An introduction to Alloy

Alcino Cunha

"I conclude there are two ways of constructing a software design: one way is to make it so simple there are obviously no deficiencies, and the other way is to make it so complicated that there are no obvious deficiencies."

Tony Hoare

“I conclude there are two ways of constructing a software design: one way is to make it so simple there are obviously no deficiencies, and the other way is to make it so complicated that there are no obvious deficiencies.”

Tony Hoare

“The first principle is that you must not fool yourself, and you are the easiest person to fool.”

Richard Feynman

“The core of software development is the design of abstractions.”

“An abstraction is not a module, or an interface, class, or method; it is a structure, pure and simple - an idea reduced to its essential form.”

“I use the term ‘model’ for a description of a software abstraction.”

Daniel Jackson

Alloy in a nutshell

- Declarative modeling language
- Automated analysis
- Lightweight formal methods

Alloy in a nutshell

- Declarative modeling language
- Automated analysis
- Lightweight formal methods

http://alloy.mit.edu

Key ingredients

Key ingredients

- Everything is a relation

Key ingredients

- Everything is a relation
- Non-specialized logic

Key ingredients

- Everything is a relation
- Non-specialized logic
- Counterexamples within scope

Key ingredients

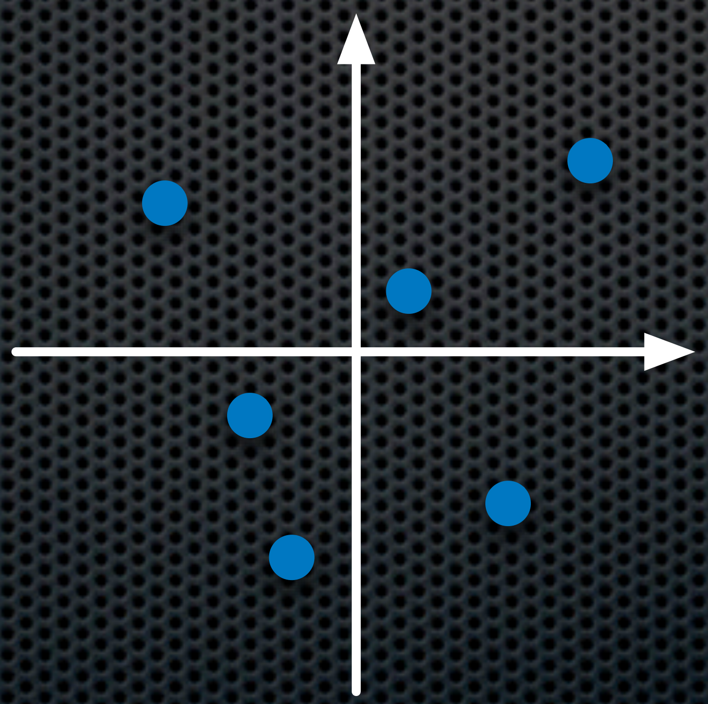
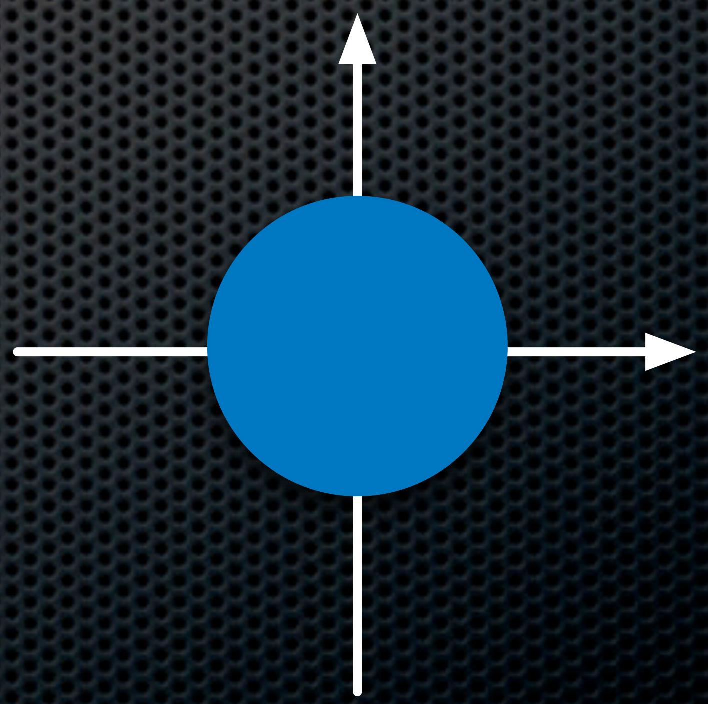
- Everything is a relation
- Non-specialized logic
- Counterexamples within scope
- Analysis by SAT

Small scope hypothesis

- Most bugs have small counterexamples
- Instead of building a proof look for a refutation
- A scope is defined that limits the size of instances

Small scope hypothesis

- Most bugs have small counterexamples
- Instead of building a proof look for a refutation
- A scope is defined that limits the size of instances

Relations

Relations

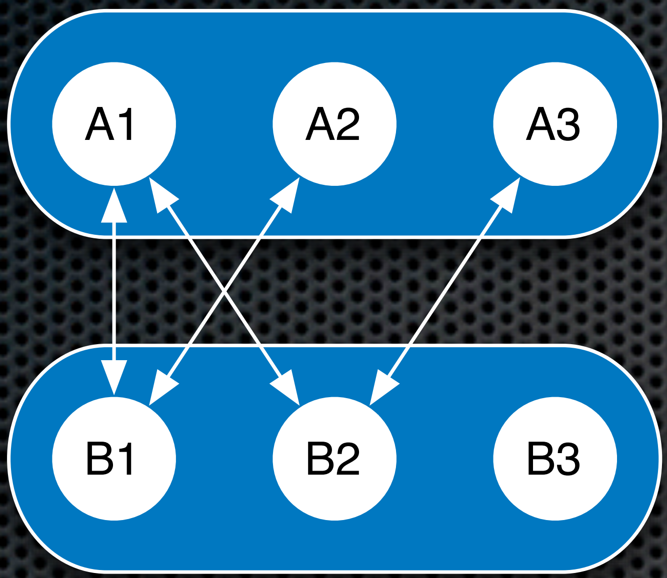
```
{(A1,B1),(A1,B2),(A2,B1),(A3,B2)}
```

Relations

A1	B1
A1	B2
A2	B1
A3	B2

```
{(A1,B1),(A1,B2),(A2,B1),(A3,B2)}
```

Relations



A1	B1
A1	B2
A2	B1
A3	B2

{(A1,B1),(A1,B2),(A2,B1),(A3,B2)}

Relations

- Sets are relations of arity 1
- Scalars are relations with size 1
- Relations are first order... but we have multirelations

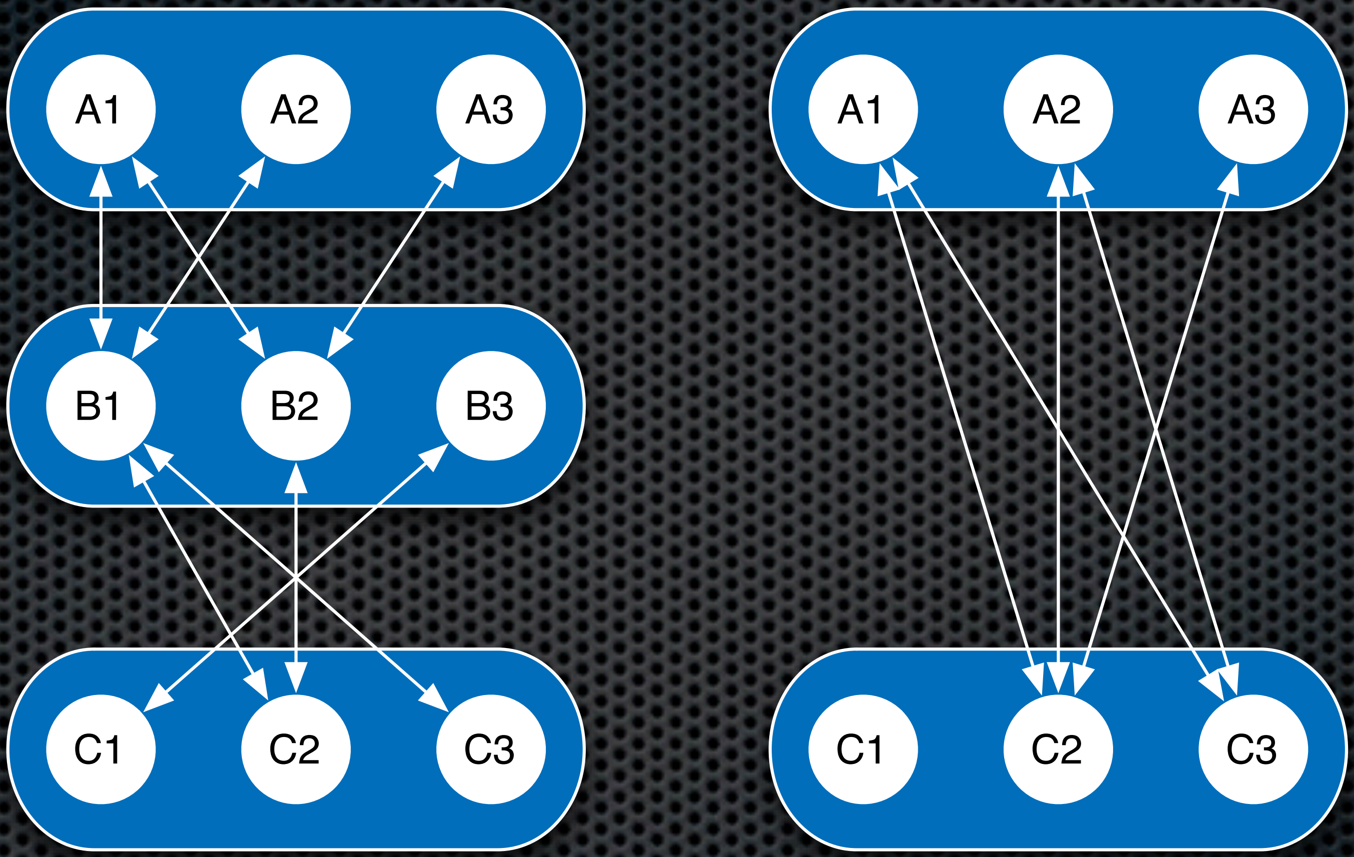
```
File   = {(F1),(F2),(F3)}
Dir    = {(D1),(D2)}
Time   = {(T1),(T2),(T3),(T4)}
root   = {(D1)}
now    = {(T4)}
path   = {(D2)}
parent = {(F1,D1),(D2,D1),(F2,D2)}
log    = {(T1,F1,D1),(T3,D2,D1),(T4,F2,D2)}
```

The special ones

`none`	empty set
`univ`	universal set
`iden`	identity relation

```
File = {(F1),(F2),(F3)}
Dir  = {(D1),(D2)}
none = {}
univ = {(F1),(F2),(F3),(D1),(D2)}
iden = {(F1,F1),(F2,F2),(F3,F3),(D1,D1),(D2,D2)}
```

Composition




```
R   = {(A1,B1),(A1,B2),(A2,B1),(A3,B2)}
S   = {(B1,C2),(B1,C3),(B2,C2),(B3,C1)}
R.S = {(A1,C2),(A1,C3),(A2,C2),(A2,C3),(A3,C2)}
```

Composition

- The swiss army knife of Alloy
- It subsumes function application
- Encourages a navigational (point-free) style
- `R.S[x] = x.(R.S)`

```
Person = {(P1),(P2),(P3),(P4)}
parent = {(P1,P2),(P1,P3),(P2,P4)}
me = {(P1)}
me.parent = {(P2),(P3)}
parent.parent[me] = {(P4)}
Person.parent = {(P2),(P3),(P4)}
```

Operators

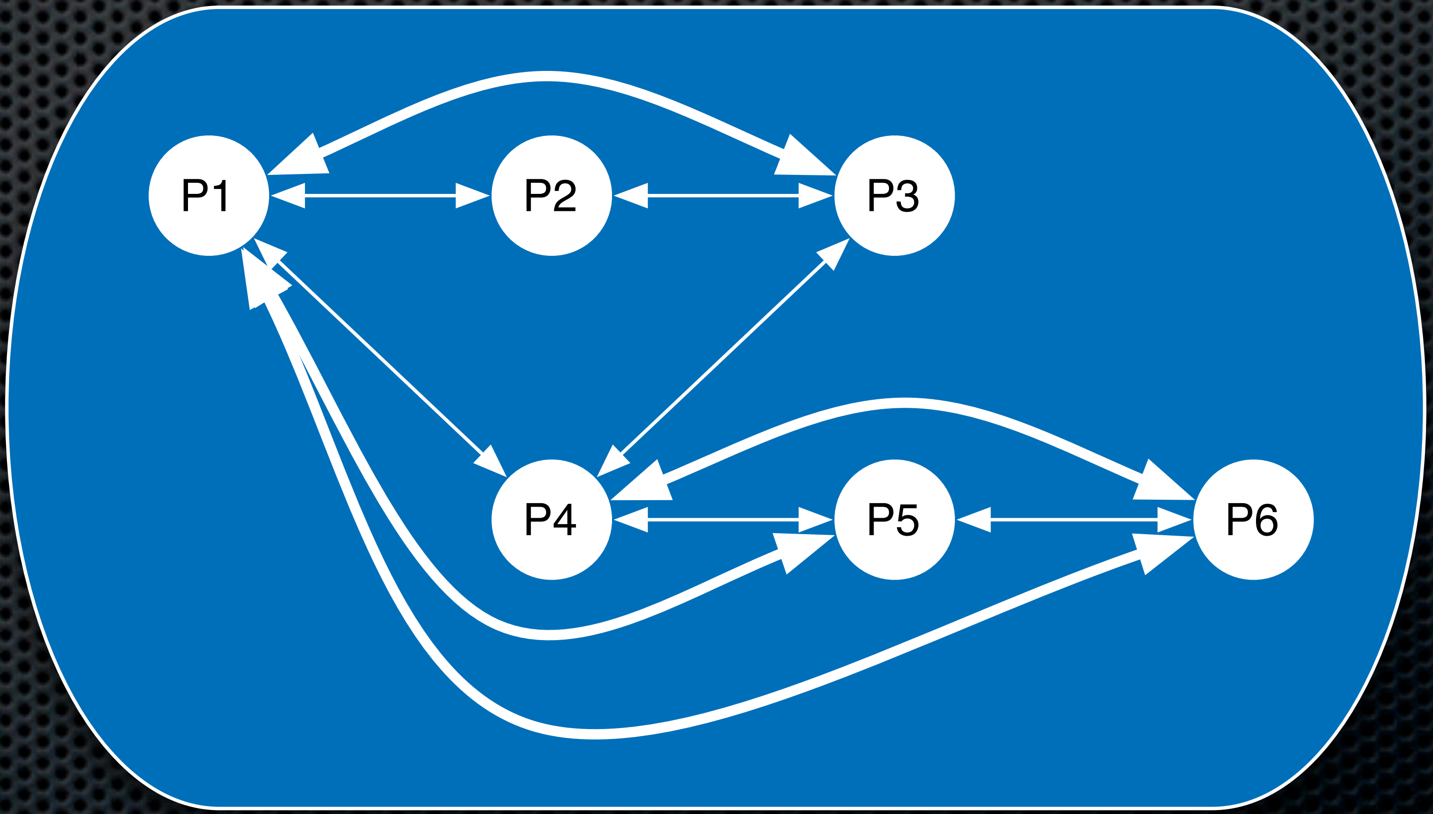
.	composition
+	union
++	override
&	intersection
-	difference
->	cartesian product
<:	domain restriction
:>	range restriction
~	converse
^	transitive closure
*	transitive-reflexive closure

Operators

```
File   = {(F1),(F2),(F3)}
Dir    = {(D1),(D2)}
root   = {(D1)}
new    = {(F3,D2),(F1,D1),(F2,D1)}
parent = {(F1,D1),(D2,D1),(F2,D2)}
File + Dir = {(F1),(F2),(F3),(D1),(D2)}
parent +  new = {(F1,D1),(D2,D1),(F2,D2),(F3,D2),(F2,D1)}
parent ++ new = {(F1,D1),(D2,D1),(F3,D2),(F2,D1)}
parent -  new = {(D2,D1),(F2,D2)}
parent &  new = {(F1,D1)}
parent :> root = {(F1,D1),(D2,D1)}
File -> root = {(F1,D1),(F2,D1),(F3,D1)}
new -> Dir = {(F3,D2,D1),(F3,D2,D2),(F1,D1,D1),(F1,D1,D2)}
~parent = {(D1,F1),(D1,D2),(D2,F2)}
```

Closures

- No recursion... but we have closures
- `^R = R + R.R + R.R.R + ...`
- `*R = ^R + iden`



Multiplicities

`A m -> m B`	
`set`	any number
`one`	exactly one
`some`	at least one
`lone`	at most one

Bestiary

A lone -> B	A -> some B	A -> lone B	A some -> B
injective	entire	simple	surjective

Bestiary

A lone -> B	A -> some B	A -> lone B	A some -> B
injective	entire	simple	surjective

A lone -> some B	A -> one B	A some -> lone B
representation	function	abstraction

A lone -> one B	A some -> one B
injection	surjection

A one -> one B
bijection

Signatures

- Signatures allow us to introduce sets
- Top-level signatures are mutually disjoint

```
sig File {}
sig Dir {}
sig Name {}
```

Signatures

- A signature can extend another signature
- The extensions are mutually disjoint
- Signatures can be constrained with a multiplicity

```
sig Object {}
sig File extends Object {}
sig Dir extends Object {}
sig Exe,Txt extends File {}
one sig Root extends Dir {}
```

Signatures

- A signature can be abstract
- They have no elements outside extensions
- Arbitrary subset relations can also be declared

```
abstract sig Object {}
abstract sig File extends Object {}
sig Dir extends Object {}
sig Exe, Txt extends File {}
one sig Root extends Dir {}
sig Temp in Object {}
```

Fields

- Relations can be declared as fields
- By default binary relations are functions
- The range can be constrained with a multiplicity

```
abstract sig Object {
   name: Name,
   parent: lone Dir
}
sig File extends Object {}
sig Dir extends Object {}
sig Name {}
```

Fields

- Multirelations can also be declared as fields
- Fields can depend on other fields
- Overloading is allowed for non-overlapping signatures

```
abstract sig Object {}
sig File, Dir extends Object {}
sig Name {}
sig FileSystem {
  objects: set Object,
  parent: objects -> lone (Dir & objects),
  name: objects lone -> one Name
}
```

Command `run`

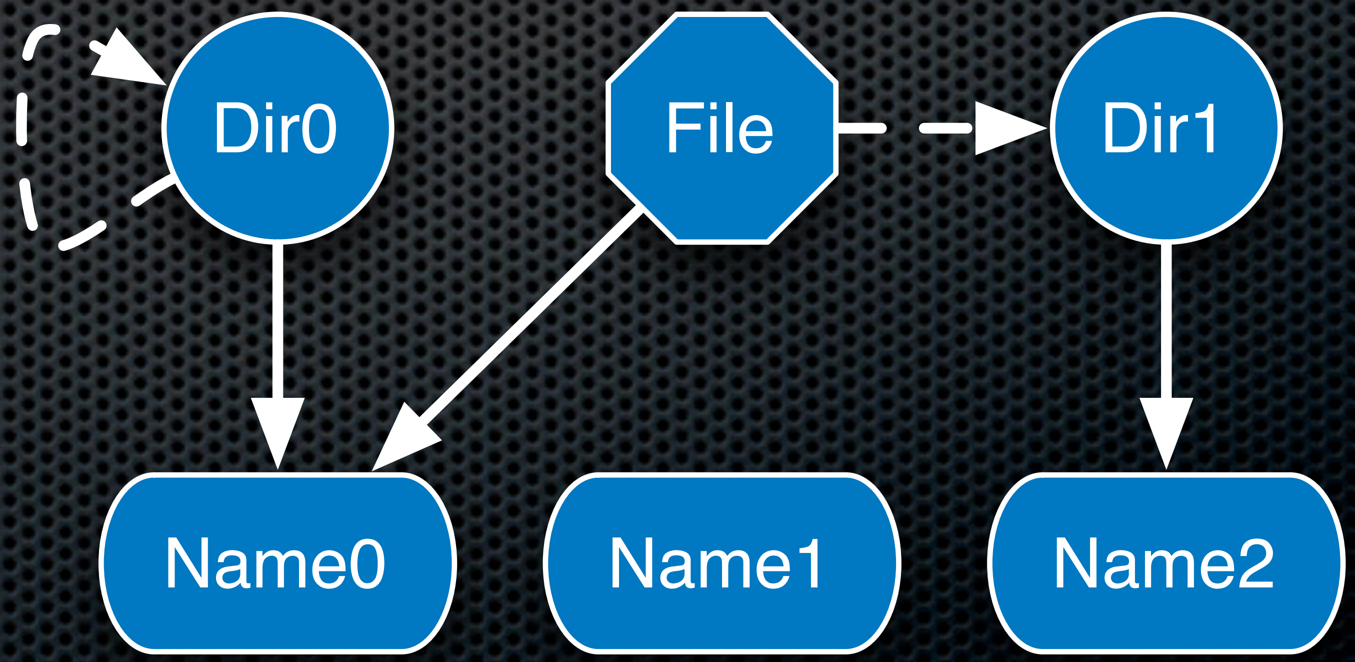
- Instructs analyser to search for instances within scope
- Scope can be fine tunned for each signature
- The default scope is 3
- Instances are built by populating sets with atoms up to the given scope
- Atoms are uninterpreted, indivisible, immutable
- It returns all (non-symmetric) instances of the model

Command `run`

```
abstract sig Object {
   name: Name,
   parent: lone Dir
}
sig File, Dir extends Object {}
sig Name {}
run {} for 3 but 2 Dir, exactly 3 Name
```

Command `run`

```
abstract sig Object {
   name: Name,
   parent: lone Dir
}
sig File, Dir extends Object {}
sig Name {}
run {} for 3 but 2 Dir, exactly 3 Name
```



Facts

- Constraints that are assumed to always hold
- Be careful what you wish for...
- First-order logic + relational calculus

```
abstract sig Object {
   name: Name,
   parent: lone Dir
}
sig File, Dir extends Object {}
sig Name {}
fact AllNamesDifferent {}
fact ParentIsATree {}
```

Operators

<table>
<tr><td>!</td><td>not</td><td>negation</td></tr>
<tr><td>&&</td><td>and</td><td>conjunction</td></tr>
<tr><td>||</td><td>or</td><td>disjunction</td></tr>
<tr><td>=></td><td>implies</td><td>implication</td></tr>
<tr><td><=></td><td>iff</td><td>equivalence</td></tr>
<tr><td colspan="3">A => B else C <=> (A && B) || (!A && C)</td></tr>
</table>

Operators

`=`	equality
`!=`	inequality
`in`	is subset
`no`	is empty
`some`	is not empty
`one`	is a singleton
`lone`	is empty or a singleton

Quantifiers

<table>
<tr><th colspan="2">Δ x:A | P[x]</th></tr>
<tr><td>all</td><td>P holds for every x in A</td></tr>
<tr><td>some</td><td>P holds for at least one x in A</td></tr>
<tr><td>lone</td><td>P holds for at most one x in A</td></tr>
<tr><td>one</td><td>P holds for exactly one x in A</td></tr>
<tr><td>no</td><td>P holds for no x in A</td></tr>
<tr><td colspan="2">Δ disj x,y:A | P[x,y] <=> Δ x,y:A | x!=y => P[x,y]</td></tr>
</table>

A question of style

A question of style

- The classic (point-wise) logic style

```
all disj x,y : Object | name[x] != name[y]
```

A question of style

- The classic (point-wise) logic style

```
all disj x,y : Object | name[x] != name[y]
```

- The navigational style

```
all x : Name | lone name.x
```

A question of style

- The classic (point-wise) logic style

```
all disj x,y : Object | name[x] != name[y]
```

- The navigational style

```
all x : Name | lone name.x
```

- The multiplicities style

```
name in Object lone -> Name
```

A question of style

- The classic (point-wise) logic style

```
all disj x,y : Object | name[x] != name[y]
```

- The navigational style

```
all x : Name | lone name.x
```

- The multiplicities style

```
name in Object lone -> Name
```

- The relational (point-free) style

```
name.~name in iden
```

A static filesystem

```
abstract sig Object {
   name: Name,
   parent: lone Dir
}
sig File, Dir extends Object {}
sig Name {}
fact AllNamesDifferent {
   name in Object lone -> Name     // name is injective
}
fact ParentIsATree {
   all f : File | some f.parent    // no orphan files
   lone r : Dir | no r.parent      // only one root
   no o : Object | o in o.^parent // no cycles
}
```

Assertions and check

- Assertions are constraints intended to follow from facts of the model
- check instructs analyser to search for counterexamples within scope

```
assert AllDescendFromRoot {
   lone r : Object | Object in r.*(~parent)
}

check AllDescendFromRoot for 6

check {name in Object lone -> Name <=> name.~name in iden}
```

Predicates and functions

- A predicate is a named formula with zero or more declarations for arguments
- A function also has a declaration for the result

```
fun content [d : Dir] : set Object {
   parent.d
}

pred leaf [o : Object] {
   o in File || no content[o]
}
```

Lets and comprehensions

let x = e | P[x]
$\{x_1 : A_1, \ldots, x_n : A_n \mid P[x_1,\ldots,x_n]\}$

```
fun siblings [o : Object] : set Object {
   let p = o.parent | parent.p
}
check {all o : Object | o in siblings[o]}

fun iden : univ -> univ {
   {x,y : univ | x = y}
}
```

Dynamic modeling

- Define the signatures that capture your state
- Define the invariants that constrain valid states
- Model operations with predicates
  - Relationship between pre and post-states
  - Do not forget frame conditions
- Check that operations are safe
- Check for consistency using `run`
- Be careful with over-specification

A dynamic filesystem

```
abstract sig Object {}
sig File, Dir extends Object {}
sig FS {
   objects : set Object,
   parent : Object -> lone Dir
}

pred inv [fs : FS] {
   fs.parent in fs.objects -> fs.objects
   all f : fs.objects & File | some fs.parent[f]
   lone r : fs.objects & Dir | no fs.parent[r]
   no o : fs.objects | o in o.^(fs.parent)
}
run inv for 3 but exactly 1 FS
```

A dynamic filesystem

```
pred rmdir [fs,fs' : FS, d : Dir] {
    d in fs.objects && no fs.parent.d
    fs'.objects = fs.objects - d
    fs'.parent = fs.parent - (d -> Object)
}

pred rmdir_consistent [fs,fs' : FS, d : Dir] {
    inv[fs] && rmdir[fs,fs',d]
}
run rmdir_consistent for 3 but 2 FS
assert rmdir_safe {
    all fs,fs':FS,d:Dir | inv[fs]&&rmdir[fs,fs',d]=>inv[fs']
}
check rmdir_safe for 3 but 2 FS
```

Modules

- `util/ordering[elem]`
  - Creates a single linear ordering over atoms in `elem`
  - Constrains all the permitted atoms to exist
  - Good for abstracting time, model traces, ...
- `util/integer`
  - Collection of utility functions over integers

Integers

- Scope limits bitwidth
- 2's complement arithmetic: be careful with overflows
- `Int` versus `int`

```
open util/integer
check {all x,y : Int | pos[y] => gt[add[x,y],x]}
sig Student {partial : set Int} {
   all i : partial | nonneg[i]
}
fun total[s : Student] : Int {
   Int[int[s.partial]]
}
```

Demos

- Filesystem
- I'm my own grandpa
- Bank accounts
- Train station
- ...