LIGHT FORMAL METHODS

Daniel Jackson
MIT Lab for Computer Science

Formal Methods Europe

Berlin, March 14, 2001

what I'll tell you about

Alloy

- a micromodelling language
- first-order logic, so declarative
- new modularity mechanism

Alloy Analyzer

- checking & simulation
- sound and comprehensive

… and along the way

- thoughts about formal methods

two schools

Pittsburgh, home of SMV
- focus on analysis
- get errors, discard model

Good for systems with
- intricate workings

Oxford, home of Z
- focus on language
- build model, maybe analyze

Good for systems with
- intricate specs

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

the boston school (almost halfway)

lightweight formal methods

- “less trouble than they’re worth”
- modelling & analysis

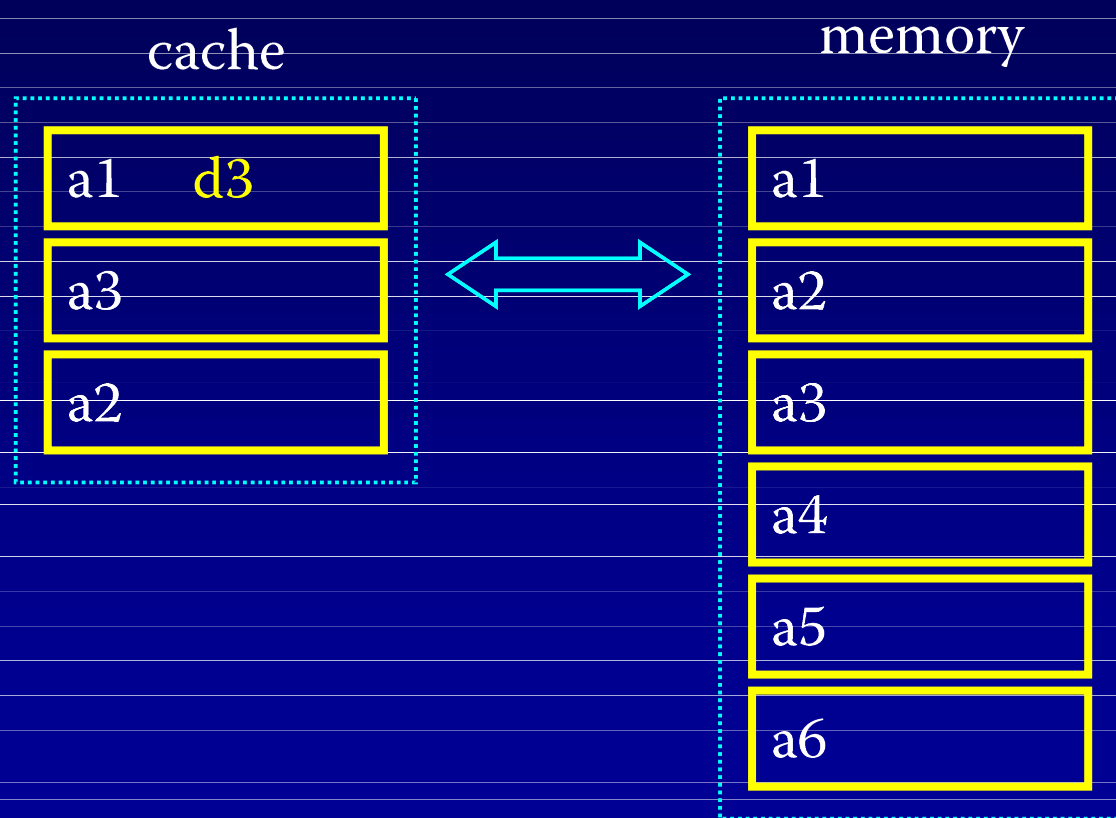
combine

- declarative models
    - unusually small & simple language
    - exploit partiality and abstraction
- automatic analysis
    - no user intervention
        - proof strategy, test cases
    - constructive output

a simple memory system

Write(a1,d3)

Read (a1,d3)



part 1: the language

semantics: types

basic types

Addr, Data, Memory

relational types

⟨Memory⟩

⟨Addr, Data⟩

⟨Memory, Addr, Data⟩

e, (e), {e}, {(e)}
all represented as {(e)} !

basic type stands for its set

Memory means universal relation of type ⟨Memory⟩

set: a unary relation

tuple: a singleton relation

scalar: a singleton set

semantics: operators

standard operators

- union (+), intersection (&), difference (-)
- transpose (~), closure (+p)

funky operators

dot

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

arrow

$[\![p \text{ -> } q]\!] = \{ (p_1, \ldots p_n, q_1, \ldots q_m) \mid (p_1, \ldots p_n) \in [\![p]\!] \wedge (q_1, \ldots q_m) \in [\![q]\!] \}$

puns

p.q, x.r, s.f	join, dereference, image
A -> B, x -> y	cross product, pair

semantics: formulas

elementary formulas

```
p in q        subset
p : q         same, but adds #p=1 when q is a set
```

quantified formulas

```
all x: e | F     universal
some x: e | F    existential
```

examples

```
p: A -> B        p is a relation from A to B
```

a puzzle

```
one x: e | F     some x: e | {y: e | F} = x
```

syntax: signatures

signature

- denotes a set of individuals
- fields are relations, organized by first type

```
sig Addr {}
sig Data {}
sig Memory {
  val: Addr ->! Data
  }
```

introduces

- sets Addr, Data, Memory of type ⟨Addr⟩ , ⟨Data⟩, ⟨Memory⟩
- relation val of type ⟨Memory, Addr, Data⟩

syntax: facts

fact

- an axiom, or global property

```
fact {
    all m: Memory | all a: Addr | one a.(m.val)
    }
```

syntax: functions

function

- a parameterized formula that can be invoked

```
fun Read (m: Memory, a: Addr, d: Data) {
    d = a.(m.val)
    }

fun Write (m, m': Memory, a: Addr, d: Data) {
    m'.val = m.val - (a->Data) + (a->d)
    }
```

syntax: assertions

assertion

- adds intentional redundancy
- analyzer tries to satisfy negation

```
assert {
  all m, m': Memory, a: Addr, d, d': Data |
    Write (m, m', a, d) && Read (m', a, d') => d = d'
  }
```

idioms: hierarchy

```
sig Cache {
    addrs: set Addr,
    val: addrs ->! Data
    }

sig System {
    cache: Cache,
    memory: Memory
    }
```

idioms: promotion

```
fun CacheRead (s, s': System, a: Addr, d: Data) {
    d = a.(s'.cache.val)
    if a in s.cache.addrs then s' = s else Load (s, s', a)
    }

fun Load (s, s': System, a: Addr) {
    some drop: s.cache.addrs - a {
        s'.cache.addrs = s.cache.addrs - drop + a
        s'.cache.val = s.cache.val - drop->Data +  a->a.(s.memory.val)
        Write (s.memory, s'.memory, drop, drop.(s.cache.val))
        }
    }
```

idioms: extension

subsignature

- semantically, just a subset
- fields added to supersignature too

```
sig WriteBackCache extends Cache {
  dirty: set addrs
  }
```

introduces

- set WriteBackCache of type ⟨Cache⟩
- relation dirty of type ⟨Cache, Addr⟩
  whose domain is a subset of WriteBackCache

exploiting dirty bits

```
sig WBSystem extends System {}
fact {all s: WBSystem | s.cache in WriteBackCache}

fun WBWrite (s, s': WBSystem, a: Addr, d: Data) {
    CacheWrite (s, s', a, d)
    s'.cache.dirty = s.cache.dirty + a - (s.cache.addrs - s'.cache.addrs)
    }

assert {
    all s, s': WBSystem, a: Addr, d: Data |
        WBWrite (s, s', a, d) &&
        no (s.cache.addrs - s'.cache.addrs) & c.cache.dirty
        => s'.memory.val = s.memory.val
    }
```

idioms: refinement

elements

- define Alpha maps concrete to abstract states
- check inductive assertion

```
fun Alpha (s: System, m: Memory) {
   m.val = s.memory.val - (s.cache.addrs->Data) + s.cache.val
   }

assert ReadOK {
   all s, s’: System, m,m’: Memory, a: Addr, d: Data |
      Alpha (s, m) &&
      Alpha (s’, m’) &&
      CacheRead (s, s', a, d) => Read (m, a, d) && m.val =m’.val
   }
```

idioms: traces

trace

- sequence of ticks
- each tick associated with a state

```
sig Tick {}
sig Trace {
    ticks: set Tick,
    first, last: ticks,
    next: (ticks - last) !->! (ticks - first)
    }

fact {all tr: Trace | (tr.first).*(tr.next) = tr.ticks}
```

traces, ctd

specialize traces

- state maps ticks to cache system states

```
sig CacheSystemTrace extends Trace {
  state: ticks ->! CacheSystem,
  }

fact {
  all tr : CacheSystemTrace |
    all t : tr.ticks - tr.last |
      some a: Addr, d: Data, s = t.(tr.state), s’ = t.(tr.next).(tr.state) |
        CacheRead (s, s’, a, d) || CacheWrite (s, s’, a, d)
  }
```

a temporal assertion

```
assert {
    all tr: CacheSystemTrace |
        all t: tr.ticks |
            t.(tr.state).cache.val in t.(tr.state).memory.val
    }
```

what our tool can do

find model of function or counterexample of assertion

simulation

- check consistency of state invariant
- execute an operation
  - forwards
    - CacheRead (s, s’, a, d) && a !in s.cache.addrs
  - backwards
    - CacheRead (s, s’, a, d) && d != a.(s’.memory.val)
  - sideways
    - CacheRead (s, s’, a, d) && some s.cache.addrs - s’.cache.addrs

check

- assertions about states, operations, refinements, traces

part 2: the analysis

model finding

every analysis is model finding

- execution
  - find s, s' such that op(s,s')
- checking
  - find s, s' such that ! f(s,s')

wanted

- if model reported, must be real
- if model exists, likely to find it

but

- language is undecidable …

scope

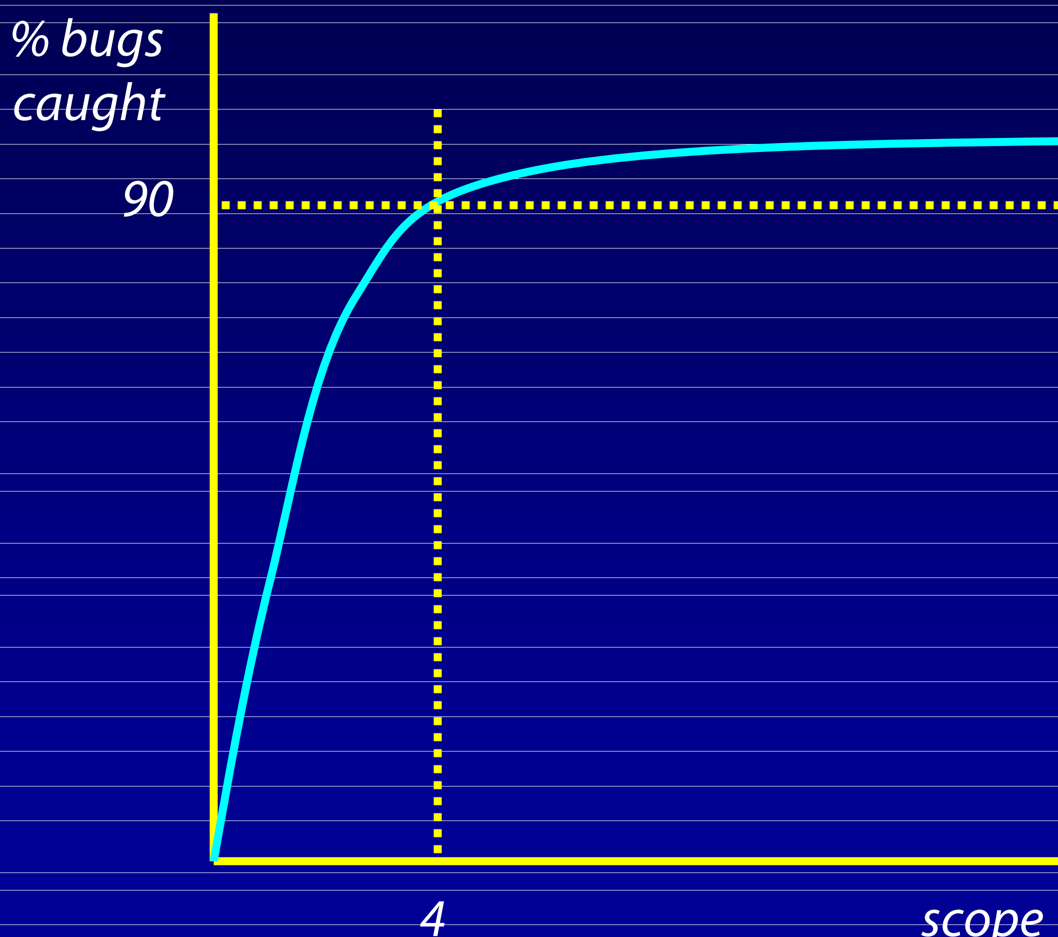
type-based bounds

- for each basic type T, pick Scope(T) $\in 1..10$
- limit search to scope

features

- comprehensive -- not testing!
  no test cases required
  3 addresses, data values => 2^24 cache system transitions
  5 network nodes => 2^25 topologies
- decouples analysis from description
  specification retains abstract form
- can focus analysis
  eg, more topologies; fewer messages; longer traces

small scope hypothesis



exploiting SAT

what you learnt in CS101

- boolean SAT first NP-complete problem
- to show problem is hard, reduce SAT to it

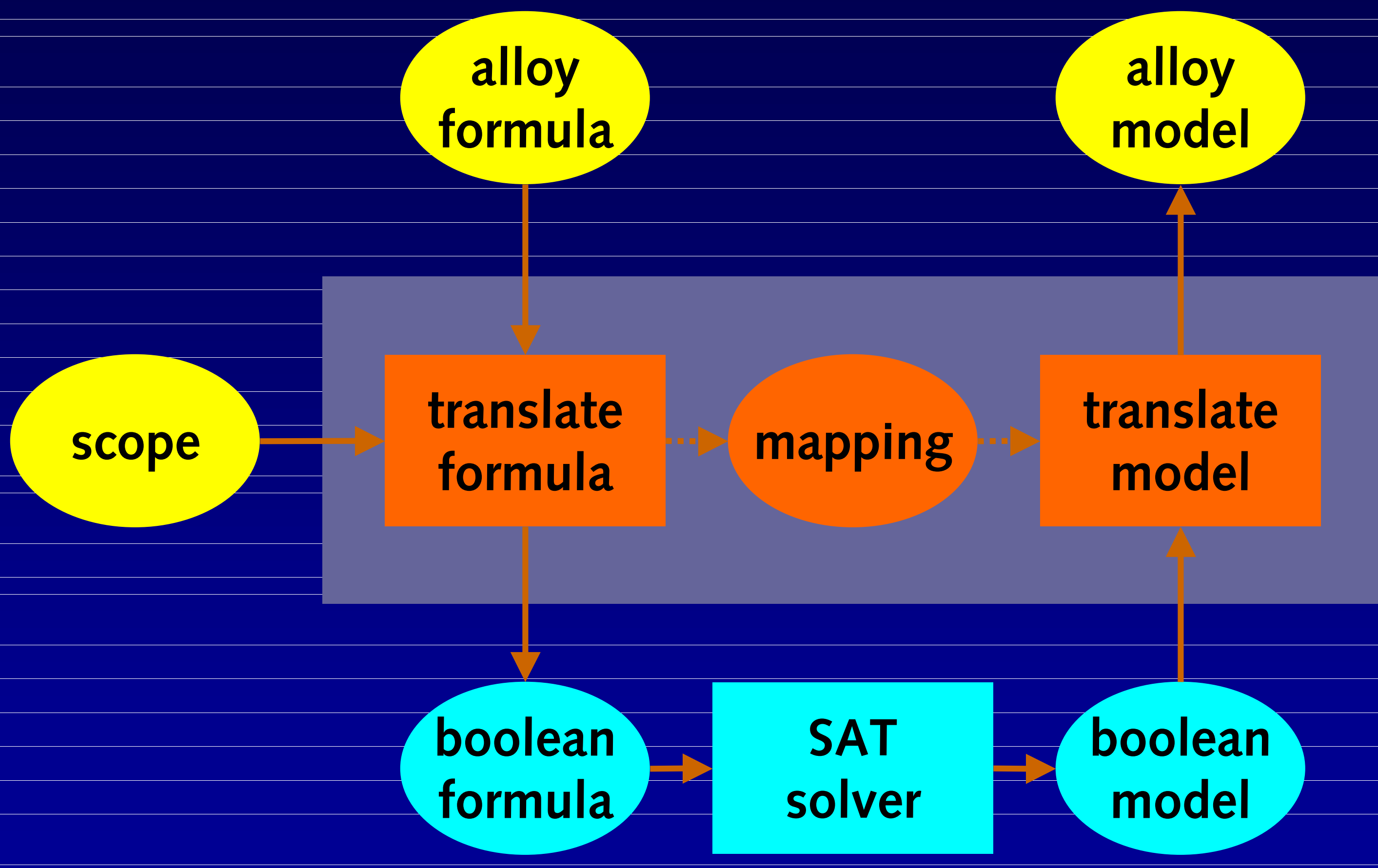
what we now know

- boolean SAT is easy
- to solve a problem, reduce it to SAT

advantages of SAT

- fine-grained goal-directedness
  fill out relations one arc at a time, short path to refutation
- exploit work of SAT community
  analyzer backend is solver-independent

analyzer architecture



translating to SAT

if a relation value	p: S -> T
is a matrix of bits	p[0,0]=1 means (S0,T0) ∈ p
· then a relation variable	
is a matrix of boolean variables	p[i,j]=1 iff (Si,Tj) ∈ p
· an expression	
· is a matrix of boolean formulas	e[i,j]=1 iff (Si,Tj) ∈ e
· a relational formula	
is a boolean formula	

example

· in scope of 2, p = ~p gives

$p[0,0] \Rightarrow p[0,0] \wedge p[0,1] \Rightarrow p[1,0]$

$\wedge\ p[1,0] \Rightarrow p[0,1] \wedge p[1,1] \Rightarrow p[1,1]$

other tricks

Shlyakhter & Kokotov

skolemization

- some r: A-> B | F
- higher-order quantifier, but turn r into a free variable

symmetry

- basic types are uninterpreted
- lots of assignments are equivalent
- add symmetry-breaking boolean formulas

improving SAT

- preprocess boolean formula
- parallelize the solver

part 3: experience

experience: INS

Khurshid (ASE 2000)

a resource discovery scheme [Balakrishnan, SOSP99]
- lookup takes attribute/value tree

we found that
- scheme did not have props claimed
- fixes to code weren't correct either
    900 lines of testing code vs. 100 lines of Alloy
- didn't satisfy monotonicity property

then we
- developed a notion of conformance
- fixed the scheme and showed it worked

experience: COM

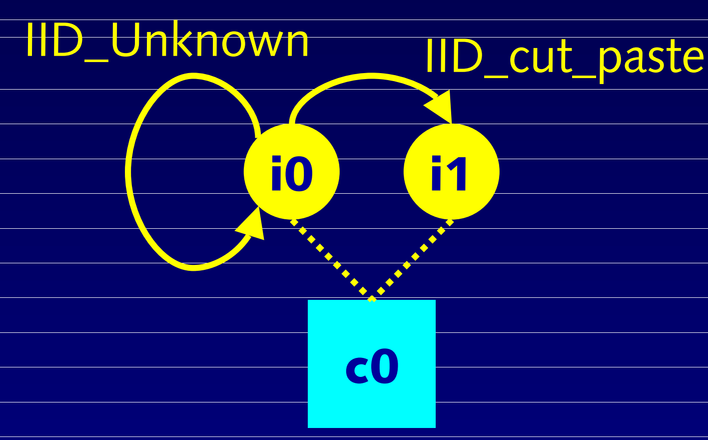
Sullivan (FSE 2000)

previous model in Z

- expressed rules of interface negotiation

what we did

- recast into Alloy
- used analyzer to check reformulations
- checked new theorems (many wrong)
- found which rules theorem required



Alloy in education

why is Alloy good for teaching?

- very few concepts to learn
- tool is easy to use
- makes abstract modelling very concrete

universities that have used Alloy in course include:

- Carnegie Mellon (Jha), Hawaii (Corbett), Irvine (Rosenblum), Kansas State (Dwyer), Michigan (Jha), Purdue (Young), Queen's (Dingel), Rochester (Lutz), Waterloo (Atlee)

ongoing projects

security domains (with BBN)

- modelling role-based access control

network topology propagation (with DERA)

- checking algorithms that propagate topology changes
- model checkers require fixed topology

air-traffic control (with NASA)

- design and analysis of new Direct-To
- parameterizing away trajectory synthesis

part 4: lessons (ie, opinions)

lessons: declarative specs

general advantages

- emphasis on properties -- often the essence
  - eg, what should restore-from-trash do?
- partiality -- replace ops by invariant
  - eg, well-formed resource database vs. spec of registration
- minimality -- of mechanism, environmental assumptions
  - eg, allow arbitrary train movements

tool-specific benefits

- masking errors
  - can conjoin negation of counterexample
- incrementality
  - can use properties in place of mechanism

lessons: executability

declarative specs are not necessarily not executable

Alloy Analyzer lets you

- execute forwards
  - op (s,s') && cond (s)
- execute backwards
  - op (s,s') && cond (s')
- execute sideways
  - op (s,s') && cond (s,s')

without

- ad hoc language restrictions
- hidden cost for declarative style
- test cases

lessons: KISS

"keep it simple, stupid"
- the biggest challenge
- new Alloy: only relations
- even Z can be simpler

researchers and UML
- walking at the back of the elephant parade?
  researchers should lead, not follow
- UML should be part of solution, not problem
- don't sanction complexity with formalism

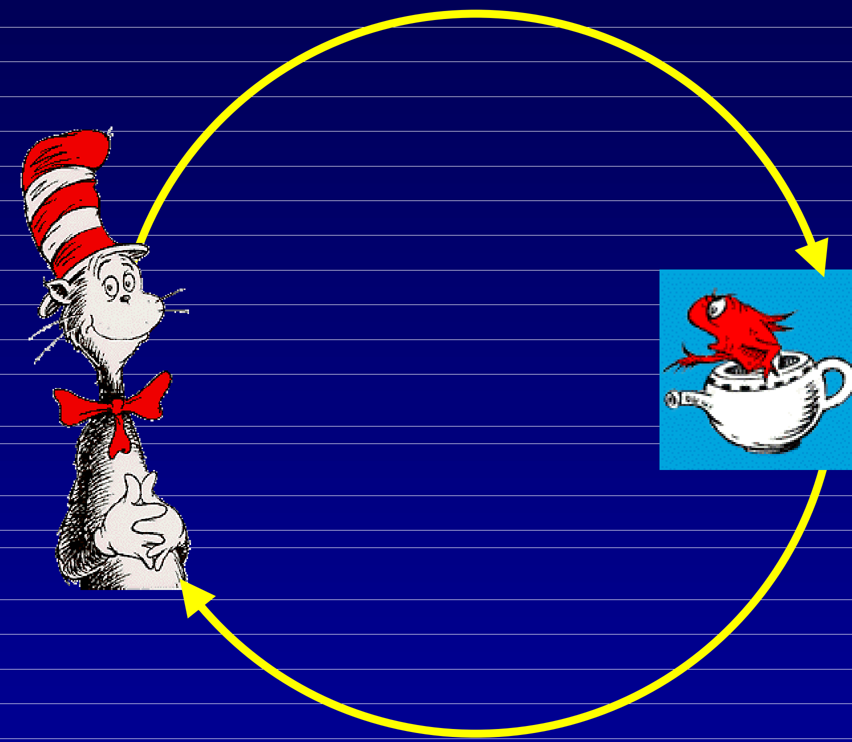
lessons: tools give meta-help too

implementing an analyzer forces

- a coherent semantics
- resolution of all details
- above all, simplicity

a procrustean bed? no!

- code mirrors language complexity
  - indexed relations
  - schema decoration



summary

let's be less apologetic to the outside

- Z is simpler & better than UML
- the force is with us (analysis, semantics)
- aim to persuade engineers, not amigos

let's be open to new ideas

- Z is not the last word
- other SAT-like analyses out there?

Alloy

- interim release available
- new tool available May 2001

Free while supplies last!
http://sdg.lcs.mit.edu/alloy