February 11, 2002
6898: Advanced Topics in Software Design
MIT Lab for Computer Science
Daniel Jackson

Alloy
Course admin

Schedule

- Presents on JML, OCL, Z
- Organizer for peer review
- Scribe for today
- Tasks

4 students present models for discussion peer review

Mon: JML, OCL, Z
Wed: peer review
Mon: holiday

First problem set out; due 2 weeks later
Wed: modelling idioms
Today: Alloy language

New version of Bill's notes online
Software Blueprints

- Captures just essence
- Clear abstract design
- Why? captures just essence
- Clear abstract design
- Software Blueprints

- Why?
  - Fewer showstopper flaws
  - Major refactoring less likely
  - Easier coding, better performance
  - Identify risky aspects
  - Develop model incrementally
  - Simulate & analyze as you go

- How?
  - What?
alloy: a new approach

- Based on new SAT technology
- Generates counterexamples to theorems
- Mouse-click automation
- Analyzer
- A logic, so declarative: incremental & implicit
- Static structures, or dynamic behaviours
- A flexible notation for describing structure
- Simulation without test cases
- Fully automatic checking of theorems
- Alloy & analyzer designed hand-in-hand

alloy: a new approach
SAT backend, new solvers every month

Flexible structuring mechanisms

Full logic with quantifiers & any-arity relations

Alloy (Jackson, Shvakhite, Sritharan, 1997-2002)

Z subset (no quantifiers), explicit search

Nipick (Jackson & Damon, 1995)

10^100 states, but low-level & for hardware

SMT model checker (CMU, 1989)

Elegant & powerful, but no automation


Roots & influences
Experience with Alloy

Georgia Tech, Queen’s, Michigan State, Imperial Colorado

CMU, Waterloo, Wisconsin, Rochester, Kansas State, Iowa, Waterloo, Wisconsin, Rochester, Kansas State, Iowa,

Twente

State, Twente

taught in courses at

Red-black tree invariants (Vaziri)
Firewire leader election (Jackson)
Classic distributed algorithms (Shyalkher)
Microsoft COM (Sullivan)
Internal Naming (Khurshid)
Access control (Wee)
Chord peer-to-peer lookup (Wee)
Applications
Elements of Alloy Project

decouples Alloy from SAT
currently Chaff & BerkMin
framework for plug-in solvers

customizable visualization
exploiting symmetry & sharing
skolemization, grounding out
scheme for translation to SAT

Flexible, clean syntax, all F.O.
Language design

typechecker
translator
visualizer
dot
SATlab
Chaff
Berkmin
RelSAT
alloy type system

Basic types
- scalars
- relational types
- a universe of atoms, partitioned into basic types.

Examples
- sets are unary relations; scalars are singleton sets
- relational type is sequence of basic types

The set of routers that is up

maps router to table
from to: {LINK, ROUTER}
ip: {ROUTER, IP}
up: {ROUTER}

the set of links to routers
maps link to ip
maps router to its Ip addr
for relations $p$ and $q$, $p \in q$ is set subset

for scalar $e$ and set $s$, $e \in s$ is set membership

$p \in q = \text{every tuple in } p \text{ is also in } q$

$p + q, p - q, p \cdot q \text{ are union, difference, intersection}$

set operators

for sets $s$ and $t$, $s \cdot t$ is cross product

$\{ b \in \mathbb{D} \mid (\exists h \in \mathbb{D} \cdot (u \cdot b \cdot \ldots \cdot d) \cap (u \cdot d)) \cap (u \cdot d) \cap (u \cdot d) \}$

product

for sets $s$ and $t$, $s \cdot t$ is cross product

$\text{[d]}q \text{ is syntactic variant of } p \cdot q$

for set $s$ and relation $r$, $s \cdot r$ is relational image

for binary relations $p$ and $q$, $p \cdot q$ is composition

$\{ b \in \mathbb{D} \mid (\exists h \in \mathbb{D} \cdot (u \cdot b \cdot \ldots \cdot d) \cap (u \cdot d)) \cap (u \cdot d) \cap (u \cdot d) \}$

join

Relational operators
module routing
-- declare sets & relations

```alloy
sig Up extends Router {
    up: (ROUTER)

    nexts: (ROUTER, ROUTER)
    table: (ROUTER, IP, LINK)
    ip: (ROUTER, IP)
    Router: (ROUTER)

    from, to: (LINK, ROUTER)
    Link: (LINK)
}
```

```
sig Lnk from, to: Router {
    from, to: Router
}
```

```
sig Ip {
    ip: IP
}
```

-- declare sets & relations
module routing
a sample network
interlude: identity etc

contstants

identity: maps each atom of type \( t \) to itself

universal: contains every tuple of type \( t \)

zero: contains no tuple of type \( t \)

identity etc

examples

\[
\text{sig Router} \{ \text{ip}: \text{IP}, \text{table}: \text{IP} \rightarrow \text{? Link}, \text{nexts}: \text{set Router} \}
\]

\[
\text{fact NoSelfLinks} \{ \text{all } r: \text{Router} | r \notin r.\text{nexts} \}
\]
alloy constraints

// router table refers only to router's links
r.table[I].from = r

// nexts are routers reachable in one step
r.nexts = r.table[I].to

// router doesn't forward to itself
no r.table[r.ip]

// ip addresses are unique
no disj r1, r2: Router | r1.ip = r2.ip

fun Consistent() {
    // table forwards on plausible link
    all r: Router, i: IP | r.table[i].to in i.~ip.*~nexts
}

{ no disj r1, r2: Router | r1.ip = r2.ip

    // ip addresses are unique
    no r.table[r.ip]

    // router doesn't forward to itself
    r.nexts = r.table[I].to

    // nexts are routers reachable in one step
    r.table[I].from = r

    // router table refers only to router's links
    all r: Router
}

tact Basics
Simulation commands

run Inconsistent for 2
{ not Consistent ()

-- show me one that doesn't

run Consistent for 2

-- show me a network that satisfies the Consistent constraint
an inconsistent state
assertions & commands

-- define forwarding operation
-- packet with destination d goes from at to at'
fun Forward (d: IP, at, at': Router) { at' = at.table[d].to }

-- assert that packet doesn't get stuck in a loop
assert Progress { all d: IP, at, at': Router | Consistent() && Forward (d, at, at') => at != at' }

-- issue command to check assertion
check Progress for 4

assert Progress
-- assert that packet doesn't get stuck in a loop

fin Forward (d: IP, at, at': Router)

-- packet with destination d goes from at to at'
-- define forwarding operation

assertions & commands
introducing mutation

sig State {nexts: Router -> Router}
-- put router connectivity here
-- state is just an action

-- one table per state

sig Router {ip: IP, table: State -> IP -> ? Link}

sig Link {from: State, to: State -> ? Router}
-- links now depend on state

introducing mutation
fun Consistent (s: State) {
    all r: Router, i: IP | (r.table[s][i].to)[s] = r.s.nexts[s].nexts
    (r.table[s][i].to)[s][r.ip] = r
}

fact state in constraints {
    no disj r1, r2: Router | r1.ip = r2.ip
    { all r: Router, s: State |
        [s][r.ip] = [r.table[s][r.ip].from][s][r.ip]
        s.nexts[r] = (r.table[s][r.ip].to)[s]
    }
}
Propagation

- easier than writing operationally
- more possibilities, better checking
- declarative spec

```
propagate (s, s': State) { all r: Router | r.table[s'] in r.table[s] + r.(s.neexts).table[s] }
```

Run Propagate (s, s': State)

- drop entries
- incorporate a neighbor's entries

... in one step, each router can
check PropagationOK for 2

{ 
  Consistent (s) \&\& Propagate (s,s') \Rightarrow Consistent (s')
| all s', s :: State |
} assert PropagationOK

does propagation work?
try again...
still broken!
language recap (1)

```plaintext
define a theorem intended to follow from the facts
{\cdots} \text{assert } A

define a constraint to be instantiated
{\cdots} \text{fun } F (\cdots) (\cdots)

introduces a global constraint
{\cdots} \text{fact}

(\forall x : X \mid \text{all } x : X \text{ in } y \text{ exists } x.f)

\langle a \text{ constraint } \forall x : X \mid \text{all } x : X \text{ in } y \text{ exists } x.f, \langle a \text{ relation } f \text{ with type } \langle T_X, T_Y \rangle \rangle \rangle

\langle a \text{ type } T_X \text{ associated with } X \langle a \text{ set } X \rangle \langle a \text{ sig } X \mid \text{all } x : X \text{ declares } \langle \forall y : Y \rangle \langle x \rangle \rangle

(1)
```
Language recap (2)

- Run F for 3 instructions analyzer to
- Find example of F
- Check A for 5 but 2 X instructions analyzer to
- Find counterexample of A
- Run F for 3 instructions analyzer to
- Using 3 atoms for each type
Other features (3)

- Arbitrary expressions in decls
- Signature extensions
- Signature extensions in imports
- Integers
- Open models/trees
- Modules
- Poly/morphism
- Fun Acyclic[t] (t: t->t) no √ t & iden[t]
- #r.table[IP] > r,fanout
models, validity & scopes

semantic elements

assignment: function from free variables to values

meaning functions $E : Expression \rightarrow Ass \rightarrow Relation$

$F : Formula \rightarrow Ass \rightarrow Bool$

examples

value: $\{(bob)\}$

likes = $\{(bob, alice), (carol, alice)\}$

person = $\{(alice)\}$

$Al\hspace{0.1em}ice = \{(alice)\}$

assignment: $\{Al\hspace{0.1em}ice.~likes\}$
Alice in p.

\[
\text{likes} = \{(\text{bob}, \text{alice}), (\text{carol}, \text{alice})\}
\]
\[
\text{Person} = \{(\text{alice}), (\text{bob}), (\text{carol})\}
\]
\[
\text{Alice} = \{(\text{alice})\}
\]

\[
\text{assignment:}
\]

\[
\text{Alice in p.} \mid \text{Person}
\]

\[
\text{formula: all p: Person = p.}
\]

\[
\text{value: true}
\]
\[
\text{likes} = \{(\text{bob}, \text{alice}), (\text{carol}, \text{alice})\}
\]
\[
\text{Person} = \{(\text{alice}), (\text{bob}), (\text{carol})\}
\]
\[
\text{Alice} = \{(\text{alice})\}
\]
\[
P = \text{p.}\}
\]

\[
\text{assignment:}
\]

\[
\text{Alice in p.} \mid \text{likes}
\]

\[
\text{value: false}
\]
Validity, satisfiability, etc

A model of negation of a theorem is a counterexample if intended to be valid, so try to show that negation is sat.

Validity \( \models \text{PROPERTY} \)

Checking assertion

\[
\begin{align*}
\text{Valid} \ (f) &= \{ \text{a in Models}(f) | \text{ass}(f) \land \text{F}[f][a] = \text{true} \} \\
\text{Satisfiable} \ (f) &= \{ \text{a in Models}(f) | \text{ass}(f) \land \text{F}[f][a] = \text{true} \} \\
\end{align*}
\]

Validity, satisfiability, etc
A scope is a function from basic types to natural numbers. Assignment a is within scope s iff

$$\#a(t) \leq s(t)$$

for basic type t. From basic types to natural numbers, a scope is a function.
what you've seen

- Concrete output
- Checking over large spaces
- Simulation, even of implicit operations
- Fully automatic analysis

- Tractable
- Expressive
- Inexpressive
- Intractable

- Expressive: no fixed idiom
- Static & dynamic constraints
- Properties in same notation
- Expressive but first-order
- Simple notation

What you've seen...
incrementality

- declarative
- operational
- no behaviours
- all behaviours
- a safety property
questions are on web page

reading operations and traces
object-oriented structure
frame conditions
mutation idioms

next time