Formal Methods in Software Engineering
Alloy

Fathiyeh Faghih

University of Tehran

Content taken from http://alloy.mit.edu/

Presentation outline

Introduction

Logic

Language

Modeling
- Static
- Dynamic

What is Alloy?

- A language for describing structures and a tool for exploring them
- Used in a wide range of applications from finding holes in security mechanisms to designing telephone switching networks
- An Alloy model is a collection of constraints that describes (implicitly) a set of structures
  - All the possible security configurations of a web application
  - All the possible topologies of a switching network

What is Alloy?

- The Alloy Analyzer is a solver that takes the constraints of a model and finds structures that satisfy them.
- It can be used both to explore the model by generating sample structures, and to check properties of the model by generating counterexamples.
- Structures are displayed graphically, and their appearance can be customized for the domain at hand.

What is Alloy?

- Alloy is a great tool for learning about first-order logic
- The difference between model finding and model checking. Alloy is a model finder
- The Alloy Analyzer works by reduction to SAT. Version 4 was a complete rewrite that included Kodkod, a new model finding engine that optimizes the reduction.

What is Alloy?

- Notation inspired by Z
  - sets and relations
  - but not easily analyzed
- Analysis inspired by SMV
  - billions of cases in seconds
  - 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]

Alloy Case Studies

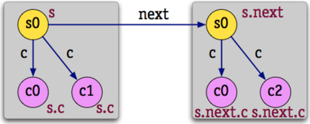
- Multilevel security (Bolton)
- Multicast key management (Taghdiri)
- Rendezvous (Jazayeri)
- Firewire (Jackson)
- Intentional naming (Khurshid)
- Java views (Waingold)
- Access control (Zao)
- Proton therapy (Seater, Dennis)
- Chord peer-to-peer (Kaashoek)
- Unison file sync (Pierce)
- Telephone switching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

Four Key Ideas

1. Everything is a relation
2. Non-specialized logic
3. Counterexamples & scope
4. Analysis by SAT

Everything is a Relation!

- Alloy uses relations for:
  - All data types – even sets, scalars, tuples
- key operator is **dot** join:
  - Field navigation
  - Function application

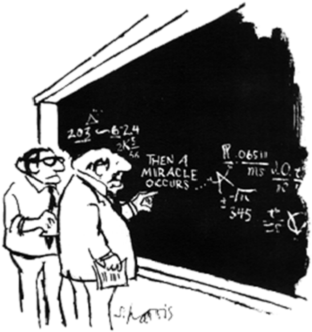


Why Relations?

- Easy to understand
- Binary relation is a graph or mapping
- Easy to analyze

Non-Specialized Logic

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



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

Counterexamples and Scope

Observations about design analysis: Most flaws have small counterexamples...

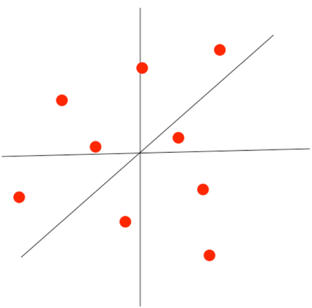


Figure : testing: a few cases of arbitrary size

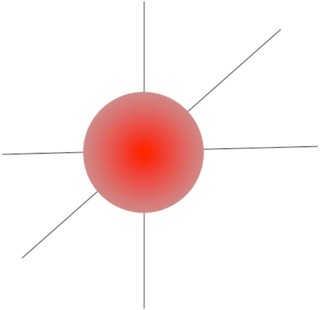
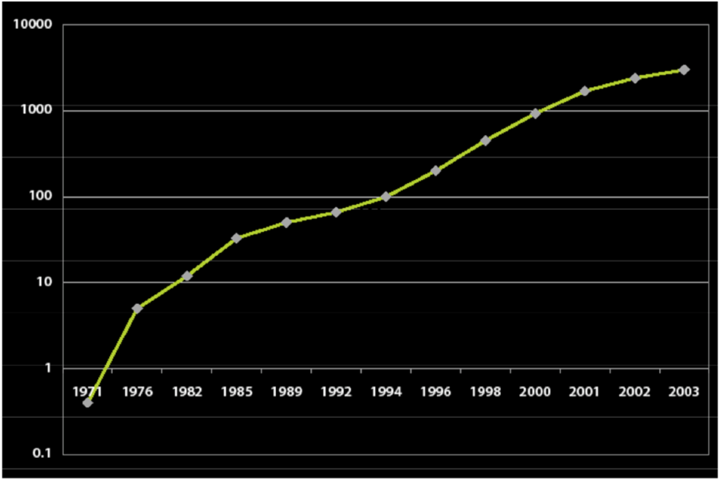


Figure : scope-complete: all cases within a small bound

Analysis by SAT

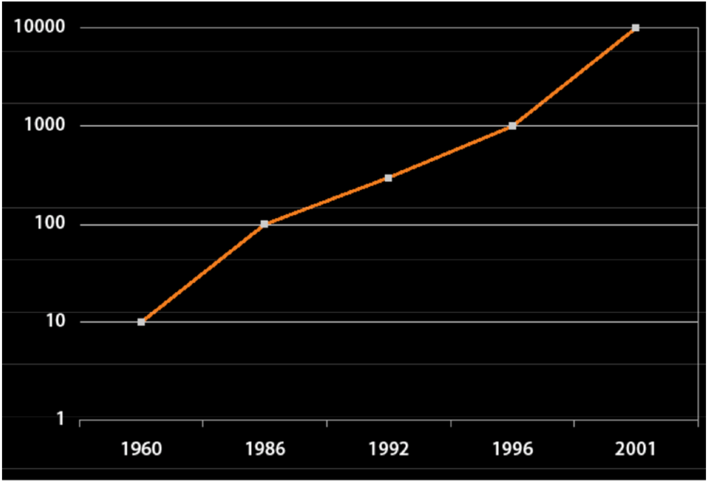
- SAT, the quintessential hard problem (Cook 1971)
  - SAT is hard, so reduce SAT to your problem.
- SAT, the universal constraint solver (Kautz, Selman, ... 1990's)
  - SAT is easy, so reduce your problem to SAT
  - solvers: Chaff (Malik), Berkmin (Goldberg & Novikov), ...

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



Install the Alloy Analyzer

- Requires Java 5 runtime environment
  - http://java.sun.com/
- Download the Alloy Analyzer 4.2
  - http://alloy.mit.edu/
- Run the Analyzer
  - Double click alloy4.jar or
  - Execute java -jar alloy4.jar at the command line

Verify the Installation

- Click the “file” menu, then click “open sample models” to open examples/toys/ceilingsAndFloors.als
- Click the “Execute” icon output shows graphic
- Need troubleshooting? http://alloy.mit.edu/

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
```

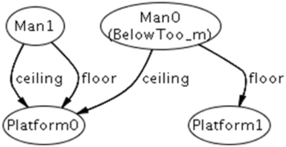
Modeling “Ceilings and Floors”

```
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 2or less platforms and men?
```

Clicking “Execute” ran this command. A counterexample found, shown in graphic.

Counterexample to “BelowToo”



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

Presentation outline

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 $\langle 1 column \rangle$ relations
  - Name = $\{\langle N0 \rangle, \langle N1 \rangle, \langle N2 \rangle\}$
  - Addr = $\{\langle A0 \rangle, \langle A1 \rangle, \langle A2 \rangle\}$
  - Book = $\{\langle B0 \rangle, \langle B1 \rangle\}$
- Scalars are singleton sets
  - myName = $\{\langle N1 \rangle\}$
  - yourName = $\{\langle N2 \rangle\}$
  - myBook = $\{\langle B0 \rangle\}$
- Binary relation
  - names = $\{\langle B0, N0 \rangle, \langle B0, N1 \rangle, \langle B1, N2 \rangle\}$
- Ternary relation
  - addrs = $\{\langle B0, N0, A0 \rangle, \langle B0, N1, A \rangle, \langle B1, N1, A2 \rangle, \langle B1, N2, A2 \rangle\}$

Constants

none	empty set
univ	universal set
iden	identity relation

Example

Name = $\{\langle N_0\rangle, \langle N_1\rangle, \langle N_2\rangle\}$
Addr = $\{\langle A_0\rangle, \langle A_1\rangle, \langle A_2\rangle\}$

none=

Constants

none	empty set
univ	universal set
iden	identity relation

Example

Name = $\{\langle N_0\rangle, \langle N_1\rangle, \langle N_2\rangle\}$
Addr = $\{\langle A_0\rangle, \langle A_1\rangle, \langle A_2\rangle\}$

none= $\{\}$
univ=

Constants

none	empty set
univ	universal set
iden	identity relation

Example

Name = $\{\langle N_0 \rangle, \langle N_1 \rangle, \langle N_2 \rangle\}$
Addr = $\{\langle A_0 \rangle, \langle A_1 \rangle, \langle A_2 \rangle\}$

none= $\{\}$
univ= $\{\langle N_0 \rangle, \langle N_1 \rangle, \langle N_2 \rangle, \langle A_0 \rangle, \langle A_1 \rangle, \langle A_2 \rangle\}$
iden=

Constants

none	empty set
univ	universal set
iden	identity relation

Example

Name = $\{\langle N_0\rangle, \langle N_1\rangle, \langle N_2\rangle\}$
Addr = $\{\langle A_0\rangle, \langle A_1\rangle, \langle A_2\rangle\}$

none= $\{\}$
univ= $\{\langle N_0\rangle, \langle N_1\rangle, \langle N_2\rangle, \langle A_0\rangle, \langle A_1\rangle, \langle A_2\rangle\}$
iden= $\{\langle N_0, N_0\rangle, \langle N_1, N_1\rangle, \langle N_2, N_2\rangle, \langle A_0, A_0\rangle, \langle A_1, A_1\rangle, \langle A_2, A_2\rangle\}$

Set Operators

+	union
&	intersection
-	difference
in	subset
=	equality

Example

cacheAddr = $\{\langle N0, A0\rangle, \langle N1, A1\rangle\}$
diskAddr = $\{\langle N0, A0\rangle, \langle N1, A2\rangle\}$

cacheAddr **+** diskAddr =

Set Operators

+	union
&	intersection
-	difference
in	subset
=	equality

Example

cacheAddr = $\{\langle N0, A0\rangle, \langle N1, A1\rangle\}$
diskAddr = $\{\langle N0, A0\rangle, \langle N1, A2\rangle\}$

cacheAddr **+** diskAddr = $\{\langle N0, A0\rangle, \langle N1, A1\rangle, \langle N1, A2\rangle\}$
cacheAddr **&** diskAddr =

Set Operators

+	union
&	intersection
-	difference
in	subset
=	equality

Example

cacheAddr = $\{\langle N0, A0\rangle, \langle N1, A1\rangle\}$
diskAddr = $\{\langle N0, A0\rangle, \langle N1, A2\rangle\}$

cacheAddr **+** diskAddr = $\{\langle N0, A0\rangle, \langle N1, A1\rangle, \langle N1, A2\rangle\}$
cacheAddr **&** diskAddr = $\{\langle N0, A0\rangle\}$
cacheAddr **=** diskAddr =

Set Operators

+	union
&	intersection
-	difference
in	subset
=	equality

Example

cacheAddr = $\{\langle N0, A0\rangle, \langle N1, A1\rangle\}$
diskAddr = $\{\langle N0, A0\rangle, \langle N1, A2\rangle\}$

cacheAddr **+** diskAddr = $\{\langle N0, A0\rangle, \langle N1, A1\rangle, \langle N1, A2\rangle\}$
cacheAddr **&** diskAddr = $\{\langle N0, A0\rangle\}$
cacheAddr **=** diskAddr = *false*
cacheAddr **in** diskAddr =

Set Operators

+	union
&	intersection
-	difference
in	subset
=	equality

Example

cacheAddr = $\{\langle N0, A0\rangle, \langle N1, A1\rangle\}$
diskAddr = $\{\langle N0, A0\rangle, \langle N1, A2\rangle\}$

cacheAddr **+** diskAddr = $\{\langle N0, A0\rangle, \langle N1, A1\rangle, \langle N1, A2\rangle\}$
cacheAddr **&** diskAddr = $\{\langle N0, A0\rangle\}$
cacheAddr **=** diskAddr = *false*
cacheAddr **in** diskAddr = *false*

Product Operators

Example

Name = $\{\langle N0\rangle, \langle N1\rangle\}$
Addr = $\{\langle A0\rangle, \langle A1\rangle\}$
Book = $\{\langle B0\rangle\}$

Name $->$ Addr =

Product Operators

Example

Name = $\{\langle N0\rangle, \langle N1\rangle\}$
Addr = $\{\langle A0\rangle, \langle A1\rangle\}$
Book = $\{\langle B0\rangle\}$

Name $->$ Addr = $\{\langle N0, A0\rangle, \langle N0, A1\rangle, \langle N1, A0\rangle, \langle N1, A1\rangle\}$
Book$->$Name$->$Addr =

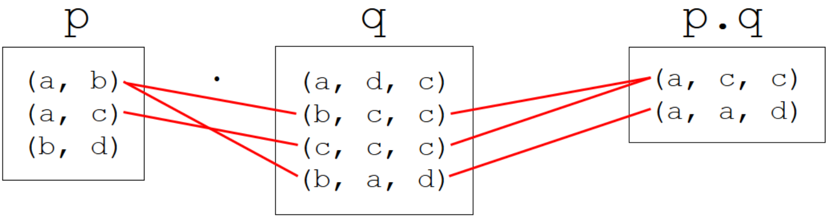
Product Operators

Example

Name = $\{\langle N0\rangle, \langle N1\rangle\}$
Addr = $\{\langle A0\rangle, \langle A1\rangle\}$
Book = $\{\langle B0\rangle\}$

Name $->$ Addr = $\{\langle N0, A0\rangle, \langle N0, A1\rangle, \langle N1, A0\rangle, \langle N1, A1\rangle\}$
Book$->$Name$->$Addr =
$\{\langle B0, N0, A0\rangle, \langle B0, N0, A1\rangle, \langle B0, N1, A0\rangle, \langle B0, N1, A1\rangle\}$

Relational Join



Relational Operators

.	dot join
[]	box join

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

Example

Name = $\{\langle N0\rangle, \langle N1\rangle\}$
Book = $\{\langle B0\rangle\}$
myName = $\{\langle N1\rangle\}$
myAddr = $\{\langle A0\rangle\}$
address = $\{\langle B0, N0, A0\rangle, \langle B0, N1, A0\rangle, \langle B0, N2, A2\rangle\}$
next = $\{\langle N0, N1\rangle, \langle N1, N2\rangle, \langle N2, N3\rangle\}$

Book.address =

Relational Operators

.	dot join
[]	box join

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

Example

Name = $\{\langle N0 \rangle, \langle N1 \rangle\}$
Book = $\{\langle B0 \rangle\}$
myName = $\{\langle N1 \rangle\}$
myAddr = $\{\langle A0 \rangle\}$
address = $\{\langle B0, N0, A0 \rangle, \langle B0, N1, A0 \rangle, \langle B0, N2, A2 \rangle\}$
next = $\{\langle N0, N1 \rangle, \langle N1, N2 \rangle, \langle N2, N3 \rangle\}$

Book.address = $\{\langle N0, A0 \rangle, \langle N1, A0 \rangle, \langle N2, A2 \rangle\}$
Book.address[myName] =

Relational Operators

.	dot join
[]	box join

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

Example

Name = $\{\langle N0\rangle, \langle N1\rangle\}$
Book = $\{\langle B0\rangle\}$
myName = $\{\langle N1\rangle\}$
myAddr = $\{\langle A0\rangle\}$
address = $\{\langle B0, N0, A0\rangle, \langle B0, N1, A0\rangle, \langle B0, N2, A2\rangle\}$
next = $\{\langle N0, N1\rangle, \langle N1, N2\rangle, \langle N2, N3\rangle\}$

Book.address = $\{\langle N0, A0\rangle, \langle N1, A0\rangle, \langle N2, A2\rangle\}$
Book.address[myName] = $\{\langle A0\rangle\}$
Book.address.myName =

Relational Operators

.	dot join
[]	box join

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

Example

Name = $\{\langle N0 \rangle, \langle N1 \rangle\}$
Book = $\{\langle B0 \rangle\}$
myName = $\{\langle N1 \rangle\}$
myAddr = $\{\langle A0 \rangle\}$
address = $\{\langle B0, N0, A0 \rangle, \langle B0, N1, A0 \rangle, \langle B0, N2, A2 \rangle\}$
next = $\{\langle N0, N1 \rangle, \langle N1, N2 \rangle, \langle N2, N3 \rangle\}$

Book.address = $\{\langle N0, A0 \rangle, \langle N1, A0 \rangle, \langle N2, A2 \rangle\}$
Book.address[myName] = $\{\langle A0 \rangle\}$
Book.address.myName = $\{\}$

Unary Operators on Binary Relations

$\sim$	transpose
$\hat{}$	transitive closure
$*$	reflexive transitive closure

$$\hat{}r = r + r.r + r.r.r + \ldots$$

$$*r = iden + \hat{}r$$

Example

Node = $\{\langle N0\rangle, \langle N1\rangle, \langle N2\rangle, \langle N3\rangle\}$
next = $\{\langle N0, N1\rangle, \langle N1, N2\rangle, \langle N2, N3\rangle\}$

$\sim$next =

Unary Operators on Binary Relations

$\sim$	transpose
$\hat{}$	transitive closure
$*$	reflexive transitive closure

$$\hat{}r = r + r.r + r.r.r + \ldots$$

$$*r = iden + \hat{}r$$

Example

Node = $\{\langle N0\rangle, \langle N1\rangle, \langle N2\rangle, \langle N3\rangle\}$
next = $\{\langle N0, N1\rangle, \langle N1, N2\rangle, \langle N2, N3\rangle\}$

$\sim$next = $\{\langle N1, N0\rangle, \langle N2, N1\rangle, \langle N3, N2\rangle\}$
$\hat{}$next =

Unary Operators on Binary Relations

$\sim$	transpose
$\hat{}$	transitive closure
$*$	reflexive transitive closure

$$\hat{}r = r + r.r + r.r.r + \ldots$$

$$*r = iden + \hat{}r$$

Example

Node = $\{\langle N0 \rangle, \langle N1 \rangle, \langle N2 \rangle, \langle N3 \rangle\}$
next = $\{\langle N0, N1 \rangle, \langle N1, N2 \rangle, \langle N2, N3 \rangle\}$

$\sim$next = $\{\langle N1, N0 \rangle, \langle N2, N1 \rangle, \langle N3, N2 \rangle\}$
$\hat{}$next = $\{\langle N0, N1 \rangle, \langle N0, N2 \rangle, \langle N0, N3 \rangle, \langle N1, N2 \rangle, \langle N1, N3 \rangle, \langle N2, N3 \rangle\}$

Logic: Boolean Operators

!	**not**	negation
&&	**and**	conjunction
||	**or**	disjunction
=>	**implies else**	implication alternative
<=>	**iff**	double implication

Example of equivalent formulas

F => G else H
F implies G else H

Logic: Boolean Operators

!	**not**	negation
&&	**and**	conjunction
||	**or**	disjunction
=>	**implies else**	implication alternative
<=>	**iff**	double implication

Example of equivalent formulas

F => G else H
F implies G else H
(F && G) || ((!F) && H)

Logic: Boolean Operators

!	**not**	negation
&&	**and**	conjunction
||	**or**	disjunction
=>	**implies else**	implication alternative
<=>	**iff**	double implication

Example of equivalent formulas

F => G else H
F implies G else H
(F && G) || ((!F) && H)
(F and G) or ((not F) and H)

Quantifiers

all	for every
some	at least one
no	no
lone	at most one
one	exactly one

Example

some n: Name, a: Address | a in n.address

Quantifiers

all	for every
some	at least one
no	no
lone	at most one
one	exactly one

Example

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

all n: Name | lone a: Address | a in n.address

Quantifiers

all	for every
some	at least one
no	no
lone	at most one
one	exactly one

Example

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

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

Subset Declarations

set	any number
one	exactly one
lone	zero or one
some	one or more

Example

RecentlyUsed: **set** Name

Subset Declarations

set	any number
one	exactly one
lone	zero or one
some	one or more

Example

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

Subset Declarations

set	any number
one	exactly one
lone	zero or one
some	one or more

Example

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

senderAddress: **one** Addr

Subset Declarations

set	any number
one	exactly one
lone	zero or one
some	one or more

Example

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

senderAddress: **one** Addr
senderAddress is a singleton subset of Addr

Subset Declarations

set	any number
one	exactly one
lone	zero or one
some	one or more

Example

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

senderAddress: **one** Addr
senderAddress is a singleton subset of Addr

senderName: **lone** Name

Subset Declarations

set	any number
one	exactly one
lone	zero or one
some	one or more

Example

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

senderAddress: **one** Addr
senderAddress is a singleton subset of Addr

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

Subset Declarations

set	any number
one	exactly one
lone	zero or one
some	one or more

Example

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

senderAddress: **one** Addr
senderAddress is a singleton subset of Addr

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

receiverAddresses: **some** Addr

Subset Declarations

set	any number
one	exactly one
lone	zero or one
some	one or more

Example

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

senderAddress: **one** 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

Quantified Expression

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

Example

some Name
set of names is not empty

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

no (address.Addr - Name)

Quantified Expression

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

Example

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

Quantified Expression

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

Example

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

Two Logics in One

"Everybody loves a winner."

- predicate logic:

Two Logics in One

"Everybody loves a winner."

- predicate logic:

$$\forall w. Winner(w) \rightarrow \forall p. Loves(p, w)$$

- relational calculus:

Two 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

Two 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

Two 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

Two 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

Example

Is there a relation that is transitive, irreflexive, symmetric, functional, one-to-one, total, and onto?

Example

Is there a relation that is transitive, irreflexive, symmetric, functional, one-to-one, total, and onto?

```
pred show {
some r: univ -> univ {
some r   // non empty
```

Example

Is there a relation that is transitive, irreflexive, symmetric, functional, one-to-one, total, and onto?

```
pred show {
some r: univ -> univ {
some r   // non empty
r.r in r    // transitive
```

Example

Is there a relation that is transitive, irreflexive, symmetric, functional, one-to-one, total, and onto?

```
pred show {
some r: univ -> univ {
some r   // non empty
r.r in r    // transitive
no iden & r // irreflexive
```

Example

Is there a relation that is transitive, irreflexive, symmetric, functional, one-to-one, total, and onto?

```
pred show {
some r: univ -> univ {
some r   // non empty
r.r in r    // transitive
no iden & r // irreflexive
~r in r     // symmetric
```

Example

Is there a relation that is transitive, irreflexive, symmetric, functional, one-to-one, total, and onto?

```
pred show {
some r: univ -> univ {
some r   // non empty
r.r in r    // transitive
no iden & r // irreflexive
~r in r     // symmetric
~r.r in iden // functional
```

Example

Is there a relation that is transitive, irreflexive, symmetric, functional, one-to-one, total, and onto?

```
pred show {
some r: univ -> univ {
some r   // non empty
r.r in r    // transitive
no iden & r // irreflexive
~r in r     // symmetric
~r.r in iden // functional
r.~r in iden // one-to-one
```

Example

Is there a relation that is transitive, irreflexive, symmetric, functional, one-to-one, total, and onto?

```
pred show {
some r: univ -> univ {
some r   // non empty
r.r in r    // transitive
no iden & r // irreflexive
~r in r     // symmetric
~r.r in iden // functional
r.~r in iden // one-to-one
univ in r.univ // total
```

Example

Is there a relation that is transitive, irreflexive, symmetric, functional, one-to-one, total, and onto?

```
pred show {
some r: univ -> univ {
some r   // non empty
r.r in r    // transitive
no iden & r // irreflexive
~r in r     // symmetric
~r.r in iden // functional
r.~r in iden // one-to-one
univ in r.univ // total
univ in univ.r // onto
}
}
run show for 4
```

Example: Distributivity

```
assert union {
all s: set univ, p, q: univ -> univ |
s.(p + q) = s.p + s.q
}
assert difference {
all s: set univ, p, q: univ -> univ |
s.(p - q) = s.p - s.q
}
assert intersection {
all s: set univ, p, q: univ -> univ |
s.(p & q) = s.p & s.q
}
check union for 4
check difference for 4
check intersection for 4
```

Presentation outline

Introduction

Logic

Language

Modeling

- Static
- Dynamic

Example: 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 {
no p: Person |
p in p.^(mother + father)
wife = ~husband
}
assert noSelfFather {
no m: Man | m = m.father
}

check noSelfFather
```

Example: Grandpa, Continued

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

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

run ownGrandpa for 4Person
```

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

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

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

abstract sig A {}
sig B extends A {}
sig C extends A {} //A partitioned by disjoint subsets B and C
```

Signatures: Grandpa

```
abstract sig Person {
. . .
}

sig Man extends Person {
. . .
}

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

Signatures: Grandpa

```
abstract sig Person {
. . .
}

sig Man extends Person {
. . .
}

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

- All men and women are persons.

Signatures: Grandpa

```
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.

Signatures: Grandpa

```
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.

Fields: Grandpa

```
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.

Facts

Facts introduce constraints that are assumed to always hold.

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

```
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
```

Facts: Grandpa

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

Facts: Grandpa

```
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.

Functions

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]
}
```

Predicates

A predicate is a named formula with declaration parameters.

Predicates

A predicate is a 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, d]
}
```

Predicates: Grandpa

```
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.

Assertions

```
sig Node {
children: set Node
}

one sig Root extends Node {}

fact {
Node in Root.*children
}

assert someParent {
all n: Node | some children.n
}
```

Assertions

```
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
}
```

Check Command

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

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

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

Check Command

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

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

check noSelfFather
```

Run Command

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

Instructs analyzer to search for instance of predicate within scope.

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

Instructs analyzer to search for instance of function within scope.

Barber Paradox

- Consider the set of all sets that do not contain themselves as members.
  - Does it contain itself?
- This paradox was discovered by Bertrand Russell in 1901.
- A variant of the paradox, also attributed to Bertrand Russell, asks: in a village in which the barber shaves every man who doesn't shave himself, who shaves the barber?

Barber Paradox

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

Barber Paradox

Feminists have noted that the paradox disappears if the existence of women is acknowledged.

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

run { }
```

Other Solutions to Barber Paradox

Dijkstra: No barber...

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

run { }
```

Other Solutions to Barber Paradox

Dijkstra: No barber...

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

run { }
```

More than one barber...

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

run { }
```

Presentation outline

Static vs. Dynamic Modeling

- Static models
  - Describes states, not behaviors
  - Properties are invariants.
  - e.g. that a list is sorted
- Dynamic model
  - Describe transitions between states.
  - Properties are operations.
  - e.g. how a sorting algorithm works

Outline

MIT Course Scheduler

Course catalog and graduation requirements:

- Designed and built by Vincent Yeung
- Web application backed by Alloy engine
- Generate a course schedule to satisfy MIT degree requirements given past courses

Outline

Model of an Address Book

```
abstract sig Target {}
sig Name extends Target {}
sig Addr extends Target {}
sig Book { addr: Name -> Target }
pred init [b: Book] { no b.addr }
pred inv [b: Book] {
let addr = b.addr | all n: Name {
n not in n.^addr
some addr.n => some n.addr
}
}
fun lookup [b: Book, n: Name] : set Addr {
n.^(b.addr) & Addr
}
assert namesResolve {
all b: Book | inv[b] =>
all n: Name | some b.addr[n] => some lookup[b, n]
}
check namesResolve for 4
```

What about Operations?

How is a name and address added to a book?

- No built-in model of execution
- No notion of time or mutable state
- Need to model time/state explicitly

Can use a new book after each mutation:

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

Delete Operation

Write a predicate for a delete operation:

- Removes a name-target pair from a book

Delete Operation

Write a predicate for a delete operation:

- Removes a name-target pair from a book

```
pred remove [b, b': Book, n: Name, t: Target] {
b'.addr = b.addr - n -> t
}
```

Delete Operation

Assert and check that delete is the undo of add:

Delete Operation

Assert and check that delete is the undo of add:

```
assert isequiv{
all b,b’: Book, n: Name, t: Target |
add[b,b’,n,t] => remove[b’,b,n,t]
}
```

Delete Operation

Assert and check that delete is the undo of add:

```
assert isequiv{
all b,b': Book, n: Name, t: Target |
add[b,b',n,t] => remove[b',b,n,t]
}
```

Why does this fail?

Delete Operation

Edit your assertion, so that it is satisfied.

Delete Operation

Edit your assertion, so that it is satisfied.

```
assert isequiv{
all b,b': Book, n: Name, t: Target | (not(n->t in b.addr)) =>
(add[b,b',n,t] => remove[b',b,n,t])
}
```