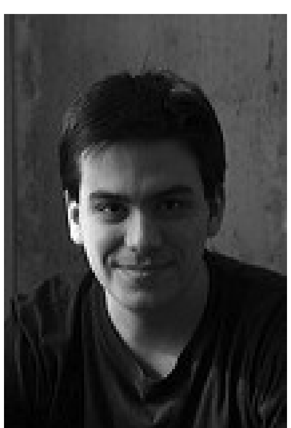
FM 2006 Alloy Tutorial

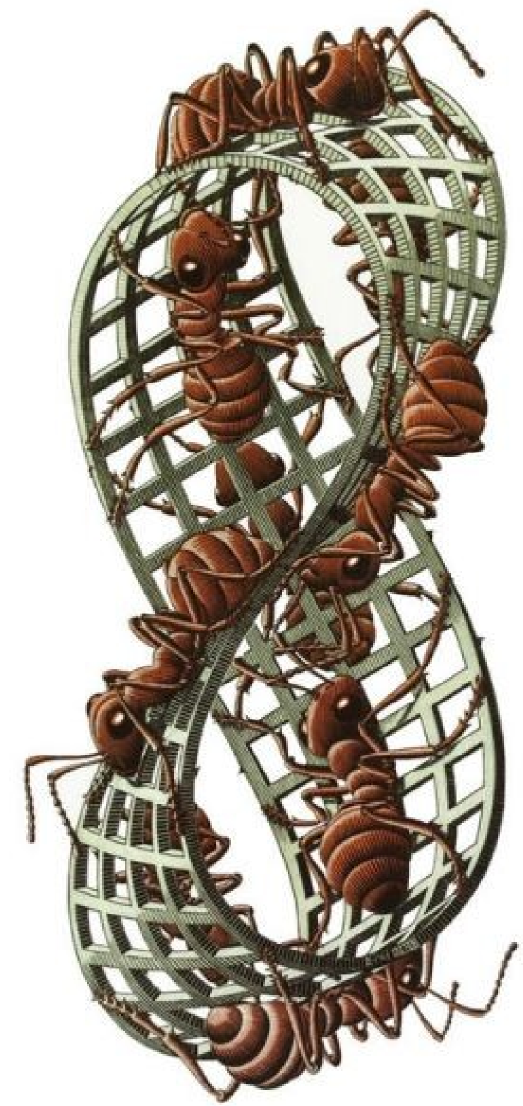
Session 1: Intro and Logic

Greg Dennis and Rob Seater
Software Design Group, MIT



agenda

- Session 1: Intro & Logic
  - break
- Session 2: Language & Analysis
  - lunch
- Session 3: Static Modeling
  - break
- Session 4: Dynamic Modeling



M.C. Escher

trans-atlantic analysis

Oxford, home of Z

- notation inspired by Z
  - sets and relations
  - uniformity
  - *but* not easily analyzed

Pittsburgh, home of SMV

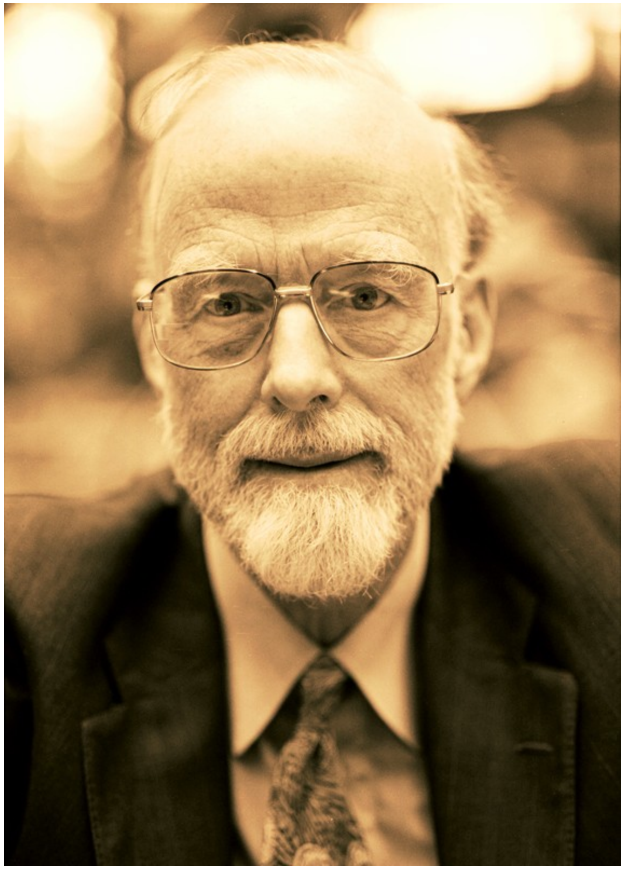
- analysis inspired by SMV
  - billions of cases in seconds
  - counterexamples not proofs
  - *but* not declarative

why declarative design?

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.

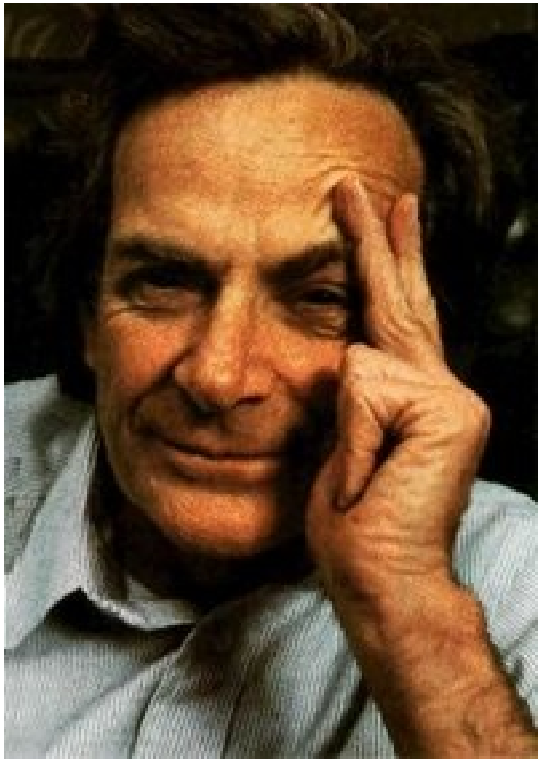
– Tony Hoare [Turing Award Lecture, 1980]



why automated analysis?

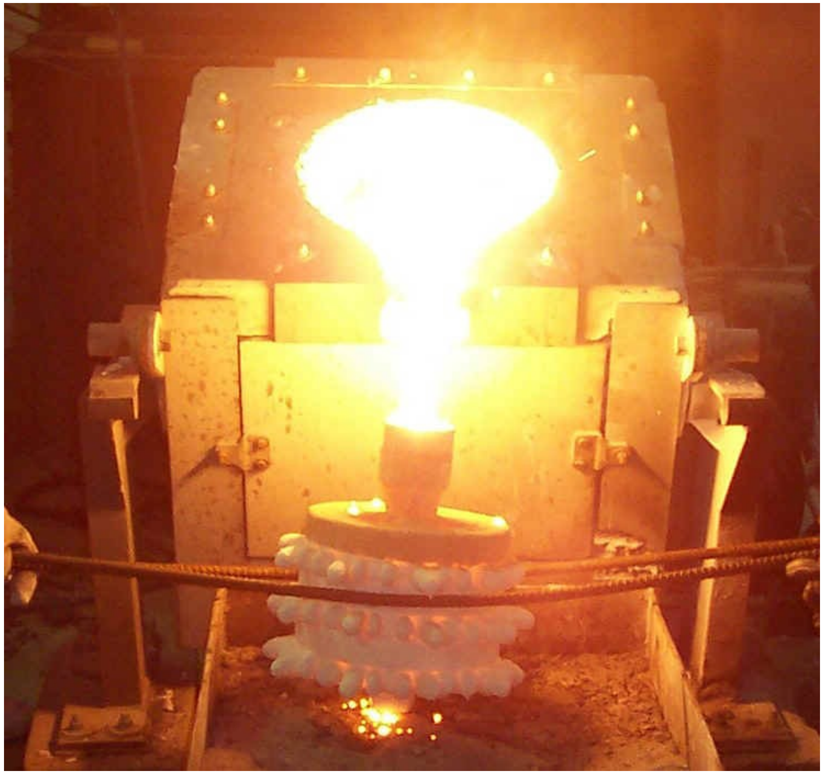
The first principle is that you must not fool yourself, and you are the easiest person to fool.

– Richard P. Feynman



alloy case studies

- Multilevel security (Bolton)
- Multicast key management (Taghdiri)
- Rendezvous (Jazayeri)
- Firewire (Jackson)
- Intentional naming (Khurshid)
- Java views (Waingold)
- Access control (Zao)
- Proton therapy (Seater, Dennis)
- Chord peer-to-peer (Kaashoek)
- Unison file sync (Pierce)
- Telephone switching (Zave)



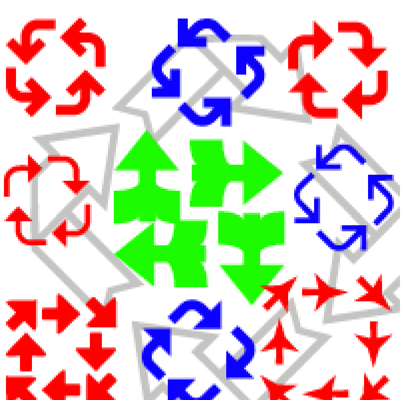
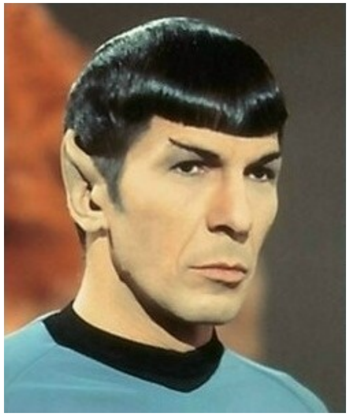
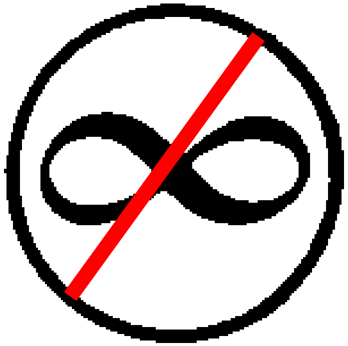
four key ideas . . .

1) everything is a relation

2) non-specialized logic

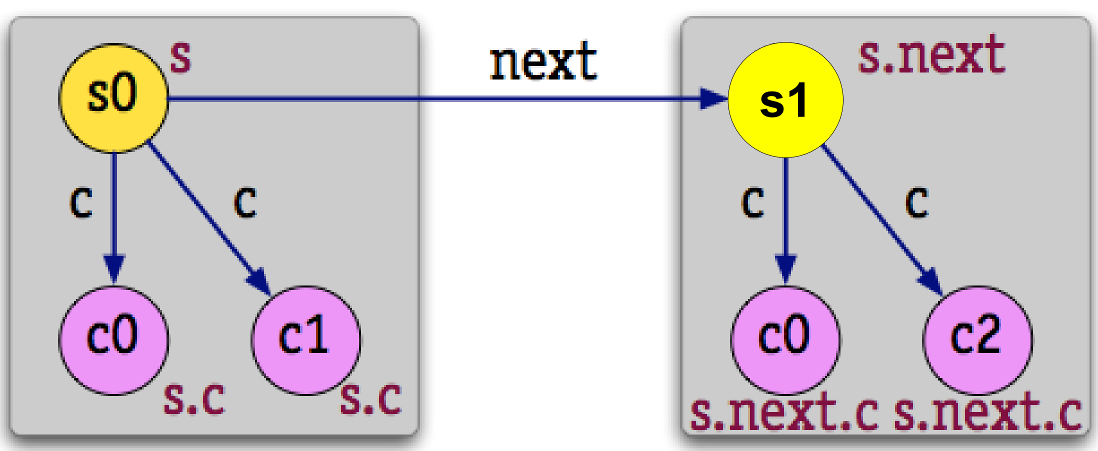
3) counterexamples & scope

4) analysis by SAT


1) everything's a relation

- Alloy uses relations for
  - all datatypes – even sets, scalars, tuples
  - structures in space and time
- 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 books B

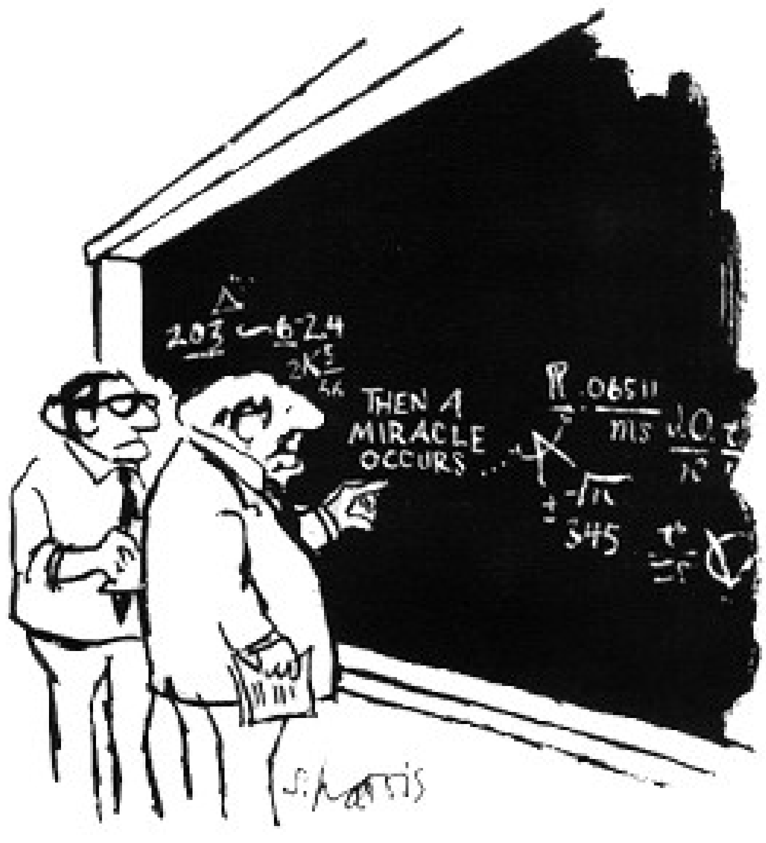
```
Alloy:  n.(B.addr)
Z:      ∪ { b: B • b.addr (| {n} |)}
OCL:    B.addr[n]->asSet()
```

> There is no problem in computer science that cannot be solved by an extra level of indirection.
> – David Wheeler

Wheeler

2) non-specialized logic

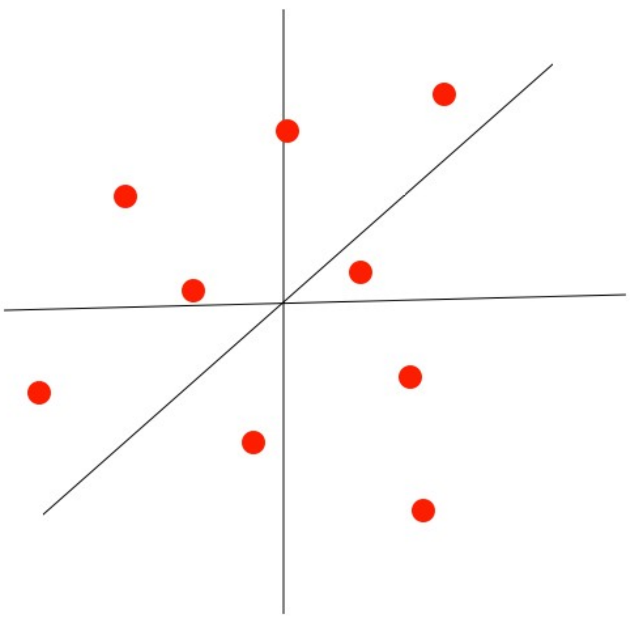
- No special constructs for state machines, traces, synchronization, concurrency . . .



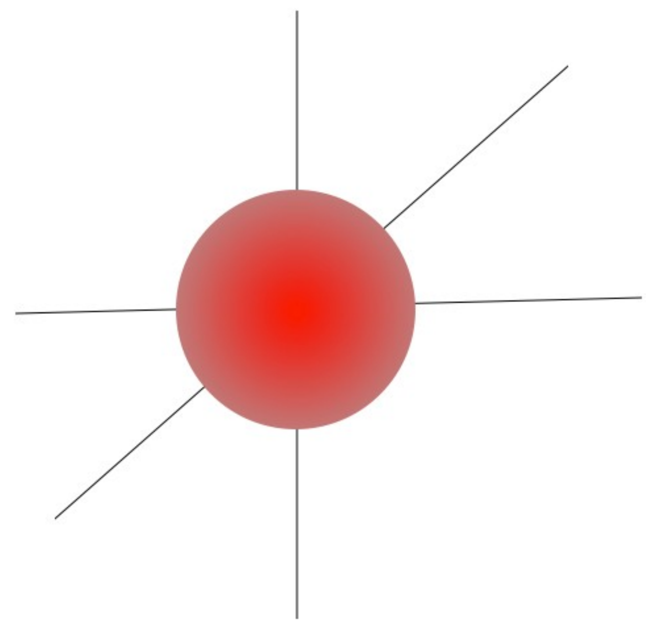
"I think you should be more explicit here in step two."

3) counterexamples & scope

- observations about design analysis:
  - most assertions are wrong
  - most flaws have small counterexamples



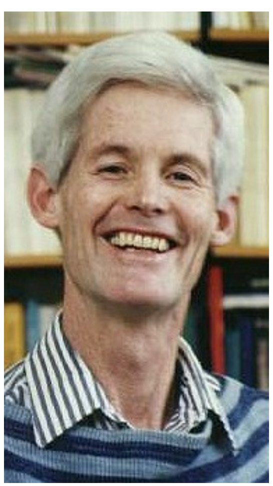
testing:
a few cases of arbitrary size



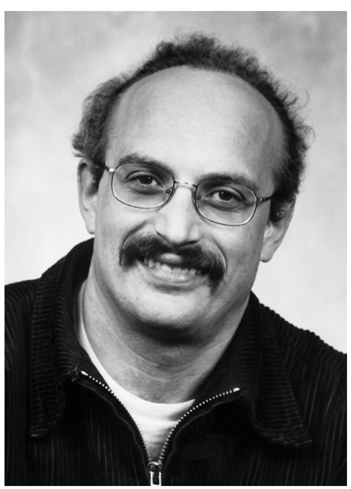
scope-complete:
all cases within a small bound

4) analysis by SAT

- SAT, the quintessential hard problem (Cook 1971)
  - SAT is hard, so reduce SAT to your problem
- SAT, the universal constraint solver (Kautz, Selman, ... 1990's)
  - SAT is easy, so reduce your problem to SAT
  - solvers: Chaff (Malik), Berkmin (Goldberg & Novikov), ...



Stephen Cook



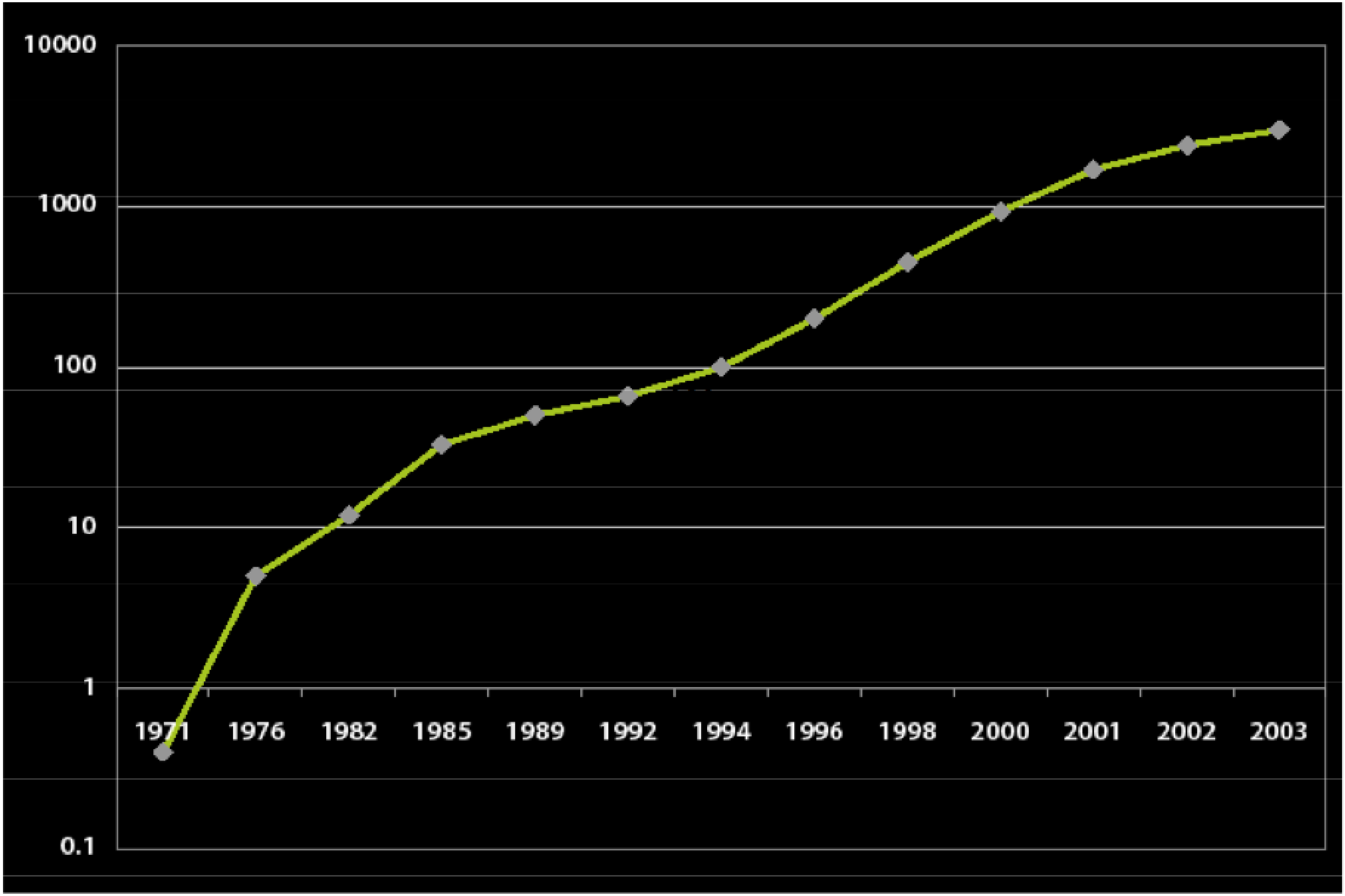
Eugene Goldberg

Henry Kautz

Sharad Malik

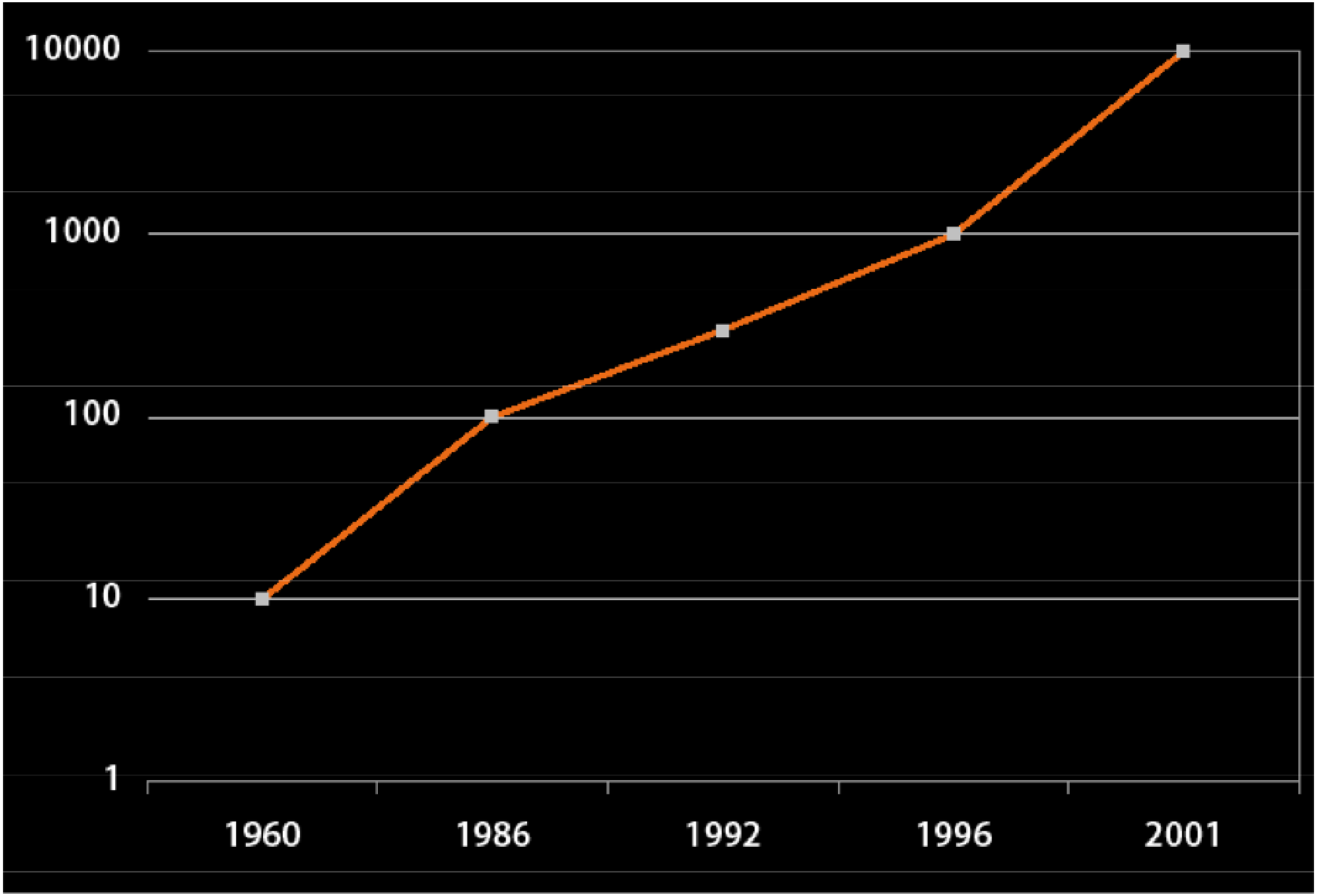
Yakov Novikov

Moore's Law



10000
1000
100
10
1
0.1
1971
1976
1982
1985
1989
1992
1994
1996
1998
2000
2001
2002
2003

SAT performance



SAT trophies

install the Alloy Analyzer

- requires Java 1.4 Runtime Environment
  - http://java.sun.com

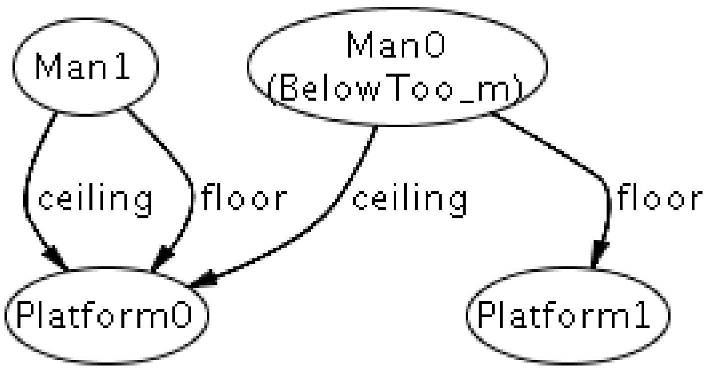
➢ download the Alloy Analyzer
  - http://alloy.mit.edu

➢ run the Analyzer
  - double click *alloy.jar* or
  - execute *java -jar alloy.jar* at the command line

➢ this bullet indicates something *you* should do

verify the installation

- open *examples/toys/ceilingsAndFloors.als*
- click the “Build” icon
  - output reads “Compilation successful”
- click the “Execute” icon
  - output shows graphic



- need troubleshooting?
  - http://alloy.mit.edu/downloads.php

modeling "ceilings and floors"

```
sig Platform {}
```

there are "Platform" things

```
sig Man {ceiling, floor: Platform}
```

each Man has a ceiling and a floor Platform

```
pred Above(m, n: Man) {m.floor = n.ceiling}
```

Man m is "above" Man n if m's floor is n's ceiling

```
fact {all m: Man | some n: Man | Above (n,m)}
```

"One Man's Ceiling Is Another Man's Floor"

checking “ceilings and floors”

```
assert BelowToo {
  all m: Man | some n: Man | Above (m,n)
}
```

"One Man's Floor Is Another Man's Ceiling"?

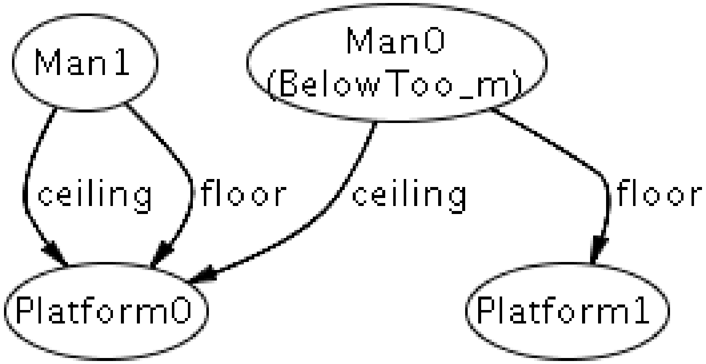
```
check BelowToo for 2
```

check "One Man's Floor Is Another Man's Ceiling"

counterexample with 2 or less platforms and men?

- clicking “Execute” ran this command
  - counterexample found, shown in graphic

counterexample to “BelowToo”

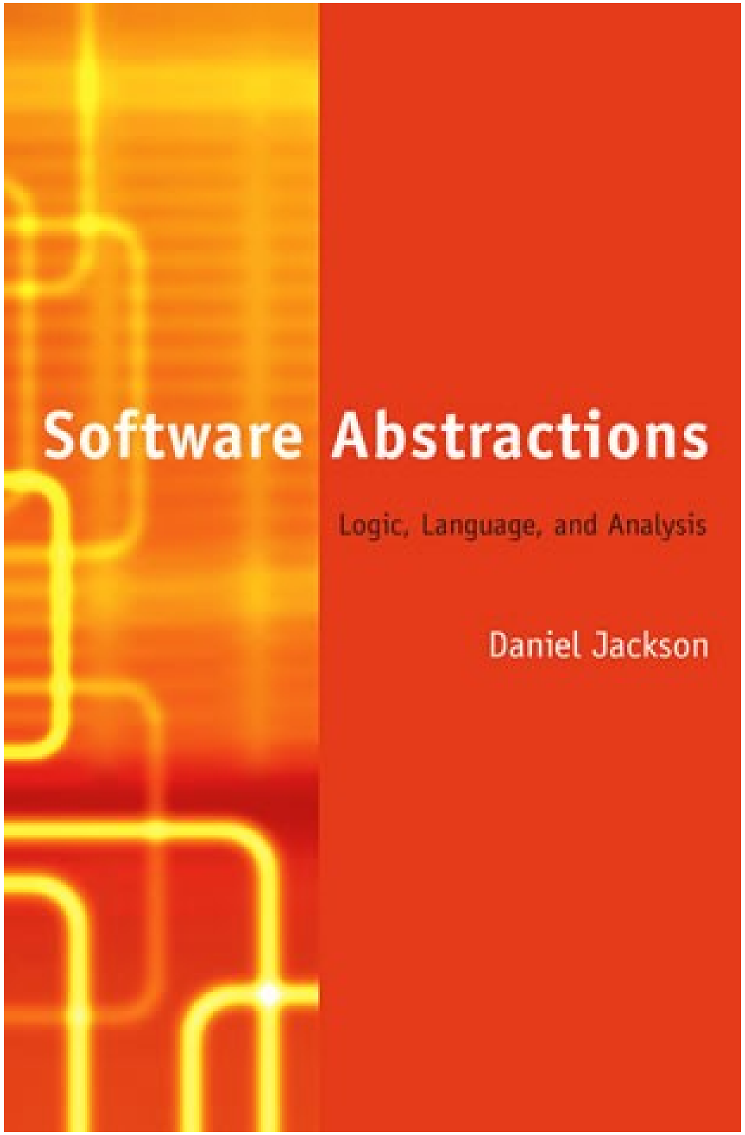


McNaughton

Alloy = logic + language + analysis

- logic
  - first order logic + relational calculus
- language
  - syntax for structuring specifications in the logic
- analysis
  - bounded exhaustive search for counterexample to a claimed property using SAT

software abstractions



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's 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**

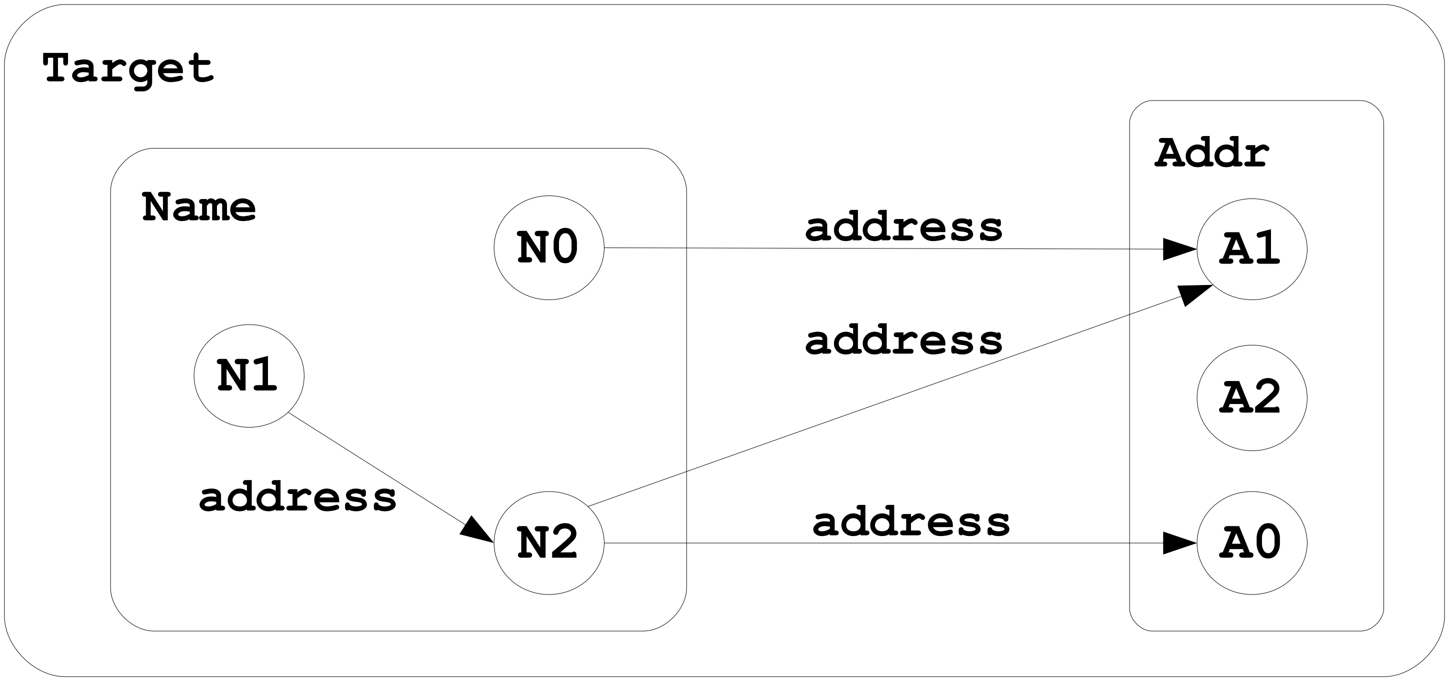
size = 4

arity = 3

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

logic: address book example

```
Name = {(N0), (N1), (N2)}
Addr = {(A0), (A1), (A2)}
Target = {(N0), (N1), (N2), (A0), (A1), (A2)}
address = {(N0, A1), (N1, N2), (N2, A1), (N2, A0)}
```



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*

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

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

```
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
```

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

cacheAddr + diskAddr =
cacheAddr & diskAddr =
cacheAddr = diskAddr =
```

logic: set operators

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

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

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

```
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
```

```
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' =

b->address + b'->address' =
```

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

p.q ≡ p • q =

p
<a, b>
<a, c>
<b, d>

q
<a, d, c>
<b, c, c>
<c, c, c>
<b, a, d>

<a, c, c>
<a, a, d>

---

x.f ≡ x • f =

x
<c>

f
<a, b>
<b, d>
<c, a>
<d, a>

<a>

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] =
address.host =
```

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)}

~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*

```
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 = [redacted]
```

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

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

<table>
<tr><td><code>!</code></td><td>not</td><td>negation</td></tr>
<tr><td><code>&&</code></td><td>and</td><td>conjunction</td></tr>
<tr><td><code>||</code></td><td>or</td><td>disjunction</td></tr>
<tr><td><code>=></code></td><td>implies</td><td>implication</td></tr>
<tr><td><code>,</code></td><td>else</td><td>alternative</td></tr>
<tr><td><code><=></code></td><td>iff</td><td>bi-implication</td></tr>
</table>

four equivalent constraints:

```
F => G , 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
```
[redacted]

```
all n: Name | lone a: Address | a in n.address
```
[redacted]

```
all n: Name | no disj a, a': Address | (a + a') in n.address
```
[redacted]

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

```
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
```

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*

```
Q e <=> Q e | true
```

`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: comprehensions

```
{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

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

four equivalent constraints:

```
all n: Name |
  some n.workAddress => n.address = n.workAddress
    else n.address = n.homeAddress

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

all n: Name |
  let w = n.workAddress |
    n.address = if some w then w else n.homeAddress

all n: Name |
  n.address = let w = n.workAddress |
    if some w then 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

2 logics in one

- "everybody loves a winner"
- predicate logic
  - $\forall$`w | Winner(w)` $\Rightarrow$ $\forall$`p | Loves(p, w)`
- relational calculus
  - `Person` $\times$ `Winner` $\subseteq$ `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`

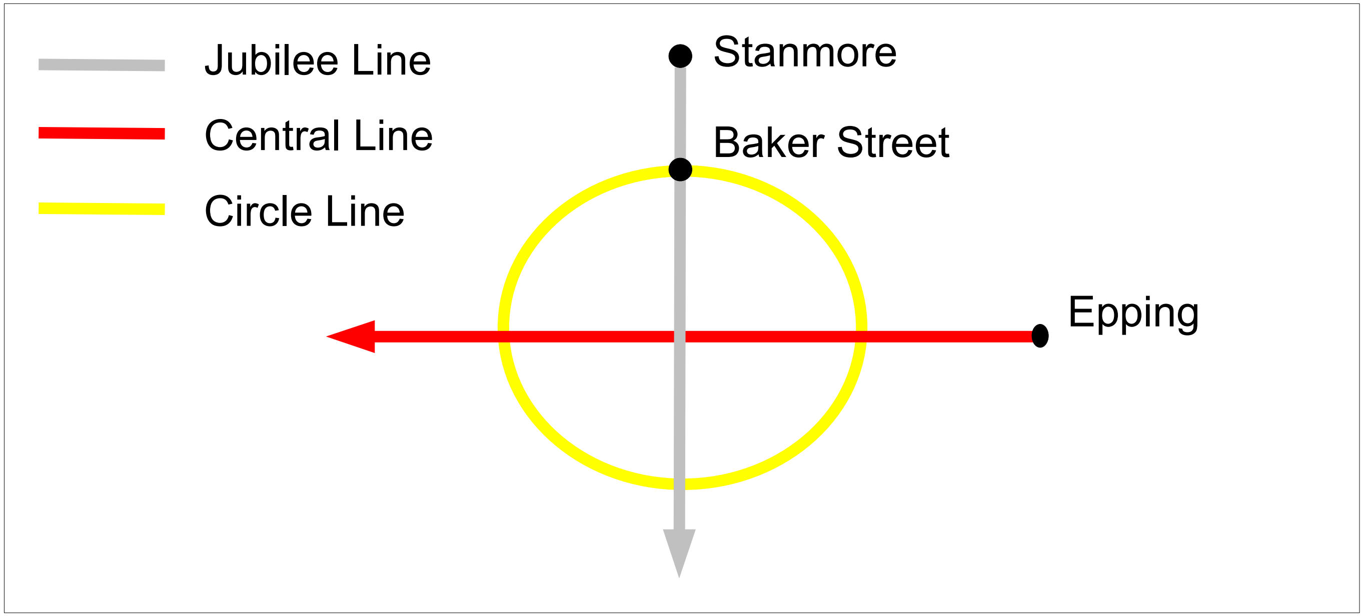
logic exercises: binary relations & join

- open *examples/tutorial/properties.als*
  - explores properties of binary relations
- open *examples/tutorial/distribution.als*
  - explores the distributivity of the join operator
- follow the instructions in the models
- don't hesitate to ask questions

logic exercise: modeling the tube

- open *examples/tutorial/tube.als*

- a simplified portion of the London Underground:



- follow the instructions in the model