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

Daniel Jackson

Alloy
Course admin

Schedule

- First problem set out; due 2 weeks later
- Mon: JML, OCL, Z
- Weds: peer review
- Mon: holiday
- Weds: modelling idioms
- Today: Alloy language

Tasks

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

4 students present models for discussion

Peer review

- New version of Bill's notes online
Software Blueprints

What?

Clear abstract design

Why?

Captures just essence

How?

Develop model incrementally

Identify risky aspects

Simulate & analyze as you go

Fewer showstopper flaws

Major refactoring less likely

Easier coding, better performance

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
- language & analyzer designed hand-in-hand
- alloy: a new approach
Roots & Influences

Z notation (Oxford, 1980-1992)› elegant & powerful, but no automation

SMV model checker (CMU, 1989)

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

10100 states, but low-level & for hardware

Alloy (Jackson, Shiljakhter, Strichman, 1997-2002)

Full logic with quantifiers & any-arity relations

Flexible structuring mechanisms

Elegant & powerful, but no automation

Explicit search

Z subset (no quantifiers), explicit search

Nitpick (Jackson & Damron, 1995)

SAT backend, new solvers every month

Flexible structuring mechanisms

Elegant & powerful, but no automation
experience with Alloy

Georgia Tech, Queen’s, Michigan State, Imperial Colorado State, Rochester, Wisconsin, Kansas State, Irvine, CMU, Waterloo, Twente, State, Twente, State, Twente, State, Twente

taught in courses at

Red-black tree invariants (Vazirani)
Firewire Leader election (Jackson)
Classic distributed algorithms (Shlyakhter)
Microsoft COM (Sullivan)
Intentional Naming (Khurshid)
Access control (WEE)
Chord peer-to-peer lookup (WEE)
Applications
elements of alloy project

develops 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
alloy type system

sets are unary relations; scalars are singleton sets
relational type is sequence of basic types
a universe of atoms, partitioned into basic types

basic types

ROUTER, IP, LINK

examples

the set of routers that's up
maps router to its IP addr
maps link to routers

ip: (ROUTER, IP)
from: (LINK, ROUTER)
table: (ROUTER, IP, LINK)
up: (ROUTER)
relations
for relations \( p \) and \( q \), \( p \subseteq q \) if \( q \) is set subset

for scalar \( e \) and set \( s, e \in s \) is set membership

\( p \) in \( q \) = every tuple in \( p \) is also in \( q \)

\( p + q \), \( p - q \), \( p \cap q \) = union, difference, intersection

set operators

for sets \( s \) and \( t \), \( s \prec t \) is cross product

\( \{ b \in s \mid b \subseteq (u_1 \cdots u_n)(d_1 \cdots d_m) \} = s \prec t \)

product

\( q[p] \) is syntactic variant of \( p \cdot q \)

for sets \( s \) and relation \( r \), \( s \cdot r \) is relational image

for binary relations \( p \cdot q \), \( p \cdot q \) is composition

\( \{ b = (u_1 \cdots u_n)(d_1 \cdots d_m) \mid (u_1 \cdots u_n)(d_1 \cdots d_m) \subseteq (u_1 \cdots u_n)(d_1 \cdots d_m) \} = p \cdot q \)

join

relational operators
module routing -- declare sets & relations

sig IP

sig Link {from, to: Router}

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

sig Up extends Router

{} sig Up extends Router

{} sig Router

{} sig Lnk from, to: Router

{} sig Lnk from: Link

{} sig Ip

-- declare sets & relations module routing

Alloy declarations
a sample network
interlude: identity etc

signature Router {ip: IP, table: IP ->? Link, nexts: set Router}

fact NoSelfLinks {no Router$nexts & iden [Router]}

examples

zero: contains no tuple of type t

unity: contains every tuple of type t

identity: maps each atom of type t to itself

constants

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

fun Consistent ()

{ no dist r1, r2: Router | r1.ip = r2.ip
  ip addresses are unique //
  { [r:table[r].ip]
    no r:table[r].ip
    router doesn't forward to itself //
    r.nexts = r.table[r].ip
    to
    nexts are routers reachable in one step //
    r.table[i].from = r
    router table refers only to router's links //
  } all r: Router

} 

factBasics

alloy constraints
run Inconsistent for 2
\{ \text{not Consistent()} \}
run Consistent for 2
\rightarrow show me a network that satisfies the Consistent constraint

Simulation commands
an inconsistent state
check Progress for 4

-- Issue command to check assertion

{Consistent() \&\& Forward (d', at', at) \n= at' \n= at => at' \n= at

assert Progress

assert that packet doesn't get stuck in a loop

{at' = at:table[d][to]

} (d: IP, at, at: Router)

run Forward (d: IP, at, at: Router)

-- Packet with destination d goes from at to at'

-- Define forwarding operation

assertions & commands
Lack of progress
introducing mutation

sig State {nexts: Router -> Router}
-- put router connectivity here

-- state is just an atom

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

-- one table per state

sig Link {from, to: State -> ? Router}

-- links now depend on state
fun Consistent (s: State) {
    { 
        (r.table[s][r].to)[s] in r.~ip.*~(s.nexts)
        | all r: Router, i: IP 
    } 
}

{ 
    no disjoint R1, R2: Router | R1.~ip = R2.~ip 
    { 
        { 
            no r.~ip
            [r.table[s][r].~ip]
            [s].nexts[r] = [r.table[s][r].~ip][s] 
            (r.table[s][r].~ip)[s] = r 
        } 
        all r: Router, s: State 
        } 
    }
in one step, each router can incorporate a neighbour's entries…

...
does propagation work?

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

cHECK PropagationOK for 2
no!
Check PropagationOK for 4 but 2 State

\[
\begin{align*}
&\text{\{ } \\
&\quad \forall x \text{ Propagate} (s,s') \Rightarrow \text{Consistent} (s) \\
&\quad \text{Consistent} (s) \land \text{NoTopologyChange} (s,s') \\
&\quad \forall s, s' : \text{State} \mid \\
&\text{assert PropagationOK} \text{.} \\
\end{align*}
\]

\[
\begin{align*}
&\text{\{ } \\
&\quad \text{fun NoTopologyChange} (s,s') : \text{State} \rightarrow \text{State} \\
&\quad \text{try again...} \\
\end{align*}
\]
still broken!
language recap (1)

sig X {f: Y} declares a set X

a type TX associated with X

a relation f with type TX,TY

one x.f

h

constraint (all x: X | x.f in Y && one x.f)

fact {…}

introduces a global constraint

fun F (…) {…}

assert A {…}

declares a constraint to be instantiated

declares a theorem intended to follow from the facts

fact {…}
language recap (2)

run $F$ for 3 instructs analyzer to

check $A$ for $5$ but $2 \times$ instructs analyzer to

find counterexample of $A$

using $5$ atoms for each type, but $2$ for type $TX$
other features (3)
models, validity & scopes
value: false

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

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

\{\text{Alice}\} = \text{Alice}(\text{alice})

assignment:

\text{formula: all: Person | Alice in p. likes}

value: true

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

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

\{\text{Alice}\} = \text{Alice}(\text{alice})

\{\text{(bob)}\} = \text{p}

assignment:

\text{formula: Alice in p. likes}
validity, satisfiability, etc

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

SYSTEM \subset\text{PROPERTY}

checking assertion

\begin{align*}
\text{Valid}(\varphi) & = \text{Satisfiable}(\varphi) \\
\text{Satisfiable}(\varphi) & = \{ \text{some } a : \text{Ass}(\varphi, a) \in \text{Models}(\varphi) \} \\
\text{Valid}(\varphi) & = \{ \text{all } a : \text{Ass}(\varphi, a) \in \text{Models}(\varphi) \} \\
\text{Models}(\varphi) & = \{ \text{true} \} \\
\text{Ass}(\varphi) & = \{ \text{set of all well-typed assignments for formula } \varphi \}
\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 the theorems $f$ that arise in practice, i.e., many errors can be found in small scopes, a small scope hypothesis,

$$\text{(for basic type } t, \#a(t) \leq s(t)) \Rightarrow \text{assignment } a \text{ is within scope } s$$

from basic types to natural numbers.

A scope is a function scope.
fully automatic analyses

flexible: no fixed idiom

simple notation

expressive but first-order

properties in same notation

static & dynamic constraints

simulation, even of implicit operations

checking over large spaces

concrete output

tractable

inexpressive

intractable

what you’ve seen
Incrementality
next time

questions are on web page
reading
operations and traces
object-oriented structure
frame conditions
mutation
idioms

answers to me by mail before class