micromodels of software

Daniel Jackson
MIT Lab for Computer Science
MSU · November 13, 2001

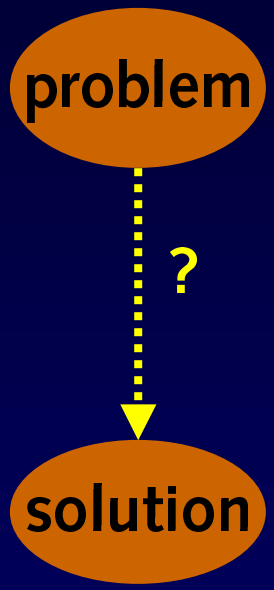
how to get good software?

traditional approach
- sacrifice a forest to write a function list
- flaws found late; depends on heavy testing

CASE tool approach
- draw a class diagram & turn the crank
- automates the easy part; adds complexity

strong formal methods
- specify function & refine to code
- expensive; can't handle modularity mechanisms

extreme programming
- start basic & refactor
- small projects only; code is poor exploratory medium



micromodelling

steal
- flexibility of traditional approach
- emphasis on models of CASE approach
- precision & clarity of formal methods
- nimbleness of extreme programming

risk-driven approach
- build tiny models of tricky aspects
- simulate & analyze for flaws
- justify aspects of code against models

why models?

cheap & effective exploration
- immediacy of coding
- coverage without test cases

conceptual basis for clean design
- strong & simple abstractions
- robust user concepts

medium for communication
- "we have become dangerously dependent on large software systems whose behavior is not well understood"
  -- PITAC report, February 1999

a complete micromodel

One man's ceiling is another man's floor
-- Paul Simon, 1973

but is one man's floor another man's ceiling?

```
module CeilingsAndFloors
sig Platform {}
sig Man {ceiling, floor: Platform}
fun Above (m, n: Man) {m.floor = n.ceiling}
fact {all m: Man | some n: Man | Above (n,m)}
assert BelowToo {all m: Man | some n: Man | Above (m,n)}
check BelowToo for 3
```

example: numbering paragraphs

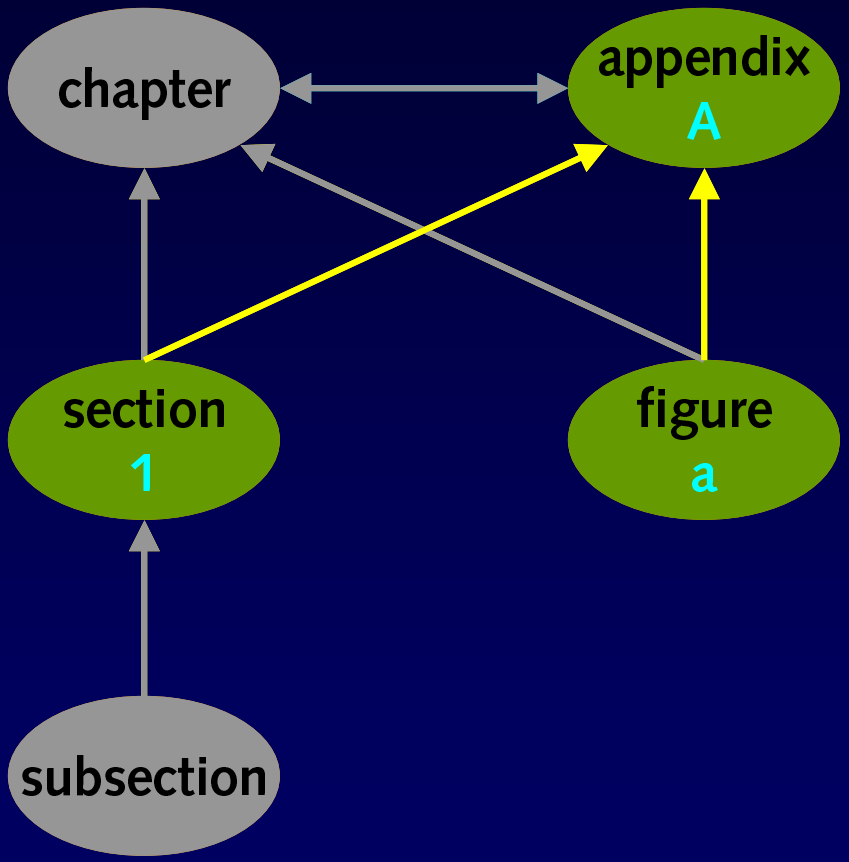
problem

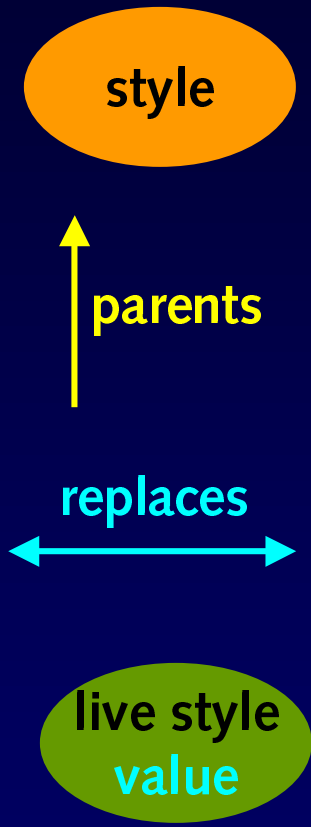
- number paragraphs tagged by style names
- interleaving (figure vs. section)
- variants (chapter vs. appendix)

\chapter Introduction
\section Motivation
\subsection Why?
\figure A nice diagram
\section Overview
\chapter Conclusions
\section Unrelated Work
\appendix Glossary
\section Terms

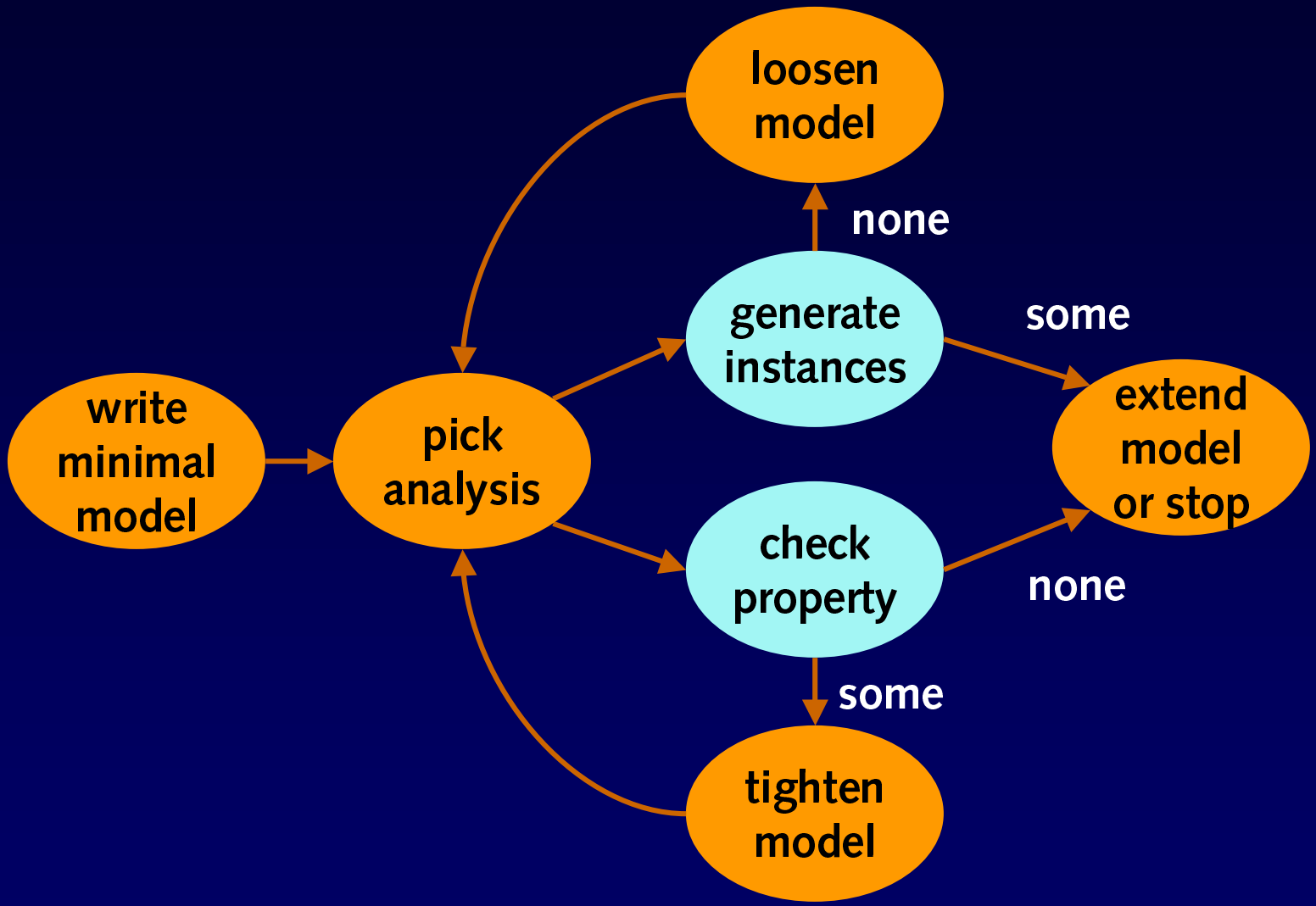
Chapter 1: Introduction
1.1 Motivation
1.1.1 Why?
Figure 1a. A nice diagram
1.2 Overview
Chapter 2: Conclusions
2.1 Unrelated Work
Appendix A: Glossary
A.1 Terms

a representation





how to use Alloy



style sheet

```
sig Style {
    replaces, parents: set Style
    }

fact ReplacesProps {
    all x: Style {
       x.replaces.parents.replaces = x.parents
       all y,z: x.parents | y in z.replaces
       }
    Equivalence (Style$replaces)
    Acyclic (Style$parents)
    }
```

state

```
sig State {
    context: set Style,
    ancestors: Style -> Style
    }

fact DefineAncestors {
    all s: State, x: Style |
      s.ancestors [x] = x.*parents & s.context
    }
```

well-formed state

```
fun Forest (s: State) {
    all x: s.context |
      some root: s.ancestors[x] {
          no root.parents
          all y: s.ancestors[x] - root | one y.parents & s.context
          }
    all x: Style | sole x.replaces & s.context
    }
```

incrementing

```
fun Increment (s, s': State, style: Style) {
    s'.context = s.context - style.replaces + style
    }
```

a check

```
assert PreserveForest {
    all s, s': State, x: Style |
      Forest(s) && Increment (s,s',x) => Forest(s')
    }

check PreserveForest for 4
```

fixing increment

```
fun Increment (s, s': State, style: Style) {
    all x: style.^parents | some x.replaces & s.context
    s'.context = s.context - style.replaces + style
    }
```

refactoring

```
fun Forest (s: State) {
    all x: s.context |
      some root: s.ancestors[x] {
          no root.parents
          all y: s.ancestors[x] - root | one y.parents & s.context}
    all x: Style | sole x.replaces & s.context
    }

fun Forest' (s: State) {
    some root: s.context {
      no root.parents
      all y: s.context - root | one y.parents & s.context}
    all x: Style | sole x.replaces & s.context
    }
```

checking a refactoring

```
assert SameForest {
    all s: State | Forest(s) iff Forest'(s)
    }

check SameForest for 2
```

adding numbering

```
sig Value {next: Value}

sig State {context: set Style, value: Style ->! Value}

fun increment (s, s': State, style: Style) {
    all x: style.^parents | some x.replaces & s.context
    s'.context = s.context - style.replaces + style
    s'.value[style] =
      if style in s.context
      then s.value[style].next
      else style.initial
    all x: Style - style | s'.value[x] =
          if style in x.^parents then x.initial else s.value[x]
    }
```

what I haven't shown you

constructs

- modules
- polymorphism
- nifty shorthands

idioms

- execution traces
- object models
- mutable objects

analyzability

assumption

¬ (declarative ∧ executable)

proof or counter?

- theorem provers fail badly
- optimizing for common case

making it work

- limit to first-order
- search in finite scope
- translate to SAT

Pittsburgh, home of SMV

Oxford, home of Z

Oxford photo from Jonathan Bowen's collection; Pittsburgh photo by Norm Schumm, with permission

making alloy first-order

objectification
- no composite structures
- everything's an atom

relations only
- every expression is a relation
- no special treatment of
  - functions: many-one binary relations
  - scalars: singleton sets

dereferencing
- takes n-ary relation and joins on first column

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

- no partial functions, no flattening, no higher-order objects

example

```
sig State {
    value: Style ->! Value
    }
```

conventional approach

value is a function from State to Style → Value

s.value is a partial function application

~value is higher-order

our approach

value is a ternary relation on ⟨State, Style, Value⟩

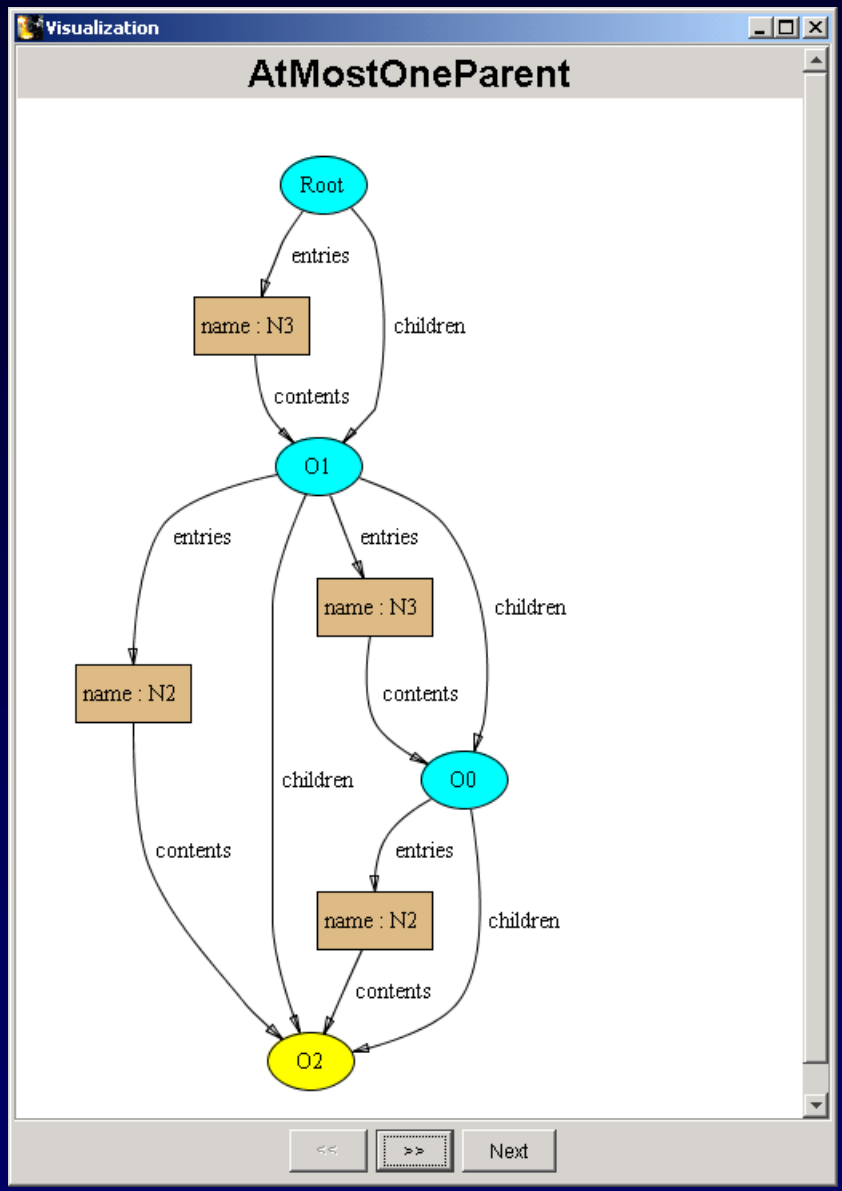
s.value joins ⟨State⟩ with ⟨State, Style, Value⟩

~value is just a ternary relation on ⟨Value, Style, State⟩

expressing structure

model checking
- great for event sequencing
- but not for state structure

examples
- topology of network
- links between Java objects
- naming relationships



being declarative

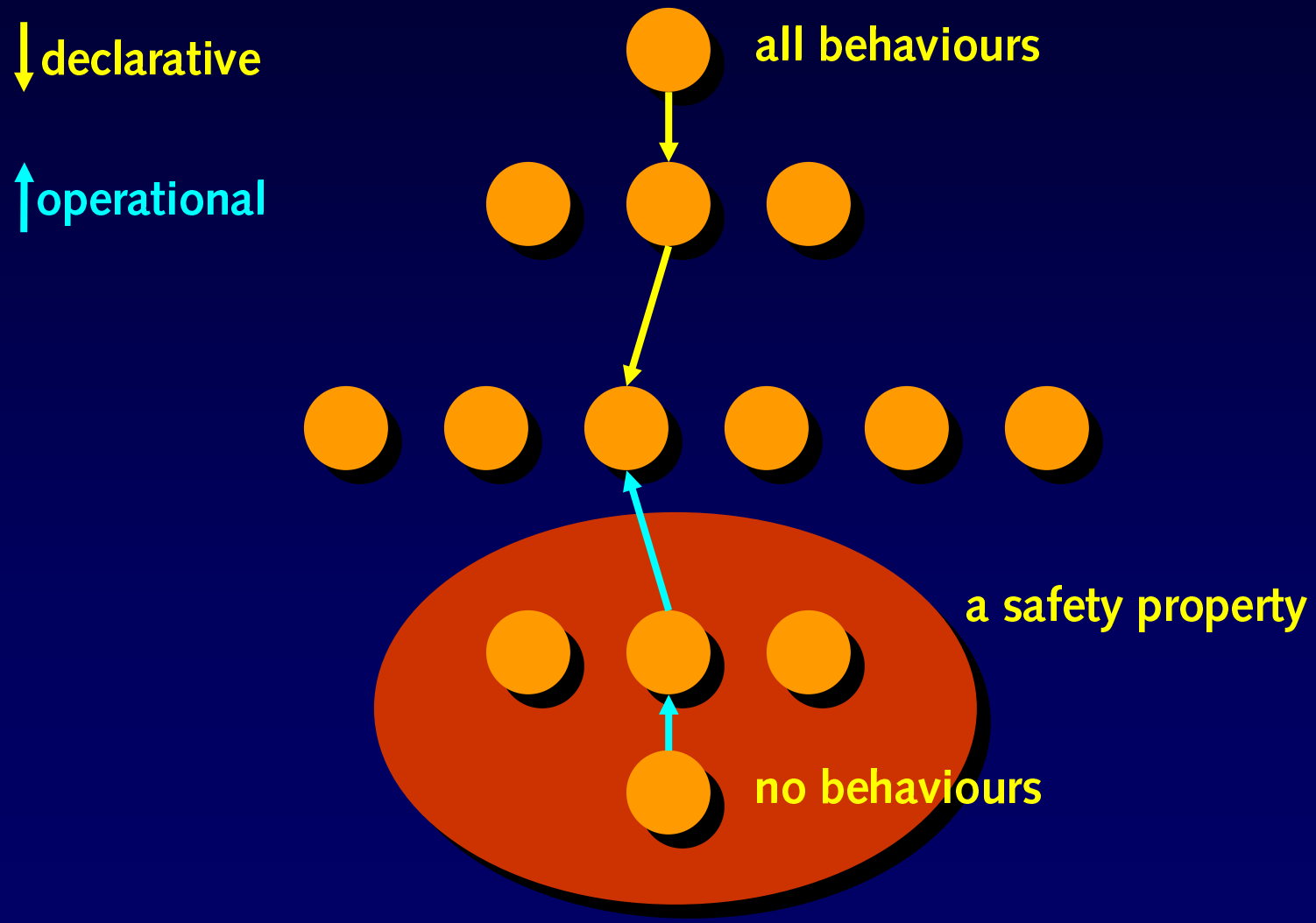
if my system has property P, will property Q follow?

an approach based on logic
- everything's a constraint
- one language, for model & theorems

advantages
- partiality: can say very little
- clarity: no need for mechanism
- masking: just conjoin failed property

the less said, the better
- environment model -- less to rely on
- design model -- less to implement

incrementality



applications of alloy

case studies
- Intentional Naming System
- COM interface rules
- Chord peer-to-peer nameservice
- rule-based access control

ongoing work
- models of conflict probe
- resource control in interpreter
- network topology protocols
- object interaction in Java

education
- taught in courses at a dozen universities

finding the last bug?

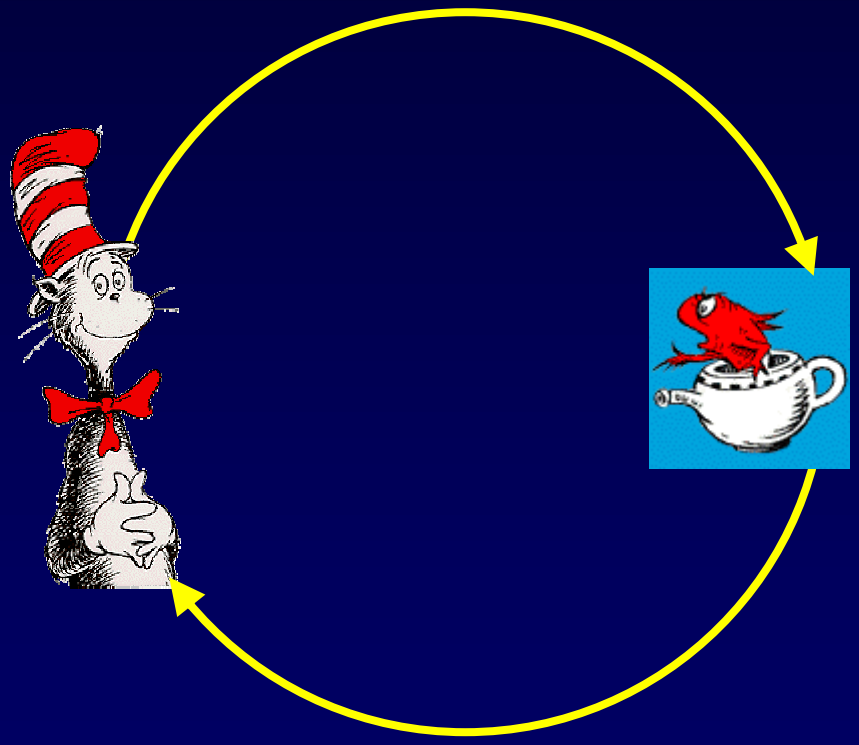
model checking
- benefit is detection of showstopper bug
- mostly batch

alloy analysis
- benefit is shaping of model
- like programming, but abstract
- very interactive

iterative design

interplay

- between the designer
- and the skeptic (the tool)



related research

analyses
- Wahls & Leavens constraint solver (2000)
- USE tool for OCL (2000)
- IFAD VDM animator
- using SAT: planners, model checking

related languages
- Z specification language (1982)
- McCarthy's situation calculus (1969)

modelling approaches
- Fowler's Analysis Patterns (1997)

links

sdg.lcs.mit.edu/~dnj/publications
- alloy language
- reduction to SAT
- case studies
- translating code to alloy

sdg.lcs.mit.edu/alloy
- alloy analyzer
- release on november 16

acknowledgments

Alloy team
- Ilya Shlyakhter
- Manu Sridharan
- Dan Kokotov
- Brian Lin
- Jesse Pavel
- Sarfraz Khurshid
- Mandana Vaziri
- Gregory Dennis
- Michal Mirvis
- Hoeteck Wee

aspects to model

1. problem domain
   Problem Frames, Michael Jackson, AW 2001
2. invented concepts
   Analysis Patterns, Martin Fowler, AW 1997
3. code structure
   Design Patterns, Erich Gamma et al, AW 1995

I'll illustrate (2) with a complete micromodel

scalability

current limits
- < 20 relations, scope < 5 usually ok
- state space approx 10^100 states
- gets stuck unpredictably

SAT technology
- Chaff much faster than current solver
- we have a parallel backend

Moore's law
- couldn't have done this in 1980
- 3 hours --> 3 seconds