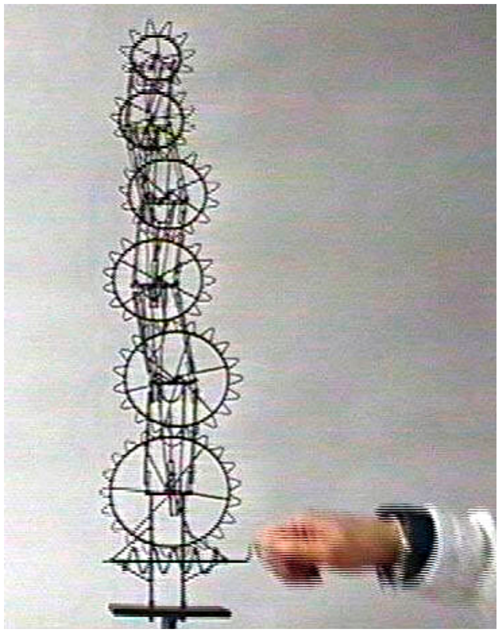
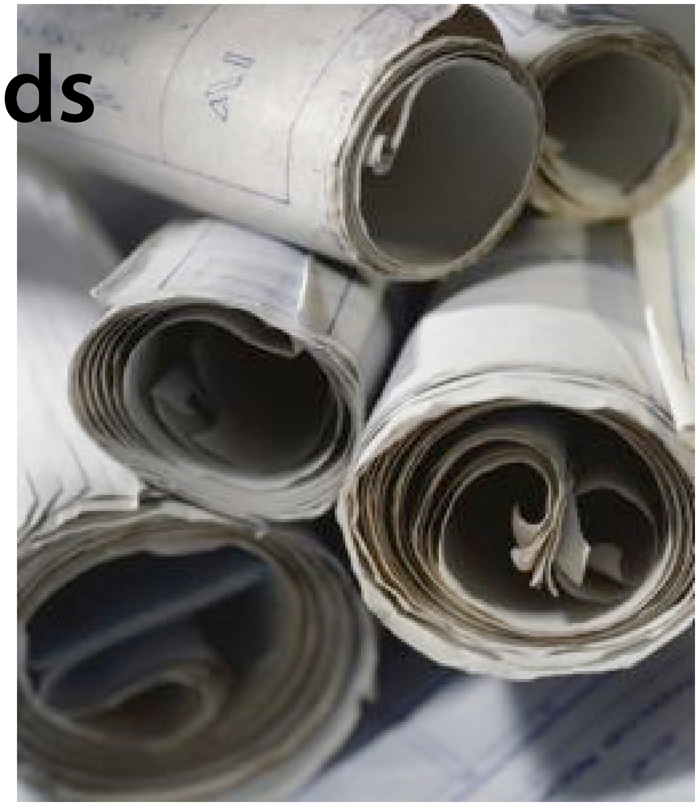
alloy: a logical modelling language

Daniel Jackson, MIT
Ilya Shlyakhter
Manu Sridharan

ZB 2003
Turku, June 4, 2003


Small Tower of 6 Gears, Arthur Ganson

lightweight formal methods



ingredients
- › incremental, risk-driven
- › small, focused models
- › rapid feedback from analysis

language must be
- › small and simple
- › expressive, esp. for structure
- › declarative (for partiality)

analysis must be
- › fully automatic
- › semantically deep

alloy: a structural, analyzable logic

a notation inspired by Z
- just (sets and) relations
- everything's a formula
- but not easily analyzed

an analysis inspired by SMV
- billions of cases in second
- counterexamples, not proof
- but not declarative

Oxford, home of Z

Pittsburgh, home of SMV

why not …?

animators
- › non-declarative sublanguage
- › limited coverage of space
- › manually driven (eg, by test cases)

theorem provers
- › still too hard for many users
- › failure hard to diagnose

model checkers
- › no support for data structures
- › language is often operational

demo

lengthy illustration of use of Alloy
to model and analyze an address book

a whirlwind tour

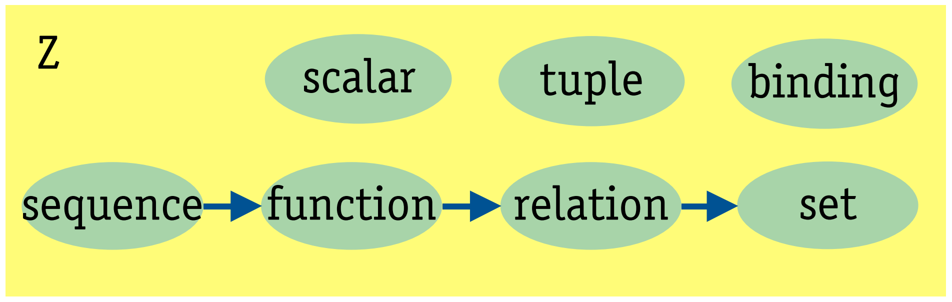
language ideas (versus Z)

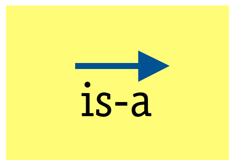
- kernel: everything's a relation -- not a set
- signatures: structure by atoms+projection -- not bindings
- functions: explicit parameterization -- not free variables

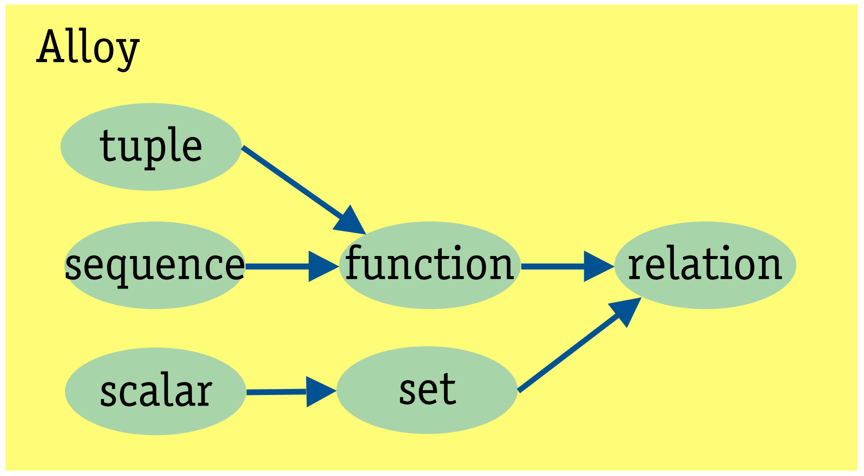
analysis ideas

- scope: exhaustive search in finite bounds
- engine: reduction to SAT

everything's a relation







relational kernel

all values are represented as relations

- {(a),(b)} for a set
- {(a)} for a scalar
- {(a,b)} for a tuple

operators

- p + q, p - q, p & q, ~p, *p, ^p
- p in q
- $p\ .\ q = \{(p_1, \dots p_{n-1}, q_2, \dots q_m) \mid (p_1, \dots p_n) \in p \wedge (p_n, q_2, \dots q_m) \in q\}$
- $p \text{ -> } q = \{(p_1, \dots p_n, q_1, \dots q_m) \mid (p_1, \dots p_n) \in p \wedge (q_1, \dots q_m) \in q\}$

example

- b′.addr = b.addr + n->a
- b = {(B0)}, b′ = {(B1)}, n = {(N0)}, a = {(A0)}, addr = {(B1,N0,A0)}

consequences

good

- › no function application: avoid partiality tarpit
- › uniform navigation expressions: no flattening, lifting, etc
- › simple semantics: easy to grasp, easy to implement

bad

- › partial function problem isn’t gone

```
no p: Person | p.wife in p.siblings
```

implies that everyone has a wife; instead say

```
no p: Person | some p.wife & p.siblings
```

first-order puns

`r: A -> B` means `r □ A□B` not `r □ A□ B`

signatures

key idea

› signatures denote sets of atoms
› fields denote global relations
› extension is subset

example

```
sig X, Y {}
sig A {f: X}
sig B extends A {g: X -> Y, h: Y}
```

X, Y, A, B are atom sets	f is a relation on A -> X
X, Y, A are disjoint	g is a relation on B -> X -> Y
B is a subset of A	h is a relation on B -> Y

a.h is empty if **a** not in **B**

consequences

good

› quantification over signature sets is first-order
› simpler semantics than Z's schema bindings
› no casts needed

bad

› existentials don't always mean what you think

```
    all b: Book | some b′: Book | b′.addr = b.addr + n->a
```

no classification by schemas in Z

```
sig Target {}
sig Addr extends Target {u: User}
sig Book {addr: Name -> Target}
fun SimpleAdd (b, b′: Book, n: Name, a: Addr) {b′.addr = b.addr + n->a}
```

```
Target ⌜ [] BOGUS!
Addr ⌜ [Target; u: User]
Book ⌜ [addr: Name ↦ Target]
Add ⌜ [□Book, n: Name, a: Addr | addr′ = addr □ {(n,a)}] ERROR!
```

idioms for change of state

- ‘established strategy’

```
sig Book {addr: Name -> Addr}
fun Clear (b, b′: Book) {no b′.addr}
```

- object-oriented heap

```
sig State {deref: Ref -> Book}
fun Clear (s, s′: State, br: Ref) {no s′.deref[br]}
```

- asynchronous processes

```
sig BookProcess {addr: Name -> Addr -> Time}
fun Clear (t, t′: Time, bp: BookProcess) {no bp.addr.t′}
```

- explicit events

```
sig Event {t: Time}
sig ClearEvent extends Event {bp: BookProcess}
fun trans (e: Event) {e in ClearEvent => no e.bp.addr.t ,…}
```

parameterization

functions are parameterized formulas
- › semantics is just renaming/inlining
- › can handle recursion if args are scalar

good
- › simple, clear semantics
- › no tricky variable capture
- › type checking catches errors
- › modular implementation

bad
- › can be more verbose than Z
- › can't factor out argument sublist

promotion in Alloy

```
sig Name, Addr {}
sig Book {addr: Name -> Addr}
fun AddLocal (b, b′: Book, n: Name, a: Addr) {
  b′.addr = b.addr + n->a
  }

sig BookID {}
sig Email {book: BookID ->! Book}
fun Add (e, e′: Email, b: BookID, n: Name, a: Addr) {
  AddLocal (e.book[b], e′.book[b], n, a)
  all bx: BookID - b | e′.book[bx] = e.book[bx]
  }
```

promotion in Z

[Name, Addr]
Book ≙ [addr: Name <-> Addr]
AddLocal ≙ [□Book; n: Name; a: Addr | addr′ = addr □ {(n,a)}]

[BookID]
Email ≙ [book: BookID ↦ Book]
Add ≙ □ □Book | AddLocal □
[
□Email; □Book; bid: BookID |
book bid = □Book
book′ bid = □Book′
□bid′: BID • bid′ != bid | book′ bid′ = book bid′
]

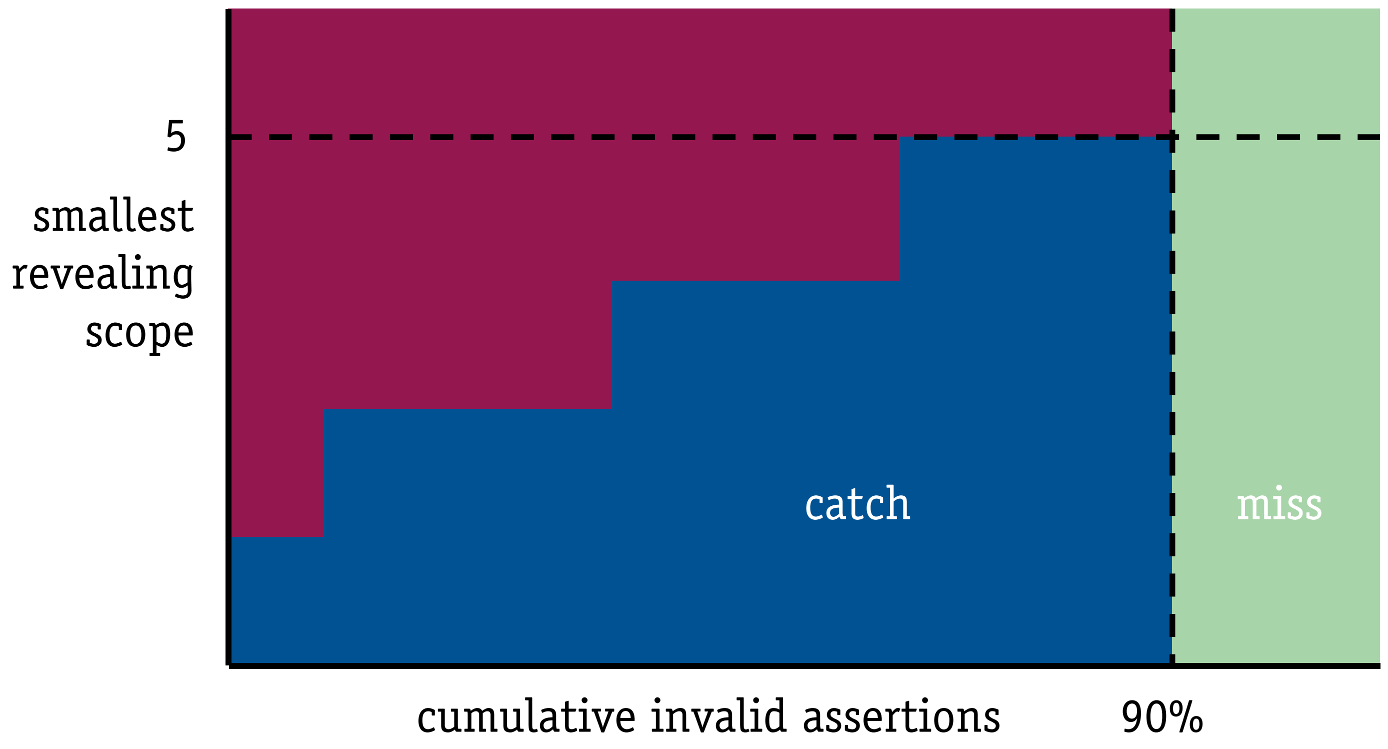
scope

language is undecidable
- › so no sound & complete algorithm

“try all small tests”
- › model proper is unbounded
- › user defines scope in command
- › scope bounds each basic type

small scope hypothesis
- › many bugs have small counterexamples
- › … and models often have many bugs

small scope hypothesis



consequences
› sound: no false alarms
› incomplete: can’t prove anything

engine: reduction to SAT

space is huge

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

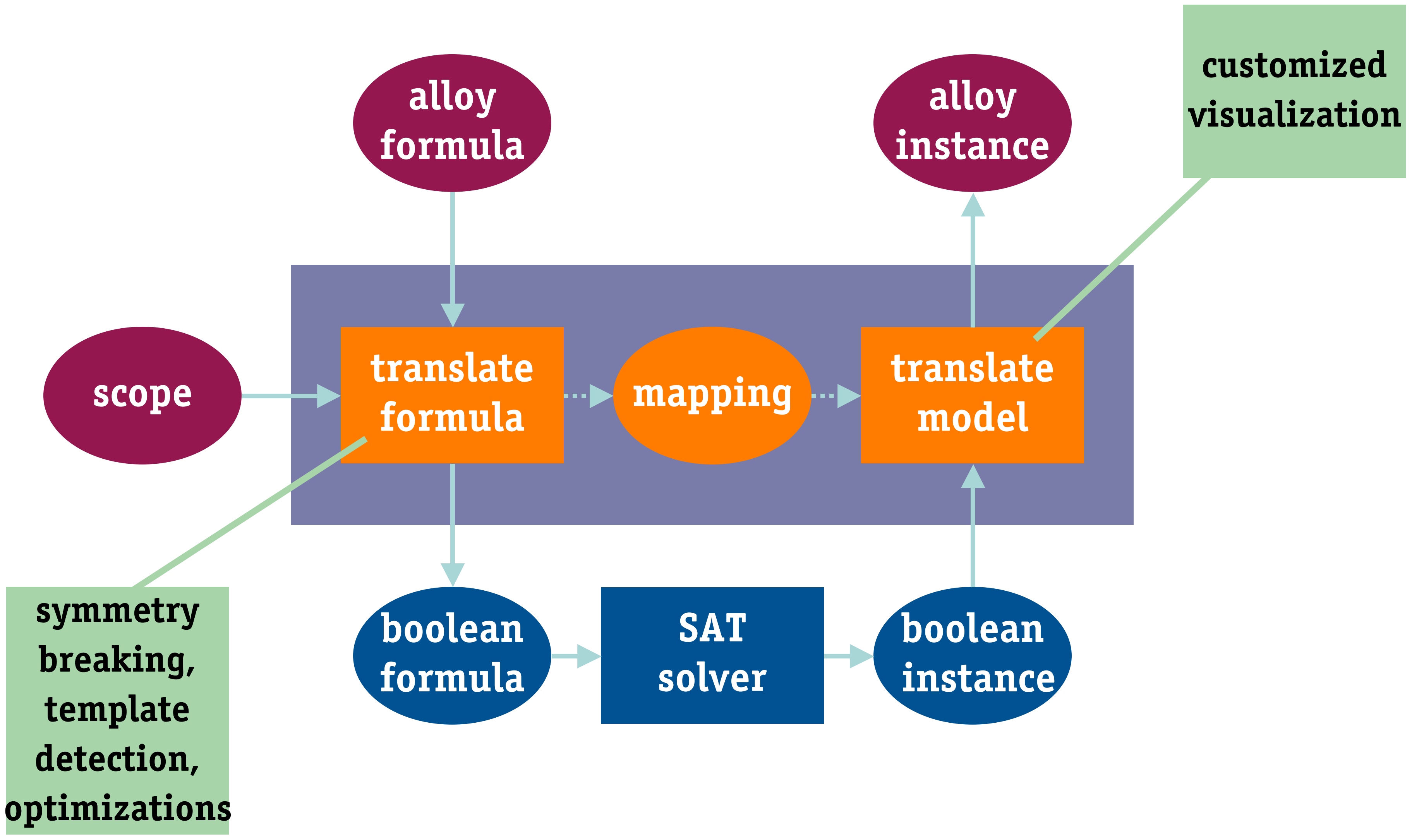
SAT to the rescue

- › SAT is hard (Cook, 1971)
- › SAT is easy (Kautz, Selman et al, 1990's)
- › Chaff, Berkmin: thousands vars, millions clauses

translating to SAT

- › view relation as a graph
- › space of possible values: each edge is present or not
- › label edge with boolean variable
- › compositional mapping from relational to boolean formula

analyzer architecture



analysis idioms

- refactoring

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

- abstraction

```
fun abs {c: Concrete, a: Abstract) {…}
fun opC (c, c′: Concrete) {…}
fun opA (a, a′: Abstract) {…}
assert refines {all a, a′: Abstract, c, c′: Concrete |
  opC(c,c′) and abs(c,a) and abs(c′,a′) => opA(a,a′) }
```

- machine diameter

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

reflections

executable and abstract specifications?

› can have your cake and eat it
› … if you eat slowly

is first-order enough?

› most uses of higher-order features are gratuitous
› but minimization is a problem

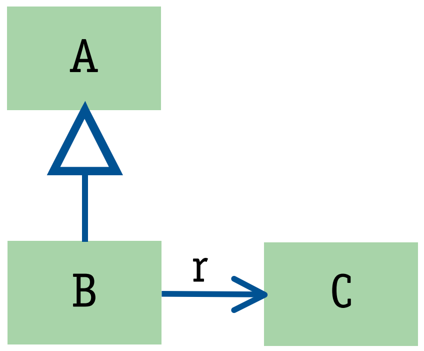
tool implementation

› strong sanity check on language design

Alloy and UML

because of these Alloy features
- signature extension
- implicit typing
- flexible declaration syntax

you can transcribe an object model or ER diagram



```
sig A {}
sig B extends A {r: set C}
sig C {}

sig Atyp, Ctyp {}
sig State {
  A: set Atyp, C : set Ctyp,
  B: set A, r: B -> C}
```

the UML dilemma

UML’s constraint language, OCL
- › complicated, inexpressive, not modular, not well defined

what to do?

path A
- › develop formal semantics, and sanction its complexity
- › call this an industrial application of formal methods
- › embrace UML in teaching

path B
- › explain why it’s broken, and suggest how it might be fixed
- › get on with applying better approaches to real problems
- › snub UML in teaching and teach stronger, simpler notations

experience: general

amazing number of flaws
- › blatant and subtle
- › in every model

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

experience: design analyses

case studies

› about 30 completed
› serious flaws in published designs found

distinguishing features

› complex data structures (eg, file synchronization)
› network protocol over all topologies (eg, firewire, chord)
› partial model; only some operations (eg, intentional naming)
› not state machine (eg, ideal address translation)

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

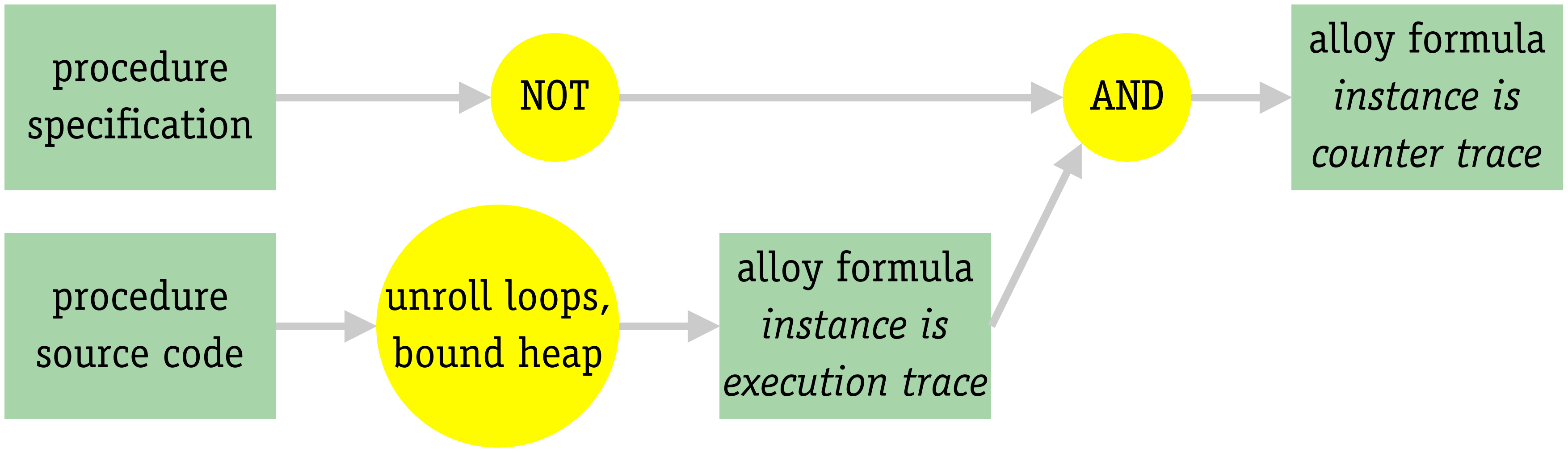
used in courses at

- Imperial, U. Iowa, Kansas State, CMU, Waterloo, Wisconsin, Rochester, Irvine, Georgia Tech, Queen’s, Michigan State, Colorado State, Twente, WPI, USC, MIT, …

how long to learn?

- undergraduate, no formal methods background
- can build and analyze small models in 2 weeks

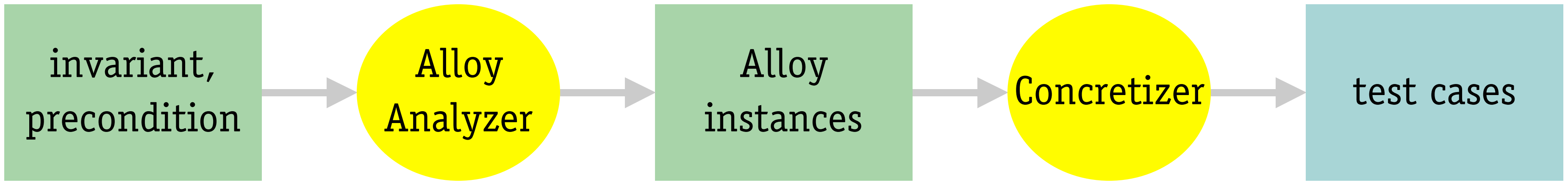
applications: code analysis


procedure specification
NOT
AND
alloy formula *instance is counter trace*
procedure source code
unroll loops, bound heap
alloy formula *instance is execution trace*

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

ongoing research projects

scalability: dancing around the intractability tarpit
› circuit minimization

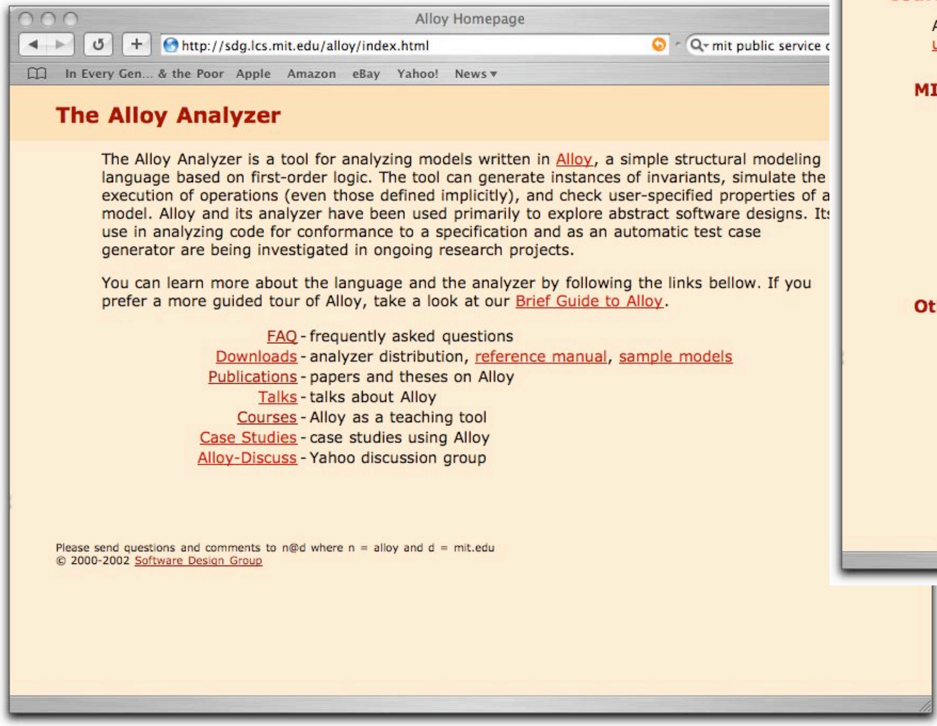
overconstraint: the dark side of declarative models
› unsat core prototype
› highlights contradicting formulas

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

alloy.mit.edu

› downloads for OS X, windows, linux
› courses, talks, case studies, papers
› coming: tutorial, book


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

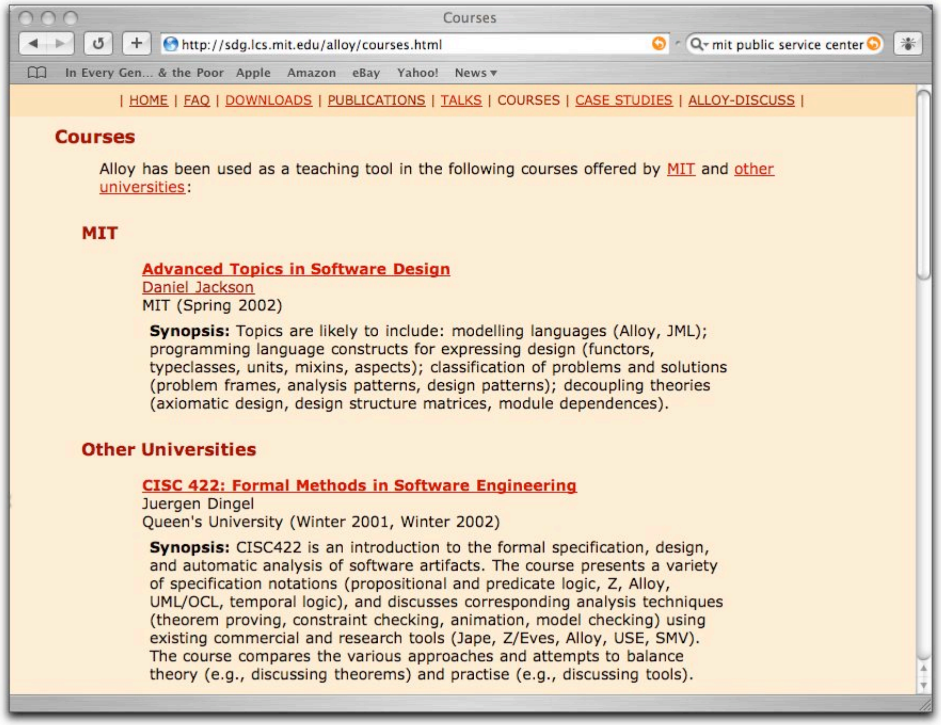
The Alloy Analyzer

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 practice (e.g., discussing tools).