Alloy

Alloy's Resources

- The book is the main resource
- Many examples, and online tutorial
- Everything (code, documentation, tool) on:

http://alloy.mit.edu/

"Some slides are adapted from
Greg Dennis and Rob Seater
Software Design Group, MIT"

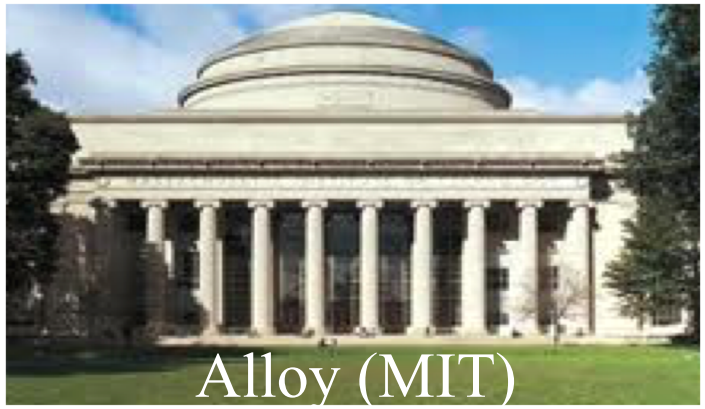
Alloy

- Alloy is a formal notation for specifying models of systems and software
  - ✓ Looks like a declarative OO language
  - ✓ But also has a strong mathematical foundation

- Alloy comes with a supporting tool to simulate specifications and perform property verification

Why is Alloy Important?

- Alloy is declarative
- It has been created to offer an expressive power similar to the Z language as well as the strong automated analysis of the SMV model checker



Alloy for Declarative Modeling

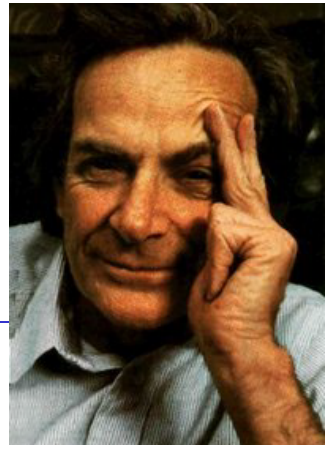
- Alloy is used for abstractions and conceptual modeling in a declarative manner
- ***Declarative approach to programming and modeling****:*
  - "declarative programming is a programming paradigm that expresses the logic of a computation without describing its control flow"
  - "describing *what* the program should accomplish, rather than describing *how* to go about accomplishing it"

Alloy's Applications for SE

- In RE Alloy can be used to
  - formally describe the domain and its properties, or
  - operations that the machine has to provide
- In software design to formally model components and their interactions

Alloy for Automated Analysis

- Any real-life system has a set of properties and constraints
- Specification analysis can help check whether or not the properties will be satisfied by the system and the constraints will never be violated
- "*The first principle is that you must not fool yourself, and you are the easiest person to fool.*"
  ~ Richard P. Feynman

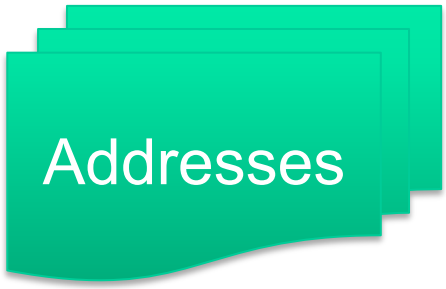


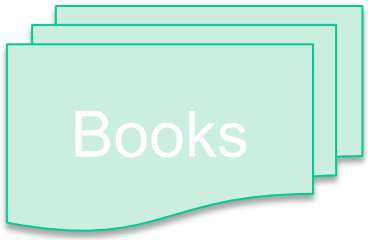
What is Alloy?

- First order predicate logic + relational calculus
- Carefully chosen subset of relational algebra
  - uniform model for individuals, sets and relations
  - no higher-order relations
- Almost no arithmetic
- Modules and hierarchies
- Suitable for small, exploratory specifications
- Powerful and fast analysis tool
  - is this specification satisfiable?
  - is this predicate true?

Let's Get Started!

- Imagine that we are asked to model a very simple address book
- The books that contain a bunch of addresses linked to the corresponding names





What to model?

How to model?

AddressBook in Alloy

- Name and Addr are two types of entities
- Book is another type of entity
- addr is linking Name to Addr within the context of Book (it is a ternary relation `b->n->a`)
- The keyword “lone”: each Name can correspond to at most one Addr

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

Logic: Everything is a Relation

- Sets are unary (1 column) relations

```
Name = {(N0),      Addr = {(A0),      Book = {(B0),
        (N1),              (A1),              (B1)}
        (N2)}              (A2)}
```

- Scalars are singleton sets

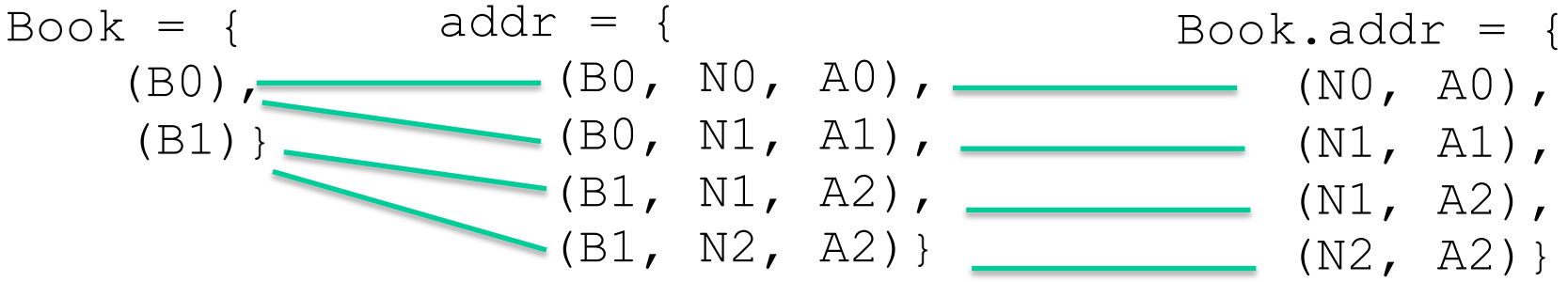
```
myName   = {(N1)}
yourName = {(N2)}
myBook   = {(B0)}
```

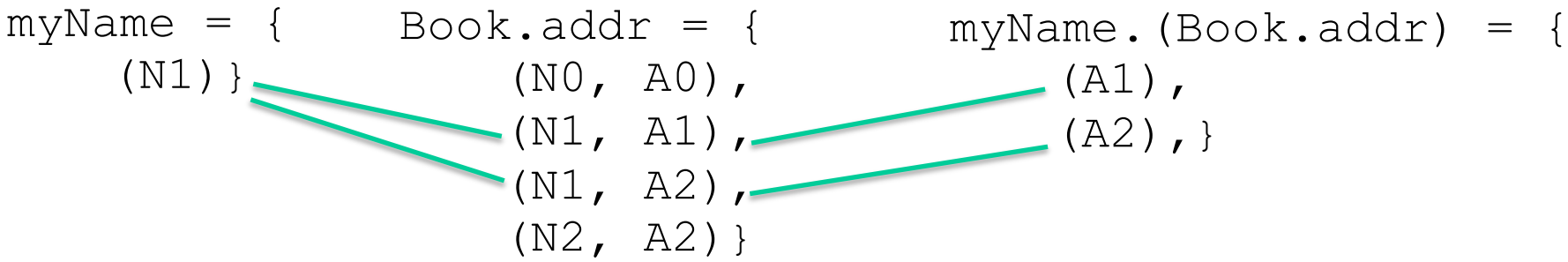
- Ternary relation

```
addr = {(B0, N0, A0),
        (B0, N1, A1),
        (B1, N1, A2),
        (B1, N2, A2)}
```

what is myName.(Book.addr)?

myName.(Book.addr)





Static Analysis

- Let's open the Alloy tool and play a bit!
- We add an empty predicate "show" by using keyword "pred" to find the instances of the entities involved in the modeling
- In this case, the state exploration is limited to 3 for each entity except Book that is set to 1

```
pred show {}

run show  for 3  but 1 Book
```

Static Analysis

- Adding a constraint on the number of (Name,Address) relations in a given book

```
pred show [b: Book] {
    #b.addr > 1
    }
```

- Constraining the number of different addresses that appear in the book

```
pred show [b: Book] {
   #b.addr > 1
   #Name.(b.addr) > 1 }
```

- Can

```
pred show [b: Book] {
   #b.addr > 1
   some n: Name | #n.(b.addr) > 1}
```

Dynamic Analysis

- Let's model some operations.
  - A predicate that adds an address and name to a book

```
pred add [b, b': Book, n: Name, a: Addr] {
    b'.addr = b.addr + n -> a
}
```

  - b and b' model two versions of the book, respectively before and after the operation
- In the tool we can invoke an operation

```
pred showAdd [b, b': Book, n: Name, a: Addr] {
   add[b, b', n, a]
   #Name.(b'.addr) > 1
}

run showAdd
```

Dynamic Analysis

- Deleting a name from the address book

```
pred del [b, b': Book, n: Name] {
      b'.addr = b.addr - n->Addr
}
```

Assertions and counterexamples

- What happens if we run a delete after an add predicate? Will this take us to the initial state?

```
assert delUndoesAdd {
      all b, b', b'': Book, n: Name, a: Addr |
            add[b, b', n, a] and del[b', b'', n]
            implies
            b.addr = b''.addr
}
check delUndoesAdd for 3
```

- Counterexample is a scenario in which the assertion is violated
- While checking an assertion, Alloy searches for counterexamples
- Do we find a counterexample in this case?

Resolving the Counterexample

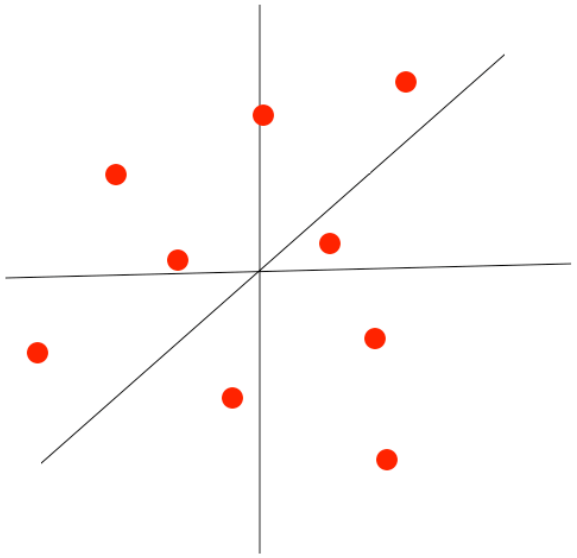
- … Yes! When add tries to add a name that already exists and delete will delete the name and the corresponding address
- Here is how we can solve the problem:

```
assert delUndoesAdd {
      all b, b', b'': Book, n: Name, a: Addr |
            no n.(b.addr) and add[b, b', n, a]
                          and del[b', b'', n]
            implies
            b.addr = b''.addr
}
```

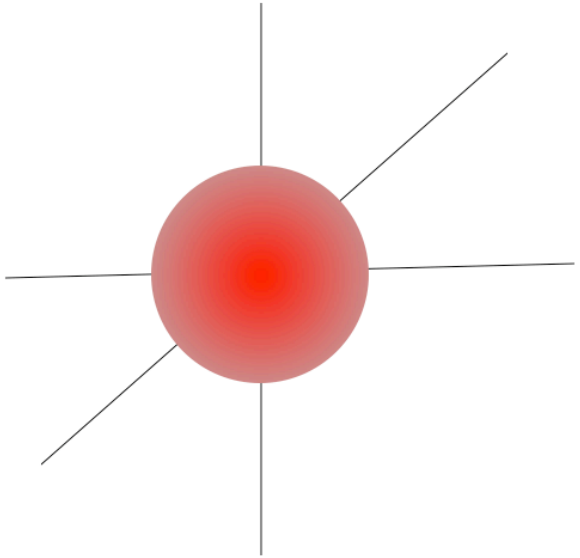
Counterexample in Alloy

- The analysis by Alloy is always bounded to a defined scope
- This means that there is NO guarantee that the assertions always hold if no counterexample is found in a certain scope!

Testing vs Counterexample



testing:
a few cases of arbitrary size



scope-complete:
all cases within a small bound

Functions: An Example

- Can we get the addresses in the book?

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

- Example of usage of lookup

```
assert addLocal {
      all b, b': Book, n, n': Name, a: Addr |
            add[b, b', n, a] and n != n'
            implies
            lookup[b, n'] = lookup[b', n']
}
check addLocal for 3 but 2 Book
```

Alloy: Syntax and Semantics

- Alloy is a language
  - It has a syntax *how do I write a right specification?*
  - It has a semantics *what does it mean?*
- In a programming language
  - Syntax defines correct programs (i.e., allowed programs)
  - Semantics defines the meaning of a program as its possible (many?) computations
- In Alloy
  - Syntax as usual...
  - Semantics defines the meaning of a specification as the collection of its models, i.e., of the worlds that make our Alloy description true

Alloy = logic + language + analysis

- Logic
  - first order logic + relational calculus
- Language
  - syntax for structuring specifications in the logic
- Analysis
  - shows bounded snapshots of the world that satisfy the specification
  - bounded exhaustive search for counterexample to a claimed property using SAT

Logic Elements

- Relations
- Constants
- Set Operators
- Join Operators
- Quantifiers and Boolean Operators
- Set and Relation Declaration
- If and let

Logic: relations of atoms

- Atoms are Alloy's primitive entities
  - indivisible, immutable, uninterpreted
- Relations associate atoms with one another
  - set of tuples, tuples are sequences of atoms

Logic: everything is a relation

- Sets are unary (1 column) relations

```
Name = {(N0),      Addr = {(A0),      Book = {(B0),
        (N1),              (A1),              (B1)}
        (N2)}              (A2)}
```

- Scalars are singleton sets

```
myName   = {(N1)}
yourName = {(N2)}
myBook   = {(B0)}
```

- Ternary relation

```
addr = {(B0, N0, A0),
        (B0, N1, A1),
        (B1, N1, A2),
        (B1, N2, A2)}
```

Relations

- Relation = set of ordered n-tuples of atoms
  - n is called the *arity* of the relation
- Relations in Alloy are typed
  - Determined by the declaration of the relation
  - Example: a relation with type
    `Person -> String`
    only contains pairs
    - whose first component is a Person
    - and whose second component is a String

Logic: constants

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

```
Name = {(N0), (N1), (N2)}
Addr = {(A0), (A1)}

none = {}
univ = {(N0), (N1), (N2), (A0), (A1)}
iden = {(N0,N0), (N1,N1), (N2,N2), (A0,A0),
  (A1,A1)}
```

Logic: set operators

+	*union*
&	*intersection*
-	*difference*
in	*subset*
=	*equality*

```
Name  = {(N0), (N1), (N2)}
Alias = {(N1), (N2)}
Group = {(N0)}
RecentlyUsed = {(N0), (N2)}

Alias + Group = {(N0), (N1), (N2)}
Alias & RecentlyUsed  = {(N2)}
Name - RecentlyUsed   = {(N1)}
RecentlyUsed in Alias = false
RecentlyUsed in Name  = true
Name = Group + Alias  = true
```

```
greg = {(N0)}
rob = {(N1)}

greg + rob    = {(N0), (N1)}
greg = rob    = false
rob in none   = false
```

```
cacheAddr = {(N0, A0), (N1, A1)}
diskAddr = {(N0, A0), (N1, A2)}

cacheAddr + diskAddr = {(N0, A0), (N1, A1), (N1, A2)}
cacheAddr & diskAddr = {(N0, A0)}
cacheAddr = diskAddr = false
```

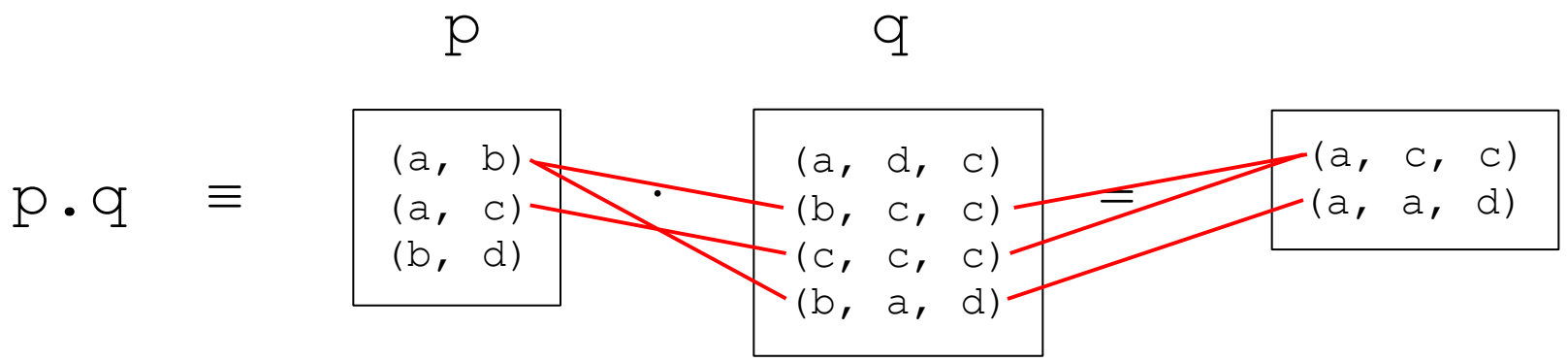
Logic: product operator

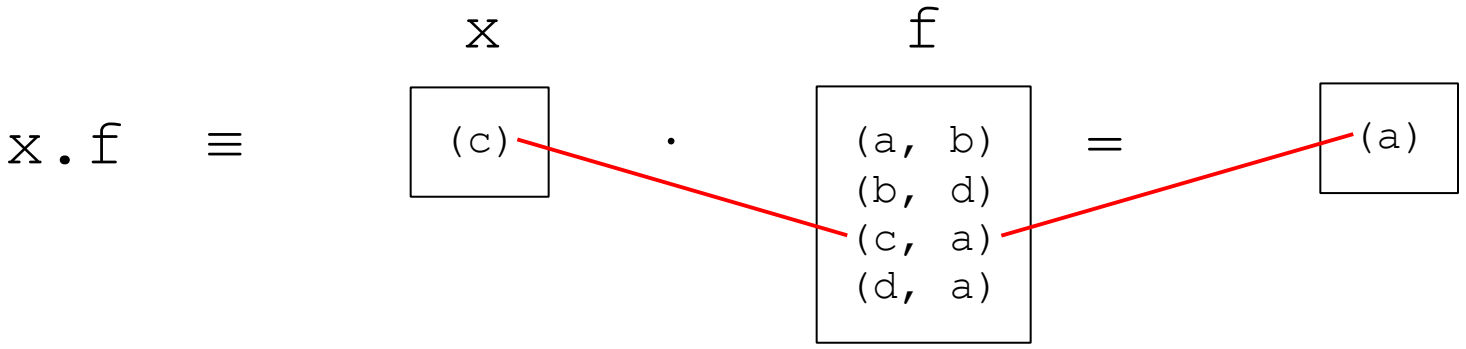
-> *cross product*

```
Name = {(N0), (N1)}
Addr = {(A0), (A1)}
Book = {(B0)}

Name->Addr = {(N0, A0), (N0, A1),
              (N1, A0), (N1, A1)}
Book->Name->Addr =
  {(B0, N0, A0), (B0, N0, A1),
   (B0, N1, A0), (B0, N1, A1)}
```

```
b  = {(B0)}
b' = {(B1)}
address  = {(N0, A0), (N1, A1)}
address' = {(N2, A2)}

b->b' = {(B0, B1)}

b->address + b'->address' =
  {(B0, N0, A0), (B0, N1, A1), (B1, N2, A2)}
```

Logic: relational composition dot join





in db join the match of columns is by name, not by position
and the matching column is not dropped

Logic: join operators

`.`	*dot join*
`[]`	*box join*

```
e1[e2] = e2.e1
a.b.c[d] = d.(a.b.c)
```

```
Book = {(B0)}
Name = {(N0), (N1), (N2)}
Addr = {(A0), (A1), (A2)}
Host = {(H0), (H1)}

myName = {(N1)}
myAddr = {(A0)}

address = {(B0, N0, A0), (B0, N1, A0), (B0, N2, A2)}
host = {(A0, H0), (A1, H1), (A2, H1)}

Book.address = {(N0, A0), (N1, A0), (N2, A2)}
Book.address[myName] = {(A0)}
Book.address.myName = {}

host[myAddr] = {(H0)}
address.host = {(B0, N0, H0), (B0, N1, H0), (B0, N2, H1)}
```

Logic: unary operators

~ *transpose*
^ *transitive closure*
* *reflexive transitive closure*

apply only to binary relations

```
^r = r + r.r + r.r.r + …
*r = iden + ^r
```

```
Node = {(N0), (N1), (N2), (N3)}
next = {(N0, N1), (N1, N2), (N2, N3)}   a list

~next = {(N1, N0), (N2, N1), (N3, N2)}
^next = {(N0, N1), (N0, N2), (N0, N3),
         (N1, N2), (N1, N3),
         (N2, N3)}
*next = {(N0, N0), (N0, N1), (N0, N2), (N0, N3),
         (N1, N1), (N1, N2), (N1, N3),
         (N2, N2), (N2, N3), (N3, N3)}
```

```
first = {(N0)}
rest = {(N1), (N2), (N3)}

first.^next = rest
first.*next = Node
```

Logic: restriction and override

`<:`	*domain restriction*
`:>`	*range restriction*
`++`	*override*

Apply to any relations (normally binary)

```
p ++ q =
p - (domain[q] <: p) + q
```

```
Name    = {(N0), (N1), (N2)}
Alias   = {(N0), (N1)}
Addr    = {(A0)}
address = {(N0, N1), (N1, N2), (N2, A0)}

address :> Addr  = {(N2, A0)}
Alias <: address = {(N0, N1), (N1, N2)}
address :> Alias = {(N0, N1)}

workAddress = {(N0, N1), (N1, A0)}
address ++ workAddress = {(N0, N1), (N1, A0), (N2, A0)}
```

```
m' = m ++ (k -> v)
```
update map m with key-value pair (k, v)

Logic: restriction and override

- Examples of usage of these operators with non binary relations
  - `addr = {(B0, N1, A2), (B1, N1, A2), (B2, N0, A1), (B2, N1, A0)}`
  - `{B2} <: addr = {(B2, N0, A1), (B2, N1, A0)}`
  - `addr :> {A2} = {(B0, N1, A2), (B1, N1, A2)}`
  - `addr ++ {(B0, N1, A0)} = {(B0, N1, A0), (B1, N1, A2), (B2, N0, A1), (B2, N1, A0)}`
  - `addr ++ {(B0, N0, A0)} = {(B0, N0, A0), (B1, N1, A2), (B2, N0, A1), (B2, N1, A0)}`
  - `addr :> {N1} = {}`

Logic: boolean operators

`!`	`not`	*negation*
`&&`	`and`	*conjunction*
`||`	`or`	*disjunction*
`=>`	`implies`	*implication*
`,`	`else`	*alternative*
`<=>`	`iff`	*bi-implication*

four equivalent constraints:

```
F => G else H

F implies G else H

(F && G) || ((!F) && H)

(F and G) or ((not F) and H)
```

Logic: quantifiers


```
all x: e | F
all x: e1, y: e2 | F
all x, y: e | F
all disj x, y: e | F
```

`all`	*F holds for every x in e*
`some`	*F holds for at least one x in e*
`no`	*F holds for no x in e*
`lone`	*F holds for at most one x in e*
`one`	*F holds for exactly one x in e*

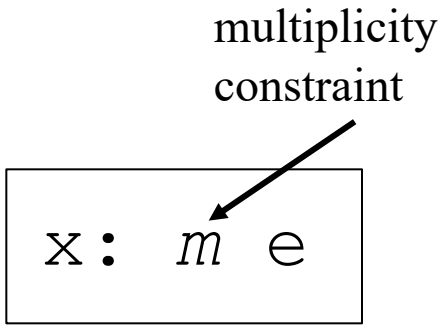
```
some n: Name, a: Address | a in n.address
```
some name maps to some address — address book not empty

```
no n: Name | n in n.^address
```
no name can be reached by lookups from itself — address book acyclic

```
all n: Name | lone a: Address | a in n.address
```
every name maps to at most one address — address book is functional

```
all n: Name | no disj a, a': Address | (a + a') in n.address
```
no name maps to two or more distinct addresses — same as above

Logic: relation declarations



`set`	*any number*
`one`	*exactly one* — default
`lone`	*zero or one*
`some`	*one or more*

`RecentlyUsed: set Name`
RecentlyUsed is a subset of the set Name

`senderAddress: Addr`
senderAddress is a singleton subset of Addr

`senderName: lone Name`
senderName is either empty or a singleton subset of Name

`receiverAddresses: some Addr`
receiverAddresses is a nonempty subset of Addr

Logic: relation declarations


```
r: A m -> n B
```

```
(r: A m -> n B) <=>
   ((all a: A | n a.r) and (all b: B | m r.b))
```

```
workAddress: Name -> lone Addr
```
each name refers to at most one work address

```
homeAddress: Name -> one Addr
```
each name refers to exactly one home address

```
members: Group lone -> some Addr
```
address belongs to at most one group
and group contains at least one address

Logic: set definitions


```
{x1: e1, x2: e2, ..., xn: en | F}
```

```
{n: Name | no n.^address & Addr}
```
set of names that don't resolve to any actual addresses

```
{n: Name, a: Address | n -> a in ^address}
```
binary relation mapping names to reachable addresses

Logic: if and let

```
f implies e1 else e2
let x = e | formula
let x = e | expression
```

four equivalent constraints:

```
all n: Name |
  (some n.workAddress
     implies n.address = n.workAddress
     else n.address = n.homeAddress)

all n: Name |
  let w = n.workAddress, a = n.address |
    (some w implies a = w else a = n.homeAddress)

all n: Name |
  let w = n.workAddress |
    n.address = (some w implies w else n.homeAddress)

all n: Name |
  n.address = (let w = n.workAddress |
    (some w implies w else n.homeAddress))
```

Logic: cardinalities

`#r`	*number of tuples in r*
`0,1,...`	*integer literal*
`+`	*plus*
`-`	*minus*

`=`	*equals*
`<`	*less than*
`>`	*greater than*
`=<`	*less than or equal to*
`>=`	*greater than or equal to*

```
sum x: e | ie
```
sum of integer expression ie for all singletons x drawn from e

```
all b: Bag | #b.marbles =< 3
```
all bags have 3 or less marbles

```
#Marble = sum b: Bag | #b.marbles
```
the sum of the marbles across all bags equals the total number of marbles

Alloy Language (1)

- An Alloy document is a “source code” unit
- It may contain:
  - **Signatures**: define types and relationships
  - **Facts**: properties of models (constraints!)
  - **Predicates/functions**: reusable expressions
  - **Assertions**: properties we want to check
  - **Commands**: instruct the Alloy Analyzer which assertions to check, and how

A predicate is **run** to find a world that satisfies it

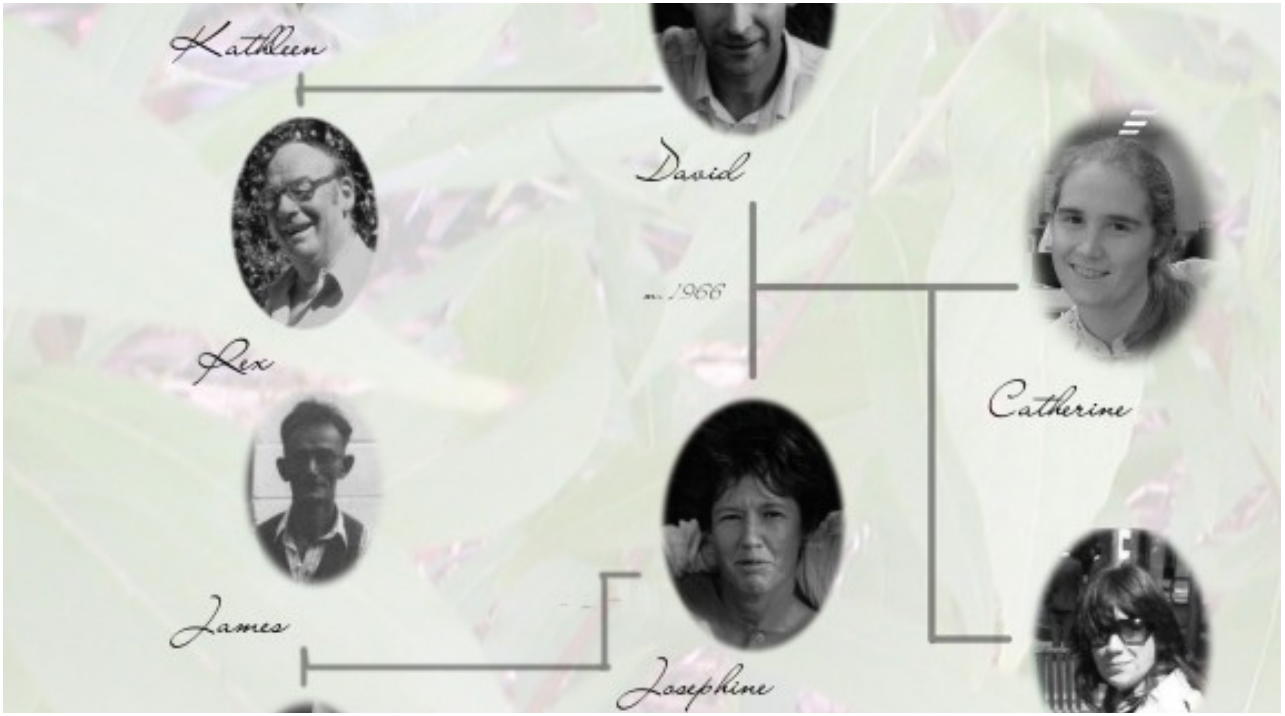
An assertion is **checked** to find a counterexample

Alloy Language (2)

- Signatures, predicates, facts and functions tell how correct worlds (models) are made
  - When the Alloy analyzer tries to build a model, it must comply with them
- Assertions and commands tell which kind of checks must be performed over these worlds
  - E.g., “find, among all the models, one that violates this assertion”
- Of course, the Analyzer cannot check all the (usually infinite) models of a specification
  - So you must also tell the Analyzer how to limit the search

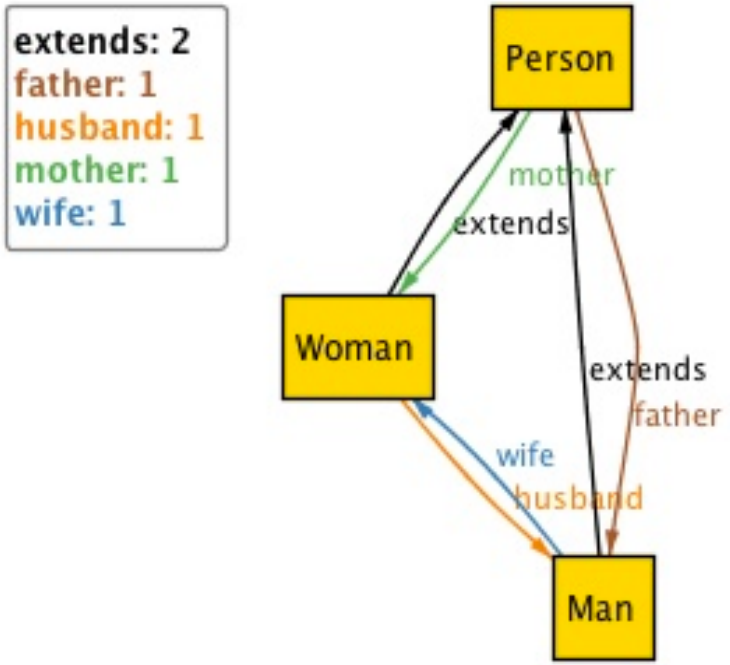
Family Relationship

- There are notions like *Person*
- There are relationships like *father, mother, wife, …*



Initial Alloy Model

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

sig Man extends Person {
      wife: lone Woman
      }

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



Meta-Model

Signatures and Fields

- Signatures by using keyword '*sig*' represents a set of atoms
- The ***fields*** of a signature are relations whose domain is a subset of the signature
- "extends" keyword is used to declare a subset of a signature
- An *abstract* signature has no element except those belonging to the signatures that extend it
- *m* sig A {} – m is to declare the multiplicity, the number of elements of A

Relations

- Relations are declared as fields of signatures

  sig A {f : e}
- A is the domain
- e is the range
- You can add multiplicity in defining the relation

  sig A {f : **m** e}
- The default multiplicity is ***one***

Can you be your own Grandpa?

- Biologically may be not, but terminologically let us check out this song:
  - I'm my own Grandpa!
    http://www.youtube.com/watch?v=eYlJH81dSiw
    (with family tree)

    https://www.youtube.com/watch?v=gkiOm-vmpcY&list=RDgkiOm-vmpcY
    (with muppets)

Adding Predicates and Functions

- The function *grandpas* returns the grandfathers of a given person P

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

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

This can be written also as `p in grandpas[p]`

Running the Model

- The model is executed by trying to see if a predicate is valid for a set of instances of the meta-model

```
run ownGrandpa for 4 Person
```

Fixing some problems in the spec

- “No one can be his\her own ancestor”
- “If X is husband of Y, then Y is the wife of X”

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

Adding Assertions

- Check if there is no man who is also his father
- Assertions are to find counterexamples which are the instances of a model

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

check NoSelfFather
```

Adding more facts to the model

- Facts can be named for readability purposes

```
fact SocialConvention {
      no ((wife+husband) & ^(mother+father))
      }
```

- It is not possible that someone has a wife or husband who is also one of his/her ancestors
- Still, one can have common ancestors!

Facts are different from Predicate?

- They are both used to express constraints

- Facts must hold globally for any atom and relation in the model
- Predicates have to be invoked

How to avoid that wives and husbands have common ancestors?


```
fact SocialConvention2 {
  all m: Man, w: Woman |
   (m.wife = w and w.husband = m)
   implies
   not( m in w.^(father+mother) or
        w in m.^(father+mother) )
}
```

Does this solve the common ancestor problem?
To which fact is this equivalent?

Answer: SocialConvention

Another solution


```
fun ancestors [p: Person]: set Person {
  p.^(mother+father)
}
fact noCommonAncestors {
  all p1: Man, p2: Woman |
    p1->p2 in wife
    implies
    ancestors[p1] & ancestors[p2] = none
}
```

Check yourself if it works

Other solution (?)


```
fact SocialConvention4 {
  no ((wife+husband) &
      (mother+father).~(mother+father))
}
```

In fact, `noCommonAncestor` is *stronger* than `SocialConvention4`

”Proof” that `noCommonAncestor` is stronger than `SocialConvention4`


```
pred noCommonAncestors {
  all p1: Man, p2: Woman |
    p1->p2 in wife
    implies
    ancestors[p1] & ancestors[p2] = none
}
```

Notice the declaration

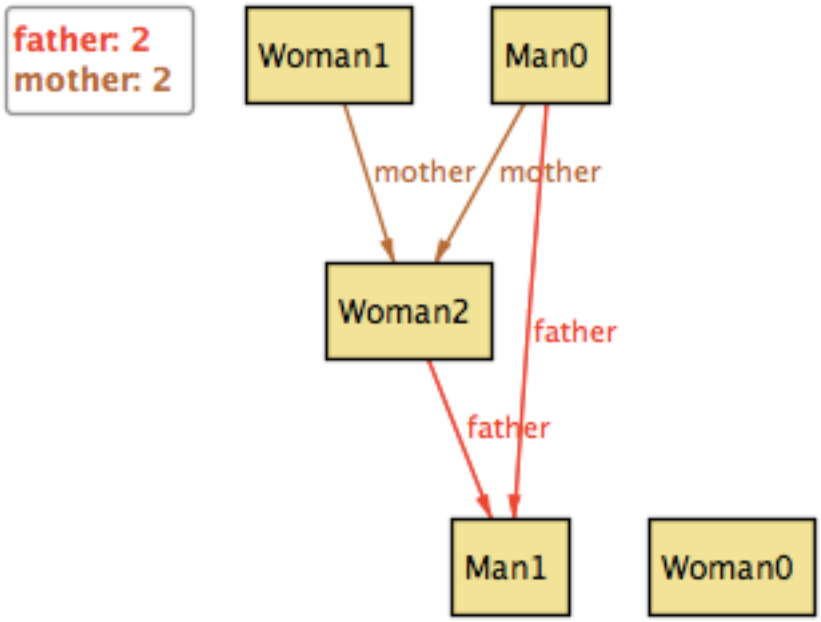
```
pred SocialConvention4 {
  no ((wife+husband) &
      (mother+father).~(mother+father))
}
```

”Proof” (cont.)


```
assert Stronger {
  noCommonAncestors implies SocialConvention4
}

check Stronger for 8
```

```
assert NotStronger {
  SocialConvention4 implies noCommonAncestors
}

check NotStronger for 8
```

Still we can have this case



Does this address the problem?


```
fact SocialConvention3 {
  all p1: Person, p2: Person |
       p1 in p2.(mother+father) implies
       p1.(mother+father) & p2.(mother+father) = none
}
```

Traffic lights

Traffic Lights

- A semaphore can have one of the following colors: GREEN and RED.
- Two semaphores are used to regulate traffic at an intersection where 2 roads meet. Each road is two-way. The semaphores must guarantee that traffic can proceed on one and only one of the two roads in both directions.
- Provide an Alloy model of the intersection, specifying the necessary invariants.

- **NB:** While building the model, focus on those part of the roads that are close to the intersection. Disregard the fact that a road can participate to more than one intersection.

Traffic Lights - Signatures


```
abstract sig Color{}
one sig GREEN extends Color{}
one sig RED extends Color{}

abstract sig Traffic{}
one sig FLOWING extends Traffic{}
one sig STOPPED extends Traffic{}

sig Semaphore {
 color: one Color
}

sig Road {
 traffic: one Traffic,
}

sig Intersection{
 connection: Semaphore ->  Road //MAPS Semaphore to Road
}{
 #connection =2
}
```

Traffic Lights - functions


```
//Retrieves all the Roads of one Intersection
fun getIRoads[i :Intersection]: set Road {
 Semaphore.(i.connection)
}

//Retrieves the Roads connected to one semaphore
fun getSRoads[s :Semaphore]: set Road {
 s.(Intersection.connection)
}

//Retrieves all the Semaphores of one Intersection
fun getSemaphores[i :Intersection]: set Semaphore {
 (i.connection).Road
}
```

Traffic Lights - properties


```
//Intersection has exactly 2 roads and 2
   semaphores
fact intersectionStructure{
 (all i : Intersection |
  (let s = getSemaphores[i] | #s=2) and
  (let r = getIRoads[i] | #r=2)
 )
 and
 // All semaphores are connected to only one road
 (all sem : Semaphore |
  (let rd = getSRoads[sem] | #rd=1)
 )
}
```

Traffic Lights - properties cont.


```
//Semaphores of one intersection should display
   different colors
fact greenIsExclusive{
 all i : Intersection |
  (let s = getSemaphores[i] |
     all s1: Semaphore, s2 : Semaphore |
       (s1 in s and s2 in s and s1 != s2)
       implies s1.color != s2.color)
}

//Traffic flows with GREEN
fact goWithGreen{
 (all s: Semaphore | let r = getSRoads[s] |
                     s.color=RED iff r.traffic=STOPPED)
 and
 (all r: Road | r.traffic=STOPPED implies
                #((Intersection.connection).r)>0)
}
```

Traffic Lights – commands to run

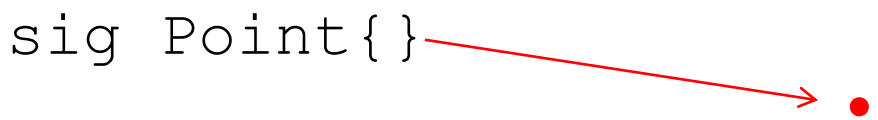

```
pred show{
 #Intersection = 1
}

run show for 8
```

Line

Line

- We wish to specify a simple fragment of geometry
- Points are defined by a signature
  - `sig Point {}`
- Starting from Point, you are to specify:
  1. A Segment; i.e., a pair of points
  2. A Line; i.e., a sequence of connected segments
  3. A Polygon; i.e., a closed Line
  4. Assuming that a predicate `intersect` is defined for two segments, write a predicate `linesIntersect` that checks intersection of two lines
  5. Specify an operation `add` that takes a line and a segment and produces a new line where the segment is appended as a last segment; of course the last point of the last segment of the line and the first point of the segment must coincide

Line - signatures

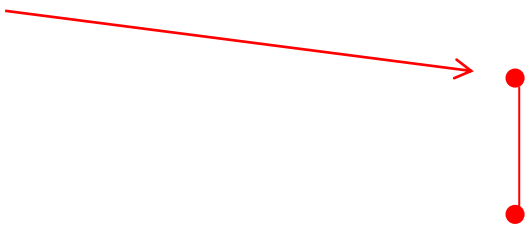


Remember: Fields **are** Relations!

- Alloy fields define relations among the signatures
  - Similar to a field in an object class that establishes a relation between objects of two classes
  - Similar to associations in OCL

Line - signatures


```
sig Point{}

sig Segment {
  start: one Point,
  end: one Point
}{???}
```



Is this enough? What is the condition that must always be true?

Line - signatures


```
sig Point{}

sig Segment {
  start: one Point,
  end: one Point
}{ start != end }
```

- Alternative formulation

```
sig Segment {
  start: one Point,
  end: one Point
}
fact {
 all s: Segment |
        s.start != s.end
}
```

Line

- We wish to specify a simple fragment of geometry
- Points are defined by a signature
  - `sig Point {}`
- Starting from Point, you are to specify:
  1. A Segment; i.e., a pair of points
  2. **A Line; i.e., a sequence of connected segments**
  3. A Polygon; i.e., a closed Line
  4. Assuming that a predicate `intersect` is defined for two segments, write a predicate `linesIntersect` that checks intersection of two lines
  5. Specify an operation `add` that takes a line and a segment and produces a new line where the segment is appended as a last segment; of course the last point of the last segment of the line and the first point of the segment must coincide

Line - signatures


```
sig Line {
  startSeg: one Segment,
  endSeg: one Segment,
  intermediateSeg: set Segment
}
{
```

My definition…

What do I have to do here?

```
??????????



}
```

Line - signatures


```
sig Line {
  startSeg: one Segment,
  endSeg: one Segment,
  intermediateSeg: set Segment
}
{
  no x: intermediateSeg   | startSeg = x or endSeg = x
}
```

There is no intermediate that is also a start or end

Line - signatures


```
sig Line {
  startSeg: one Segment,
  endSeg: one Segment,
  intermediateSeg: set Segment
}
{
  no x: intermediateSeg   | startSeg = x or endSeg = x
  lone x: intermediateSeg | startSeg.end = x.start
}
```

There is zero or one intermediate whose start tails the end of startSeg

Line - signatures


```
sig Line {
  startSeg: one Segment,
  endSeg: one Segment,
  intermediateSeg: set Segment
}
{
  no x: intermediateSeg   | startSeg = x or endSeg = x
  lone x: intermediateSeg | startSeg.end = x.start
  lone x: intermediateSeg | endSeg.start = x.end
}
```

There is zero or one intermediate whose end leads to the start of endSeg

Line - signatures


```
sig Line {
  startSeg: one Segment,
  endSeg: one Segment,
  intermediateSeg: set Segment
}
{
  no x: intermediateSeg   | startSeg = x or endSeg = x
  lone x: intermediateSeg | startSeg.end = x.start
  lone x: intermediateSeg | endSeg.start = x.end
  no x: intermediateSeg   | startSeg.start = x.end
}
```

There is no intermediate whose end is the starting point of startSeg

Line - signatures


```
sig Line {
  startSeg: one Segment,
  endSeg: one Segment,
  intermediateSeg: set Segment
}
{
  no x: intermediateSeg   | startSeg = x or endSeg = x
  lone x: intermediateSeg | startSeg.end = x.start
  lone x: intermediateSeg | endSeg.start = x.end
  no x: intermediateSeg   | startSeg.start = x.end
  no x: intermediateSeg   | endSeg.end = x.start
  all x: intermediateSeg  |
     x.end = endSeg.start or
     one x1: intermediateSeg | x.end = x1.start
  all x: intermediateSeg  |
     x.start = startSeg.end or
     one x1: intermediateSeg | x.start = x1.end

}
```

Full signature… can anyone tell me what's wrong with this?

Line - signatures


```
sig Line {
  startSeg: one Segment,
  endSeg: one Segment,
  intermediateSeg: set Segment
}
{
  no x: intermediateSeg   | startSeg = x or endSeg = x
  lone x: intermediateSeg | startSeg.end = x.start
  lone x: intermediateSeg | endSeg.start = x.end
  no x: intermediateSeg   | startSeg.start = x.end
  no x: intermediateSeg   | endSeg.end = x.start
  all x: intermediateSeg  |
     x.end = endSeg.start or
     one x1: intermediateSeg | x.end = x1.start
  all x: intermediateSeg  |
     x.start = startSeg.end or
     one x1: intermediateSeg | x.start = x1.end
  #intermediateSeg = 0 implies
       (startSeg = endSeg or startSeg.end=endSeg.start)
}
```

Line - signatures


```
sig Line {
  startSeg: one Segment,
  endSeg: one Segment,
  intermediateSeg: set Segment
}
{
  no x: intermediateSeg   | startSeg = x or endSeg = x
  lone x: intermediateSeg | startSeg.end = x.start
  lone x: intermediateSeg | endSeg.start = x.end
  no x: intermediateSeg   | startSeg.start = x.end
  no x: intermediateSeg   | endSeg.end = x.start
  all x: intermediateSeg  |
     x.end = endSeg.start or
     one x1: intermediateSeg | x.end = x1.start
  all x: intermediateSeg  |
     x.start = startSeg.end or
     one x1: intermediateSeg | x.start = x1.end
  #intermediateSeg = 0 implies
       (startSeg = endSeg or startSeg.end=endSeg.start)
}
```

Line

- We wish to specify a simple fragment of geometry
- Points are defined by a signature
  - `sig Point {}`
- Starting from Point, you are to specify:
  1. A Segment; i.e., a pair of points
  2. A Line; i.e., a sequence of connected segments
  3. **A Polygon; i.e., a closed Line**
  4. Assuming that a predicate `intersect` is defined for two segments, write a predicate `linesIntersect` that checks intersection of two lines
  5. Specify an operation `add` that takes a line and a segment and produces a new line where the segment is appended as a last segment; of course the last point of the last segment of the line and the first point of the segment must coincide

Line – signatures and predicates


```
sig Polygon extends Line {
}
{
 ??????????
}
```

Anyone?

Line

- We wish to specify a simple fragment of geometry
- Points are defined by a signature
  - `sig Point {}`
- Starting from Point, you are to specify:
  1. A Segment; i.e., a pair of points
  2. A Line; i.e., a sequence of connected segments
  3. **A Polygon; i.e., a closed Line**
  4. Assuming that a predicate `intersect` is defined for two segments, write a predicate `linesIntersect` that checks intersection of two lines
  5. Specify an operation `add` that takes a line and a segment and produces a new line where the segment is appended as a last segment; of course the last point of the last segment of the line and the first point of the segment must coincide

Line – signatures and predicates


```
sig Polygon extends Line {
}
{
  startSeg.start = endSeg.end
}
```

Line

- We wish to specify a simple fragment of geometry
- Points are defined by a signature
  - `sig Point {}`
- Starting from Point, you are to specify:
  1. A Segment; i.e., a pair of points
  2. A Line; i.e., a sequence of connected segments
  3. A Polygon; i.e., a closed Line
  4. Assuming that a **predicate `intersect`** is defined for two segments, **write a predicate `linesIntersect` that checks intersection of two lines**
  5. Specify an operation `add` that takes a line and a segment and produces a new line where the segment is appended as a last segment; of course the last point of the last segment of the line and the first point of the segment must coincide

Line – signatures and predicates


```
sig Polygon extends Line {
}
{
  startSeg.start = endSeg.end
}

// simplified version of intersect
pred intersect[s1, s2: Segment] {
   s1 != s2 and
   (s1.start = s2.start or s1.start = s2.end or
    s1.end = s2.end     or s1.end = s2.start)
}

pred linesIntersect[l1, l2: Line] {
   l1 != l2 and
   (some s1: l1.intermediateSeg, s2:l2.intermediateSeg |
      intersect[s1, s2])
}
```

Can anyone spell these out for me?

Is this definition correct?

Line – more on predicates and facts

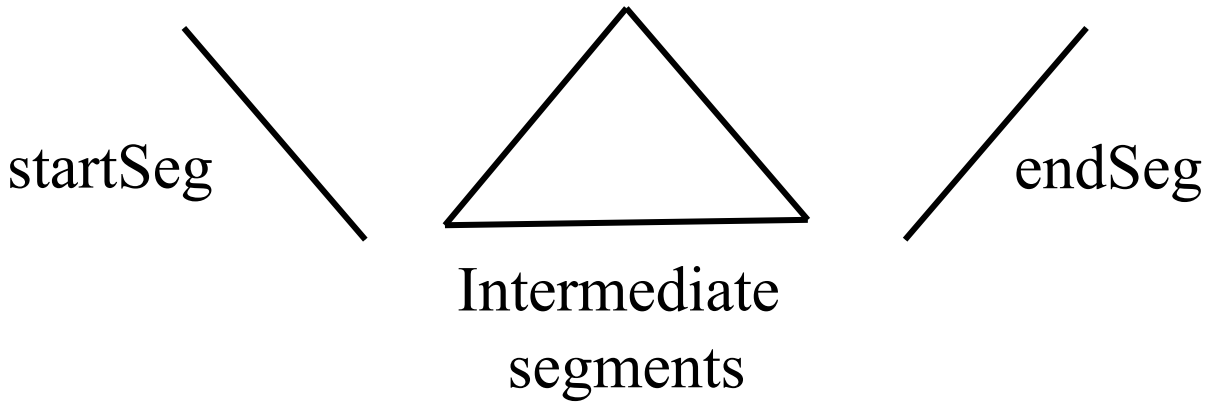

```
pred addToLine[l: Line, s: Segment, l': Line] {
// precondition
  l.endSeg.end = s.start

//postcondition
  l'.startSeg = l.startSeg and
  l'.intermediateSeg = l.intermediateSeg + l.endSeg and
  l'.endSeg = s
}

fact noTwoSegWithSameStartEnd {
  no disj x1, x2: Segment |
          x1.start = x2.start and x1.end = x2.end
}
```

What does this mean?

What does this mean?

An unwanted case

- As highlighted by one of your colleagues, we can still have a case where startSeg and endSeg are detached from the intermediate segments



An unwanted case

- We can see that this holds by defining an assertion

```
assert intermediateSegNotIsolated {
   all l: Line |    #(l.intermediateSeg) >0 implies
     some i: Segment | i in l.intermediateSeg and
     l.startSeg.end = i.start
}
check intermediateSegNotIsolated for 10
```

- Think at how to fix the specification to cope with this

An unwanted case

```
check intermediateSegNotIsolated for 10
```

Note that we need at least 4 elements in our domains to see the problem!

Executing "Check intermediateSegNotIsolated for 3"
Solver=sat4j Bitwidth=0 MaxSeq=0 SkolemDepth=1 Symmetry=20
1557 vars. 57 primary vars. 3230 clauses. 21ms.
No counterexample found. Assertion may be valid. 8ms.

Executing "Check intermediateSegNotIsolated for 4"
Solver=sat4j Bitwidth=0 MaxSeq=0 SkolemDepth=1 Symmetry=20
3060 vars. 96 primary vars. 6833 clauses. 32ms.
Counterexample found. Assertion is invalid. 19ms.

In summary: Alloy Characteristics

- Finite scope check:
  - The analysis is sound, but incomplete
- Infinite model:
  - Finite checking does not get reflected in your model
- Declarative: first-order relational logic
- Automatic analysis:
  - visualization a big help
- Structured data

Operator Precedence

```
let    all a:X|F   no a:X|F   some a:X|F   lone a:X|F
one a:x|F   sum a:x|F
||
<=>
=>      => else
&&
!
in     =     <     >    <=    >=
!in    !=    !<    !>   !<=   !>=
no X    some X    lone X    one X    set X    seq X
<<      >>        >>>
+       -
#X
++
&
->
<:
:>
[]
.
~     *      ^
```

lowest

highest

Hint: keep in mind operator precedence when writing your specs!!