analyzable models for software design

Daniel Jackson, MIT
University of York · February 4, 2004

CSAIL
MIT COMPUTER SCIENCE AND ARTIFICIAL INTELLIGENCE LABORATORY

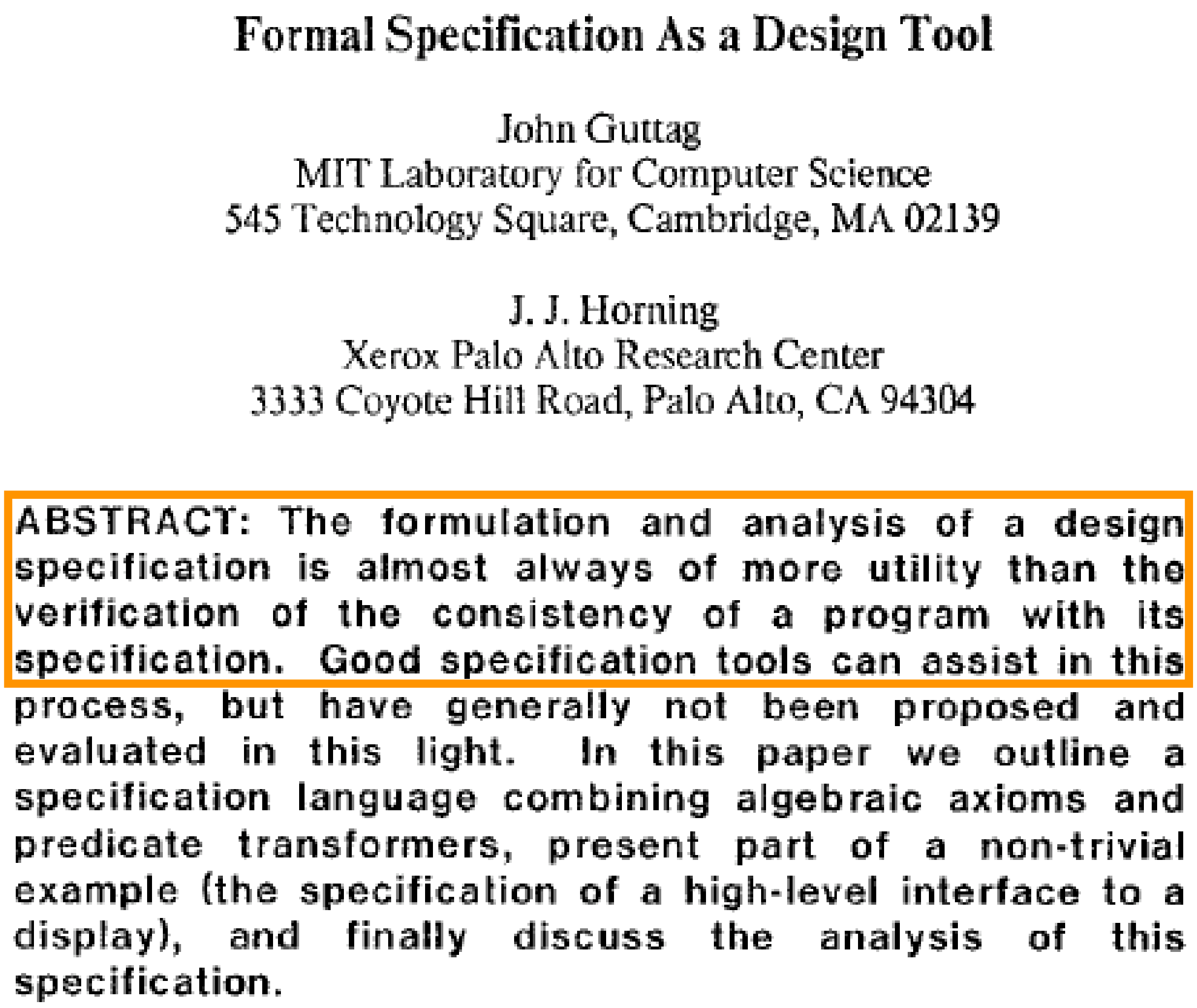
why analyzable models?

why models?
- figure out what problem you’re solving
- explore invented concepts
- communicate with collaborators

why analyzable?
- not just finding errors early
- analysis breathes life into models!

software based on simple, strong models tends to have cleaner interfaces, fewer bugs, and is easier to use and to maintain.

an inspiration (POPL, 1980)



Formal Specification As a Design Tool

John Guttag
MIT Laboratory for Computer Science
545 Technology Square, Cambridge, MA 02139

J. J. Horning
Xerox Palo Alto Research Center
3333 Coyote Hill Road, Palo Alto, CA 94304

ABSTRACT: The formulation and analysis of a design specification is almost always of more utility than the verification of the consistency of a program with its specification. Good specification tools can assist in this process, but have generally not been proposed and evaluated in this light. In this paper we outline a specification language combining algebraic axioms and predicate transformers, present part of a non-trivial example (the specification of a high-level interface to a display), and finally discuss the analysis of this specification.

desiderata

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

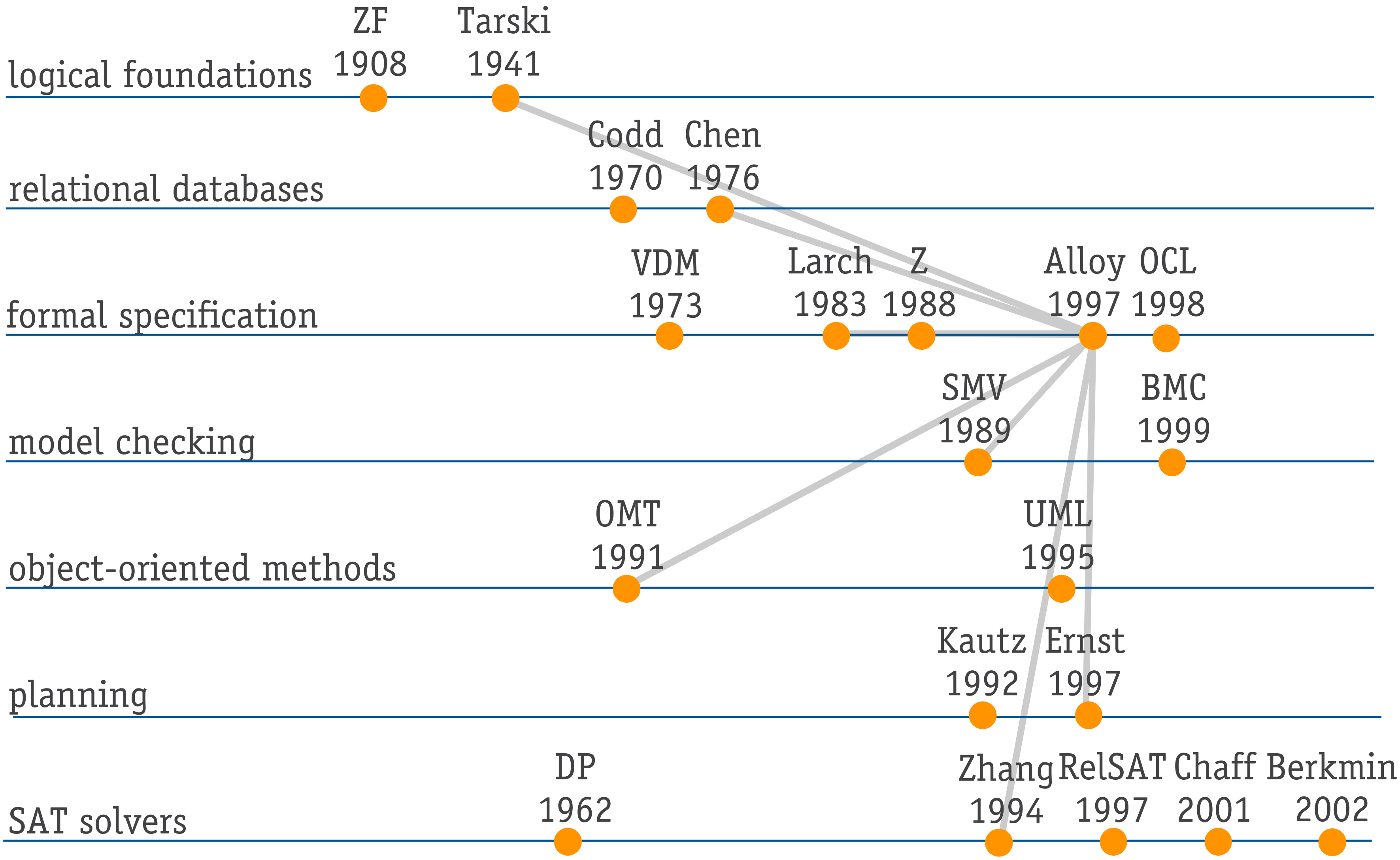
a notation inspired by Z
- just (sets and) relations
- everything's a formula
- but not easily analyzed

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

Oxford, home of Z

Pittsburgh, home of SMV

alloy's origins



logical foundations: ZF 1908, Tarski 1941

relational databases: Codd 1970, Chen 1976

formal specification: VDM 1973, Larch 1983, Z 1988, Alloy 1997, OCL 1998

model checking: SMV 1989, BMC 1999

object-oriented methods: OMT 1991, UML 1995

planning: Kautz 1992, Ernst 1997

SAT solvers: DP 1962, Zhang 1994, RelSAT 1997, Chaff 2001, Berkmin 2002

demo

ideas behind alloy

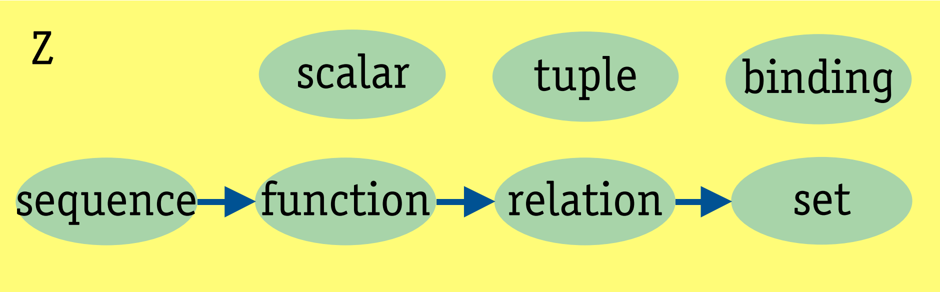
language

- every value's a relation
- everything else is a constraint
- no hard-wired idioms

analysis

- it's all constraint solving
- bounding the scope
- exploiting SAT

every value’s a relation

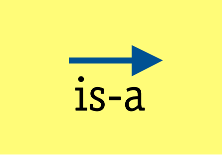

Z

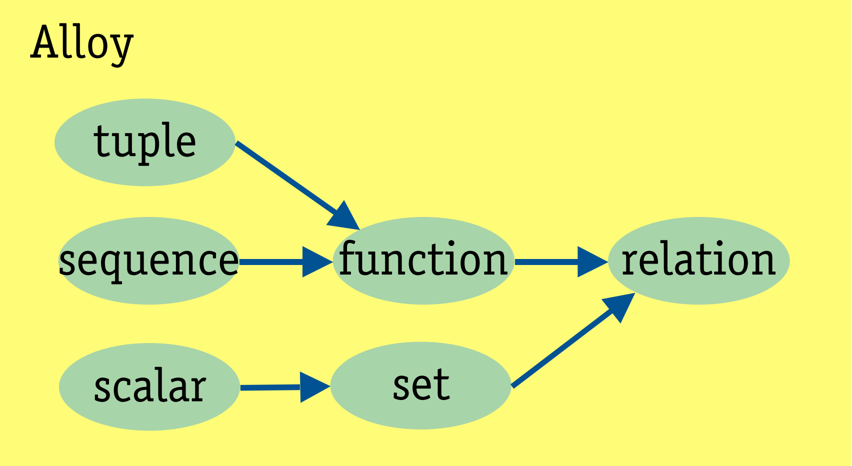
scalar

tuple

binding

sequence → function → relation → set


is-a


Alloy

tuple → function

sequence → function → relation

scalar → set → relation

signatures: making structure first order

problem: how to get composite structures, but stay first order

traditional viewpoint
- member of set Book is a record
- addr component is a (binary) relation

alloy's viewpoint
- member of set Book is an atom
- addr component is a ternary relation

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

relational operators

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 = $\{(p_1, \ldots p_{n-1}, q_2, \ldots q_m) \mid (p_1, \ldots p_n) \in p \wedge (p_n, q_2, \ldots q_m) \in q\}$
p -> q = $\{(p_1, \ldots p_n, q_1, \ldots q_m) \mid (p_1, \ldots p_n) \in p \wedge (q_1, \ldots q_m) \in q\}$

example

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

why relations are nice

easy to understand
- binary relation is a graph
- ternary relation is a graph/atom

easy to implement
- first order, so tractable
- relational kernel like compiler's IL

uniformity

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()
```

everything else is a constraint

predicates

- invariants
  ```
  pred Init (s: State) {…}
  ```
- operations
  ```
  pred Op (s, s’: State) {…}
  ```
- traces
  ```
  pred Traces () {
    Init (first ()) and all s: State - last () | Op (s, next(s)) }
  ```

assertions

- invariants are preserved
  ```
  assert Safe {all s,s’: State | Safe(s) and Op(s,s’) => Safe(s’)}
  ```
- undo works
  ```
  assert UndoOK {all s,s’,s”: State | Op(s,s’) and Undo(s’,s”) => s”= s}
  ```

no hard-wired idioms

what’s hard-wired?

- relational structure
- facts/predicates/functions/assertions
- subtypes and parametric polymorphism
- … but not: state machines, traces, attributes/associations, etc

idioms of Alloy usage

- refinement of Z-style operations (security, *Bolton*)
- asynchronous processes (key management, *Taghdiri*)
- transitions based on history (Rendezvous, *Jazayeri*)
- global synchronized events (Firewire, *Jackson*)
- recursive lookup function (Intentional Naming, *Khurshid*)
- object-oriented heap (Java views, *Waingold*)
- flat data model (access control, *Zao*)
- …

sample idioms: change of state

- ‘established strategy’

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

- object-oriented heap

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

- asynchronous processes

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

- explicit events

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

sample idioms: analysis

- refactoring

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

- abstraction

```
pred abs {c: Concrete, a: Abstract) {…}
pred opC (c, c′: Concrete) {…}
 pred 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

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

all constraint solving

‘show me some relations satisfying these constraints’

simulation

```
sig Book { addr: Name -> Addr}
pred add (b, b’: Book, n: Name, a: Addr) {…}
run add
```

relations: b, b’, n, a, Book, Name, Addr, addr

constraint: decl constraints, facts, add

checking

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

relations: b, n, Book, Name, Addr, addr, ord/next

constraint: decl constraints, facts, axioms of next, not lookupYields

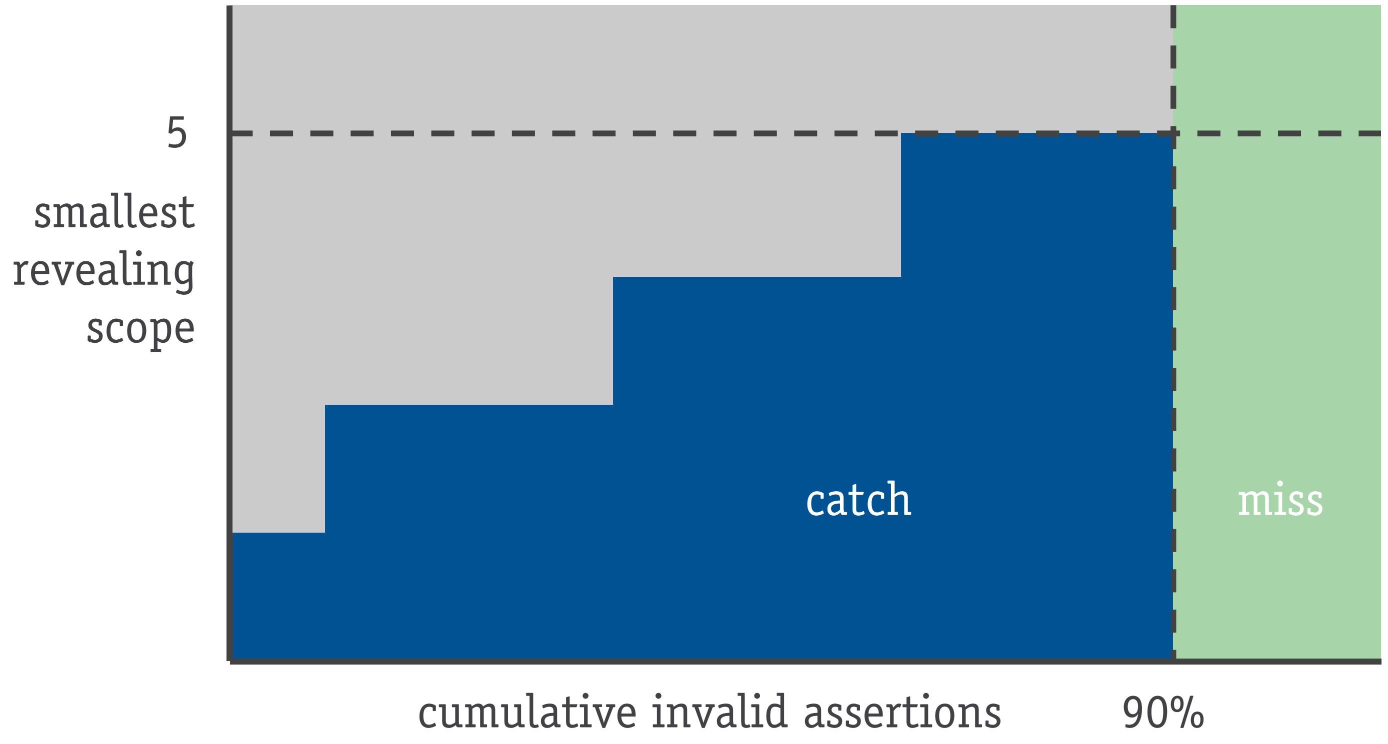
scope

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

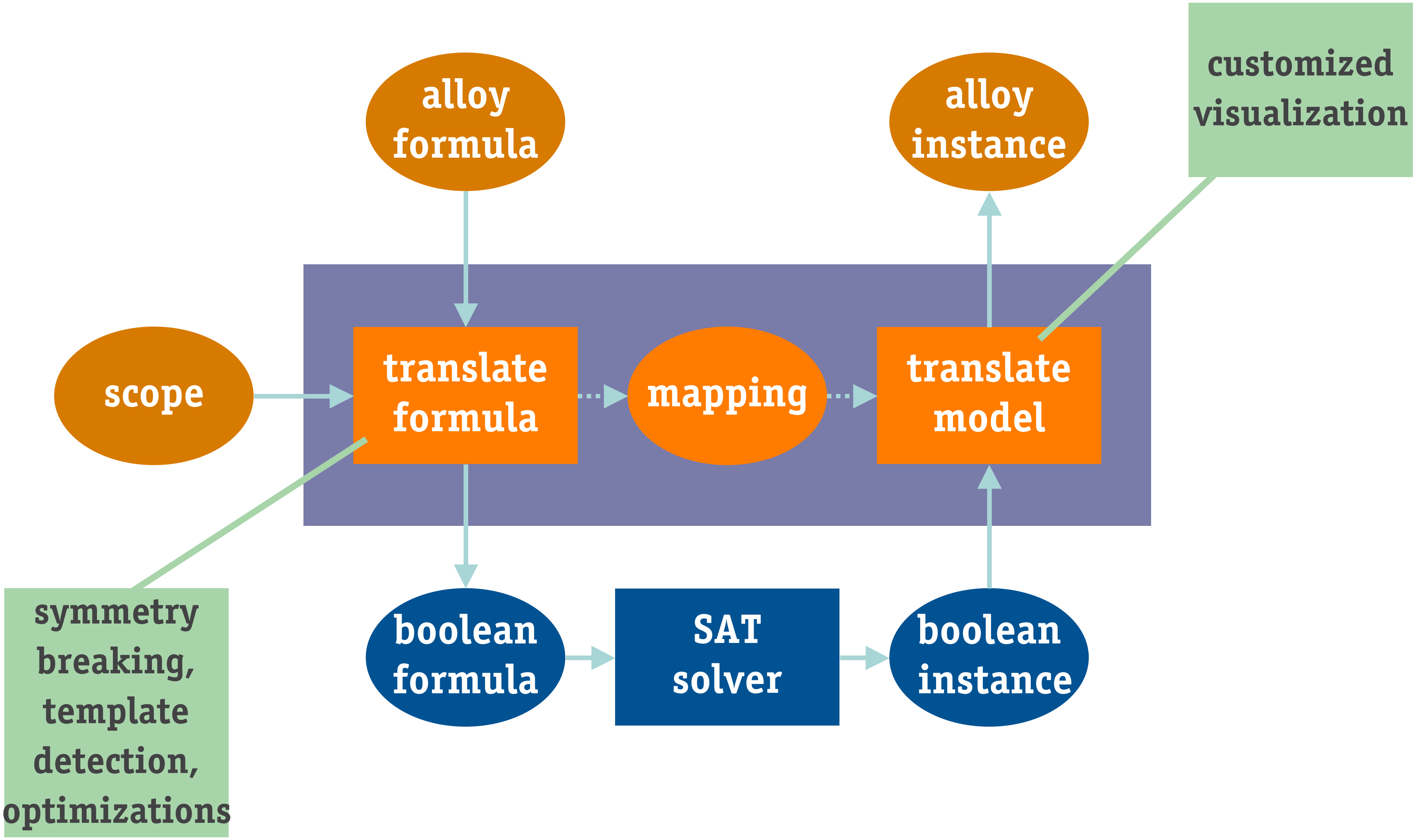
will SAT help?

- 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 translation

analyzer architecture



alloy formula

alloy instance

customized visualization

scope

translate formula

mapping

translate model

symmetry breaking, template detection, optimizations

boolean formula

SAT solver

boolean instance

what I haven't told you about…

scalability: dancing around the intractability tarpit
- implemented: symmetry, sharing, atomization
- prototyped: 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

experience: design analyses

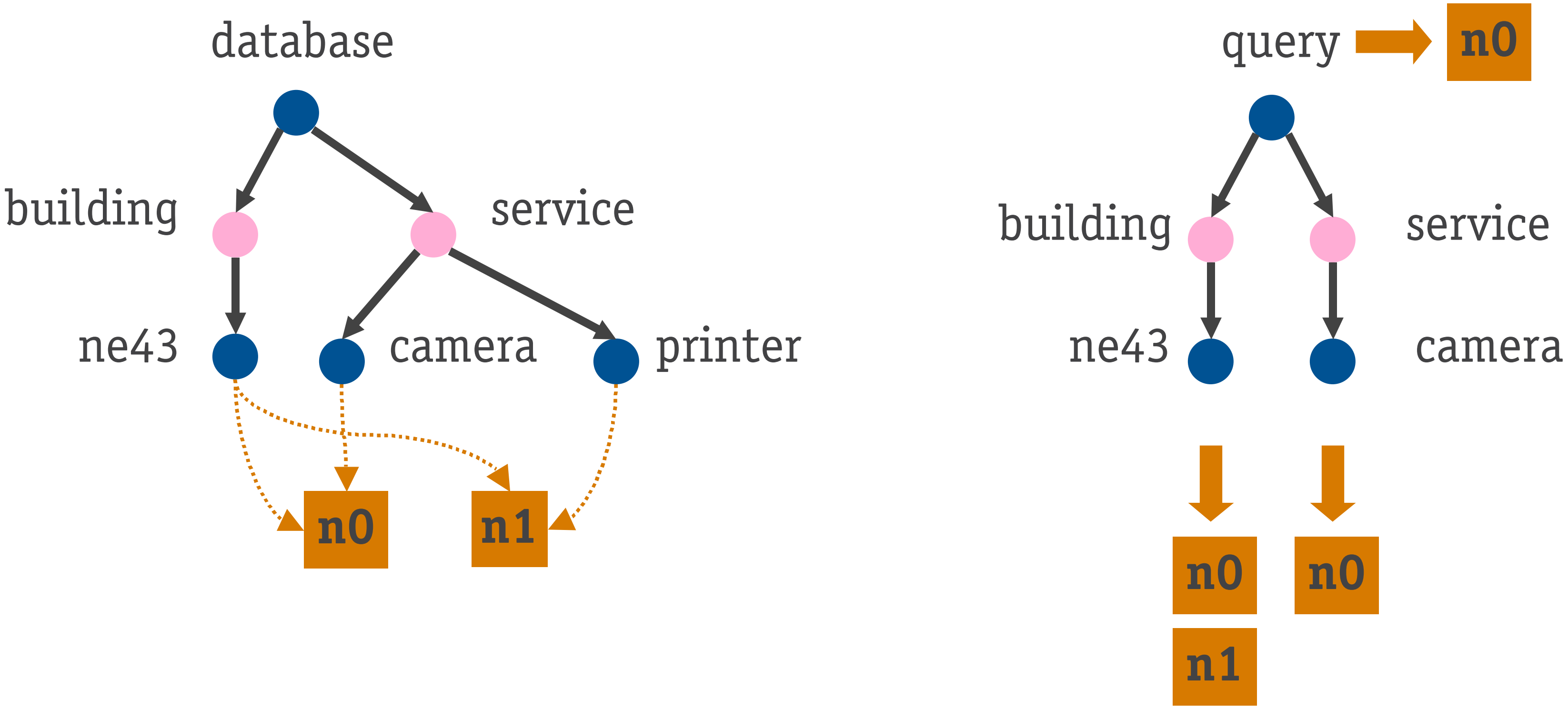
case studies
- about 30 completed
- serious flaws in published designs found

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

sample application: intentional naming

- a resource discovery scheme
- database and queries are attribute/value trees
  Balakrishnan et al, SOSP99



sample application: intentional naming

what we did

- built Alloy model from SOSP description
- checked paper's claims: none held
- checked code fixes: they didn't work either
- formulated and checked more basic claims

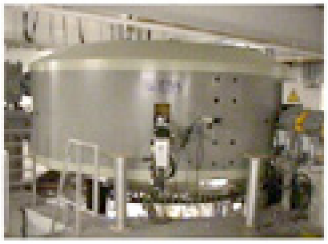
```
assert Monotone {
  all db: DB, q: Query, r: Rec | lookup(db,q) in lookup(add(db,r),q)
```

- developed notion of conformance
- fixed algorithm & code

900 lines of testing code vs. 100 lines of Alloy

Khurshid & Jackson, ASE 2000

sample application: beam scheduler



Northeast Proton Therapy Center

- 4 treatment rooms, multiplexed beam
- beam requests from treatment control rooms
- allocated by master control room
- beam scheduler automates de/allocation

what we did

- translated developer's OCL model into Alloy
- analyzed for small flaws (simulation, invariants, etc)
- checked commutativity for all operation pairs

  `Request ; Alloc = Alloc ; Request`
- found many non-commuting pairs, strange behaviours

Dennis, Jackson, Rayside, Seater

experience: education

helps teach modelling

- abstract descriptions, concrete cases
- closest useable modelling language to logic?

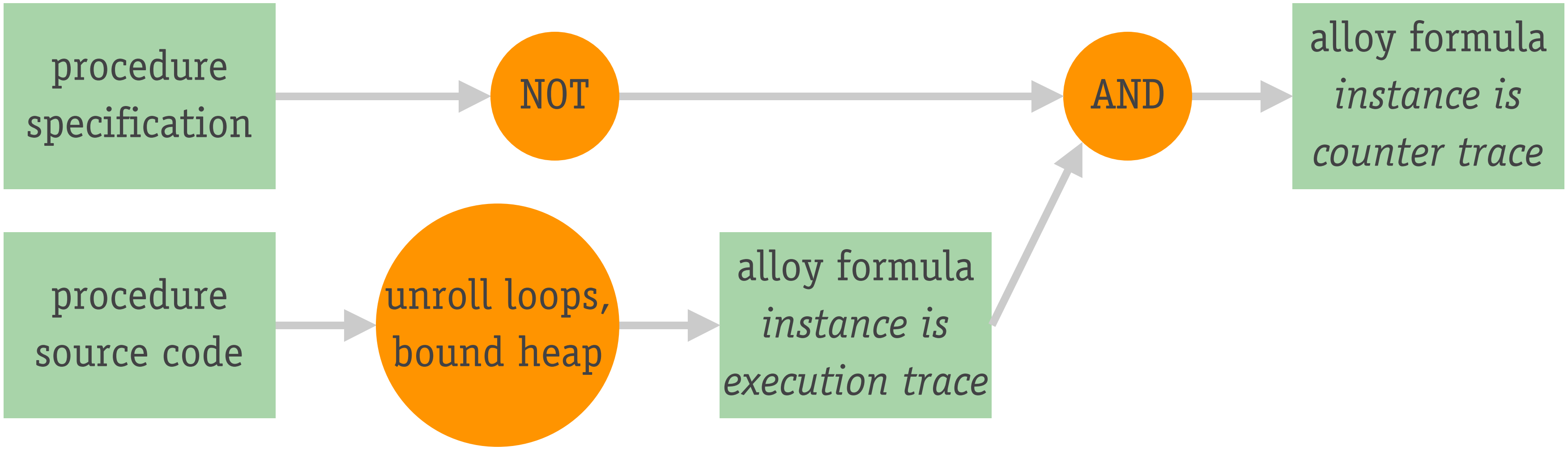
where's it's been used

- taught in about 20 courses worldwide
- mostly masters courses on modelling

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

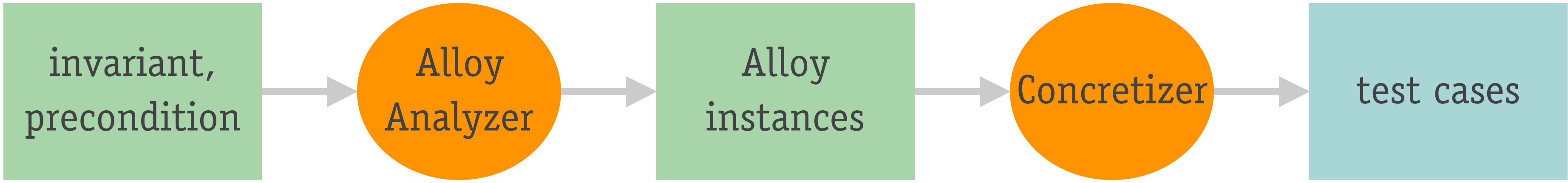
unroll loops, bound heap

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



invariant, precondition

Alloy Analyzer

Alloy instances

Concretizer

test cases

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

new views on old questions

mathematical or informal models?
- not about Greek symbols (but removing them helps)
- mathematical means simple & analyzable
- real challenge for novices is abstraction

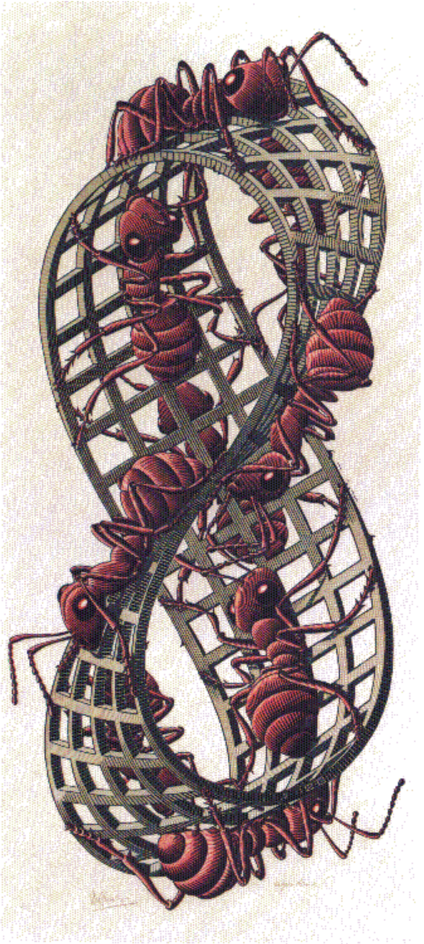
executable or abstract?
- alloy: you can have your cake and eat it (slowly)
- compromise higher order, not declarative features

simulation or verification?
- really the same: show me a good (bad) state
- it's not about subtle bugs

tool impact

developing a tool
- sanity check on language design
- complexity is intolerable
- good for implementation = good for users?
- visualization is crucial

using a tool
- amazing how many errors are exposed
- raises the bar, gives sense of confidence
- simulation is under-rated: it works!



some research based on alloy

- automatic analysis of action diagrams
  -- R. Venkatesh, TCS India
- discovery of refinements
  -- Christie Bolton, Oxford
- Ag: Alloy with dynamic logic
  -- Marcelo Frias (U. Buenes Aires)
- justifying object model transforms
  -- Paulo Borba (Pernambuco, Brazil)
- web ontology analysis
  -- Jin Sing Dong (Singapore)

acknowledgments

current developers

Ilya Shlyakhter
Emina Torlak
Sam Daitch
Jonathan Edwards
Vincent Yeung
Edmond Lau
Greg Dennis
Robert Seater
Julie Pelaez

former developers

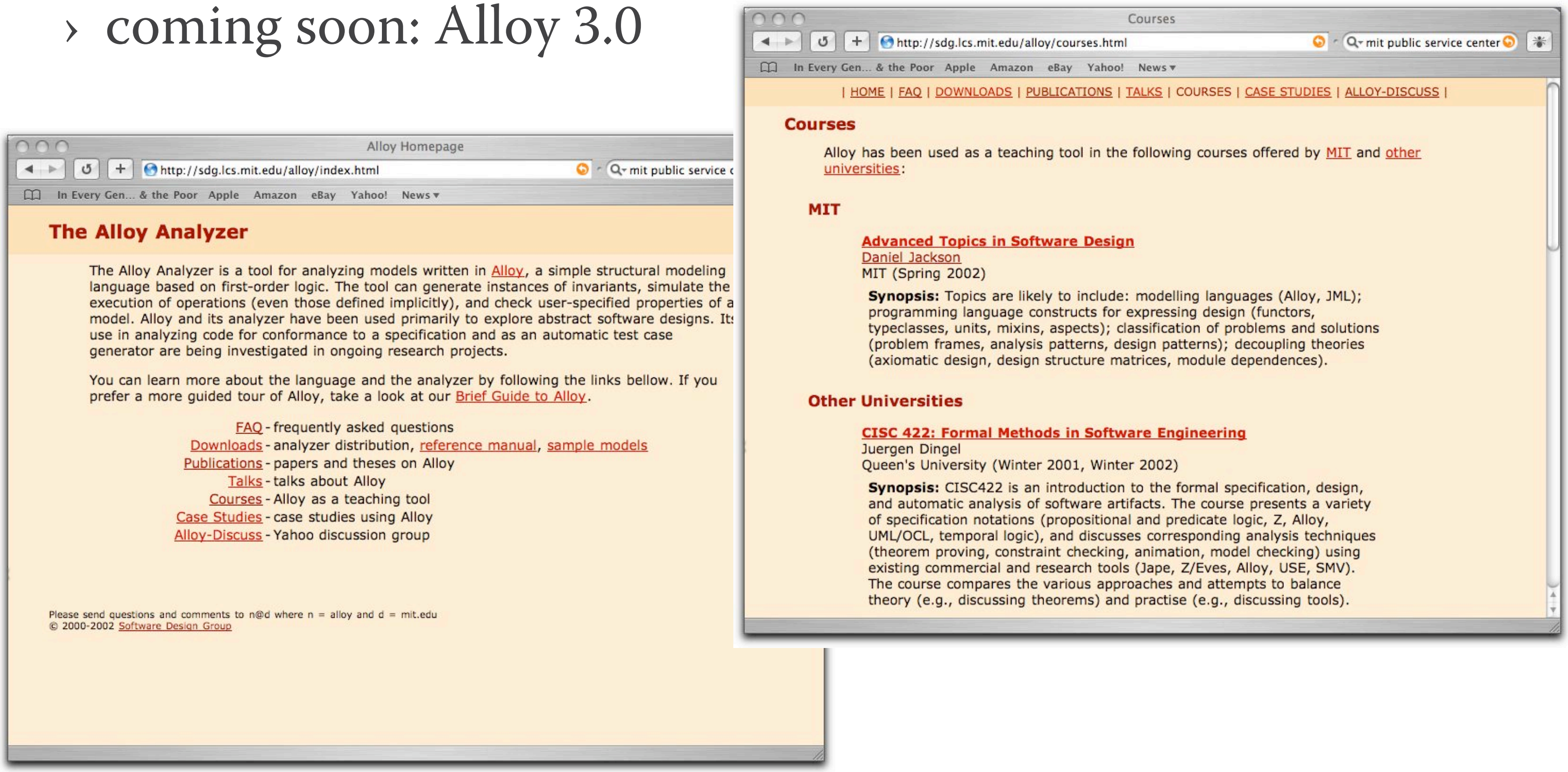
Manu Sridharan
Andrew Yip
Ning Song
Sarfraz Khurshid
Mandana Vaziri
Jesse Pavel
Ian Schechter
Li-kuo Lin
Joseph Cohen
Uriel Schafer
Arturo Arizpe

early adopters

Michael Huth
John Hatcliff
Matt Dwyer
Christie Bolton
Juergen Dingel
Cesare Tinelli
Jin Song Dong
Laura Dillon
Alain Wegmann
Marcelo Frias
Chris Wallace
Andreas Schaad
Maria Nelson
Torsten Nelson

alloy.mit.edu

- downloads for OS X, windows, linux
- courses, talks, case studies, papers, tutorial
- book in preparation: *Analyzable Models of Software*
- coming soon: Alloy 3.0



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

| 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).