modelling & analyzing software abstractions

Daniel Jackson · Computer Science & Artificial Intelligence Lab · MIT

Microsoft Research · March 21, 2005

premise #1: abstractions

the design of software abstractions is

› the essence of software design

› the key determinant of its quality

› and can be conscious, not just emergent

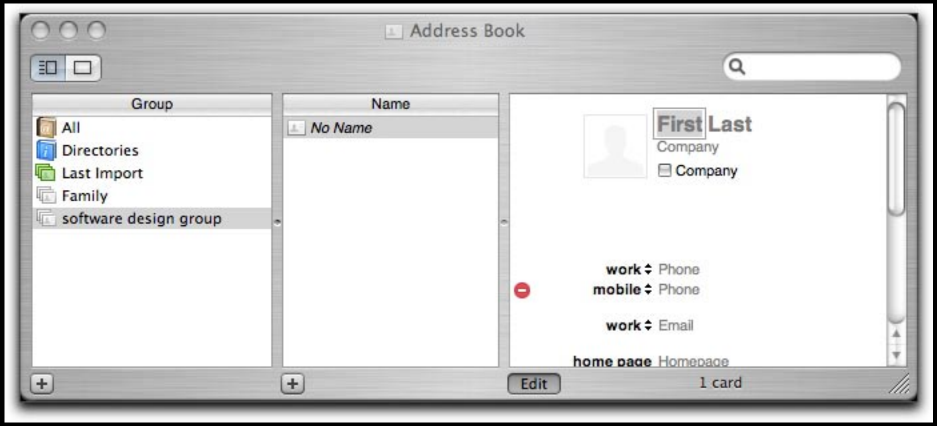
fred brooks on conceptual integrity

I will contend that conceptual integrity is the most important consideration in system design. It is better to have a system omit certain anomalous features and improvements, but to reflect one set of design ideas, than to have one that contains many good but independent and uncoordinated ideas. *1975*

I am more convinced than ever. Conceptual integrity is central to product quality. *1995*

example

- what is an address? nickname? group?
- can nicknames have nicknames?
- are empty groups ok?
- can an address appear twice in a group?



premise #2: details matter

not all bugs in code are created equal

- transcription errors ⇢ easily fixed
- bugs in abstractions ⇢ barnacles of complexity

example: fonts

- no standard font styles (bold, semibold, black, ...)
- different glyphs with same unicode id!

elements of my approach

› expressive/precise/simple language
› automatic tool, for deep analysis
› catalog of patterns, to capture state of art
› case studies for demo, evualation, education

progress to date: alloy

Alloy 3, a logic-based modeling language

Alloy Analyzer, SAT-based analyzer for simulation & checking

about 40 patterns developed, basis for current UG course

case studies

› serious flaws found in several published designs

› recent studies: beam scheduling, cryptography, interoperation

Alloy has been taught in about 20 courses worldwide

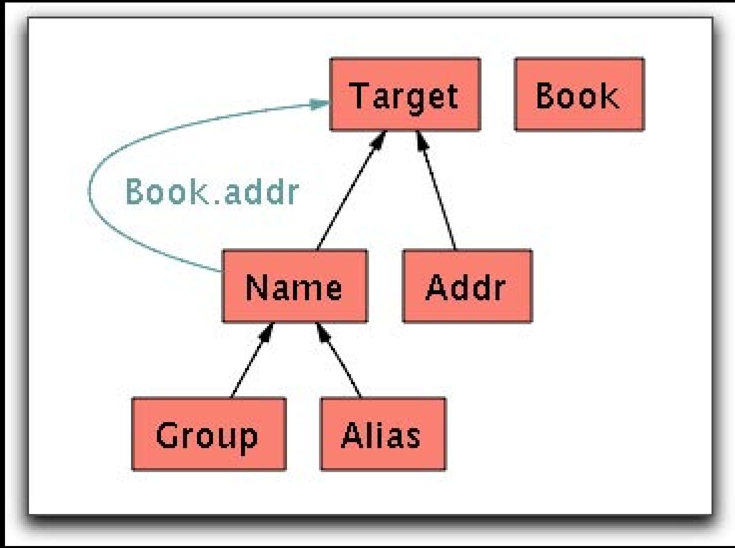
desiderata

› structure and classification
› expressiveness
› instant feedback
› declarative spec
› constraints, not test cases
› fully automatic checks

structure & classification

alloy model of an address book

```
abstract sig Target {}
abstract sig Name extends Target {}
sig Alias, Group extends Name {}
sig Addr extends Target {}
sig Book {
    addr: Name -> Target
    }
```



instant feedback

incremental analysis, of very partial models

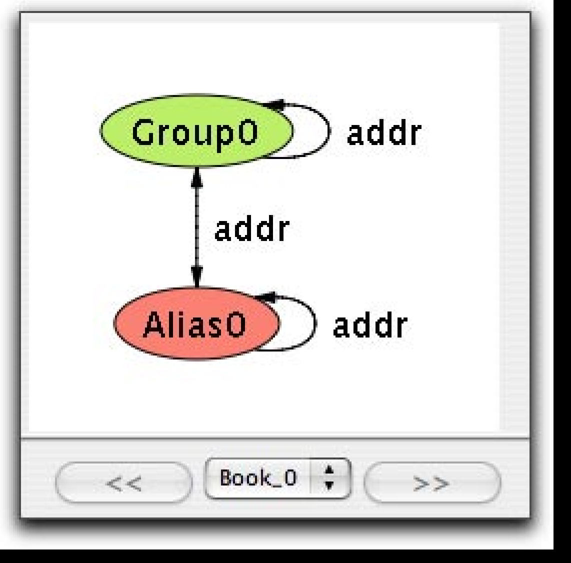
```
pred show () {}

run show
```

instant feedback

incremental analysis, of very partial models

pred show () {}

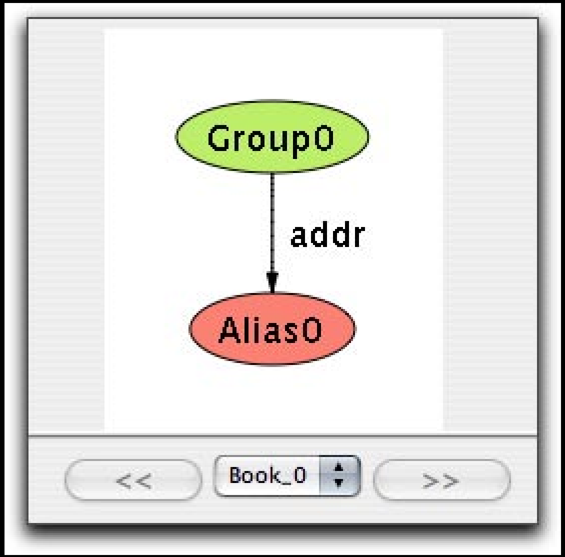
run show



expressiveness

no name refers to itself directly or indirectly

fact { all b: Book | no n: Name | n in n.^(b.addr) }



declarative spec (1)

first, an operational spec of deletion

```
pred del1 (b, b': Book, n: Name) {
    let ad = b.addr {
        some n.ad
        b'.addr = ad - (n->univ) - (univ->n) + ad.n->n.ad }
    }
```

declarative spec (2)

lookup

```
fun lookup (b: Book, n: Name): set Addr { n.^(b.addr) & Addr }
```

a declarative spec of deletion

```
pred del (b, b': Book, n: Name) {
    some n.(b.addr) and no n.(b'.addr)
    all x: Name |
        x = n implies no lookup (b', x)
        else lookup (b', x) = lookup (b, x)
    }
```

constraints, not test cases

show me an address book with more than two levels

```
pred show (b: Book) { some (b.addr).(b.addr) }
```

show me a deletion of a name at top-level

```
pred showDel (b, b': Book, n: Name) { del (b, b', n) and some n.(b.addr) and no b.addr.n }
```

show me a deletion that has no effect

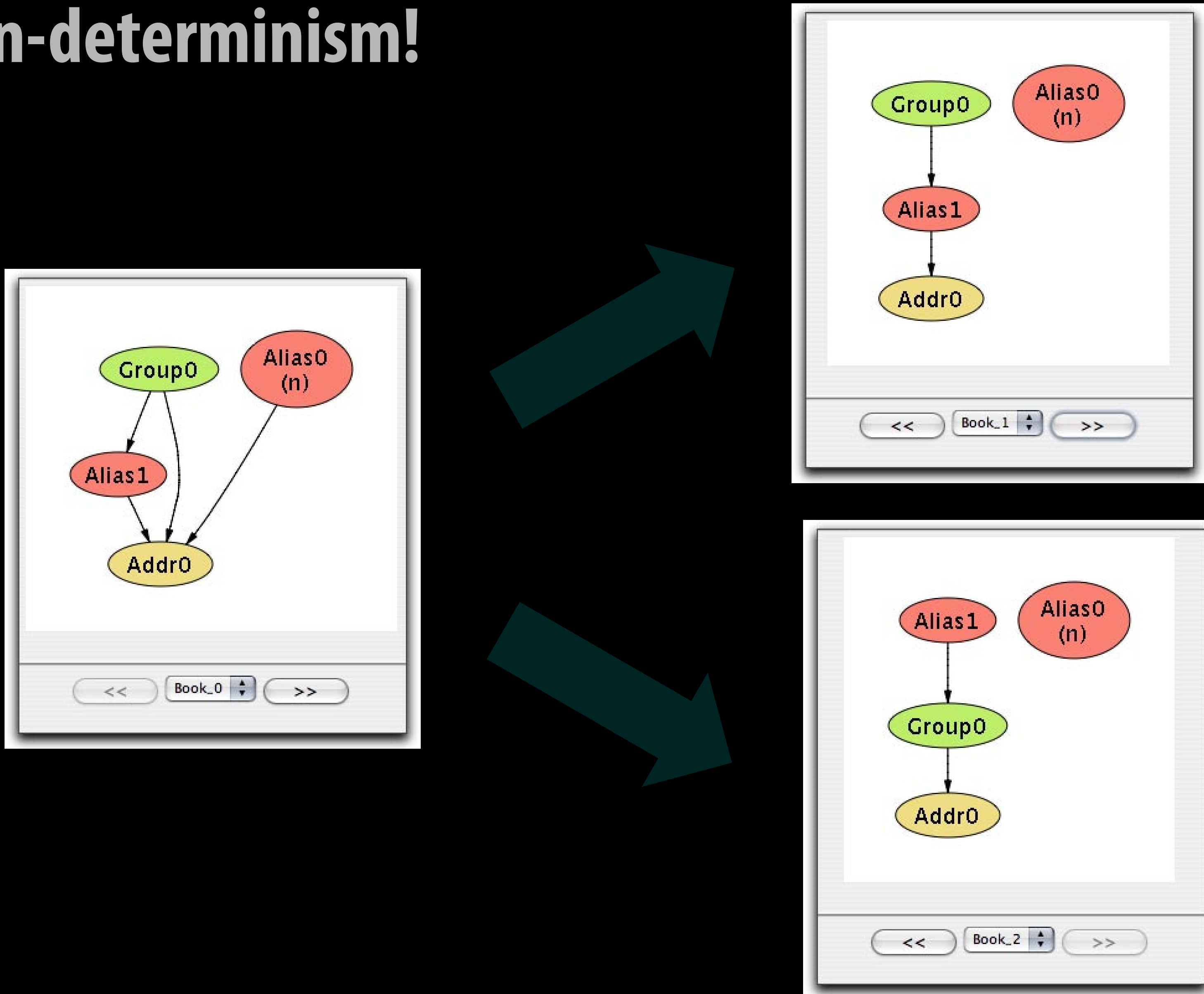
```
pred showDel (b, b': Book, n: Name) { del (b, b', n) and b.addr = b'.addr }
```

fully automatic checks

check that deletion is deterministic

```
assert delDeterministic {
    all b, b′, b″: Book, n: Name |
        del (b, b′, n) and del (b, b″, n) => b′.addr = b″.addr  }
check delDeterministic for 3
```

non-determinism!



missing complexities

no undefinedness all operators total, every expression defined

subtyping and overloading, but no casts or special type operators

composite structures but no higher-order logic

dynamics, mutation without built-in state machine idiom

resource-bounded analysis

language is undecidable

› no sound & complete algorithm

› so: try all small tests

scope: a bound on each type

› model proper is unpolluted

› user defines scope in command

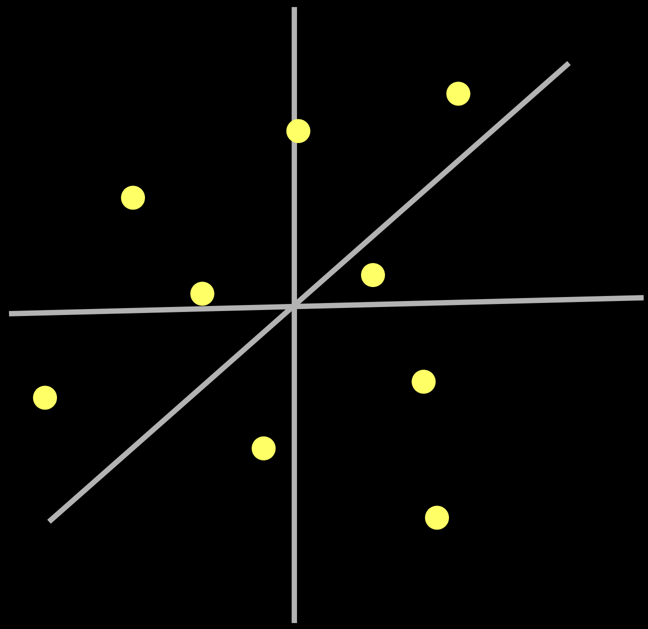
› can scope subtypes

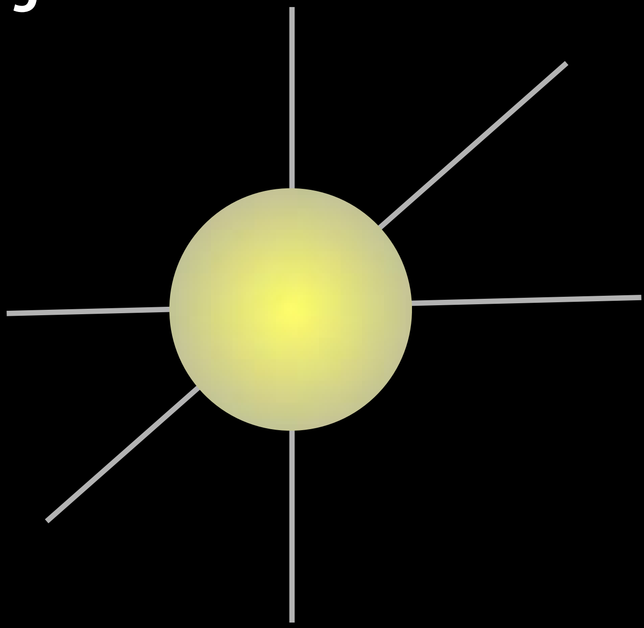
```
module book
open util/ordering [Book]
abstract sig Target {}
sig Addr extends Target {}
abstract sig Name extends Target {}
sig Alias, Group extends Name {}
sig Book {...}
assert delDeterministic {...}
check delDeterministic for
    6 but 3 Book
```

why it works in practice

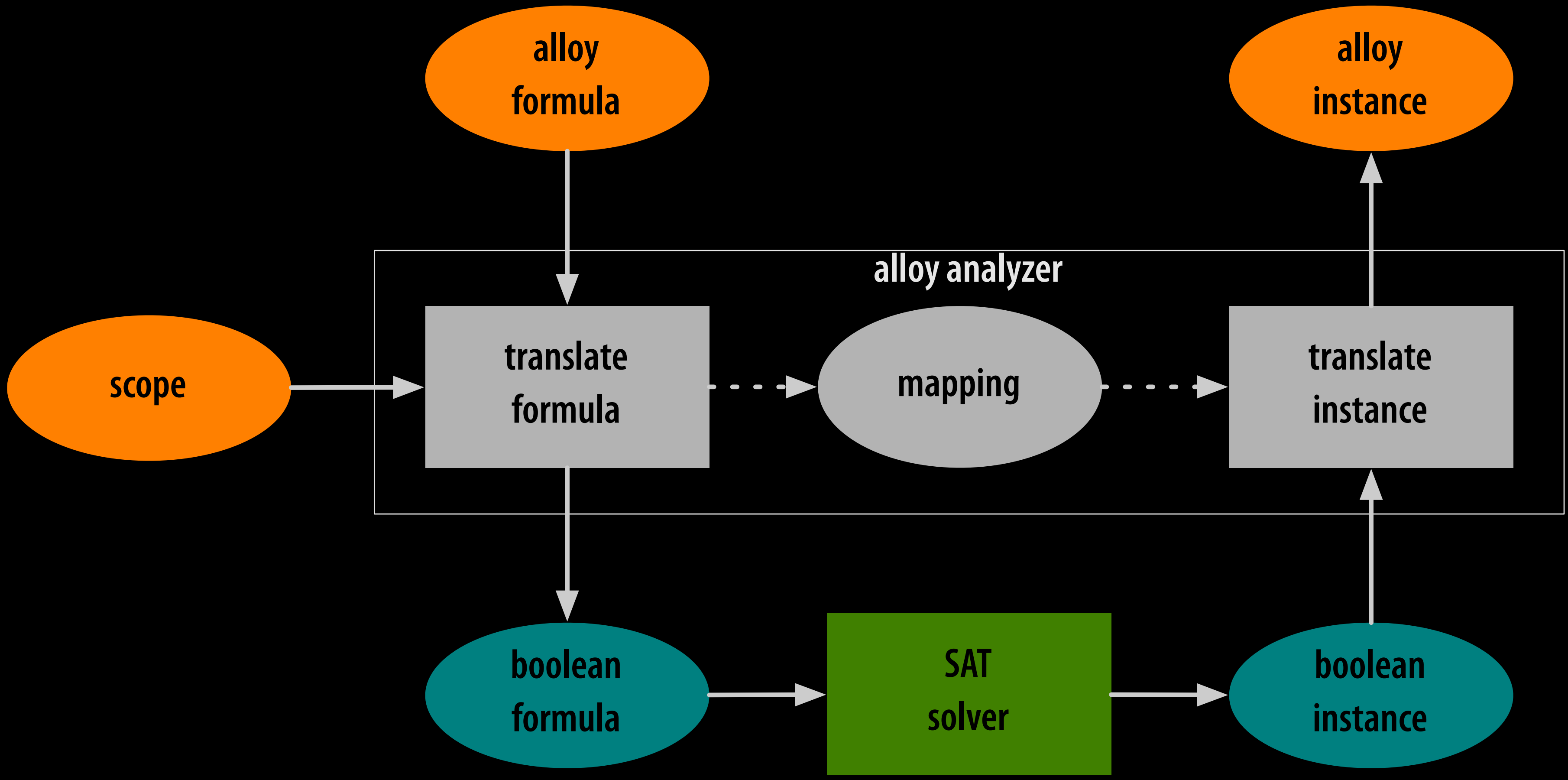
'small scope hypothesis'

- many bugs have small counterexamples
- . . . and models often have many bugs
- many more cases than traditional testing





alloy architecture



analysis by translation to SAT

analysis problem

› solve a constraint whose free variables are relations

› but in scope of 5, a ternary relation has 2^(5^3) ~ 10^30 values!

SAT: the quintessential hard problem

› SAT is hard (Cook, 1971)

› so reduce SAT to your problem

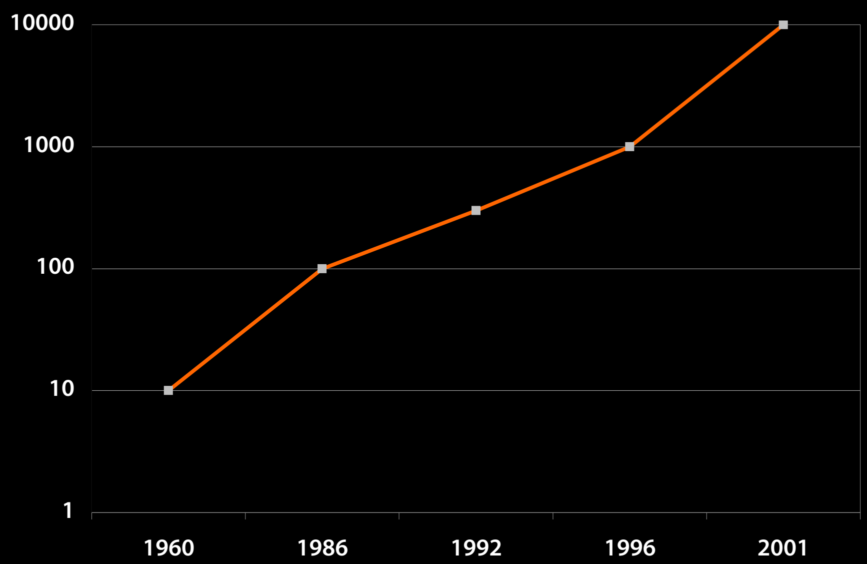
SAT: the universal constraint solver

› SAT is easy (Kautz, Selman et al, 1990's)

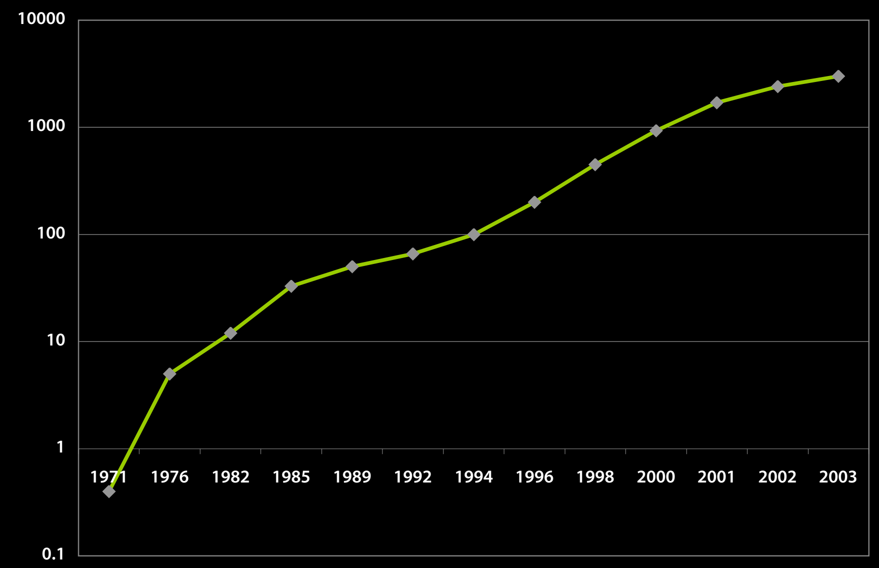
› so reduce your problem to SAT

technology advances

advances in SAT solvers
› size of solvable constraint
› in #boolean variables
from sharad malik



advances in processors
› speed in MHz
from intel.com



since 1990: factor of 100 from Moore's law, 10^{30} from SAT advances

patterns: trace

general form

```
open util/ordering [State] as so
pred op1 (s, s’: State)  {...}
pred opN (s, s’: State)  {...}
pred init (s: State)  {...}
fact traces {
    init (so/first ())
    all s: State - so/last () | let s’ = so/next (s) |
        op1 (s, s’) or ... or opN (s, s’)
    }
```

patterns: local state

instead of State as first column

sig State {f : A -> B}

a.(s.f)

make State last column

sig A {f: B -> State}

a.f.s

pattern instantiation: hotel locks

local state

```
sig Room {
    keys: set Key,
    currentKey: keys one -> Time
    }
one sig FrontDesk {
    lastKey: (Room -> lone Key) -> Time,
    occupant: (Room -> lone Guest) -> Time
    }
sig Guest { keys: Key -> Time }
```

trace

```
fact traces {
    init (to/first ())
    all t: Time - to/last() | let t' = to/next (t) |
        some g: Guest, r: Room, k: Key |
            entry (t, t', g, r, k)
            or checkin (t, t', g, r, k)
            or checkout (t, t', g)
    }
```

checking hotel locks

```
assert NoBadEntry {
    all t: Time, r: Room, g: Guest, k: Key | let t' = to/next(t) |
        entry (t, t', g, r, k) => g in FrontDesk.occupant.t [r]
    }
check NoBadEntry for 3 but 7 Time, 2 Room, 2 Guest
```

what has alloy been used for?

at MIT

› about 30 case studies, typically a few hundred lines long

› find flaws in almost everything we look at

› latest examples: beam scheduler for proton therapy, crypto

industrial uses

› animating requirements (TCS, India)

› military simulation (Northrop Grumman)

› role-based access control (BBN)

› telephony (AT&T)

test case generation

how

› characterize input tests with invariant

› have analyzer enumerate solutions

why?

› for complex structures, most random inputs are ill-formed

› Alloy's symmetry breaking reduces suite size

› less work, more coverage than manual test cases

Sarfraz Khurshid, 2003

code checking

basic idea

› model OO code with relations for fields

› extract constraint from code

› assert Code () => Spec ()

› scope sets path length within procedure, heap size, etc

so far, small systems but rich properties

› tally strategy of electronic voting software

basics + optimizations -- Mandana Vaziri, 2001/3

specification inference -- Mana Taghdiri, 2004

prospects & challenges

short term plans

› book on modelling & analyzing abstractions

› expanding Alloy user base, esp in education

long term plans

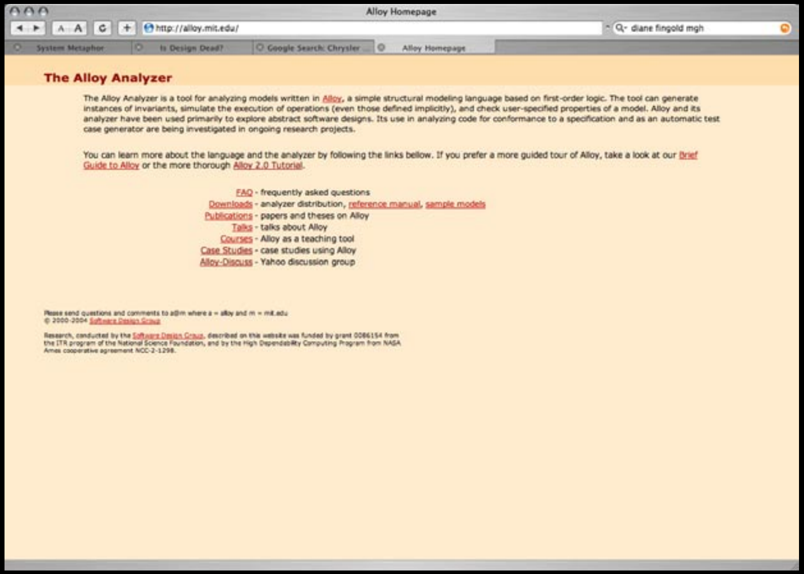
› new embedded Alloy as flexible API

› bridging design/code gap

for more information

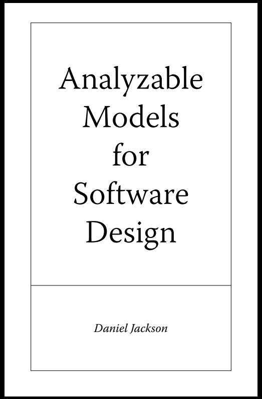
alloy.mit.edu

- › case studies
- › courses
- › tutorial
- › downloads



upcoming book (late 2005)

- › about modelling, not Alloy
- › patterns of modelling & analysis
- › lots of realistic examples



extra slides

reactions to UML

too complicated

› UML Reference Manual

576 pages; #62,915 in amazon.com

› Fowler, UML Distilled

192 pages; #1,516; 300,000 sold

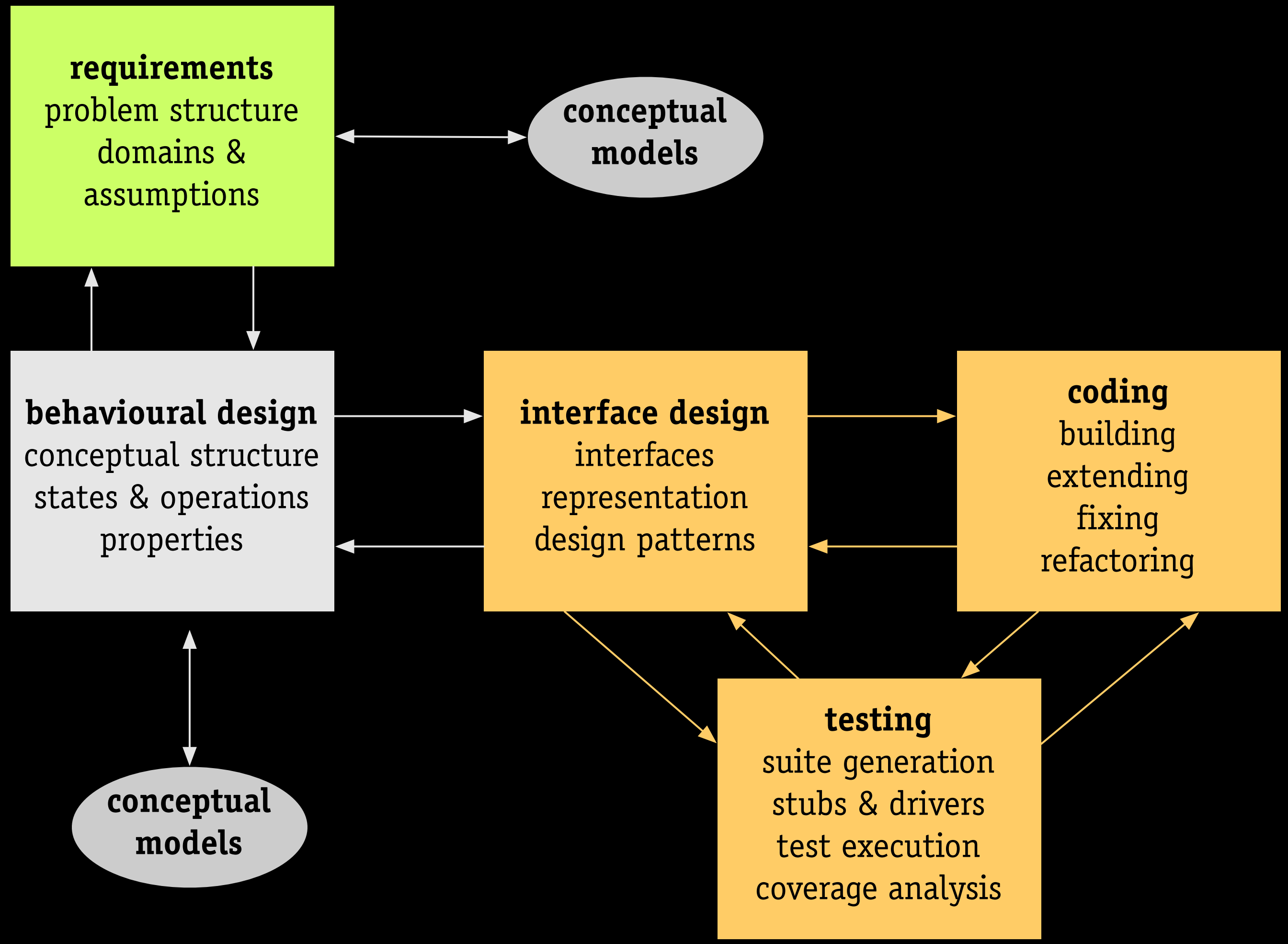
too burdensome

› inflexible process

› big documentation, little insight

revolution!

two kinds of design



origins of alloy

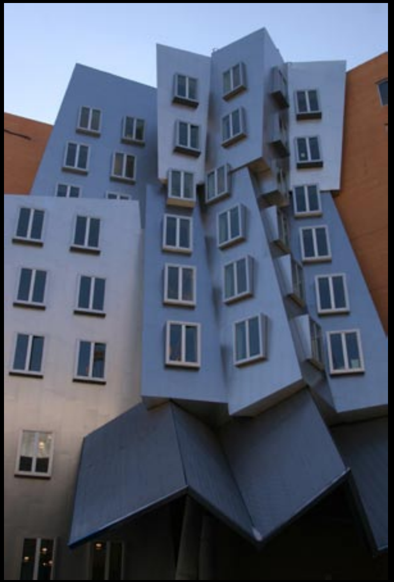
a notation inspired by Z
› just (sets and) relations
› everything's a formula
› but not easily analyzed

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

Oxford, home of Z

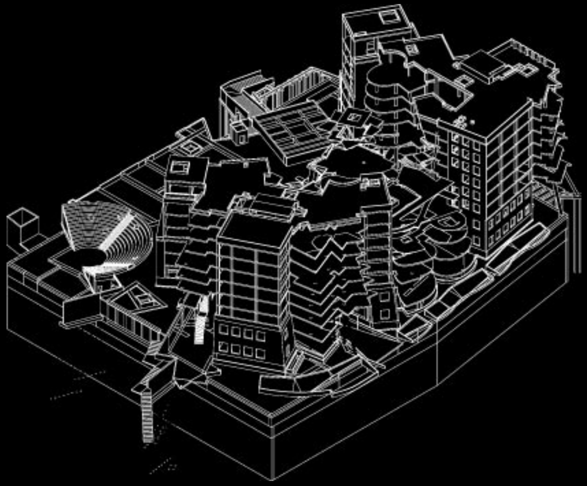
Pittsburgh, home of SMV

why analyzable models?



why models?

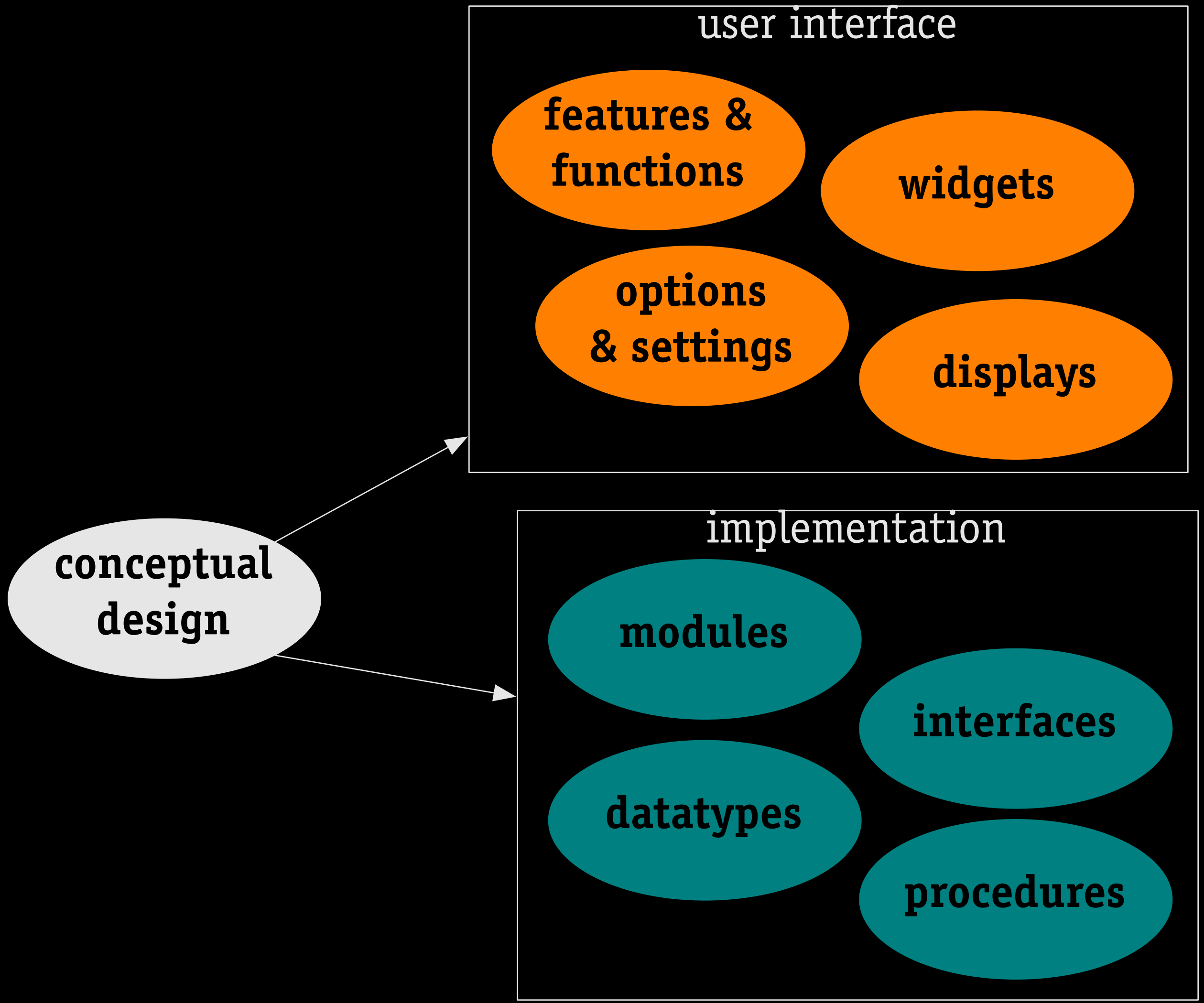
› figure out what problem you're solving
› explore invented concepts
› communicate with collaborators



why analyzable?

› not just finding errors early
› analysis breathes life into models!

impact of conceptual design



ease of use
flexibility
robustness

clean interfaces
decoupling
extensibility
dependability

xp on design models

Another strength of design with pictures is speed. In the time it would take you to code one design, you can compare and contrast three designs using pictures. The trouble with pictures, however, is that they can't give you concrete feedback… The XP strategy is that anyone can design with pictures all they want, but as soon as a question is raised that can be answered with code, the designers must turn to code for the answer. The pictures aren't saved. -- Kent Beck, Extreme Programming Explained, 2000

slide list