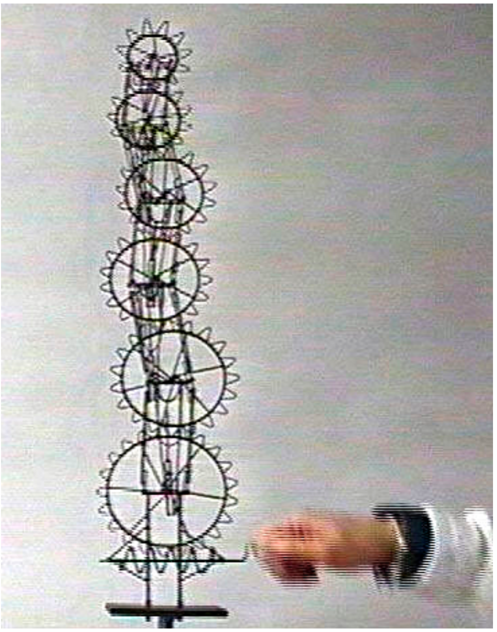
alloy:
an analyzable modelling language

Daniel Jackson, MIT
Ilya Shlyakhter
Manu Sridharan

Praxis Critical Systems
April 25, 2003



Small Tower of 6 Gears, Arthur Ganson

preaching to the choir

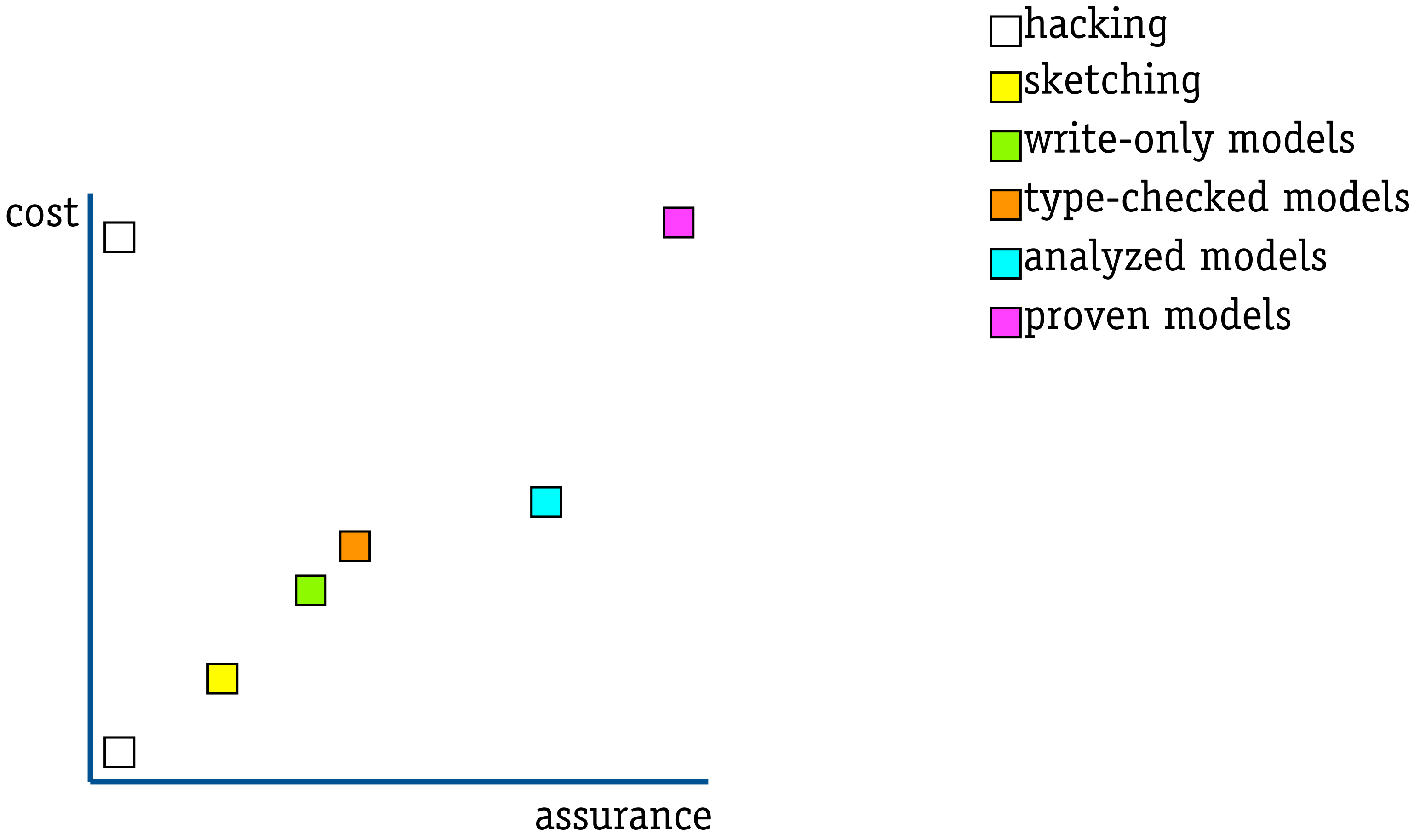
explicit models before code

- › higher quality
- › easier coding

formalism helps

- › forces simplicity
- › no wishful thinking
- › potential for tools

assurance/cost tradeoffs



lightweight formal methods

language must be

- small and simple
- well defined
- expressive enough

analysis must be

- fully automatic
- semantically deep

user doesn't want to

- provide test cases
- invent lemmas

alloy: a structural, analyzable logic

a notation inspired by Z

- just sets and relations
- familiar logical quantifiers
- simpler, less expressive, all ASCII

an analysis inspired by SMV

- billions of cases in second
- counterexamples, not proof
- declarative logic, not state machines

Oxford, home of Z

Pittsburgh, home of SMV

what to look out for

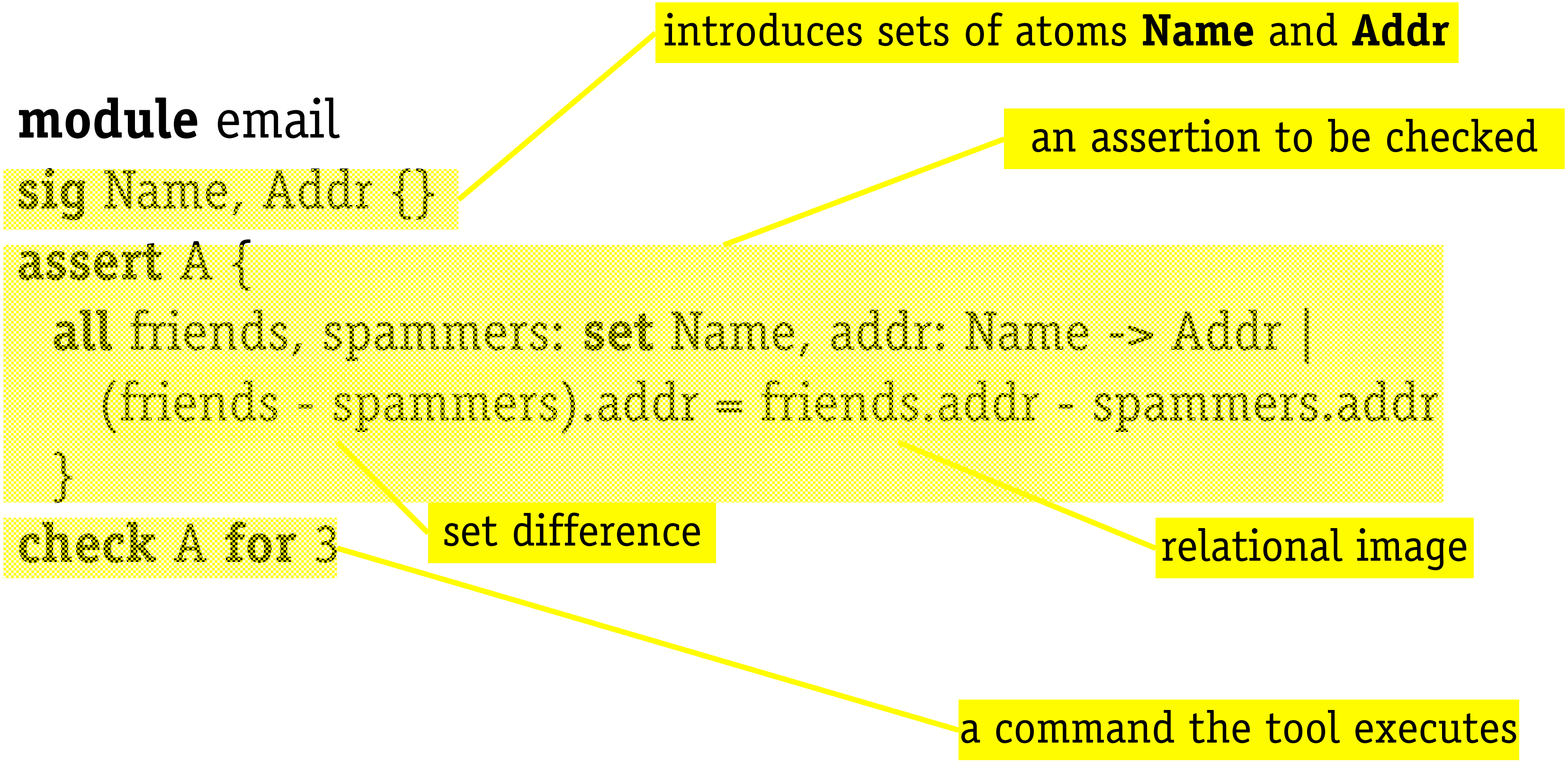
the language

- › all structure by relations
- › composites by higher-arity
- › entirely first order
- › familiar syntax by puns

the analysis

- › as in Z, everything’s a formula
- › tool tries all small tests within a “scope”
- › model itself is unbounded

a first alloy model



```
module email
sig Name, Addr {}
assert A {
  all friends, spammers: set Name, addr: Name -> Addr |
    (friends - spammers).addr = friends.addr - spammers.addr
  }
check A for 3
```

introduces sets of atoms **Name** and **Addr**

an assertion to be checked

set difference

relational image

a command the tool executes

analysis by constraint solving

```
module email
sig Name, Addr {}
assert A {
  all friends, spammers: set Name, addr: Name -> Addr |
    (friends - spammers).addr = friends.addr - spammers.addr
  }
check A for 3
```

- to check, first negate conjecture

  ```
  some f, s: set N, a: N -> A | not (f-s).a = f.a - s.a
  ```

- then skolemize away quantifiers

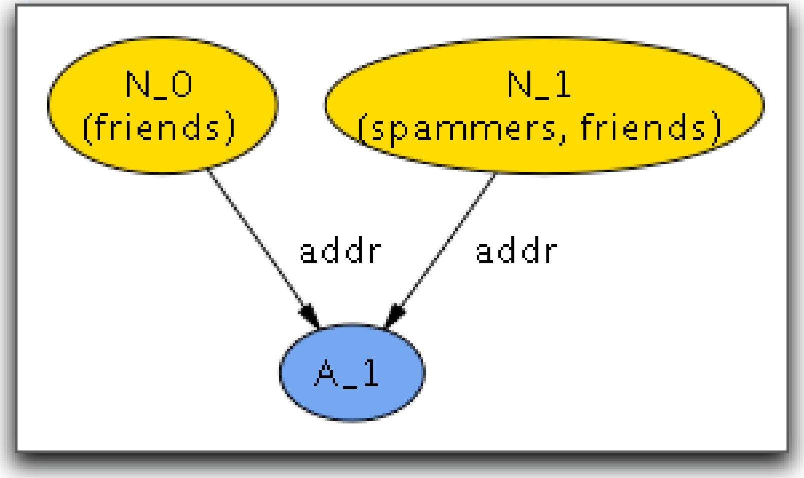
  ```
  not (f-s).a = f.a - s.a
  ```

- and now solve for constants

  ```
  f = {N0, N1} , s = {N1}, a = {N0->A1, N1->A1}
  ```

analysis by constraint solving

```
module email
sig Name, Addr {}
assert A {
  all friends, spammers: set Name, addr: Name -> Addr |
    (friends - spammers).addr = friends.addr - spammers.addr
  }
check A for 3
```



“try all small tests”

language is undecidable

- › so no sound & complete algorithm

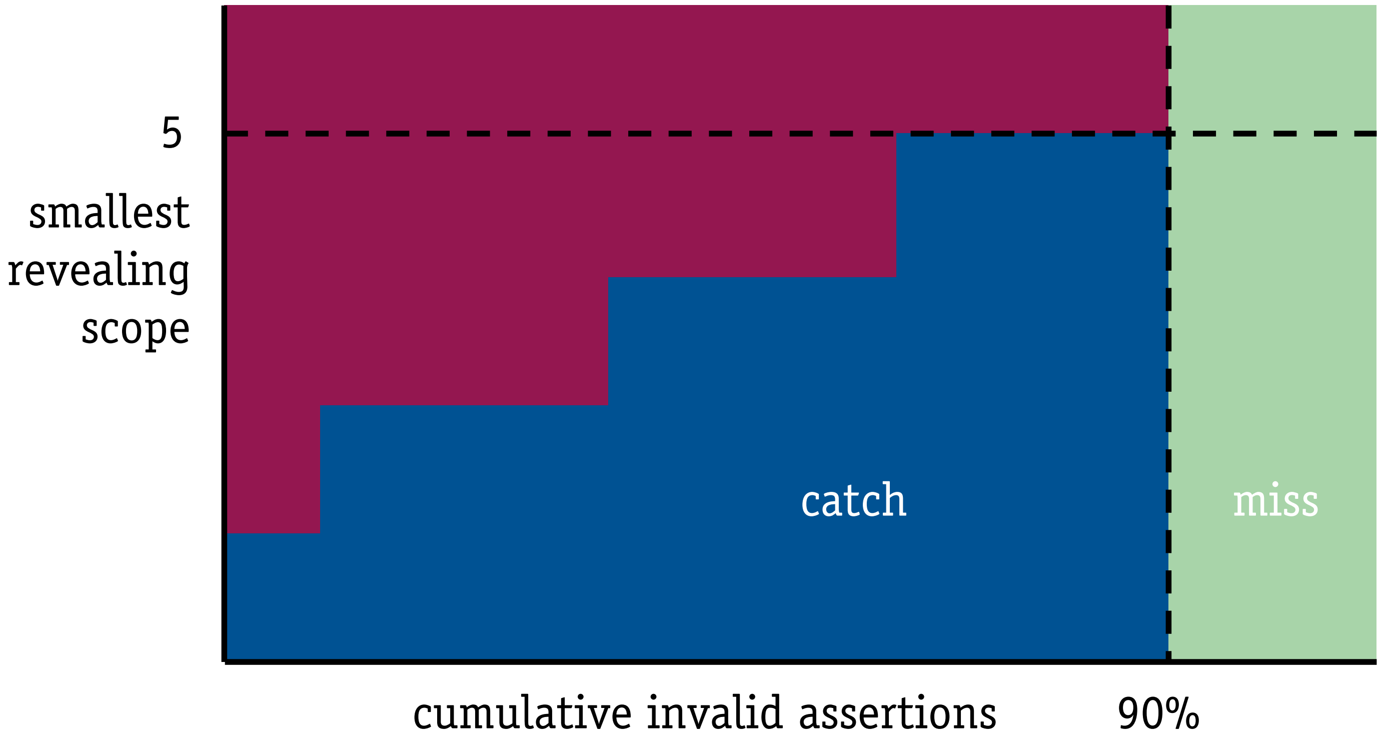
alloy’s analysis is refutation

- › look for a counterexample
- › consider all assignments of values to constants
- › user selects scope (here, 3 names and 3 addrs)

properties of models

- › usually flawed, especially in early stages
- › many bugs, even subtle ones, have small counterexamples

‘all small tests’



consequences

› sound: no false alarms

› incomplete: can’t prove anything

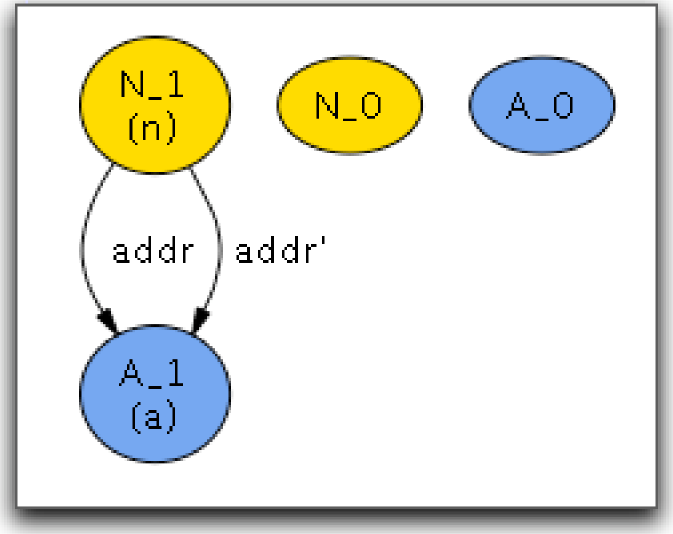
simulating an operation

declares a parameterized formula

```
module email
sig Name, Addr {}
fun add (addr, addr’: Name -> Addr, n: Name, a: Addr) {
  addr’ = addr + (n -> a)
  }
run add for 2
```

forms a tuple

set union



alloy semantics

all values are relations

{(a),(b)} is a set

{(a)} is a scalar

{(a,b)} is a tuple

higher-order values

- can't be represented directly

  AddrBook = ℙ(ℙ(Name □ Addr))

- can often be represented with higher-arity

  AddrBook -> Name -> Addr

expressions

expressions are made from variables and

- set operators

  p + q, p - q, p & q

- relational operators

  p.q, p->q, *p, ^p, ~p

  $[\![p \,.\, q]\!] = \{(p_1, \ldots\, p_{n-1}, q_2, \ldots\, q_m) \mid$

  $\quad (p_1, \ldots\, p_n) \in [\![p]\!] \wedge (q_1, \ldots\, q_m) \in [\![q]\!] \wedge p_n = q_1\}$

  $p \text{ -> } q = \{(p_1, \ldots\, p_n, q_1, \ldots\, q_m) \mid (p_1, \ldots\, p_n) \in [\![p]\!] \wedge (q_1, \ldots\, q_m) \in [\![q]\!]\}$

puns

for scalars a, b, sets S, T and relations p, q

a -> b is a tuple; S -> T is a relation

S.p is image; p.q is join

formulas

e **in** e′	e is a subset of e′
not F	
F **and** G	
F **or** G	
F **=>** G	
{ F G }	implicit conjunction
all x: X | F	
some x: X | F	
one x: X | F	there is exactly one x such that F
sole x: X | F	there is at most one x such that F
no x: X | F	
no e	there is no tuple in e; e is empty
some e	there is some tuple in e; e is non-empty
sole e	there is at most one tuple in e

fields

```
module email
sig Name, Addr {}
sig AddrBook {
  map: Name -> Addr
  }
fun add (b, b’: AddrBook, n: Name, a: Addr) {
  b’.map = b.map + n->a
  }
run add
```

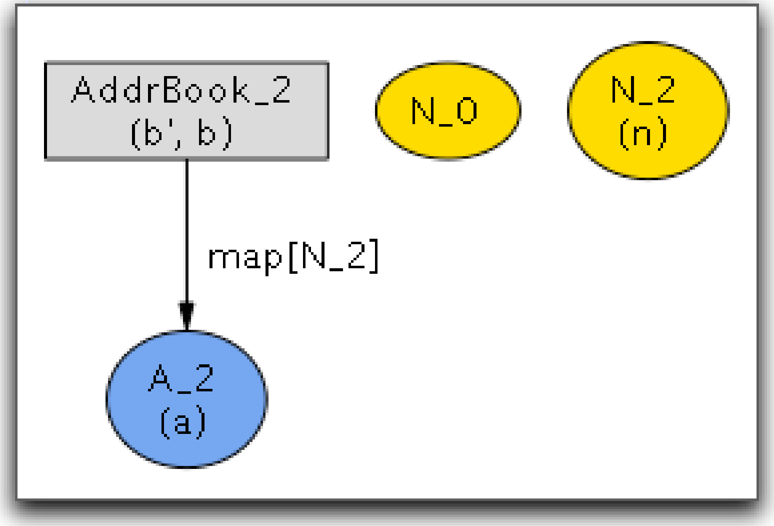
declares a ternary relation on **AddrBook**, **Name**, **Addr**

an instance

map = {AB2->N2->A2}

b = AB2, b’ = AB2

a = A2, n = N2

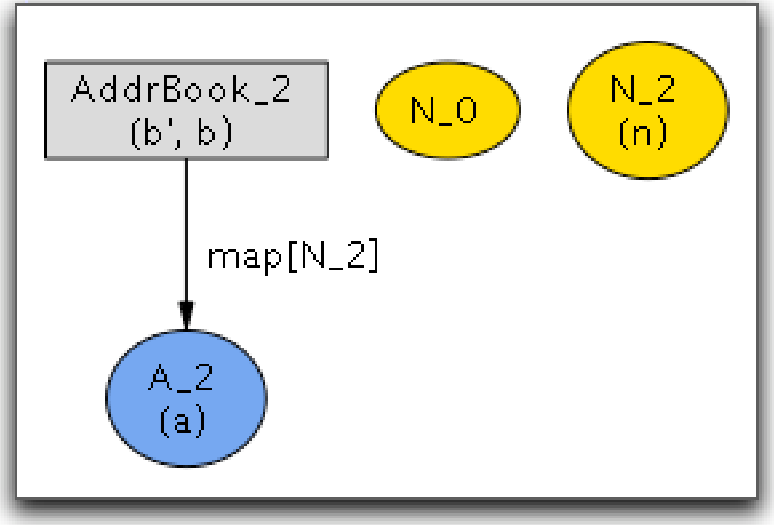


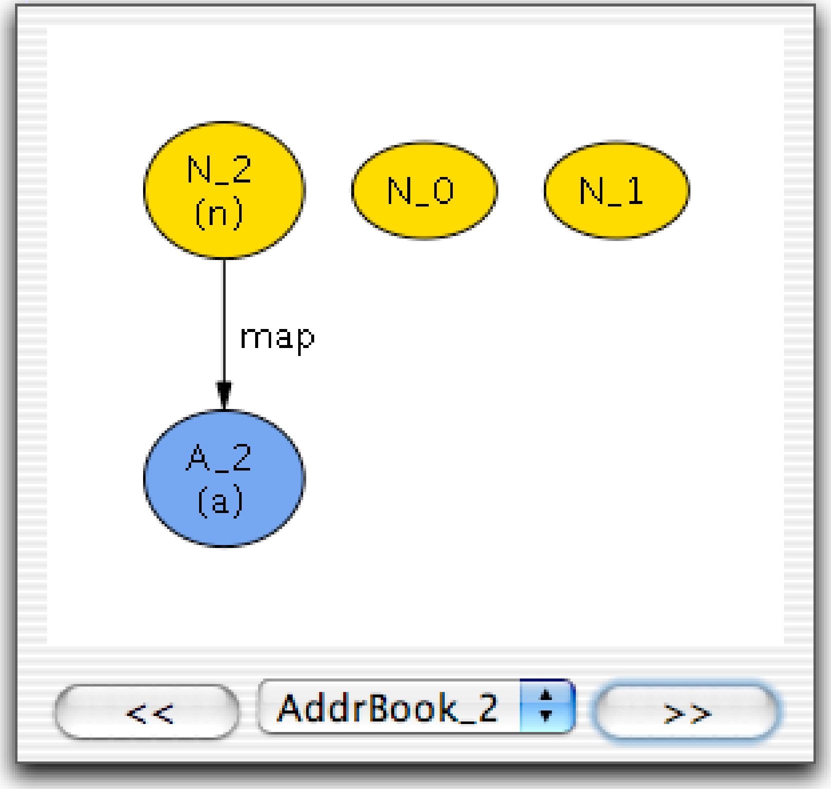
projection (a visualization technique)

to show ternary relations

› index the arcs

› or project a type





some conjectures

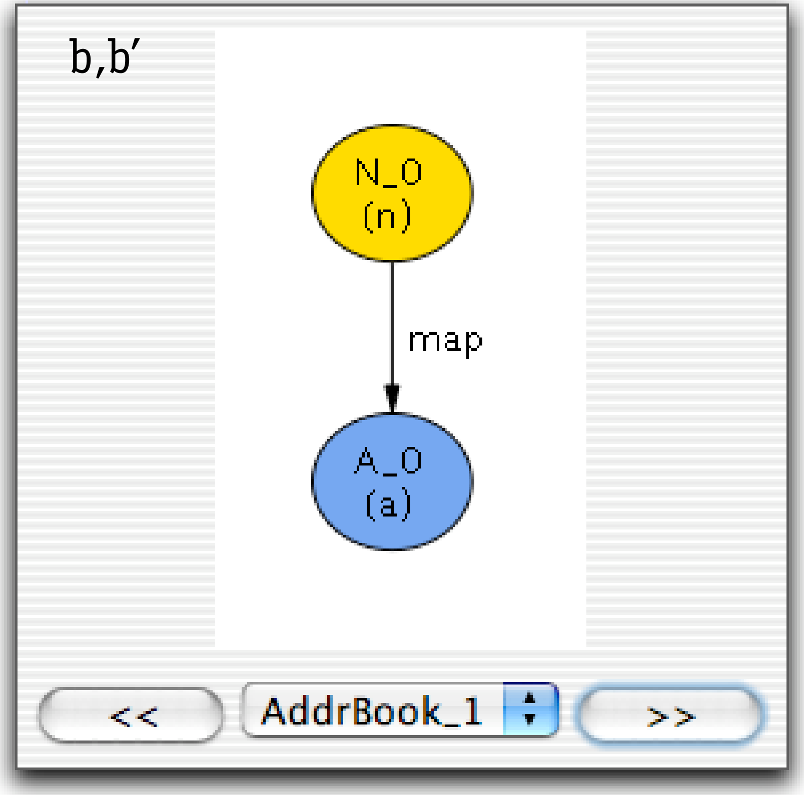
equivalent to **n.(b.map)**

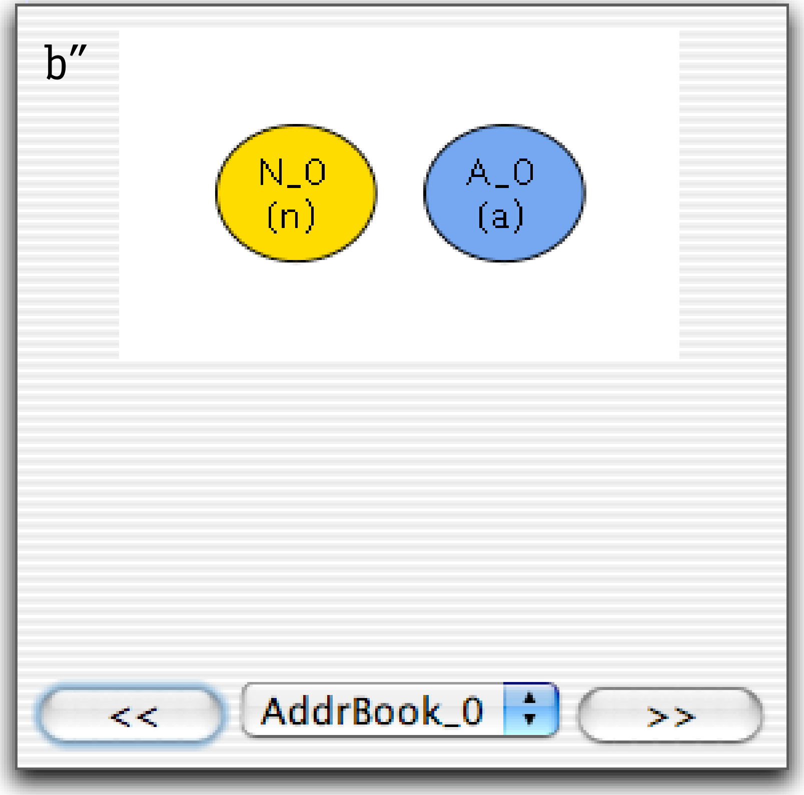
```
module email
sig Name, Addr {}
sig AddrBook {map: Name -> Addr}
fun add (b, b′: AddrBook, n: Name, a: Addr) {b′.map = b.map + n->a}
fun del (b, b′: AddrBook, n: Name) {b′.map = b.map - n->Addr}
fun lookup (b: AddrBook, n: Name): set Addr {result = b.map[n]}

✗assert delUndoesAdd {all b,b′,b″: AddrBook, n: Name, a: Addr |
   add (b,b′,n,a) and del (b′,b″,n) => b.map = b″.map }
✓assert addIdempotent {all b,b′,b″: AddrBook, n: Name, a: Addr |
   add (b,b′,n,a) and add (b′,b″,n,a) => b′.map = b″.map }
✓assert addLocal {all b,b′: AddrBook, n,n′: Name, a: Addr |
   add (b,b′,n,a) and n != n′ => lookup (b,n′) = lookup (b′,n′) }
```

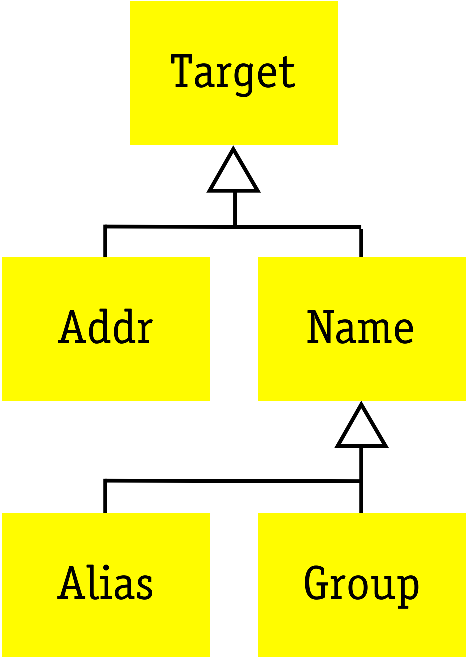
a counterexample

```
assert delUndoesAdd {all b,b′,b″: AddrBook, n: Name, a: Addr |
  add (b,b′,n,a) and del (b′,b″,n) => b.map = b″.map }
```





subsignatures

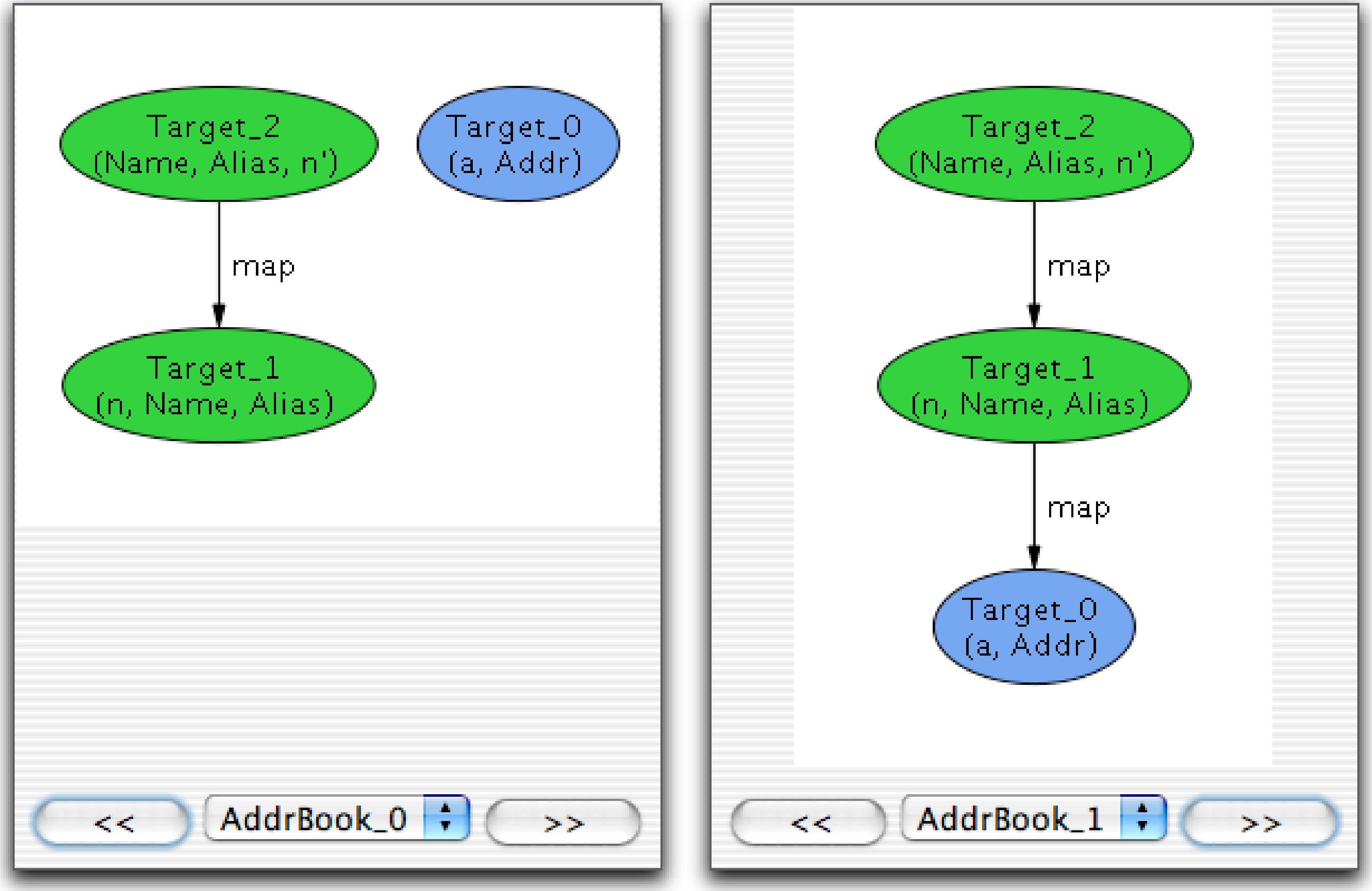


```
module email
sig Target
part sig Addr, Name extends Target {}
part sig Alias, Group extends Name {}
sig AddrBook {
  map: Name -> Target
  }

fun add (b, b': AddrBook, n: Name, t: Target) {b'.map = b.map + n->t}
fun lookup (b: AddrBook, n: Name): set Addr {
  result = n.^(b.map) & Addr }
```

counterexample

```
assert addLocal {all b,b′: AddrBook, n,n′: Name, a: Addr |
  add (b,b′,n,a) and n != n′ => lookup (b,n′) = lookup (b′,n′) }
```



fields of subsignatures

defines field **host** such that **no t.host** if **t !in Addr**

```
module email
sig Host, Target {}
disj sig Name extends Target {}
disj sig Addr extends Target {host: Host}
part sig Alias, Group extends Name {}
sig AddrBook {
  map: Name -> Target
  }{all a: Alias | sole map[a]}
fun getHosts (b: AddrBook, n: Name): set Hosts {
  result = n.^(b.map).host }
```

signature fact: **all this: AddrBook …** implicit

applies **host** to set of Target; no need to write **(expr & Addr).host**

no …

- partial functions, undefinedness, third logical value
- type casts

flexible declarations

decl says **names** is domain of **map**

```
module email
sig Target {}
part sig Addr, Name extends Target {}
part sig Alias, Group extends Name {}
sig AddrBook {names: set Name, map: names ->+ Target}

fun lookup (b: AddrBook, n: Name): set Addr {
  result = n.^(b.map) & Addr }
fun add (b, b': AddrBook, n: Name, a: Target) {
  a in Addr or some lookup(b,a)
  b'.map = b.map + n->a}
fun del (b, b': AddrBook, n: Name) {b'.map = b.map - n->Addr}
```

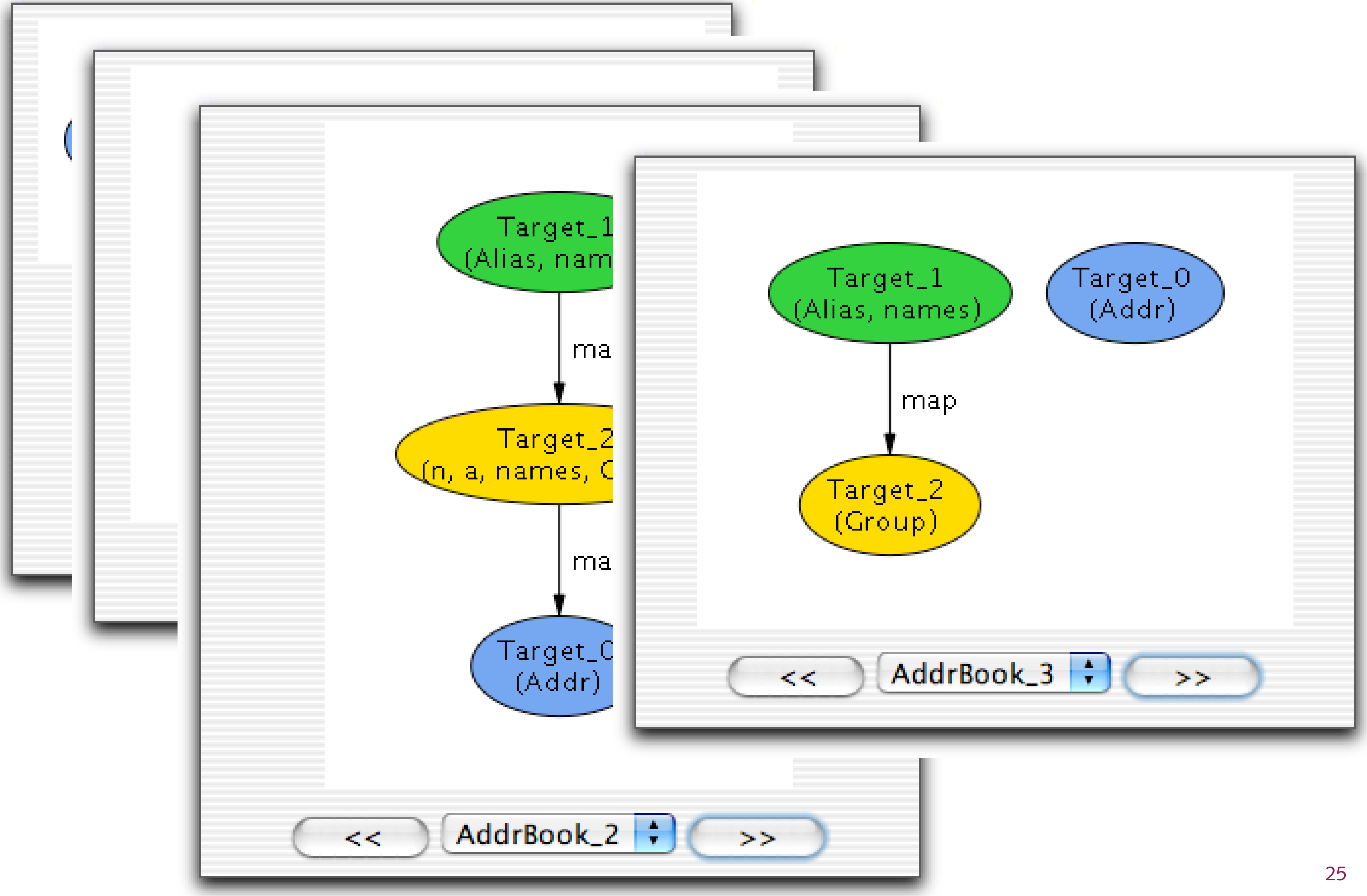
traces

```
module email
open std/ord
sig Target {}…
sig AddrBook {names: set Name, map: names ->+ Target}
fun lookup (b: AddrBook, n: Name): set Addr {…}
fun add (b, b': AddrBook, n: Name, a: Target) {…}
fun del (b, b': AddrBook, n: Name) {…}
fun init (b: AddrBook) {no b.map}

fact traces {
  all b: AddrBook - Ord[AddrBook].last | let b' = OrdNext(b) |
    some n: Name, a: Target | add (b, b', n, a) or del (b, b', n)
  init (Ord[AddrBook].first) }

assert lookupYields {all b: AddrBook, n: b.names | some lookup(b,n)}
```

counterexample



what you've seen

language

- first-order encoding
  - r: A -> B looks like r □ ℙ(A□B) but means r □ A□B
  - instead of AddrBook = ℙ(ℙ(Name□Addr))
    - define map: AddrBook -> Name -> Addr
- simple and uniform syntax
  - navigational dot, rich declarations
  - explicit parameterization

analysis

- executable and declarative
- no ad hoc constraint on language
- no test cases

not seen: modelling idioms

- schema extension

  ```
  sig AddrBook’ extends AddrBook {cache: Name -> Addr}
  ```

- object-oriented heap

  ```
  sig State {obj: Ref -> Obj}
  ```

- asynchronous processes

  ```
  sig Process {state: Time ->! State}
  ```

- explicit events

  ```
  sig Event {t: Time}
  sig AddEvent extends Event {n: Name, a: Addr}
  ```

not seen: analysis idioms

- refactoring

  ```
  fun lookup (b: AddrBook, n: Name): set Target {…}
  fun lookup′ (b: AddrBook, n: Name): set Target {…}
  assert same {all b: AddrBook, n: Name | lookup(b,n) = lookup(b′,n)
  ```

- abstraction

  ```
  fun abstract {c: ConcreteState, a: AbstractState) {…}
  fun opC (c, c′: ConcreteState) {…}
  fun opA (a, a′: AbstractState) {…}
  assert refines {all a, a′: AbstractState, c, c′: ConcreteState |
    opC(c,c′) and abstract(c,a) and abstract(c′,a′) => opA(a,a′) }
  ```

- machine diameter

  ```
  fun noRepeats {no disj b, b′: AddrBook | b.map = b′.map}
  -- when noRepeats is unsatisfiable, trace is long enough
  ```

how analyzer works

space is huge

- in scope of 5, each relation has 2^{25} possible values
- 10 relations gives 2^{250} possible assignments

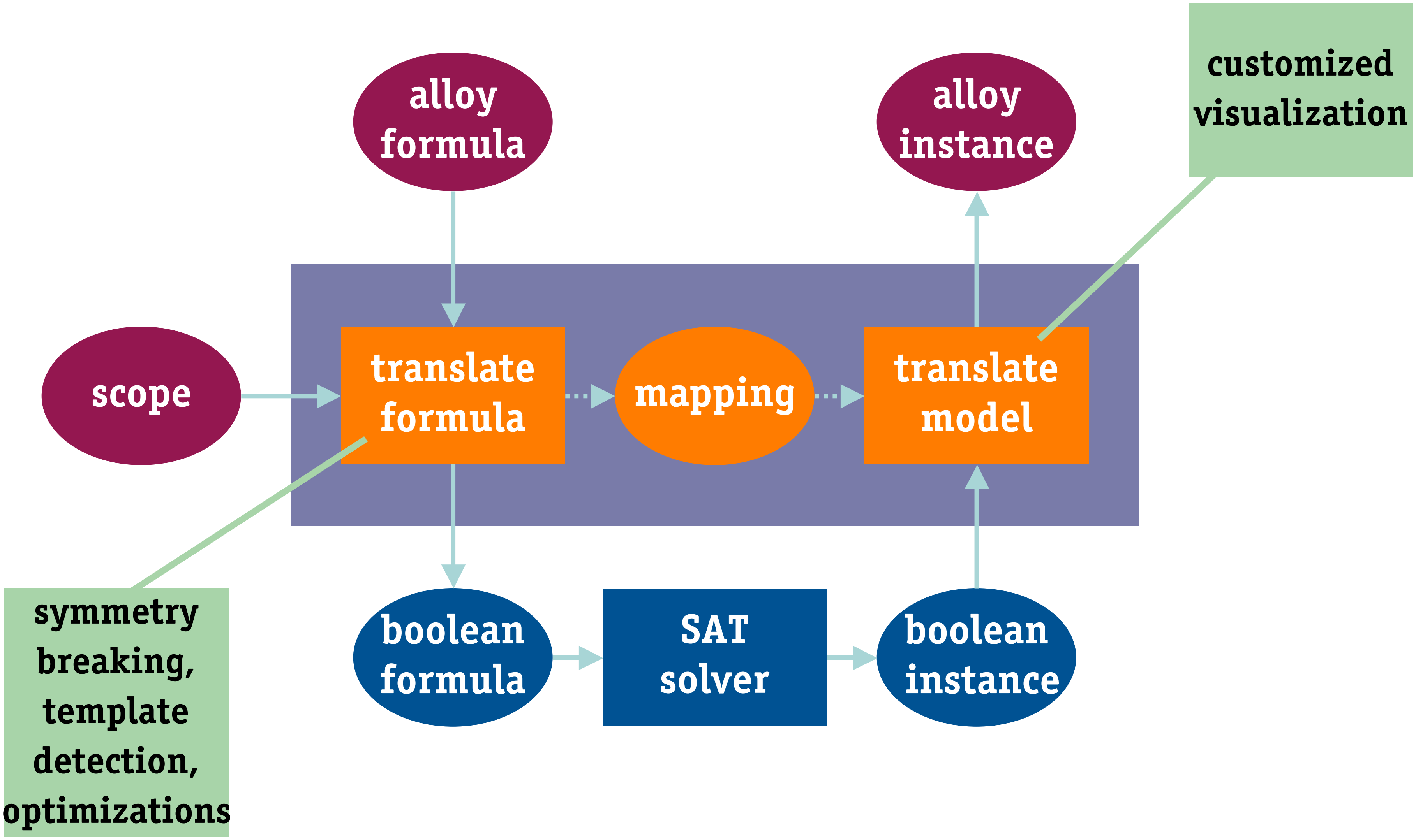
SAT to the rescue

- 1971: satisfiability problem to be shown NP-complete
- 1990's: shown to be easy in practice
- fastest solvers (Chaff, Berkmin) can handle thousands of boolean variables, millions of clauses

translating to SAT

- an instance is a graph
- for space of instances, label arcs with boolean variables

analyzer architecture



experience: general

amazing number of flaws

- blatant and subtle
- in every model

good things

- raises the bar
- sense of confidence
- compelling and fun

bad things

- encourages hacking
- over confidence

experience: design analyses

about 20 small case studies completed

- Key management (Taghdiri)
- Chord peer-to-peer storage (Wee)
- Firewire leader election (Jackson)
- Intentional Naming (Khurshid)
- Query Interface in COM (Sullivan)
- Unison file synchronizer (Nolte)
- Cellular automata (Sridharan)
- Role-based access control (Schaad et al)
- Ideal Address Translation (Seater & Dennis)

typically

- a few hundred lines of Alloy
- longest analysis time: 10 mins to 1 hour

experience: education

helps teach modelling

- abstract descriptions, concrete cases
- very close to standard first-order logic

major part of a course

- Imperial, U. Iowa, Kansas State

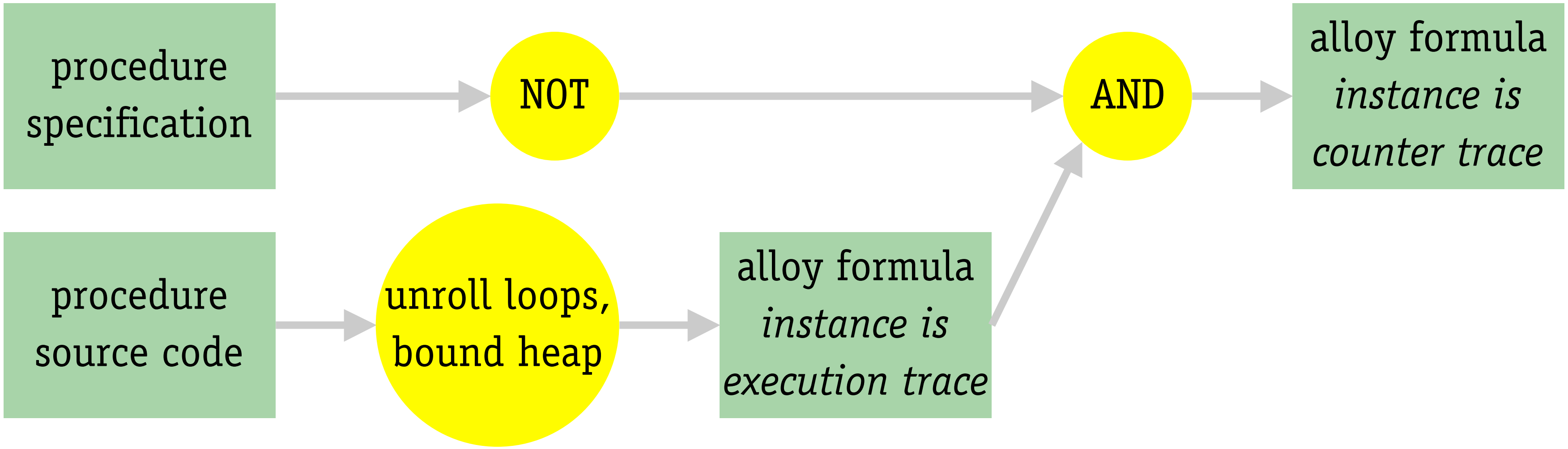
taught, usually with project

- CMU, Waterloo, Wisconsin, Rochester, Irvine, Georgia Tech, Queen's, Michigan State, Colorado State, Twente, WPI, USC, MIT

how long?

- undergraduate with no formal methods background
  can build small models in 2 weeks

applications: code analysis



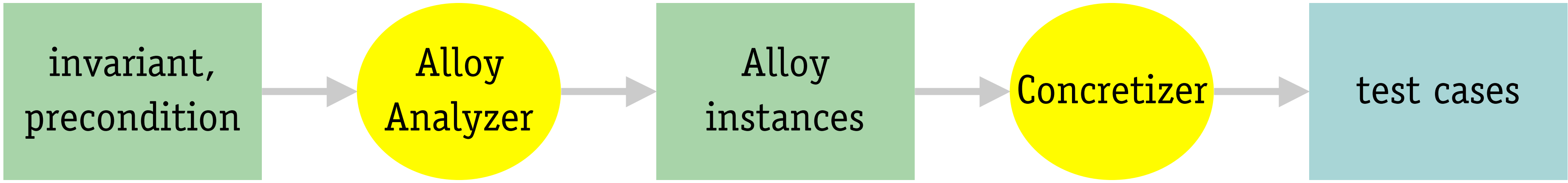
applied to small, complex algorithms

› Schorr-Waite garbage collection

› red-black trees

Mandana Vaziri's doctoral thesis

applications: test case generation



why?

- easier to write invariant than test cases
- all test cases within scope give better coverage
- symmetry breaking gives good quality quite

applied to Galileo, a NASA fault tree tool

- generated about 50,000 input trees, each less than 5 nodes
- found unknown subtle flaws

Sarfraz Khurshid's doctoral thesis

research challenges

scalability: dancing around the intractability tarpit

› circuit minimization

overconstraint: the dark side of declarative models

› unsat core prototype

› highlights contradicting formulas

new type system: real subtypes

› makes semantics fully untyped

› still no casts, down or up

› catches more errors, more flexible, better performance

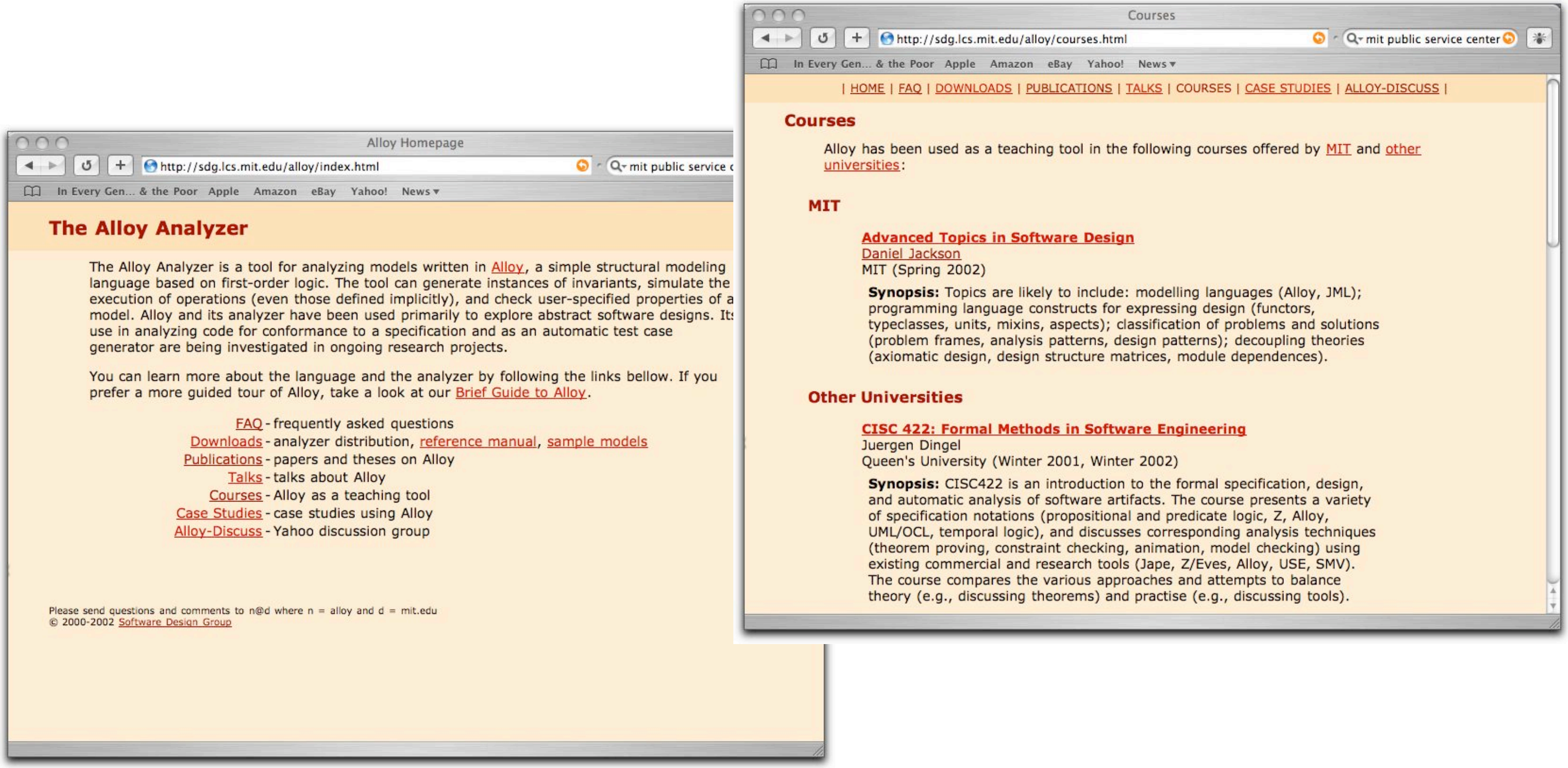
model extraction

› looking at how to extract models from code

for more information …

alloy.mit.edu

- downloads for windows, unix, macintosh
- courses, talks, case studies, papers


Alloy Homepage
http://sdg.lcs.mit.edu/alloy/index.html
mit public service c
In Every Gen... & the Poor Apple Amazon eBay Yahoo! News

The Alloy Analyzer

The Alloy Analyzer is a tool for analyzing models written in Alloy, a simple structural modeling language based on first-order logic. The tool can generate instances of invariants, simulate the execution of operations (even those defined implicitly), and check user-specified properties of a model. Alloy and its analyzer have been used primarily to explore abstract software designs. Its use in analyzing code for conformance to a specification and as an automatic test case generator are being investigated in ongoing research projects.

You can learn more about the language and the analyzer by following the links bellow. If you prefer a more guided tour of Alloy, take a look at our Brief Guide to Alloy.

FAQ	- frequently asked questions
Downloads	- analyzer distribution, reference manual, sample models
Publications	- papers and theses on Alloy
Talks	- talks about Alloy
Courses	- Alloy as a teaching tool
Case Studies	- case studies using Alloy
Alloy-Discuss	- Yahoo discussion group

Please send questions and comments to n@d where n = alloy and d = mit.edu
© 2000-2002 Software Design Group

Courses
http://sdg.lcs.mit.edu/alloy/courses.html
mit public service center
In Every Gen... & the Poor Apple Amazon eBay Yahoo! News
| HOME | FAQ | DOWNLOADS | PUBLICATIONS | TALKS | COURSES | CASE STUDIES | ALLOY-DISCUSS |

Courses

Alloy has been used as a teaching tool in the following courses offered by MIT and other universities:

MIT

Advanced Topics in Software Design
Daniel Jackson
MIT (Spring 2002)

Synopsis: Topics are likely to include: modelling languages (Alloy, JML); programming language constructs for expressing design (functors, typeclasses, units, mixins, aspects); classification of problems and solutions (problem frames, analysis patterns, design patterns); decoupling theories (axiomatic design, design structure matrices, module dependences).

Other Universities

CISC 422: Formal Methods in Software Engineering
Juergen Dingel
Queen's University (Winter 2001, Winter 2002)

Synopsis: CISC422 is an introduction to the formal specification, design, and automatic analysis of software artifacts. The course presents a variety of specification notations (propositional and predicate logic, Z, Alloy, UML/OCL, temporal logic), and discusses corresponding analysis techniques (theorem proving, constraint checking, animation, model checking) using existing commercial and research tools (Jape, Z/Eves, Alloy, USE, SMV). The course compares the various approaches and attempts to balance theory (e.g., discussing theorems) and practise (e.g., discussing tools).