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

“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

“Simplicity does not precede complexity, but follows it.”

Alan Perlis

“Design is not just what it looks like and feels like. Design is how it works.”

Steve Jobs

Alloy in a nutshell

- Declarative modeling language
- Automated analysis
- Lightweight formal methods

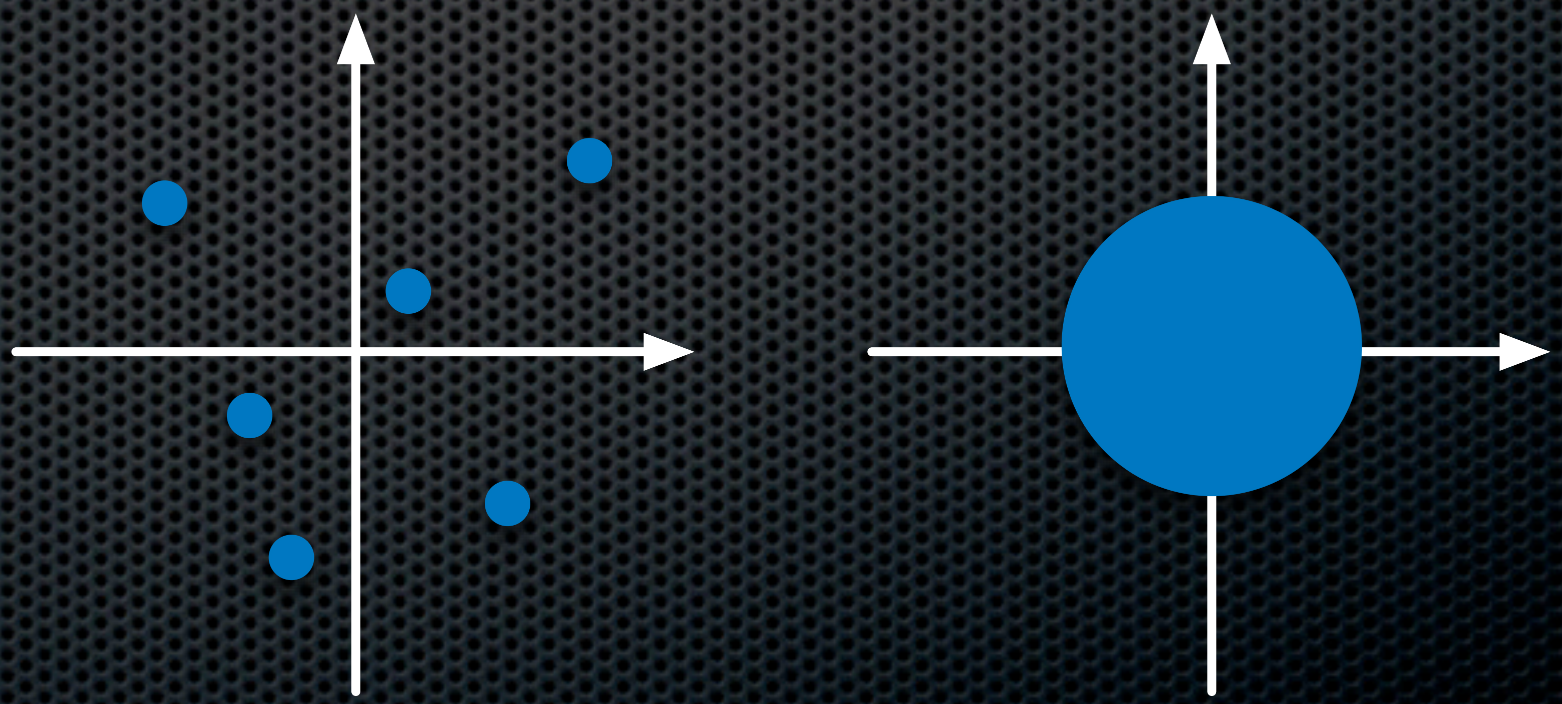
http://alloy.mit.edu

Key ingredients

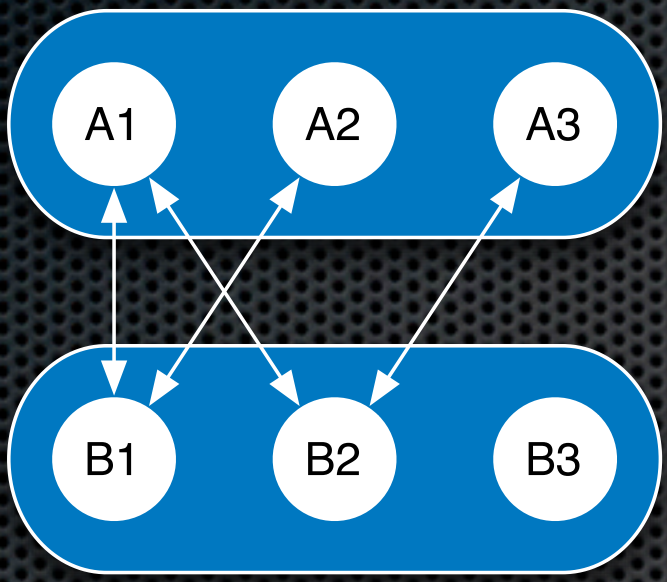
- Everything is a relation
- Non-specialized logic
- Counterexamples within scope
- Analysis by SAT

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



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

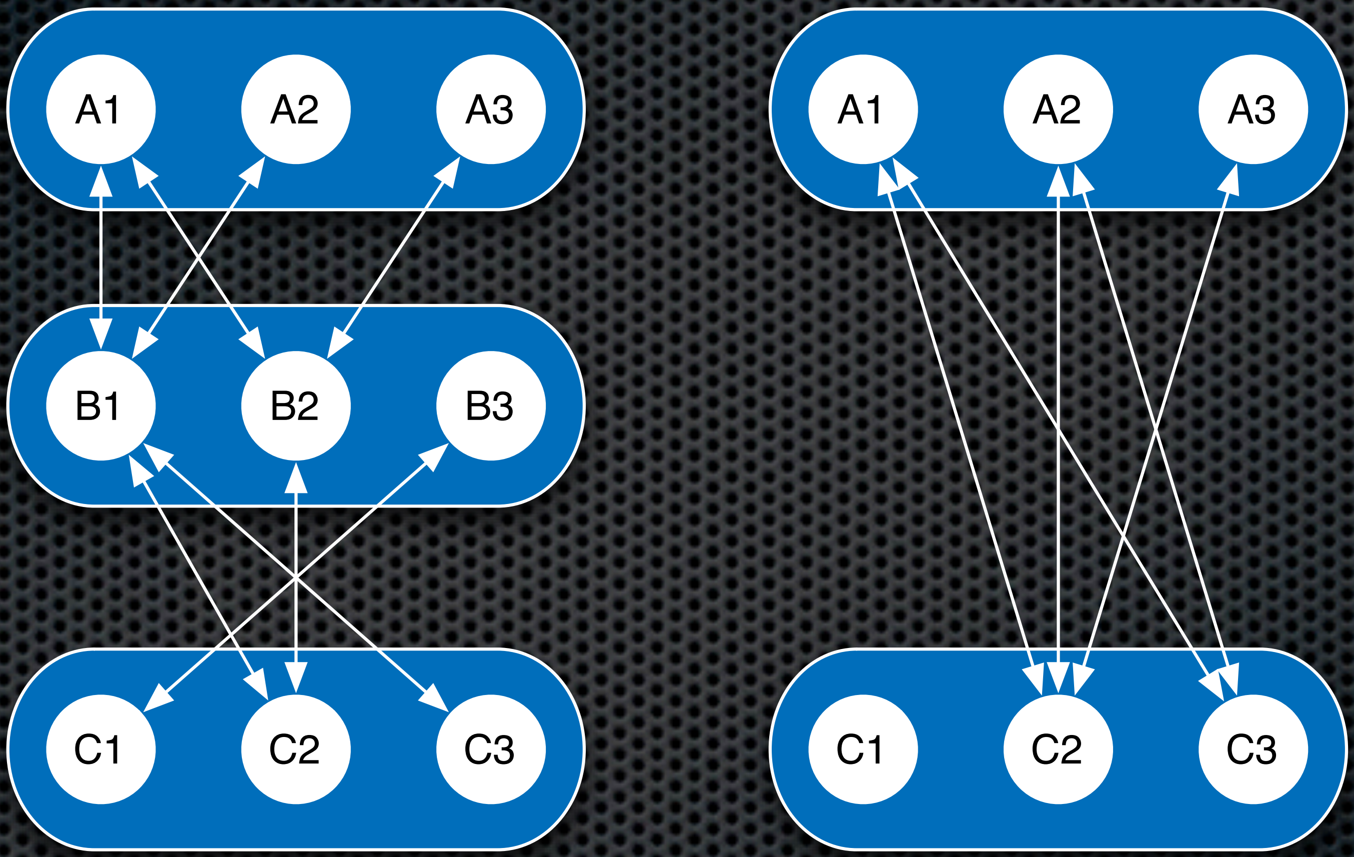
```
File   = {(F1),(F2),(F3)}
Dir    = {(D1),(D2)}
Time   = {(T1),(T2),(T3),(T4)}
root   = {(D1)}
now    = {(T4)}
path   = {(D2)}
parent = {(F1,D1),(D2,D1),(F2,D2)}
log    = {(T1,F1,D1),(T3,D2,D1),(T4,F2,D2)}
```

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

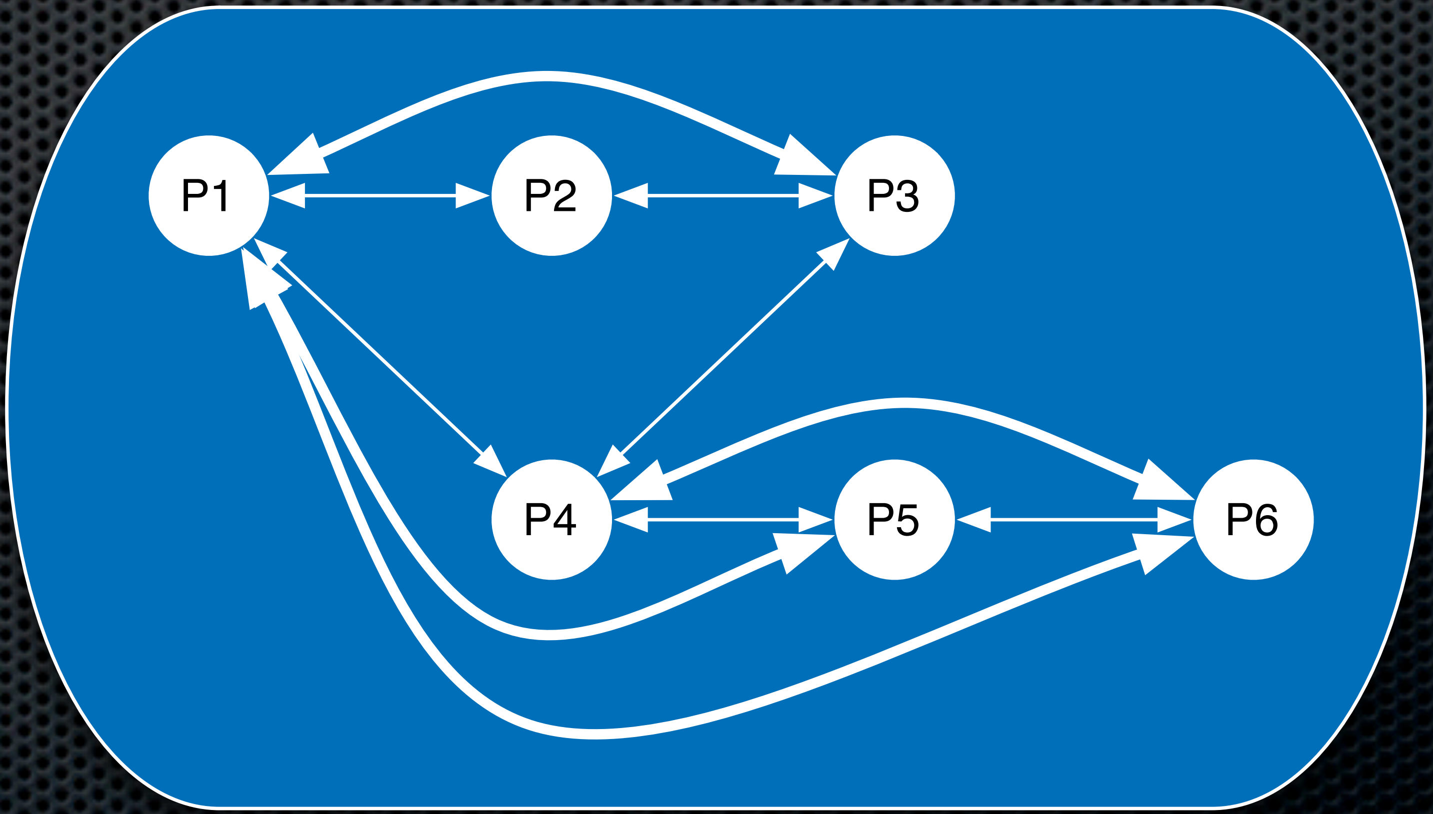
.	composition
+	union
++	override
&	intersection
-	difference
->	cartesian product
<:	domain restriction
:>	range restriction
~	converse
^	transitive closure
*	transitive-reflexive closure

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),...}
~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

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

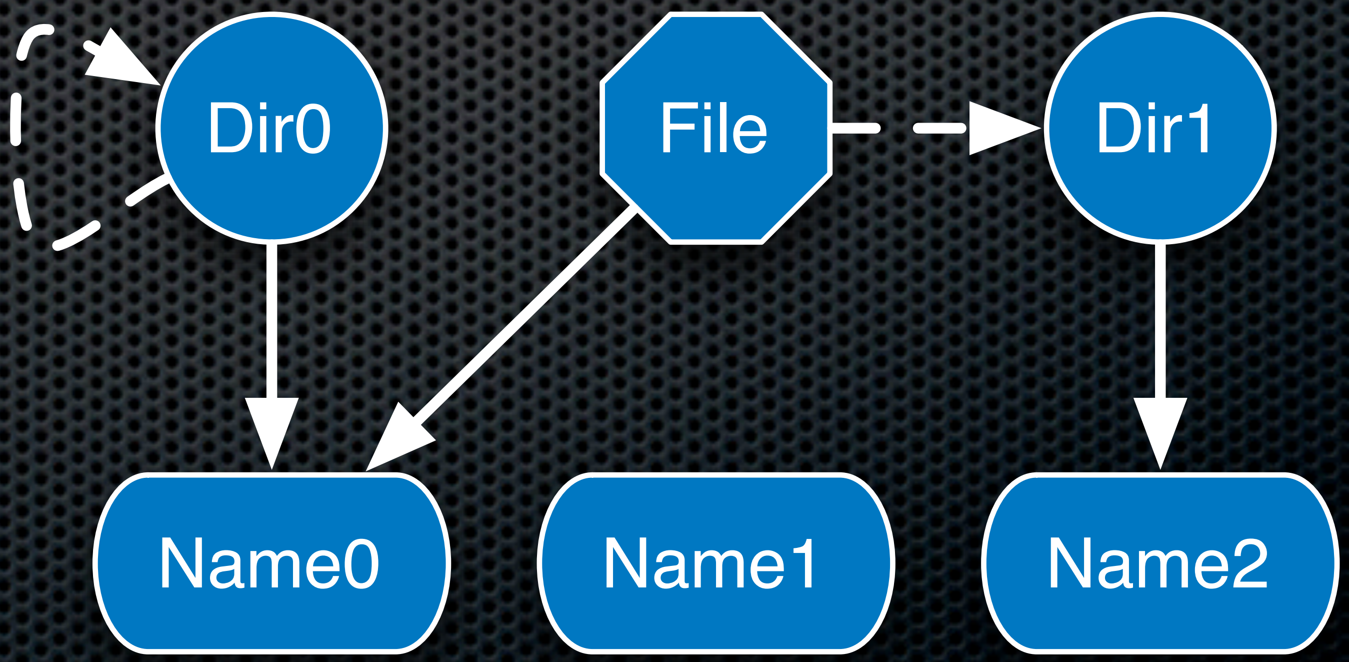
```
abstract sig Object {}
sig File, Dir extends Object {}
sig Name {}
sig FileSystem {
  objects: set Object,
  parent: objects -> lone (Dir & objects),
  name: objects lone -> one Name
}
```

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
```



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

- 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 *parent.r
}

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]]
}
```

State transition systems

- Impose ordering on the state
- Constrain initial state and valid transitions
- Bounded model checking on finite traces
- Be careful to add *nop* transitions to deadlock states

```
open util/ordering[FS]
fact {
   one first.objects and no first.parent
   all fs : FS, fs' : fs.next |
     some p,d : Dir | mkdir[fs,fs',p,d] or rmdir[fs,fs',d]
}
check { all fs : FS | inv[fs] } for 4 but 8 FS
```

Generator axioms

```
sig Set { elems : set Elem }
sig Elem {}

check {
  all s0, s1 : Set |
    some s2 : Set | s2.elems = s0.elems + s1.elems
}
```

- Counterexamples are found
- Set is not saturated enough
- A generator axiom can be enforced

Generator axioms

```
fact SetGenerator {
  some s : Set | no s.elems
  all s : Set, e : Elem |
    some s' : Set | s'.elems = s.elems + e
}
```

- Unfortunately the scope explodes
- To verify a model with *n* elements 2^n sets are needed
- Sometimes generator axioms force infinite scopes
- The risk of inconsistency is very high

Generator axioms

- As long as universal quantifiers (in runs) are *bounded* we can live without generator axioms
- *Bounded* means that the quantifier scope does not mention names of *generated* signatures

```
check {
  all s0, s1, s2 : Set |
    s0.elems + s1.elems = s2.elems =>
      s1.elems + s0.elems = s2.elems
}
```

Demos

- I'm my own grandpa
- Filesystem
- River crossing

Exercises

- Peterson's mutual exclusion algorithm
- Ebay
- Gossips

Peterson

- Model Peterson's mutual exclusion algorithm.
- Is Alloy adequate to check mutual exclusion? Deadlock absence? Liveness properties?

```
            while (true) {
idle     :    // non critical section
              flag[0] = 1; turn = 1;
wait     :    while (flag[1] && turn = 1);
critical :    // critical section
              flag[0] = 0;
            }
```

Ebay

- **Clients** can create **auctions** for **products** or **bid** on other clients' auctions.
- Define a simple Ebay model with at least the following invariants:
  - Clients do not bid on auctions for products they are also selling
  - All bids in an auction must be different
- Define and check the soundness and correctness of the following operations:
  - Create a new auction
  - Make a winning bid on a product

Gossips

- A number of **girls** initially know one distinct **secret** each. Each girl has access to a phone which can be used to **call** another girl to share their secrets. Each time two girls talk to each other they always exchange all secrets with each other. The girls can communicate only in pairs (no conference calls) but it is possible that different pairs of girls talk concurrently.
- How long does it take for n girls to know all of the secrets?

Kernel syntax

```
form := some expr
      | expr in expr
      | not form
      | form and form
      | some var : expr [, var : expr]* | form

expr := var
      | ~ expr
      | ^ expr
      | expr + expr
      | expr & expr
      | expr . expr
      | expr -> expr
      | { var : expr [, var : expr]* | form }
```

Kernel semantics

- Denotational semantics

```
F : form × binding → bool
E : expr × binding → relation
```

- Everything is a relation
- Relations are sets of tuples
- Tuples are sequences of atoms

```
binding  := var ↦ relation
relation := ℙ tuple
tuple    := <atom [, atom]*>
```

Kernel semantics

```
F(some r, Γ)  := E(r,Γ) ≠ {}
F(r in s, Γ)  := E(r,Γ) ⊆ E(s,Γ)
F(not f, Γ)   := ¬ F(f,Γ)
F(f and g, Γ) := F(f,Γ) ∧ F(g,Γ)
F(some x1:r1,…,xn:rn | f, Γ) := F(some {x1:r1,…,xn:rn | f}, Γ)
```

Kernel semantics

$$E(x, \Gamma) := \cup \{\Gamma(r) \mid name(r) = x\}$$

$$E(\sim r, \Gamma) := \{ \langle a_2, a_1 \rangle \mid \langle a_1, a_2 \rangle \in E(r,\Gamma) \}$$

$$E(\wedge\ r, \Gamma) := E(r,\Gamma) \cup E(r.r,\Gamma) \cup E(r.r.r,\Gamma) \cup \ldots$$

$$E(r + s, \Gamma) := E(r,\Gamma) \cup E(s,\Gamma)$$

$$E(r \,\&\, s, \Gamma) := E(r,\Gamma) \cap E(s,\Gamma)$$

$$E(r \,.\, s, \Gamma) :=$$

$$\{ \langle a_1,\ldots,a_{n-1},b_2,\ldots,b_m \rangle \mid \langle a_1,\ldots,a_n \rangle \in E(r,\Gamma) \wedge \langle b_1,\ldots,b_m \rangle \in E(s,\Gamma) \wedge a_n = b_1 \}$$

$$E(r \rightarrow s, \Gamma) :=$$

$$\{ \langle a_1,\ldots,a_n,b_1,\ldots,b_m \rangle \mid \langle a_1,\ldots,a_n \rangle \in E(r,\Gamma) \wedge \langle b_1,\ldots,b_m \rangle \in E(s,\Gamma) \}$$

$$E(\{x{:}r \mid f\}, \Gamma) :=$$

$$\{ \langle a \rangle \mid \langle a \rangle \in E(r,\Gamma) \wedge F(f, \Gamma \oplus x \mapsto \{\langle a \rangle\}) \}$$

$$E(\{x_1{:}r_1,\ldots,x_n{:}r_n \mid f\}, \Gamma) :=$$

$$\{ \langle a_1,\ldots,a_n \rangle \mid \langle a_1 \rangle \in E(r,\Gamma) \wedge \langle a_2,\ldots,a_n \rangle \in E(\{x_2{:}r_2,\ldots,x_n{:}r_n \mid f\}, \Gamma \oplus x_1 \mapsto \{\langle a_1 \rangle\}) \}$$

Type system

- Detect irrelevant (empty) expressions
- Low burden (no casts, overloading, ...)
- Syntactic robustness (subject reduction)
- Semantic independence (types are just warnings)
- Soundness (no false alarms)
- No completeness

Type system

- Semantic types: types are also relations
- The *bounding type* approximates the value of the expression from above
- The *relevance type* refines the bounding type given the context
- Computed by abstract interpretation
- Relevance resolves overloading: only one of the relations with the same name should be relevant

Type system

```
sig Name, Block {}
abstract sig Object { name : Name }
sig Dir  extends Object { contents : set Object }
sig File extends Object { contents : set Block }
sig Link extends Object { to : Object }
one sig Root extends Dir {}
fact {
  all o : Object | some o.name
  all b : Block | some b.name
  Root.contents in Dir
  all o : Object | some o.contents
  all o : Object | some o.(File <: contents)
  all d : Dir | d not in d.^contents.to
  no (Root.to + Root.contents.to)
  no (Root + Root.contents).to
}
```

Type system

- *Atomic types*: signatures that are not supertypes + for each non-abstract supertype T a *reminder* type $T
- Types are represented in disjunctive normal form as unions of products of atomic types

```
to : {<Link,Link>,<Link,File>,<Link,$Dir>,<Link,Root>}
```

- This canonical representation avoids subtype comparisons
- Relational operators can be used to compute types

Bounding types

```
Γ ⊢ r in s            ⇐ Γ ⊢ r : T ∧ Γ ⊢ s : U
Γ ⊢ not f             ⇐ Γ ⊢ f
Γ ⊢ f and g           ⇐ Γ ⊢ f ∧ Γ ⊢ g
Γ ⊢ some x:r | f      ⇐ Γ ⊢ r : T ∧ Γ⊕x↦T ⊢ f
Γ ⊢ x : T             ⇐ Γ(x) = T
Γ ⊢ ~r : ~T           ⇐ Γ ⊢ r : T
Γ ⊢ ^r : ^T           ⇐ Γ ⊢ r : T
Γ ⊢ r + s : T + U     ⇐ Γ ⊢ r : T ∧ Γ ⊢ s : U
Γ ⊢ r & s : T & U     ⇐ Γ ⊢ r : T ∧ Γ ⊢ s : U
Γ ⊢ r . s : T . U     ⇐ Γ ⊢ r : T ∧ Γ ⊢ s : U
Γ ⊢ r -> s : T -> U   ⇐ Γ ⊢ r : T ∧ Γ ⊢ s : U
Γ ⊢ { x:r | f } : T   ⇐ Γ ⊢ r : T ∧ Γ⊕x↦T ⊢ f
```

Bounding types

```
(Root + Root.contents).to =
(Root + Root.(contents_D+contents_F)).to

Root : {<R>}
contents_D : {<D,L>,<D,F>,<D,D>,<D,R>,
              <R,L>,<R,F>,<R,D>,<R,R>}
contents_F : {<F,B>}
to : {<L,L>,<L,F>,<L,D>,<L,R>}
contents_D+contents_F : {<D,L>,<D,F>,<D,D>,<D,R>,
                         <R,L>,<R,F>,<R,D>,<R,R>,<F,B>}
Root.(contents_D+contents_F) : {<L>,<F>,<D>,<R>}
Root + Root.(contents_D+contents_F) : {<L>,<F>,<D>,<R>}
(Root + Root.(contents_D+contents_F)).to : {<L>,<F>,<D>,<R>}
```

Relevance types

- The relevance type of an expression is relative to its context
- It is always a subset of the bounding type
- The same expression in two different contexts can have different relevance types
- A context is a term containing at most one hole, denoted by •
- Given context C and term t, C[t] denotes the term that results from filling the hole in C with t

Relevance types

$\Gamma \vdash C[\bullet \text{ in } s] \downarrow r : T \Leftarrow$

$\Gamma \vdash r : T \wedge \Gamma \vdash s : U \wedge \Gamma \vdash C \downarrow r \text{ in } s$

$\Gamma \vdash C[r \text{ in } \bullet] \downarrow s : T \,\&\, U \Leftarrow$

$\Gamma \vdash r : T \wedge \Gamma \vdash s : U \wedge \Gamma \vdash C \downarrow r \text{ in } s$

$\Gamma \vdash C[\bullet + s] \downarrow r : T \,\&\, B \Leftarrow$

$\Gamma \vdash r : T \wedge \Gamma \vdash s : U \wedge \Gamma \vdash C \downarrow r + s : B$

$\Gamma \vdash C[r + \bullet] \downarrow r : U \,\&\, B \Leftarrow$

$\Gamma \vdash r : T \wedge \Gamma \vdash s : U \wedge \Gamma \vdash C \downarrow r + s : B$

$\Gamma \vdash C[\bullet \,.\, s] \downarrow r : \{a \in T \mid \exists\, b \in U \mid a.b \in B\} \Leftarrow$

$\Gamma \vdash r : T \wedge \Gamma \vdash s : U \wedge \Gamma \vdash C \downarrow r \,.\, s : B$

$\Gamma \vdash C[r \,.\, \bullet] \downarrow s : \{b \in U \mid \exists\, a \in T \mid a.b \in B\} \Leftarrow$

$\Gamma \vdash r : T \wedge \Gamma \vdash s : U \wedge \Gamma \vdash C \downarrow r \,.\, s : B$

Relevance types

• . to ↓ Root + Root.($contents_D$+$contents_F$) : {<L>}
• + Root.($contents_D$+$contents_F$) ↓ Root : {}
Root + • ↓ Root.($contents_D$+$contents_F$) : {<L>}
Root . • ↓ ($contents_D$+$contents_F$) : {<R,L>}
• . ($contents_D$+$contents_F$) ↓ Root : {<R>}
• + $contents_F$ ↓ $contents_D$: {<R,L>}
$contents_D$ + • ↓ $contents_F$: {}

The real implementation

- Computations are performed on base instead of atomic types to get better error messages
- This forces subtype comparisons
- Empty bounding types are immediately reported as errors
- Expressions of mixed arity are rejected
- Relevance types are only used for resolving overloading: no syntactic robustness

Satisfiability

- Given propositional formula A find a model M (or valuation to boolean variables) such that $M \models A$
- Dual problem to validity: formula A is valid iff $\neg A$ is unsatisfiable
- The quintessential NP-complete problem: any problem in NP can be reduced to SAT in polynomial-time
- Naive approach using truth tables requires 2^n space for a formula with *n* variables

Conjunctive normal form

- A formula in CNF is conjunction of clauses, which are disjunctions of literals (variables or their negation)
- A formula in CNF can be represented as a set of sets of literals: it is true if it is empty; false if it has an empty set
- Any formula can reduced to CNF by applying De Morgan and distribution laws
- Unfortunately, a formula can grow exponentially
- Can be avoided by generating equisatisfiable normal forms instead

The DPLL algorithm

- Davis-Putnam-Logemann-Loveland algorithm
- Iteratively fix the value of a variable and simplify the CNF accordingly; backtrack if unsatisfiable

```
DPLL(A) | {} ≡ A     = False
        | {} ∈ A     = True
        | otherwise = DPLL(splitˣ(A)) ∧ DPLL(split¬ˣ(A))

splitˣ(A) = {c \ {-x} | c ∈ A, x ∉ c}
-(x) = ¬x
-(¬x) = x
```

- A is satisfiable iff ¬DPLL(A)

The DPLL algorithm

- Exponential in the worst case
- Highly dependent on the order variables are chosen
- Highly dependent on the data structures chosen for implementation
- Extra heuristics to avoid unnecessary branches, like *unit propagation* or *pure literal elimination*

```
DPLL(A) | {l} ∈ A = DPLL(split^l(A))
        | ∀ c ∈ A . -l ∉ c = DPLL(split^l(A))
```

Reducing Alloy to SAT

- Relational expressions are represented by matrices of boolean variables
- Relational operations are generalized to matrices
- Formulas yield boolean formulas over these variables

```
sig A {R : set B, S : set B}
sig B {}
run {R in R + S} for 3 but exactly 2 A, exactly 3 B
```

$R := \{ R_{1,1}, R_{1,2}, R_{1,3}, R_{2,1}, R_{2,2}, R_{2,3} \}$

$S := \{ S_{1,1}, S_{1,2}, S_{1,3}, S_{2,1}, S_{2,2}, S_{2,3} \}$

$R + S := \{ R_{1,1} \lor S_{1,1}, R_{1,2} \lor S_{1,2}, \ldots, R_{2,3} \lor S_{2,3} \}$

$R \text{ in } R + S := (R_{1,1} \Rightarrow R_{1,1} \lor S_{1,1}) \land \ldots \land (R_{2,3} \Rightarrow R_{2,3} \lor S_{2,3})$

Symmetry breaking

- Several optimizations are performed
- The most significant is *symmetry breaking*
- Since atoms are uninterpreted any instance is also valid for a permutation
- Symmetry-breaking constraints are conjoined to the analysis constraint
- For efficiency reasons it is not complete

Skolemization

- Since scope is finite, quantifiers could be handled by an expansion

```
sig A {}
run {some x : A | f} for 3 but exactly 3 A

A := { A1, A2, A3 }
some x : A | f ≡ f[A1/x] or f[A2/x] or f[A3/x]
```

- When an instance is generated it may not be clear which x made the formula true

Skolemization

- Free variables are implicitly existentially quantified
- Replace the bound variable by a new free variable

```
some x : A | f → (fx in A) and f[fx/x]

all x : A | some y : B | f →
(fy in A -> one B) and (all x : A | f[x.fy/y])
```

- Witnesses to bound variables are now generated
- It can handle some higher-order quantifications
- Generates smaller (equisatisfiable) formulas

Bibliography

- D. Jackson. *Software Abstractions: Logic, Language, and Analysis*. MIT Press, 2006.
- J. Edwards, D. Jackson, and E. Torlak. *A Type System for Object Models*. FSE. ACM, 2004.
- E. Torlak and D. Jackson. *Kodkod: A Relational Model Finder.* TACAS. Springer, 2007.
- J. Almeida, M. Frade, J. Pinto, S. de Sousa. *Rigorous Software Development: an Introduction to Program Verification*. Springer, 2011.