Alloy: Language and Analysis

Adapted from Greg Dennis and Rob Seater
Software Design Group, MIT

Alloy language & analysis

- Language = syntax for structuring specifications in logic
- Analysis = tool for finding solutions to logical formulas
  - searches for and visualizes counterexamples

“I'm My Own Grandpa” in Alloy

```
module grandpa

abstract sig Person {
  father: lone Man,
  mother: lone Woman
}

sig Man extends Person {
  wife: lone Woman
}

sig Woman extends Person {
  husband: lone Man
}

fact {
  no p: Person |
    p  in p.^(mother + father)
  wife = ~husband
}
```

```
assert noSelfFather {
 no m: Man | m = m.father
}

check noSelfFather

fun grandpas[p: Person] : set Person {
 p.(mother + father).father
}

pred ownGrandpa[p: Person] {
 p in grandpas[p]
}

run ownGrandpa for 4 Person
```

Can one's wife be his (grand)*mother?

```
module language/grandpa1
abstract sig Person {
  father: lone Man,
  mother: lone Woman
  }
sig Man extends Person {
  wife: lone Woman
  }
sig Woman extends Person {
  husband: lone Man
  }
fact {
  no p: Person | p in p.^(mother+father)
  wife = ~husband
  }
assert law {
       no wife & *(mother+father).mother

       }
check law
```

counterexample found!

Language: signatures

```
sig A {}
```
set of atoms A

```
sig A {}
sig B {}
```
disjoint sets A and B (no A & B)

```
sig A, B {}
```
same as above

```
sig B extends A {}
```
set B is a subset of A (B in A)

```
sig B extends A {}
sig C extends A {}
```
B and C are disjoint subsets of A (B in A && C in A && no B & C)

```
sig B, C extends A {}
```
same as above

```
abstract sig A {}
sig B extends A {}
sig C extends A {}
```
A partitioned by disjoint subsets B and C (no B & C && A = (B + C))

```
sig B in A {}
```
B is a subset of A – not necessarily disjoint from any other set

```
sig C in A + B {}
```
C is a subset of the union of A and B

```
one sig A {}
lone sig B {}
some sig C {}
```
A is a singleton set
B is a singleton or empty
C is a non-empty set

The "grandpa" example

```
abstract sig Person {
  . . .
}

sig Man extends Person {
  . . .
}

sig Woman extends Person {
  . . .
}
```

- all men and women are persons
- no person is both a man and a woman
- all persons are either men or women

Language: fields

```
sig A {f: e}
```
f is a binary relation with domain A and range given by expression e
f is constrained to be a function
(f: A -> one e) or (all a: A | a.f: e)

```
sig A {
  f1: one  e1,
  f2: lone e2,
  f3: some e3,
  f4: set  e4
}
```
(all a: A | a.fn : m e)

```
sig A {f, g: e}
```
two fields with same constraints

```
sig A {f: e1 m -> n e2}
```
(f: A -> (e1 m -> n e2)) or
(all a: A | a.f : e1 m -> n e2)

```
sig Book {
  names: set Name,
  addrs: names -> Addr
}
```
dependent fields
(all b: Book | b.addrs: b.names -> Addr)

grandpa: fields

```
abstract sig Person {
  father: lone Man,
  mother: lone Woman
}

sig Man extends Person {
  wife: lone Woman
}

sig Woman extends Person {
  husband: lone Man
}
```

- fathers are men and everyone has at most one
- mothers are women and everyone has at most one
- wives are women and every man has at most one
- husbands are men and every woman has at most one

Language: facts

```
fact { F }
fact f { F }
sig S { ... }{ F }
```

facts introduce constraints that are assumed to always hold

```
sig Host {}
sig Link {from, to: Host}

fact {all x: Link | x.from != x.to}
```
no links from a host to itself

```
fact noSelfLinks {all x: Link | x.from != x.to}
```
same as above

```
sig Link {from, to: Host} {from != to}
```
same as above, with implicit 'this.'

grandpa: fact

```
fact {
  no p: Person |
    p in p.^(mother + father)
  wife = ~husband
}
```

- no person is his or her own ancestor
- a man's wife has that man as a husband
- a woman's husband has that woman as a wife

Language: functions

```
fun f[x1: e1, ..., xn: en] : e { E }
```

functions are named expression with declaration parameters and a declaration expression as a result invoked by providing an expression for each parameter

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

fun lookup[b: Book, n: Name] : set Addr {
  b.addr[n]
}

fact everyNameMapped {
  all b: Book, n: Name | some lookup[b, n]
}
```

Language: predicates

```
pred p[x1: e1, ..., xn: en] { F }
```

named formula with declaration parameters

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

pred contains[b: Book, n: Name, d: Addr] {
  n->d in b.addr
}

fact everyNameMapped {
  all b: Book, n: Name |
    some d: Addr | contains[b, n, a]
}
```

grandpa: function and predicate

```
fun grandpas[p: Person] : set Person {
  p.(mother + father).father
}

pred ownGrandpa[p: Person] {
  p in grandpas[p]
}
```

- a person's grandpas are the fathers of one's own mother and father

Language: “receiver” syntax

```
fun f[x: X, y: Y, ...] : Z {...x...}
fun X.f[y:Y, ...] : Z {...this...}
```

```
f[x, y, ...]
x.f[y, ...]
```

```
pred p[x: X, y: Y, ...]  {...x...}
pred X.p[y:Y, ...] {...this...}
```

```
p[x, y, ...]
x.p[y, ...]
```

```
fun Person.grandpas : set Person {
  this.(mother + father).father
}

pred Person.ownGrandpa {
  this in this.grandpas
}
```

Language: assertions

```
assert a { F }
```

constraint intended to follow from facts of the model

```
sig Node {
  children: set Node
}

one sig Root extends Node {}

fact {
  Node in Root.*children
}

// invalid assertion:
assert someParent {
  all n: Node | some children.n
}

// valid assertion:
assert someParent {
  all n: Node - Root | some children.n
}
```

Language: check command

```
assert a { F }
check a scope
```

instructs analyzer to search for counterexample to assertion within scope

if model has facts M finds solution to M && !F

check a
top-level sigs bound by 3

check a **for** *default*
top-level sigs bound by default

check a **for** *default* **but** *list*
default overridden by bounds in list

check a **for** *list*
sigs bound in list, invalid if any unbound

```
abstract sig Person {}
sig Man extends Person {}
sig Woman extends Person {}
sig Grandpa extends Man {}

check a
check a for 4
check a for 4 but 3 Woman
check a for 4 but 3 Man, 5 Woman
check a for 4 Person
check a for 4 Person, 3 Woman
check a for 3 Man, 4 Woman
check a for 3 Man, 4 Woman, 2 Grandpa

// invalid:
check a for 3 Man
check a for 5 Woman, 2 Grandpa
```

grandpa: assertion check

```
fact {
  no p: Person | p in p.^(mother + father)
  wife = ~husband
}

assert noSelfFather {
  no m: Man | m = m.father
}

check noSelfFather
```

- sanity check
- command instructs analyzer to search for counterexample to *noSelfFather* within a scope of at most 3 *Persons*
- *noSelfFather* assertion follows from fact

Language: run command

```
pred p[x: X, y: Y, ...] { F }
run p scope
```

instructs analyzer to search for instance of predicate within scope

if model has facts M, finds solution to M && (some x: X, y: Y, ... | F)

```
fun f[x: X, y: Y, ...] : R { E }
run f scope
```

instructs analyzer to search for instance of function within scope

if model has facts M, finds solution to M && (some x: X, y: Y, ..., result: R | result = E)

grandpa: predicate simulation

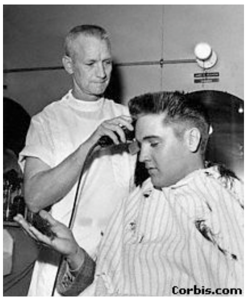
```
fun grandpas[p: Person] : set Person {
  p.(mother + father).father
}

pred ownGrandpa[p: Person] {
  p in grandpas[p]
}

run ownGrandpa for 4 Person
```

- command instructs analyzer to search for configuration with at most 4 people in which a man is his own grandfather

Example of an inconsistent world *barber paradox*




```
sig Man {shaves: set Man}
one sig Barber extends Man {}
fact {
     Barber.shaves = {m: Man | m not in m.shaves}
}
```

More on "grandpa"

```
module grandpa

abstract sig Person {
      father: lone Man,
      mother: lone Woman
      }

sig Man extends Person { wife: lone
Woman }

sig Woman extends Person {
husband: lone Man }

fact Biology { no p: Person | p in
p.^(mother+father) }

fact Terminology { wife = ~husband }
```

```
fact SocialConvention {
      no wife & *(mother+father).mother
//no wife can be one's mother, or any of the
//female ancestors
      no husband & *(mother+father).father
//same for husband
      }

fun grandpas [p: Person]: set Person {
      let parent = mother + father + father.wife +
      mother.husband | p.parent.parent & Man
      }

pred ownGrandpa [m: Man] { m in grandpas[m] }

run ownGrandpa for 4 Person
```

notice that there is no constraint on father/mother being also husband/wife….

More on "grandpa"

Man0 has a father whose wife has a mother (Woman1)
==> Woman1 is Man0's grandma
==> a grandma's husband is a grandpa

==> Man0 is his grandpa!!!