lightweight formalism

Daniel Jackson
MIT Lab for Computer Science
NASA Ames · August 14, 2001

crisis of dependability

We have become dangerously dependent on large software systems whose behavior is not well understood
-- PITAC report, February 1999

software is

- unreliable
  fails in unpredictable ways
- inflexible
  changes make it worse
- expensive
  not much reuse of designs or code

traditional approaches

testing

- can’t replicate real world
- tiny proportion of cases covered
- no help with incompleteness of spec

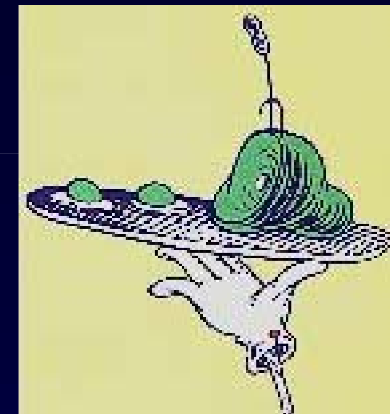
prototyping

- too costly for all but GUI
- temptation to build on prototype
- no guarantees for final product

process

- focus on human resources alone
- insensitive to nature of software
- can be major burden (hence XP, etc)

a new approach



models
- capture essence of spec & design
- much smaller than standard docs

patterns
- recognize recurring themes
- codify and reuse them

escaping from code
- not nimble enough for exploration
- poor communication medium
- our basic asset, but a huge liability

why formalism?

English + pictures
- ambiguous & imprecise
- not analyzable

formal notations
- based on mathematics
- offer simulation, calculation, analysis like conventional engineering

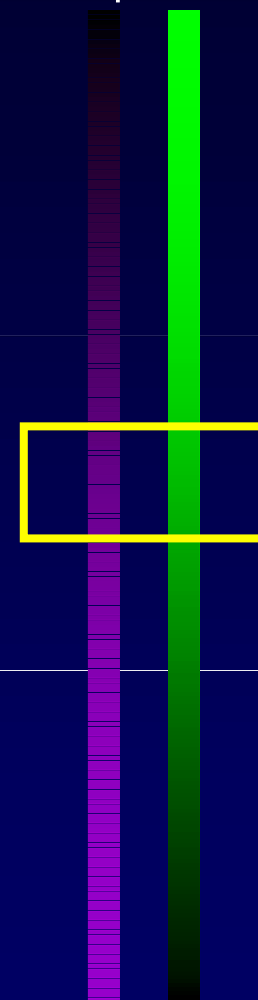
Praxis's CDIS system for UK air-traffic
- offered warranty to client
- formal specification: 8% of total cost
- low defect rate, overall cost saving

the alloy approach

automation
- simulation & checking of models
- tradeoff for full analyzability
- exploit Moore's law

process
- incremental and risk-driven
- apply in all phases of development
- at all levels from models to bytecode

origins
- Z formalism (Oxford)
- SAT solving technology (Bell Labs et al)
- semantic data models



applications of alloy

case studies

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

ongoing work

- models of conflict probe
- network topology protocols
- secure VPN
- object interaction in Java

what I'll show you

a typical problem

- from a text-processing program
- express essence in Alloy
- apply automatic analysis

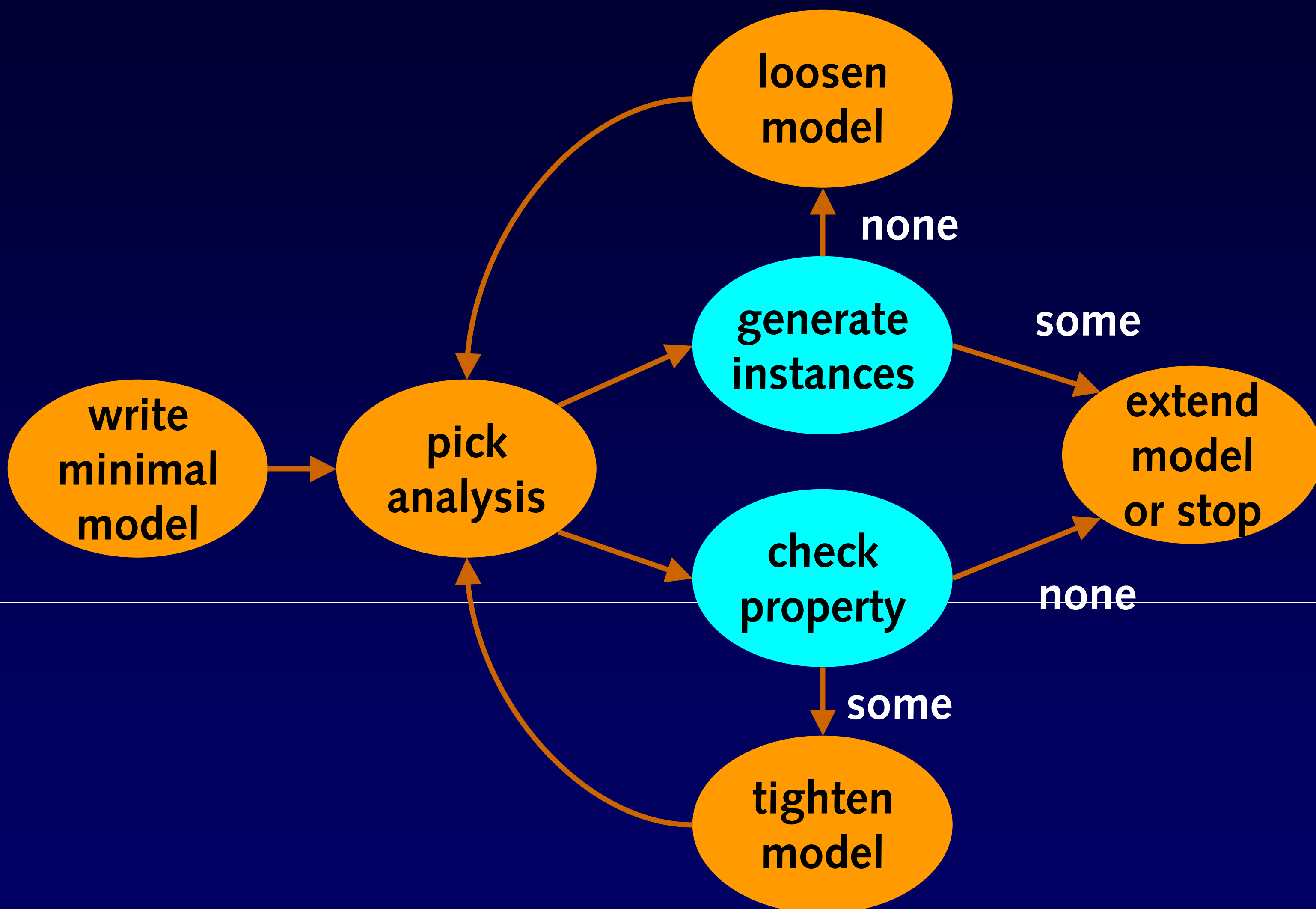
widely applicable

- based on common patterns (Command, Observer)
- same as our 1998 design of CTAS CM
- avoid failure with generic discipline

two levels: abstract design & code

- same notation, same analysis

how to use Alloy



the key to Alloy

partiality

- if my system has property P, will property Q follow?
- is environmental assumption A enough?

Alloy is a constraint language (ie, a logic)

- the less you say, the more can happen
- can express property
  without mechanism to achieve it

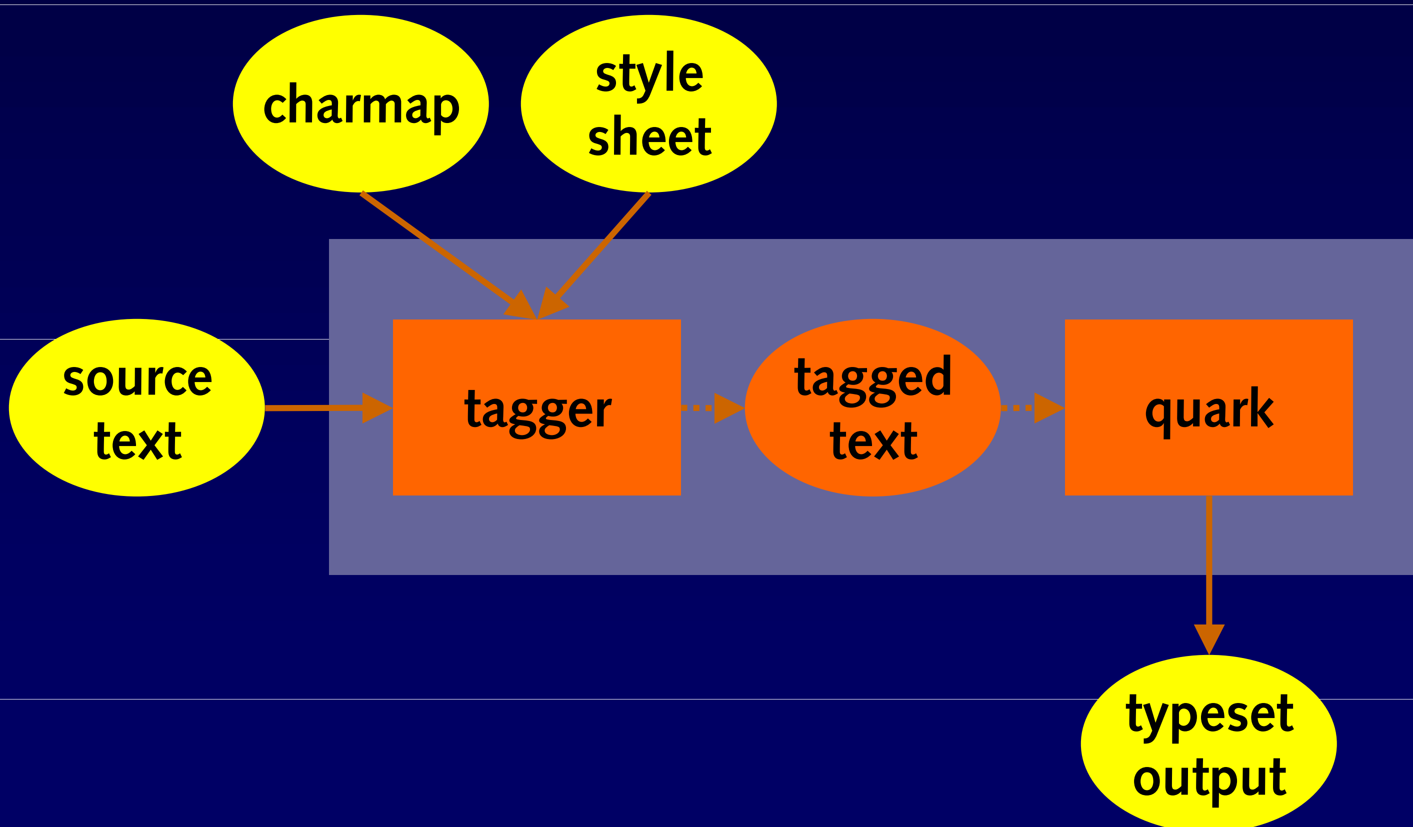
programming languages

- the less you say, the less can happen
- every property has to be achieved

example: tagger program

combine

- typographic quality & flexibility of Quark Xpress
- with convenience of textual input of TeX



sample translation

source

```
\point _A constraint solving technique_. We use a technology developed
previously [\cite{alloy-algorithm}] to analyze the skeleton
```

style sheet

```
<style:point><next:noindent><leader:\periodcentered>
```

charmap

```
<char:periodcentered><index:183>
```

tagged

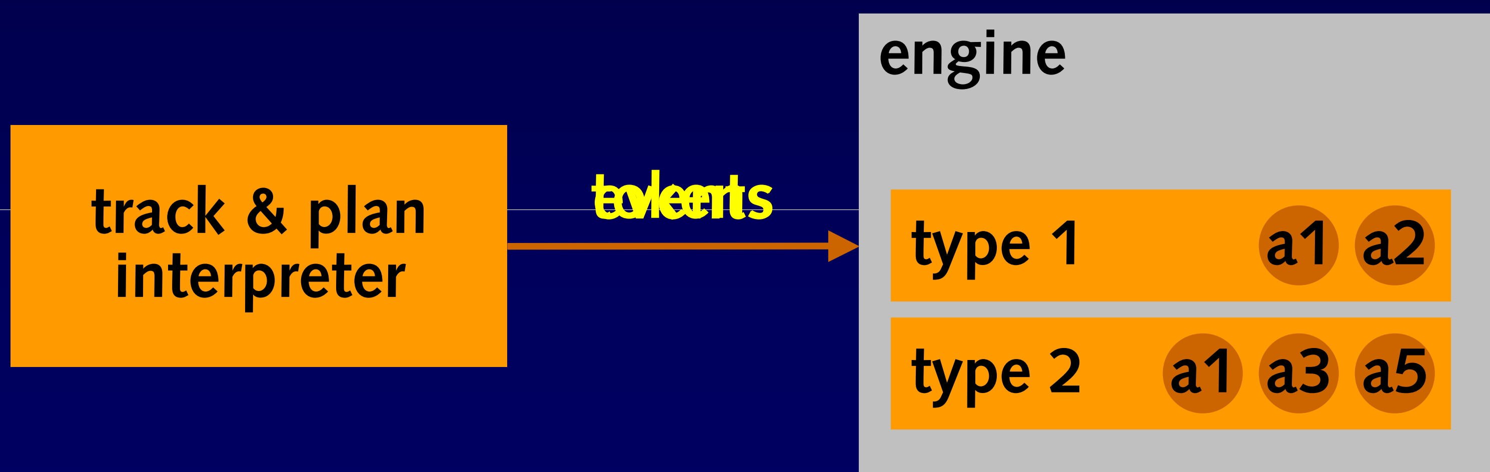
```
@point:<\#183><I>A constraint solving technique<I>. We use a technology
developed previously [6] to analyze the skeleton
```

output

· *A constraint solving technique*. We use a technology developed previously [6] to analyze the skeleton

abstract design

- source converted to stream of tokens
- each token consumed by engine
- engine executes all actions registered for token type
- actions may read & write files, register/deregister actions



why are actions nice?

- flexible, context-sensitive behaviour
- without one huge, monolithic piece of code
- easy to turn behaviours on & off
- can have subengines: eg, for numbering strings

example: math mode

- $ starts and ends ‘math mode’: puts all letters in italics
- implemented as 2 actions
  - for $ token: dollar_action
  - for alphabetic string token: italicize_action
- dollar_action
  - encapsulates mode
  - registers/deregisters italicize_action

alloy basic concepts (ABC)

atom

- an indivisible, uninterpreted entity

  a token, a token type, a person, a string, an aircraft

set

- an unordered collection of atoms

  Token, Type, Person, String, Aircraft

relation

- a set of tuples of atoms

  type: Token -> Type

  myPhoneBk: Name -> Num

  phoneBk: Name -> Name -> Num

alloy operators

set operators

s + t	union
s & t	intersection
s - t	difference

relation operators

s . r	image
p + q	union, etc

```
myPhoneBk: Name -> Num
  myFriends: set Name
  myFriends.myPhoneBk

phoneBk: Name -> Name -> Num
  me: Name
  me.phoneBk
  myFriends.phoneBk

mother: Name -> Name
  me.mother.phoneBk
  (me + me.mother).phoneBk
  me.phoneBk - me.mother.phoneBk
```

a design model of an action machine

```
sig Token {type: Type}
sig Type {}
sig Action {regs, deregs: Type -> Action}
sig Engine {table: Type -> Action}

fun execute (e, e': Engine, t: Token) {
   doActions (e, e', t.type.(e.table))
   }

fun doActions (e, e': Engine, actions: set Action) {
   e'.table = e.table - actions.deregs + actions.regs
   }
```

simulation

Token	= to1, to2	
Type	= ty1, ty2	
Action	= a1, a2	
Engine	= e1, e2	
type	= to1 -> ty1,	to2 -> ty2
regs	= a1 -> ty1 -> a2,	a2 -> ty2 -> a1
deregs	= a1 -> ty1 -> a1,	a1 -> ty2 -> a1
table	= e1 -> ty1 -> a1,	e2 -> ty1 -> a2
e	= e1	
e′	= e2	
t	= to1	

a design question

can implementation do actions in any order?

```
assert {
   all e, ei, e', e": Engine, t: Type,
      ax, ay: set t.(e.table) |
         doActions (e, ei, ax) && doActions (ei, e', ay)
         && doActions (e, e", ax+ay)
         => e'.table = e".table
   }
```

counterexample

the scenario

- first doAction registers an action
- second doAction deregisters it
- but aggregate doAction collects deregisters upfront

design challenge

what strategy for order independence?
- must be simple and static

here's one strategy
- group actions that reg/dereg each other
- say that ≤ 1 action of a group may register for type t
- rule out arbitrary initialization of engine

to check it
- write definition of grouping
- write registration and initialization rules
- recheck assertion

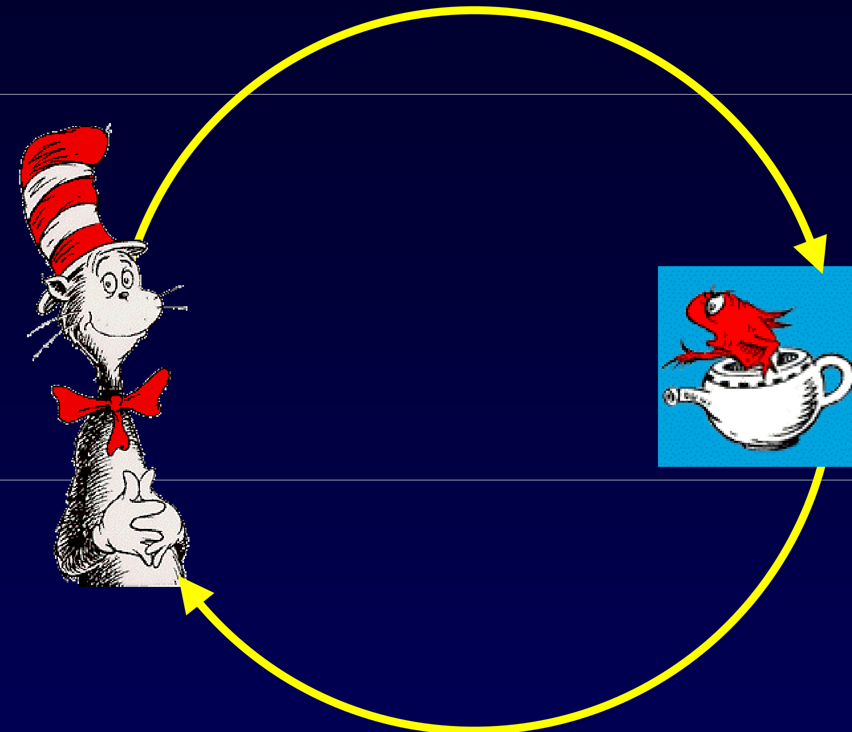
what the rest looked like …

```
sig GroupedAction extends Action {group: set Action}
fact {
   let depends = Action$regs + Action$deregs |
      Action$group = *(depends + ~depends)
   }
fact {Action = GroupedAction}

fact NoClashes () {
   all t: Type | no a: Action, a': a.group - a |
      a + a' in t.Action::regs
   }

fact NoArbitraryInits () {all e: Engine | e.table in Action.regs}
```

experience

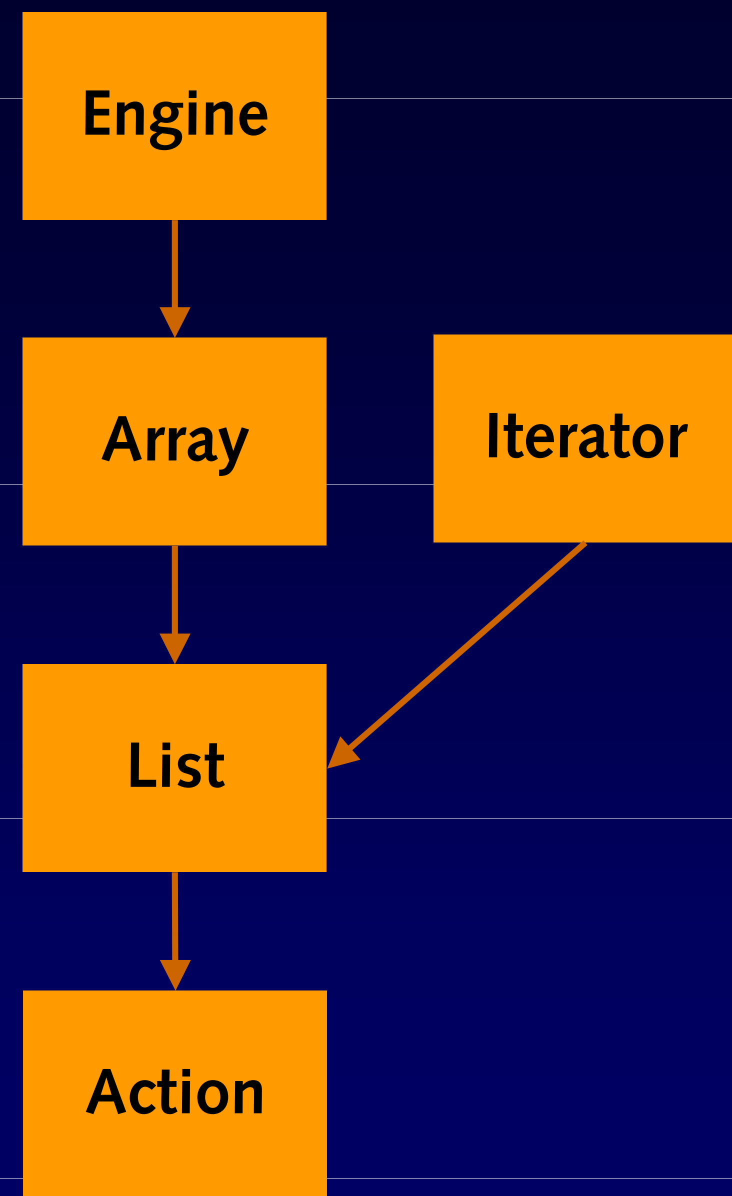


- took me 2 hours to develop this strategy
- lots of false attempts on the way
  rapidly exposed by tool
- about 100 lines of Alloy written
- for 3 actions, 10^54 cases
  counterexample < 1 second
  exhausting space < 1 minute

moving to code

Engine class
- holds action lists
- in array indexed on token type

consuming token
- creates iterator object
- yields each action in turn
  and performs it

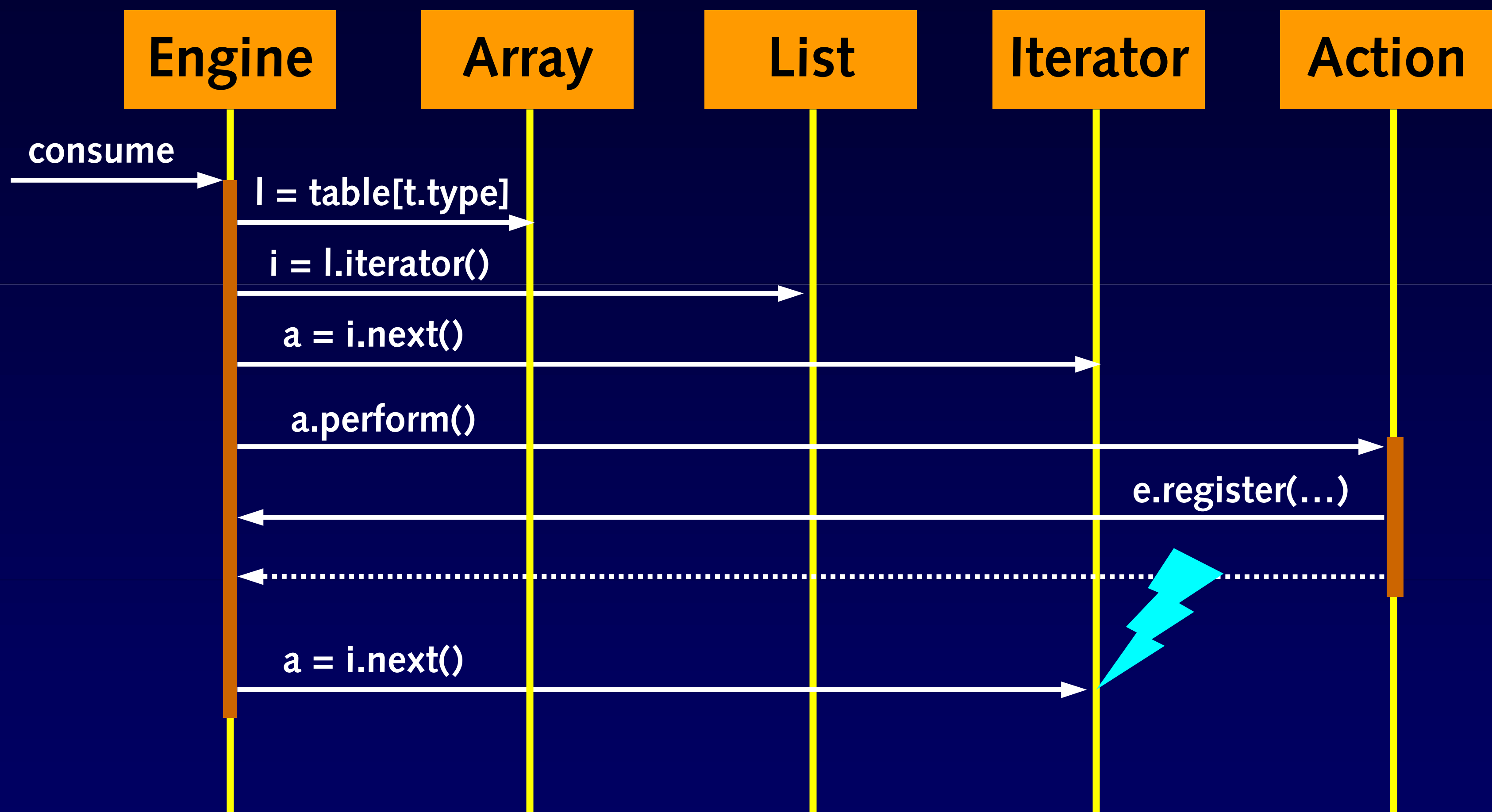


code of engine

```
private LinkedList [] table;
…
public void register (Action action, int type) {
   actions[type].add (action);
   }

public void consume_token (Token token) {
   Iterator i = table[token.type].iterator ();
   while (i.hasNext ()) {
      Action a = (Action) i.next ();
      a.perform (token, i);
      }
   }
```

failure!


Engine
Array
List
Iterator
Action
consume
l = table[t.type]
i = l.iterator()
a = i.next()
a.perform()
e.register(…)
a = i.next()

comodification

problem

- iterator shares state with list
- holds cursor into representation
- call to list's add may invalidate cursor

Java's solution

- iterator and list have version numbers
- list's iterator creates iterator with same version
- list's add/remove increments list version
- at start of iterator's next
  check versions match
  if not, throw exception

approach

build minimal Alloy models

- of list and iterator
- of engine structure

translate engine method

- into Alloy assertion
- and analyze it

to fix problem

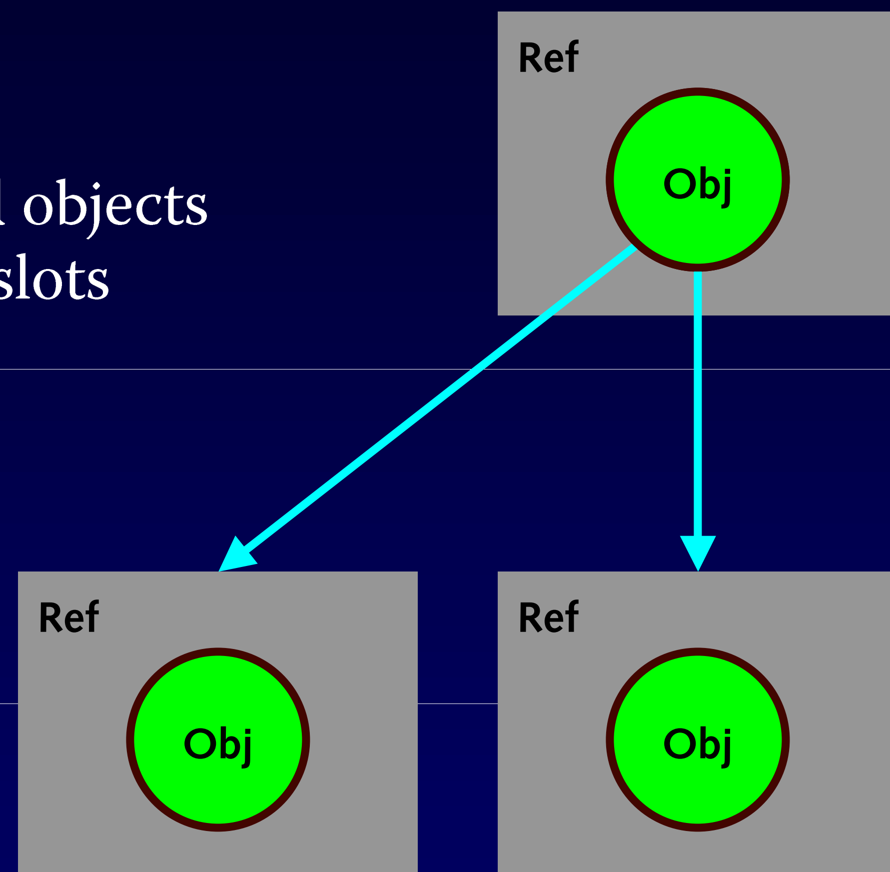
- develop general discipline
- add to model and reanalyze
- (check code against discipline)

modelling the heap

two kinds of Alloy atoms

- references: slots that hold objects
- objects: values that go in slots

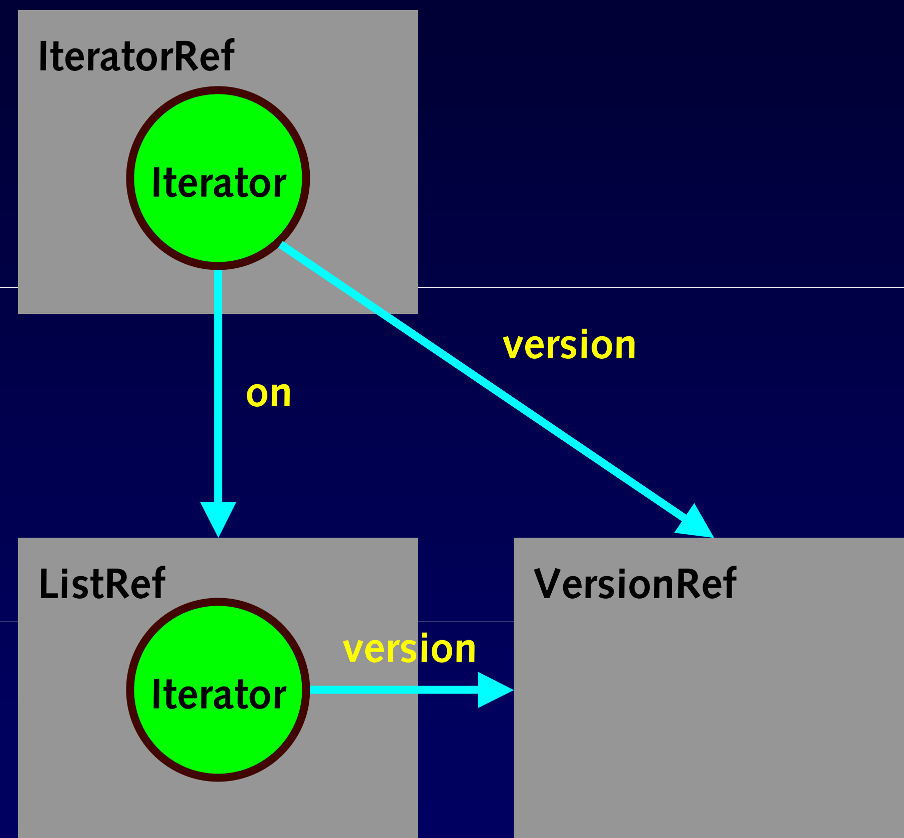
```
sig Ref {}
sig Obj {}
sig State {
    refs: set Ref,
    obj: Ref ->? Obj
    }
```



modelling iterators & lists

```
sig IteratorRef extends Ref{}
sig Iterator extends Obj {
    on: ListRef,
    left: set Ref
    version: VersionRef
    }

sig ListRef extends Ref{}
sig List extends Obj {
    elts: set Ref
    version: VersionRef
    }
```



modelling iterator methods

```
fun next (s, s': State, this: IteratorRef, x: Ref) {
    x in this.(s.obj).left
    this.(s'.obj).left = this.(s.obj).left - x
    this.(s'.obj).version = this.(s.obj).version
    modifies (s, s', this)
    }

fun next-pre (s, s': State, this: IteratorRef) {
    this.(s.obj).version = this.(s.obj).on.(s.obj).version
    }

fun modifies (s, s': State, rs: set Ref) {
    all x: s.refs - rs | x.(s'.obj) = x.(s.obj)
    }
```

modelling list methods

```
fun iterator (s, s': State, this: ListRef, result: IteratorRef) {
    result.(s'.obj).on = this
    result.(s'.obj).version = this.(s.obj).version
    result.(s'.obj).left = this.(s.obj).elts
    modifies (s, s', {})
    }

fun add (s, s': State, this: ListRef, x: Ref) {
    this.(s'.obj).elts = this.(s.obj).elts + x
    modifies (s, s', this)
    }
```

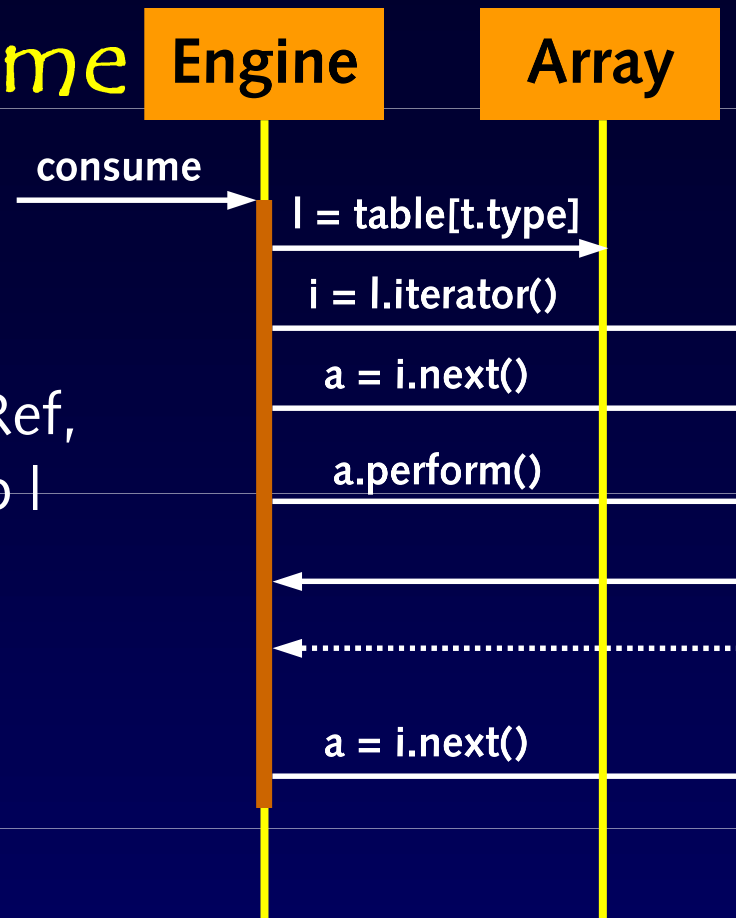
modelling the engine

```
sig Engine extends Obj {
   table: TypeRef ->! ListRef
   }

fun register (s,s': State, this: EngineRef, a: ActionRef, t: TypeRef){
   let l = t.(this.(s'.obj).table) {
      add (s, s', l, a)
      modifies (s, s', l)
      }
   }
```

translating engine_consume

```
assert {
    all s0, s1, s2, s3: State,
        e, e': EngineRef, l: ListRef, i: IteratorRef,
        t: TokenRef, a: ActionRef, a': a.group |
          {
          l = t.(s0.obj).type.(e.(s0.obj).table)
          iterator (s0, s1, l, i)
          next (s1, s2, i, a)
          register (s2, s3, e', a')
          }
          => next-pre (s3, i)
    }
```



bug!

tool finds a counterexample

- what if one list for two types?
- fix multiplicity of table
- (and check code)

```
sig Engine extends Obj {
  table: TypeRef ?->! ListRef
  }
```

so what?

nimble modelling

- models are tiny
- analysis is fast
- feedback is motivating

levels & phases

- abstract & code design
- early and late

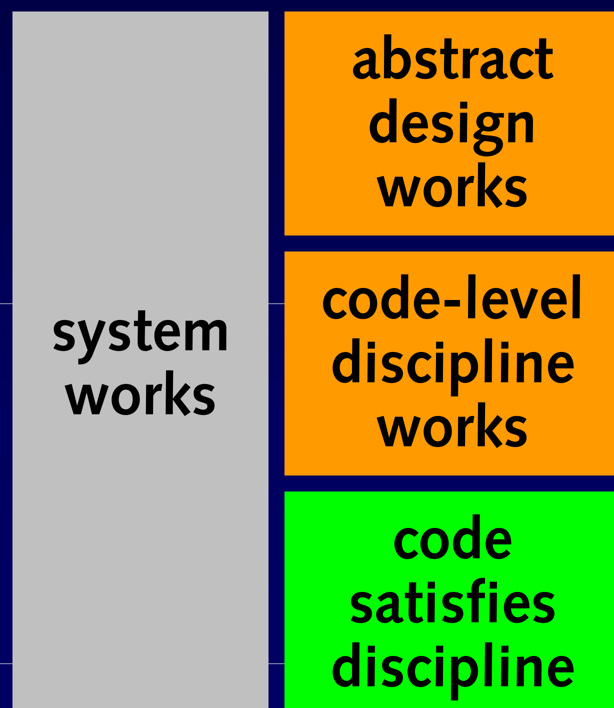
systematic

- developed general rules
- discipline for future changes

context

design conformance

- NSF/ITR grant with Martin Rinard
- combines Alloy with Rinard's shape analysis



related research

verifying state machines

- Pathfinder: applies Spin to Java (Ames)
- Bandera: translates Java to finite state model (Kansas)
- SLAM: translates code to 'boolean programs' (Microsoft)
  - for events not heap structure

lightweight code analysis

- LCLint (Maryland), PreFIX (Msft), ESC (Compaq)
  - for simpler properties, eg null derefs
  - uniform across code

acknowledgments

Alloy language and tool

- Ilya Shlyakhter
- Manu Sridharan
- Brian Lin

Alloy case studies

- Sarfraz Khrushid
- Mandana Vaziri
- Gregory Dennis
- Michal Mirvis
- Hoeteck Wee

Tagger example

- Alan Fekete

links

http://sdg.lcs.mit.edu/~dnj/publications
- research papers
  - alloy language
  - alloy analysis technology
  - object model extraction
  - object models
- lecture notes
  - software design with object models

http://sdg.lcs.mit.edu/alloy
- alloy analyzer
  - new tool released mid-september