig File extends FSObject { }
```

Alloy – properties

```
// A file system object in the file system
sig FSObject { parent: lone Dir }

// A directory in the file system
sig Dir extends FSObject { contents: set FSObject }

// A file in the file system
sig File extends FSObject { }
```

These are usually written within the sig of an entity.

They actually represent *relations* between entities.

Alloy – properties

```
// A file system object in the file system
sig FSObject { parent: lone Dir }
```

There are multiple ways of reading this:

- "There are such things as FSObjects. An FSObject has the property 'parent'. An FSObject can have zero or one parents." Or –
- "A relation 'parent' exists between FSObjects and Dirs. Whenever an FSObject appears in the relation, it can be associated with at most one Dir."

These are exactly equivalent.

Alloy – properties

```
// A file system object in the file system
sig FSObject { parent: lone Dir }
```

- The "`lone`" means "zero or one". It is a *cardinality*.
- Other possible cardinalities are:
  - "some" (one or more)
  - "one" (exactly one)
  - "set" (zero or more)
  - "none" (zero)
- When we specify a property using a colon in this way, the default multiplicity is one.
- We can use cardinalities whenever we are specifying a set or relation: since sigs also represent sets (e.g. the set of Dirs), we can give them cardinalities, too.

Cardinalities

- In set theory terms . . .
- **`one`** means the relation is a total function –
  `sig Student { name : one String }` –
  for every Student, we can map to a string which is their name.
- **`lone`** means the relation is a *partial* function –
  `sig Student { driverLicenseNum : lone String }` – \
  for every Student, we *may* be able to map to a diver's license number.
  (Here, it's assumed you can't have more than one license.)

Examples

```
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
```

Alloy – properties

```
one sig RootDir extends Dir { }
```

There exists a “RootDir”, but only one of them.

Exercise

Games:

- There are things called games.
- Games can be board games, or field games.
- There may be other sorts of games.

Relation examples

- ```
  // A file system object in the file system
  sig FSObject { parent: lone Dir }

  // A directory in the file system
  sig Dir extends FSObject { contents: set FSObject }

  // A file in the file system
  sig File extends FSObject { }
  ```
  ```

Relations

- ```
  // A file system object in the file system
  sig FSObject { parent: lone Dir }

  // A directory in the file system
  sig Dir extends FSObject { contents: set FSObject }

  // A file in the file system
  sig File extends FSObject { }
  ```
- To a first approximation, we can think of relations as behaving like *fields* in an OO language.
- `sig FSObject { parent: lone Dir }` can be read as "Things of type `FSObject` *have a parent*, which is of type `Dir`".
- **`lone`** means "at most one" – i.e., you can have zero or one parents. (We need this because the root directory has no parent.)
  ```

Relations

- ```
  // A file system object in the file system
  sig FSObject { parent: lone Dir }

  // A directory in the file system
  sig Dir extends FSObject { contents: set FSObject }

  // A file in the file system
  sig File extends FSObject { }
  ```
- More precisely, `parent` is a relation between `FSObject` and `Dir`.
  ```

Relations

- So, signature declarations will look like:

```
sig SomeName {
  field1 : FieldType,
  field2a, field2b : OtherFieldType
}
```

- The order of declarations doesn't matter – `SomeName`, `FieldType` and `OtherFieldType` could be declared in any order in a file.

Relations

- `// A directory in the file system`
  `sig Dir extends FSObject { contents: set FSObject }`
- Here, we say that a `Dir` has a field `contents`, which is a *set* of `FSObjects`.
- The could contain one item, many items, or no items.

Examples

- "A car has one engine"

  `sig Car { engine: one Engine }`, or
  `sig Car { engine: Engine }`

- "People have zero or more hobbies"

  `sig Person { hobbies: set Activity }`

Exercises

- Classes have at least one lecturer, and zero or more students.

Exercises

- Classes have at least one lecturer, and zero or more students.
- Animals have zero or more legs

Exercises

- Classes have at least one lecturer, and zero or more students.
- Animals have zero or more legs
- Some animals are carnivores

Exercises

- Classes have at least one lecturer, and zero or more students.
- Animals have zero or more legs
- Some animals are carnivores
- Textbooks have one or more pages

Alloy language – comments

```
// A file system object in the file system
sig FSObject { parent: lone Dir }

// A directory in the file system
sig Dir extends FSObject { contents: set FSObject }

// A file in the file system
sig File extends FSObject { }
```

- Comments can be written in multiple ways

Alloy language – comments

```
// A file system object in the file system
sig FSObject { parent: lone Dir }

// A directory in the file system
sig Dir extends FSObject { contents: set FSObject }

// A file in the file system
sig File extends FSObject { }
```

- Comments can be written in multiple ways
  - single-line comments with “//” or “--”

Alloy language – comments

```
// A file system object in the file system
sig FSObject { parent: lone Dir }

// A directory in the file system
sig Dir extends FSObject { contents: set FSObject }

// A file in the file system
sig File extends FSObject { }
```

- Comments can be written in multiple ways
  - single-line comments with "//" or "--"
  - multiple-line comments with "/* ... */"

Facts

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

These constraints might be about properties of sets:

```
sig Employee {}

fact atLeastTwoEmployees {
  #Employee >= 2
}

sig Manager {}

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

Alloy – facts

- How can we express that any **FSObject** must be one of either a **Dir** or a **File**?
  (i.e., there are no other sorts of **FSObject**)

Alloy – facts

- How can we express that any **`FSObject`** must be one of either a **`Dir`** or a **`File`**?
  (i.e., there are no other sorts of **`FSObject`**)
- We will use a fact:

```
sig FSObject { parent: lone Dir }
sig Dir extends FSObject { contents: set FSObject }
sig File extends FSObject { }

// All file system objects are either files or directories
fact { File + Dir = FSObject }
```

Alloy – facts

- The general syntax for a fact is

  fact *name* { *formulas* }

- *formulas* are Boolean expressions, and by putting them in a fact, we're constraining them to be true.

Alloy – abstract signatures

- (An alternative way to say that all FSObjects must be Dirs or Files would be to declare FSObject **`abstract`**)

Alloy – abstract signatures

- (An alternative way to say that all FSObjects must be Dirs or Files would be to declare FSObject **abstract**)
- (This is similar to the use of the **abstract** keyword in Java; it means there are no objects that are *directly* of type FSObject; they must be members of some subtype, instead.)

Alloy – operators

Operators are available to construct Boolean expressions.

- subset: **`in`**
  - *`set1`* **`in`** *`set2`* — *`set1`* is a subset of *`set2`*
  - informally: "some *`set2`* are *`set1`*", or "a *`set2`* may be *`set1`*"; but the set-theoretic meaning is more precise.
- set equality: **`=`**
  - *`set1`* **`=`** *`set2`* — *`set1`* equals *`set2`*
- scalar equality: **`=`**
  - `scalar` **`=`** *`value`* — *`scalar`* equals *`value`*

Alloy – subsets

- We saw that subtypes are disjoint.
- We can also declare sub*sets*:

  sig *signame* **in** *supername* { ... }

- Subsets are *not* necessarilly disjoint, and may have multiple parents

Alloy – subsets

```
sig Animal {}
sig Cat extends Animal {}
sig Dog extends Animal {}
sig FurryPet in Cat + Dog {}
```

- "`FurryPet`" is a subset of the union of `Cat` and `Dog`.
- Some dogs and cats may not be furry (hairless breeds).
- We could *make* them all furry as follows:

```
fact { Cat + Dog = FurryPet }
```

- Are there animals other than cats and dogs?
  Can they be furry?

More operators

- We can use Boolean connectives **and**, **or**, **implies**, **iff**, **not** to join Boolean expressions.
- e.g.

```
fact { A + B = C and X + Y = Z }
```

Back to the file system example

```
sig FSObject { parent: lone Dir }

sig Dir extends FSObject { contents: set FSObject }

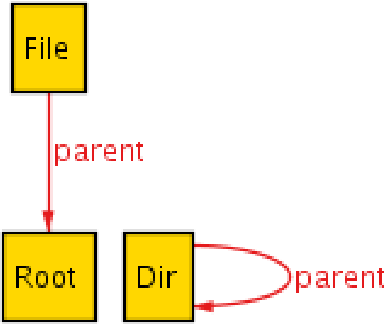
sig File extends FSObject { }

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

- FSObjects have parents, and directories have contents, and we have constrained the multiplicities . . .
- but there's currently no connection between them.

File system

- So we could have this situation:



File system

- We will 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*.
- It uses a *quantifier* ("all") – we'll look at these more in the workshop.

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
```

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)

Analyzer commands

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 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
}
```

This says "this predicate is true if the Node we apply it to has exactly one 'next' node".

Predicates are quite similar to functions. They take zero or more arguments, and can be re-used in multiple places in our model.

However, predicates always evalue to either "true" or "false" – you can think of them as always having return type `bool`.

And they contain constraints, rather than statements.

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

Alloy predicates

Note that 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
```

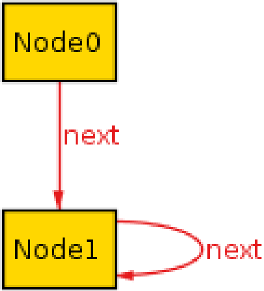
This asks Alloy to find a universe in which all Nodes have a 'next' Node – what sort of example might it come up with?

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
```

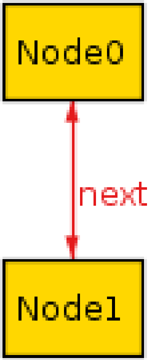
Example of run

```
sig Node { next : lone Node }

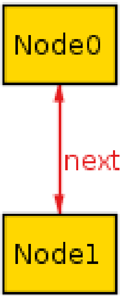
fact noSelfSuccessors {
  all n : Node | n.next != n
}

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.

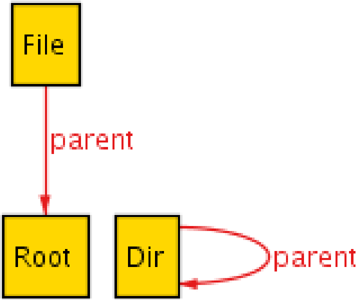
```
sig FSObject {
  parent: lone Dir
}

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
}
```

- 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.

References

- Pierce, Benjamin C. *Types and programming languages*. MIT press, 2002.
- 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.