Alloy

http://alloy.mit.edu/

some slides are adapted from

Greg Dennis and Rob Seater

Software Design Group, MIT

Documentation

- We speak of the recent version 4
  - the book refers to version 3
- There is a reference manual
- There are many examples
- There is an online tutorial
- Everything (code, documentation) is found at http://alloy.mit.edu/

Outline

- Alloy language
  - introduction
  - definition of domains, subsets, relations with multiplicity constraints
  - quantifiers and predicate forms
- Alloy analyzer
  - (the analysis algorithm)
  - use of the Alloy Constraint Analyzer

What is Alloy? (1)

- Alloy is a formal notation for specifying models of systems and software
  - Declaration syntax compatible with graphical object models
  - A set-based formula syntax
- Alloy comes with a tool to simulate and perform property verification

What is Alloy? (2)

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

Where can we use it?

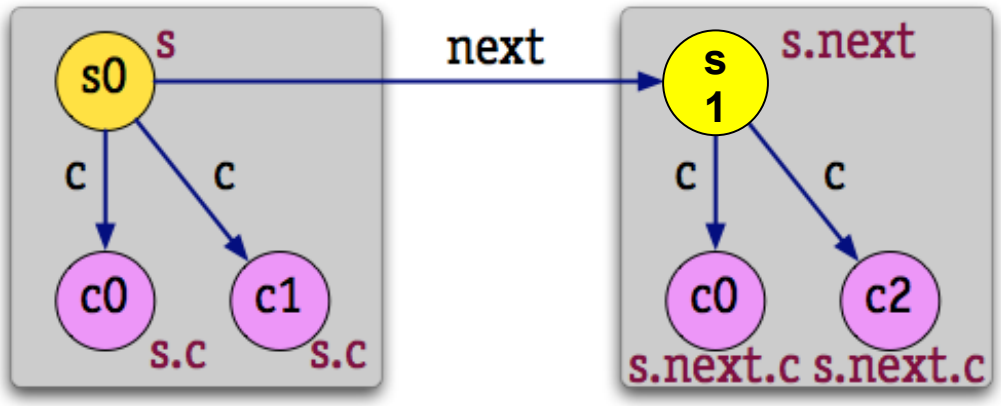
- Anywhere we need to build a model
- For example, in the requirements engineering phase, to describe the domain and its properties, or operations that the machine has to provide
- Or in the design phase, to describe certain components

Four key ideas

1) everything is a relation
2) non-specialized logic
3) counterexamples & scope
4) analysis by SAT

1) Everything is a relation

- Alloy uses relations for
  - all data types – even sets, scalars, tuples
- Key operator is **dot** join
  - relational join
  - field navigation
  - function application



Why relations?

- Easy to understand
  - binary relation is a graph or mapping
- Easy to analyze
  - first order (tractable)
- Uniform
  - set of addresses associated with name n in set of address books B

    n.(B.addr)

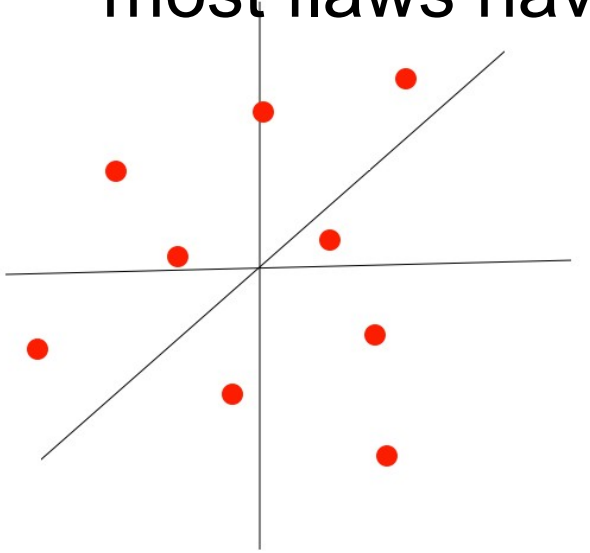
2) Non-specialized logic

- No special constructs for state machines, traces, synchronization, concurrency . . .

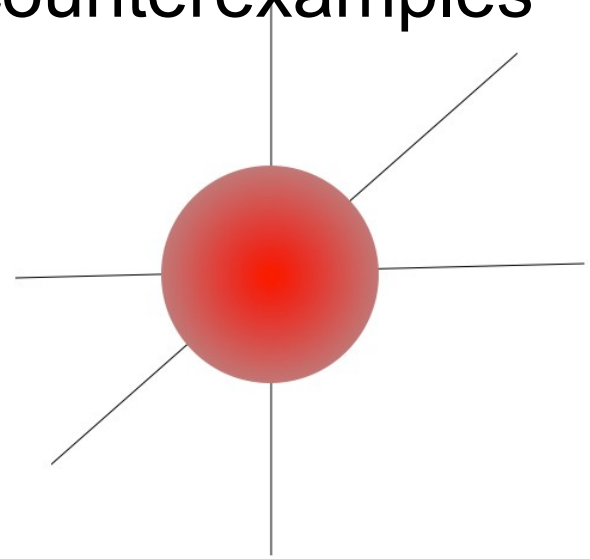
3) Counterexamples & scope

Oobservations about design analysis

- most assertions are wrong
- most flaws have small counterexamples



testing:
a few cases of arbitrary size



scope-complete:
all cases within a small bound

4) Analysis by SAT

- Satisfiability is the problem of determining if the variables of a given boolean formula can be assigned in such a way as to make the formula evaluate to TRUE
  - SAT, the quintessential hard problem (Cook 1971)
    - it is an NP complete problem
- There are high-performance algorithms for solving instances of SAT in practice
  - Chaff (Malik), Berkmin (Goldberg & Novikov), ...

Alloy features (1)

- Alloy is relational
  - It allows to define complex state structures
- Alloy is declarative
  - It is based on logic
  - It expresses the shape and relations on states
- Alloy comes with a tool that allows to perform automatic property analysis
  - Simulations
  - Check of assertions

Alloy features (2)

- Infinite model: you describe the components of a system and how they interact, but not the number
- Finite scope check: you must specify a scope (size) for your model
- Structured data - Alloy supports complex data structures, such as trees, and thus has a rich way to describe state

Alloy = logic + language + analysis

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

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
- Every value in Alloy logic is a relation!
  - relations, sets, scalars all the same thing

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)}
```

- Binary relation

```
names = {(B0, N0),
         (B0, N1),
         (B1, N2)}
```

- Ternary relation

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

Logic: relations

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

B0	**N0**	**A0**
B0	**N1**	**A1**
B1	**N1**	**A2**
B1	**N2**	**A2**

size = 4

arity = 3

- rows are unordered
- columns are ordered but unnamed
- all relations are first-order
  - relations cannot contain relations, no sets of sets

Relations (1)

- Relation = set of ordered n-tuples of atoms
  - n is called the *arity* of the relation
  - Example: name is binary (2-ary), because it contains only pairs of atoms
- 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

Relations (2)

- If R is a binary relation with type S -> T:
  - S is its *left type*, T its *right* type
  - R is *total* if every atom in the left type is mapped to at least one atom in the right type
  - R is *surjective* if every atom in the right type is mapped to at least one atom in the left type
  - R is *functional* if every atom in the left type is mapped to at most one atom in the right type
  - R is *injective* if every atom in the right type is mapped to at most one atom in the left type
- (N.B. We use -> instead of x for cartesian product of sets)

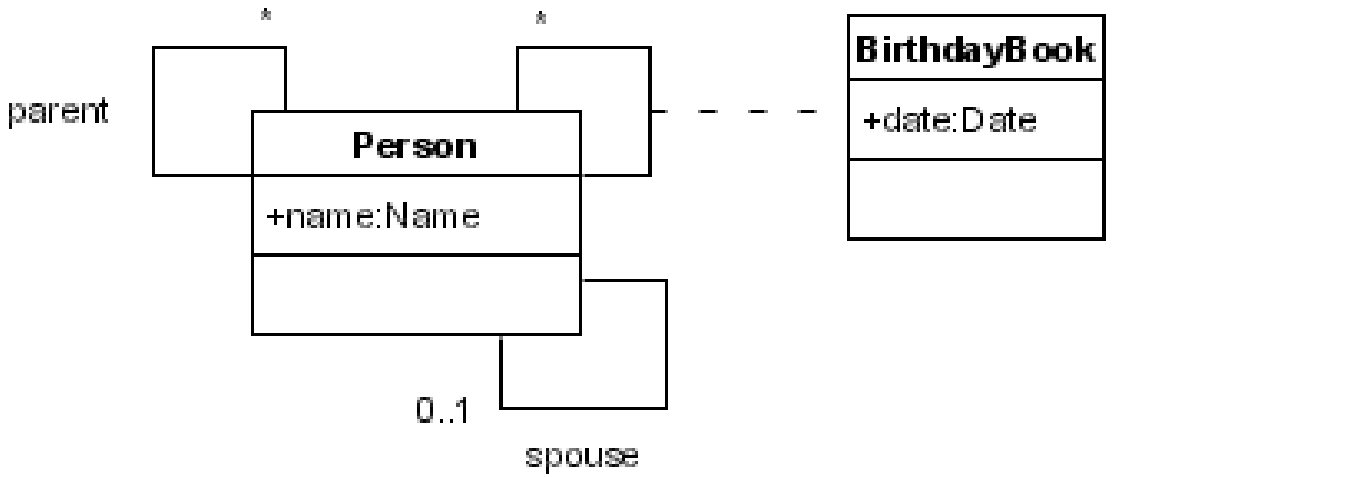
A first simple example

- A world made of people:
  - Each person has
    - Exactly one name
    - One or no spouse
    - Some parents
    - Exactly one birthday book
  - Each birthday book
    - Contains a list of persons
    - Reports, for each person in the list, his/her birthday
  - Constraints (among others)
    - No person is married with one of his/her brothers/sisters
    - Any person is in the birthday book of one of his/her friends (i.e., of one of the persons in his/her birthday book)

Rationale

- Note that the previous description is about an application domain
- We gave both domain properties (e.g., no person is married with a brother/sister)…
- …and specification of what some software has to ensure (e.g., each person is in the birthday book of a friend)

A UML model



There is no way to express the last two constraints
by means of class diagram constructs (you must use OCL)

An Alloy simplified initial specification

```
module birthday_book

sig String { }

sig Person {
  name: String,
  spouse: lone Person,
  parents: set Person
}
```

ignores birthdays book, date and constraints

means that

- *p.name is 1 String*
- *p.spouse is 0 or 1 Spouse*
- *p.parents is any number of Person*

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's worlds

- Alloy's worlds are made of
  - Atoms
  - Relations over atoms
- An atom is an object
  - Completely featureless
  - Indivisible and immutable
  - Without meaning (it can represent anything)

A possible world described by the specification

- Univ={ (ANN), (BILL), (JOHN), (LISA), (PAM), (P1), (P2), (P3), (P4), (P5)}
- String={(ANN), (BILL), (JOHN), (LISA), (PAM)}
- Person={(P1), (P2), (P3), (P4), (P5)}
- Name={(P1, ANN), (P2, LISA),(P3, BILL), (P4,JOHN), (P5, JOHN)}
- Spouse={(P1,P4), (P2,P3), (P3,P1) (P4,P4)}
- Parents={(P1,{P2, P3}), (P2, {P1, P4)}), (P3, {P4})}

The Alloy dogma

- Alloy specifications describe **only relations on atoms!!!**
  - Neither atoms alone
  - Nor tuples of atoms
  - Nor sets of atoms
  - Nor relations over sets, or other relations...
- Relations are used for
  - “establishing links” between atoms
  - “giving features” to atoms (attributes)
  - “building sets” of atoms (unary relations)
  - “detecting” single atoms (singleton unary relations)

Back to the example --constraint added--

```
module birthday_book
sig String { }
sig Person {
  name: String,
  spouse: lone Person,
  parents: set Person
}
fact {
   (no p: Person | some (p.spouse.parents & p.parents))
}
```

also

```
module birthday_book
sig String { }
sig Person {
  name: String,
  spouse: lone Person,
  parents: set Person
}
{no p:Person|
      some(p.spouse.parents&p.parents})
```

constraints restrict the set of possible worlds

*some e **means** e != none*

none is the empty set

Running the specification

- We define a predicate
  - which can be empty (no constraints)
- We run the predicate to find a model that satisfies it
- If the predicate is empty and we find an instance model, it means that the world we described is not empty
- It may also show instances that shouldn't exist
  - in this case we need to modify the specs!
    - run bthdBook1 and bthdBook2

Alloy specifications (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

Alloy specifications (2)

- Signatures, facts and functions tell how correct worlds (models) are made
  - When Alloy Analyzer tries to build a model, it must comply to 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

Example

From D. Jackson's Software Abstractions

- email client's address book (ECAB)
  - db associating addresses with short names
    - io carlo.ghezzi@polimi.it

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

for a given book, field addr maps names to addresses
lone means that each name is mapped to 0 or 1 address

addr is a 3-ary relation on Book, Name, Addr containing the triple b->n->a when in book b n maps to a

Generation of possible worlds

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

```
pred show(){}
run show for 3 but 1 Book
```

```
pred show(b: Book) {
    #b.addr > 1          or #b.addr >= 1
    }
run searches for an instance of a predicate
```

Validation (1)

- Can one name map to different addresses?

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

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

not surprising because of the multiplicity constraint in addr

Validation (2)

- Can we have more than 1 name in address book (which contains at least one entry)?

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

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

denotes the set of all addresses that may result from lookup

Adding operations (1)

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

a predicate is just a constraint

it constrains the book (b) before and after (b') the operation

here it defines an operation

adBookOp1, 2, 3

Adding operations (2)

```
pred del (b, b': Book, n: Name, a: Addr) {
    b'.addr = b.addr - n ->a
    }
```

adBookOp4

Assertions (1)

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

an assertion is a constraint that is intended to be valid for all possible cases. it can be checked by "check"

Alloy tries to find a counterexample

and here it finds it!!! WHY?????

adBookOp5

Assertions (2)

```
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, a] implies
          b.addr = b".addr
    }
```

here no counterexample found
of course, in the stated limited set of cases
we haven't proved the assertion to be valid
but we have enough confidence that it is
small scope hypothesis

adBookOp6

More assertions

```
assert addIdempotent {
    all b,b',b": Book, n: Name, a: Addr |
        add[b,b',n,a] and add[b',b",n,a] implies b'.addr = b".addrr
    }

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']
    }

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

Exercise

We model the floors of a building

```
sig Floor {
    upper, lower: one Floor
}
pred show(){
    #Floor>2  //try 0, 1, 2
}
run show
```

```
sig Floor {
    upper, lower: lone Floor
}
pred show(){
    #Floor>2  //try 0, 1, 2
}
run show
```

a floor may appear on top of itself

```
sig Floor {
    upper, lower: lone Floor
}
fact {//floors appear only once travelling up
    no f: Floor|  f in f.^upper
}

pred show(){
    #Floor>1
}
run show
```

can generate also disjoint sets of floors
may be undesirable if we model floors of a building

```
sig Floor {
    upper, lower: lone Floor
}
fact {//a floor is either upper or lower another
    all f1, f2: Floor| f1.lower=f2 <=> f2.upper=f1
}
fact {
    no f: Floor|  f in f.^upper
}
pred show(){
    #Floor>1
}
run show
```

no isolated floor (if more than one floor)

```
sig Floor {
    upper, lower: lone Floor
}
fact {
    all f1, f2: Floor| f1.lower=f2 <=> f2.upper=f1
}
fact {
    no f: Floor|  f in f.^upper
}
fact {
    #Floor>1=> all f: Floor| some (f.lower+f.upper)
}
pred show(){
    #Floor>1
}
run show
```

Logic: constants

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

```
Person = {(P1), (P2), (P3), (P4), (P5)}
String = {(ANN), (BILL), (JOHN), (LISA), (PAM)}

none = {}
univ = {(P1), (P2), (P3), (P4), (P5) (ANN),
        (BILL), (JOHN), (LISA), (PAM)}
iden = {(P1, P1), (P2, P2), (P3, P3), …
         (ANN, ANN), (BILL, BILL), … }
```

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
```

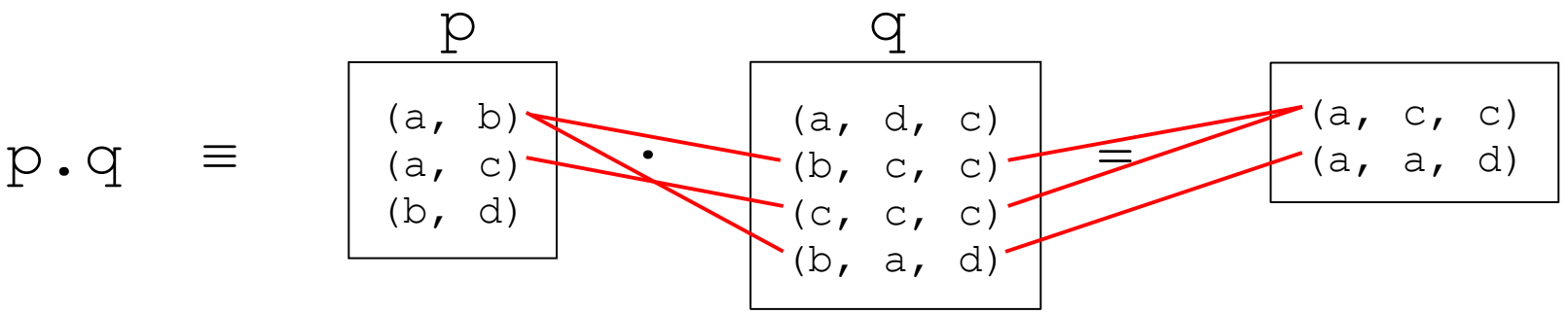
Logic: product operator

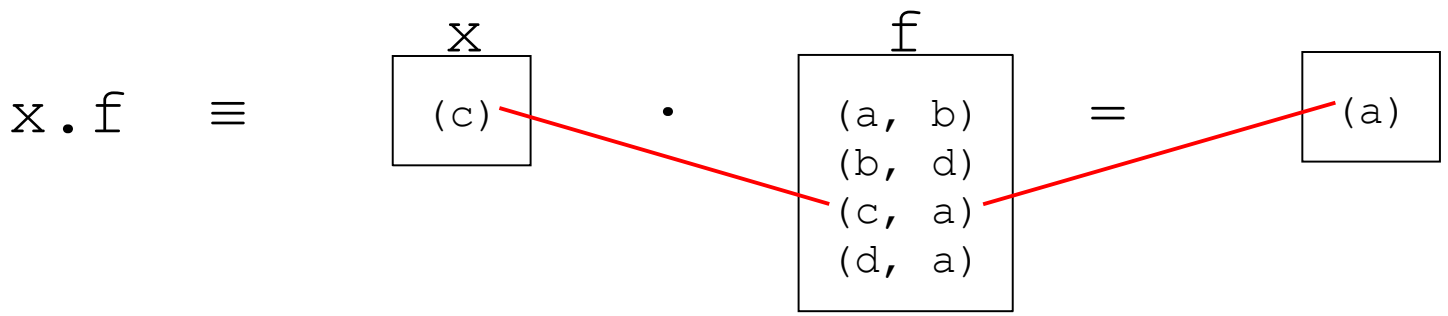
-> *cross product*

```
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 join





Logic: join operators

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

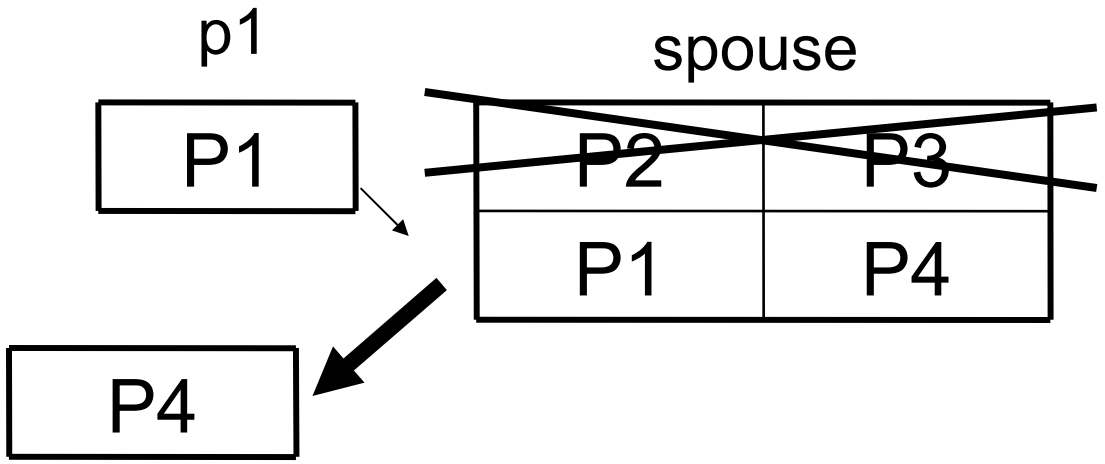
```
e1[e2] = e2.e1
a.b.c[d] = d.(a.b.c)
```

```
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)}
```

An example

Who is p1.spouse?

- p1 = { (P1) }
- spouse = { (P2, P3), (P1, P4) }



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)}

~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)}
```

Logic: restriction and override

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

```
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 = address :> Name  = {(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: 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

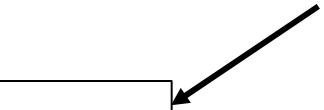
```
no n: Name | n in n.^address
```
no name can be reached by lookups from itself — address book acyclic

```
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: set declarations

multiplicity constraints



`x: m e` | `Q x: m e`

`x: e <=> x: one e`

set	*any number*
one	*exactly one*
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
Q r: A m -> n B
```

```
r: A -> B <=>
r: A set -> set B
```

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

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

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

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

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

```
r: (A m -> n B) -> C <=>
all c: C | r.c: A m -> n B
```

More examples

- f: X -> lone Y

  for each x in X f maps x to at most one element in Y
- f: X lone -> lone Y

  for each y in Y at most one element in X is mapped to y by f and viceversa
- f: X -> one Y

  for each x in X f maps x to exactly one element in Y
- f: X one-> one Y

  for each y in Y exactly one element in X is mapped to y by f and viceversa

Logic: quantified expressions

some e	*e has **at least one** tuple*
no e	*e has **no** tuples*
lone e	*e has **at most one** tuple*
one e	*e has **exactly one** tuple*

some Name
set of names is not empty

some address
address book is not empty – it has a tuple

no (address.Addr - Name)
nothing is mapped to addresses except names

all n: Name | **lone** n.address
every name maps to at most 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

Two logics in one

“everybody loves a winner”

- predicate logic

  $\forall w \mid \texttt{Winner}(w) \Rightarrow \forall p \mid \texttt{Loves}(p, w)$

- relational calculus
  - $\texttt{Person} \times \texttt{Winner} \subseteq \texttt{loves}$

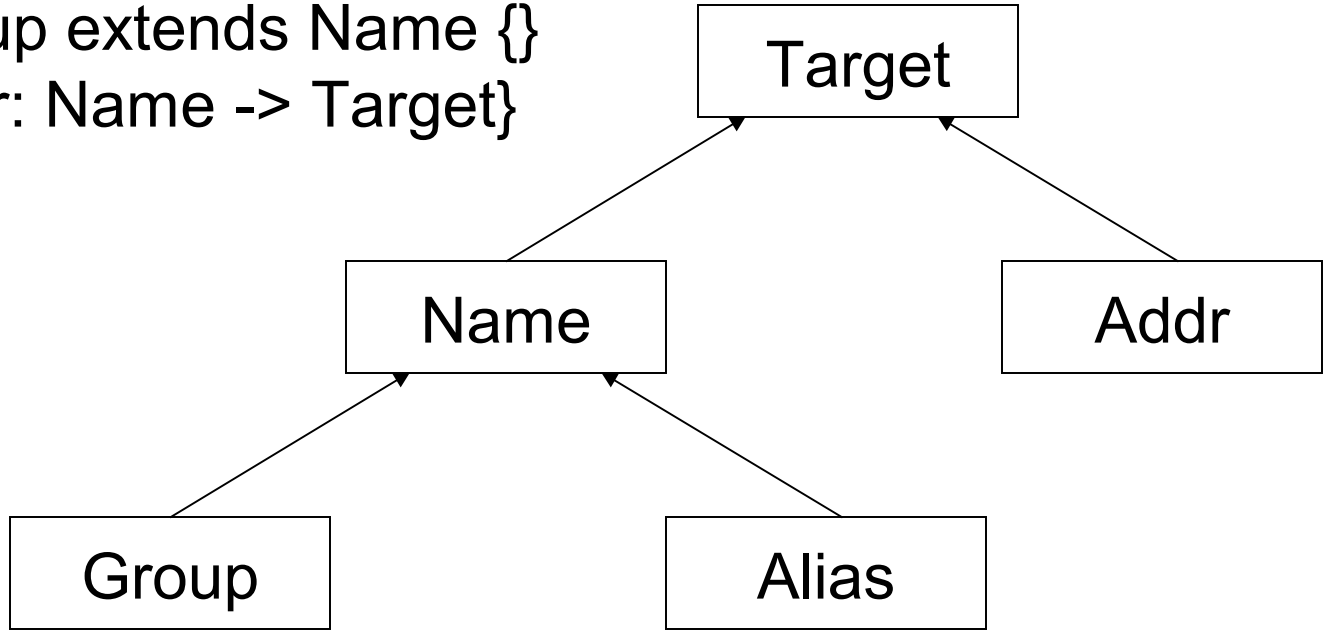
- Alloy logic – any way you want
  - ***all*** *p: Person, w: Winner | p -> w* ***in*** *loves*
  - *Person -> Winner* ***in*** *loves*
  - ***all*** *p: Person | Winner* ***in*** *p.loves*

Back to the example

A more realistic address book

- Having created an alias for an address, the alias can be the target for another alias
- An alias can name multiple targets (i.e., a group can be referred to with a single name)
- How can we do this by reusing what we did?

```
abstract sig Target {}
sig Addr extends Target {}
abstract sig Name extends Target {}

sig Alias, Group extends Name {}
sig Book {addr: Name -> Target}
```




```
pred show(b: Book) {some b.addr}
run show for 3 but 1 Book
```

Facts

- Describe a constraint that is assumed always to hold (we used it in address book earlier)

```
fact {
    all b: Book | no n: Name | n in n.^(b.addr)
}
```

^(b.addr) is the transitive closure
it says that n cannot be reached starting from n by applying addr any number of times

Embedding facts in signatures

```
abstract sig Target {}
sig Addr extends Target {}
abstract sig Name extends Target {}

sig Alias, Group extends Name {}
sig Book {addr: Name -> Target}
     {no n: Name | n in n.^addr}
```

being inside a sig, it holds for all books

Question: can an alias map to more than target?

It should not: in such a case it should be a group

Further modification

```
abstract sig Target {}
sig Addr extends Target {}
abstract sig Name extends Target
{}

sig Alias, Group extends Name {}
sig Book {addr: Name -> Target}
    {
    no n: Name | n in n.^addr
    all a: Alias | lone a.addr
    }
```

try this

```
pred show (b:Book){
      some a:  Alias | #a.(b.addr)>1
}
run show for 3 but 1 Book
```

A serious problem

- We can have names in the address book that do not correspond to addresses
- How can this be avoided?
  - we'll see shortly

One more step…

```
abstract sig Target {}
sig Addr extends Target {}
abstract sig Name extends Target {}
sig Alias, Group extends Name {}
sig Book {
        names: set Name,
        addr: names -> some Target
        }
        {
        no n: Name | n in n.^(addr)
        all a: Alias | lone a.addr
        }
fun lookup (b: Book, n: Name): set Addr {n.^(b.addr) & Addr}
assert lookupYields {
        all b: Book, n: b.names | some lookup (b, n)
}
```

generates a counterexample

Execution traces

- We describe an ordering on address books where
  - the first book satisfies some initial condition
  - any adjacent books in the ordering are related by an operation

The ordering on Books is provided by the library module util/ordering, which can order a set of any type

```
module tour/addressBook3
open util/ordering [Book]
```

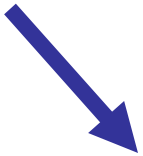
instantiation of ordering on Books

```
abstract sig Target {}
    …
pred init (b: Book) { no b.addr }
fact traces {
    init (first ())
    all b: Book - last() | let b' = next(b) |
        some n: Name, t: Target |
            add (b, b', n, t) or del (b, b', n, t)
    }
```

Empty lookup problem (1)

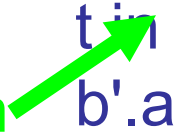
predicate add is incorrect
allows meaningless aliases to be added

```
pred add (b, b': Book, n: Name, t: Target) {
    b'.addr = b.addr + n->t
    }
```



```
pred add (b, b': Book, n: Name, t: Target) {
    t in Addr or some lookup(b,t)
    b'.addr = b.addr + n->t
    }
```

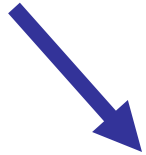
precondition



Empty lookup problem (2)

predicate delete is incorrect because a group or alias may become empty

```
pred del (b, b': Book, n: Name, t: Target) {
    b'.addr = b.addr - n->t
    }
```



```
pred del (b, b': Book, n: Name, t: Target) {
    no b.addr.n or some n.(b.addr) - t
    b'.addr = b.addr - n->t
    }
```

precondition
- n isn't mapped to
- it is mapped on some target besides t

The Alloy analyzer

Alloy analyzer

- Two kinds of analyses are possible:
  - **Checking**: given a statement S, find a world not satisfying S (*counterexample*)
  - **Simulation**: given a statement S, find a world satisfying S (*instance*)
- Actually the analysis is the same
  - just negate S!

The analysis algorithm

- Translates constraint to be solved into boolean constraints
- These are fed to a SAT solver
  - SAT stands for satisfiability
    - a solution of a boolean formula is an assignment of values to variables that satisfies the formula

Unsolv(-ed? -able?) problems

- If a counterexample is not found, this does **NOT** means that everything is OK. However:
  - If one is in scope, the Analyzer WILL find it
  - Sometimes we are content with finite scopes (e.g., Firewire bus has max 63 devices)
  - If scope is “large enough”, no counterexample *likely* means that everything is OK indeed
- How do we make specification and code agree?
  - Not much available by now (abstraction functions...)
  - Specifying is nevertheless beneficial!
    - More interesting (specifiable) checks than compilers
    - Abstraction eases understanding
    - Pinpoint analysis & design issues before they arise in code