alloy: an analyzable modelling language

Daniel Jackson, MIT
Ilya Shlyakhter
Manu Sridharan

Praxis Critical Systems
April 25, 2003
preaching to the choir

explicit models before code
› higher quality
› easier coding

formalism helps
› forces simplicity
› no wishful thinking
› potential for tools
assurance/cost tradeoffs

- hacking
- sketching
- write-only models
- type-checked models
- analyzed models
- proven models
lightweight formal methods

language must be
› small and simple
› well defined
› expressive enough

analysis must be
› fully automatic
› semantically deep

user doesn’t want to
› provide test cases
› invent lemmas
alloy: a structural, analyzable logic

a notation inspired by Z
› just sets and relations
› familiar logical quantifiers
› simpler, less expressive, all ASCII

an analysis inspired by SMV
› billions of cases in second
› counterexamples, not proof
› declarative logic, not state machines

Oxford, home of Z

Pittsburgh, home of SMV
what to look out for

the language
› all structure by relations
› composites by higher-arity
› entirely first order
› familiar syntax by puns

the analysis
› as in Z, everything’s a formula
› tool tries all small tests within a “scope”
› model itself is unbounded
a first alloy model

module email

sig Name, Addr ()

assert A {
    all friends, spammers: set Name, addr. Name -> Addr |
    (friends - spammers).addr = friends.addr - spammers.addr
}

check A for 3

introduces sets of atoms Name and Addr

an assertion to be checked

set difference

relational image

a command the tool executes
analysis by constraint solving

module email
sig Name, Addr {}
assert A {
    all friends, spammers: set Name, addr: Name -> Addr |
    (friends - spammers).addr = friends.addr - spammers.addr
}
check A for 3

› to check, first negate conjecture
    some f, s: set N, a: N -> A | not (f-s).a = f.a - s.a
› then skolemize away quantifiers
    not (f-s).a = f.a - s.a
› and now solve for constants
    f = {N0, N1} , s = {N1}, a = {N0->A1, N1->A1}
analysis by constraint solving

module email

sig Name, Addr {}

assert A {
    all friends, spammers: set Name, addr: Name -> Addr |
        (friends - spammers).addr = friends.addr - spammers.addr
}

check A for 3
“try all small tests”

language is undecidable
› so no sound & complete algorithm

alloy’s analysis is refutation
› look for a counterexample
› consider all assignments of values to constants
› user selects scope (here, 3 names and 3 addrs)

properties of models
› usually flawed, especially in early stages
› many bugs, even subtle ones, have small counterexamples
‘all small tests’

consequences
  › sound: no false alarms
  › incomplete: can’t prove anything
simulating an operation

module email
sig Name, Addr {}
fun add (addr, addr': Name -> Addr, n: Name, a: Addr) {
    addr' = addr x (n -> a)
}
run add for 2

decares a parameterized formula
forms a tuple
set union
alloy semantics

all values are relations
{(a),(b)} is a set
{(a)} is a scalar
{(a,b)} is a tuple

higher-order values
› can’t be represented directly
   AddrBook = \(\mathbb{P}(\mathbb{P}(\text{Name} \sqcup \text{Addr}))\)
› can often be represented with higher-arity
   AddrBook -> Name -> Addr
expressions

expressions are made from variables and
› set operators
   p + q, p - q, p & q
› relational operators
   p.q, p->q, *p, ^p, ~p

\[ [p \cdot q] = \{(p_1, \ldots, p_{n-1}, q_2, \ldots, q_m) | (p_1, \ldots, p_n) \in [p] \land (q_1, \ldots, q_m) \in [q] \land p_n = q_1\} \]

\[ p \rightarrow q = \{(p_1, \ldots, p_n, q_1, \ldots, q_m) | (p_1, \ldots, p_n) \in [p] \land (q_1, \ldots, q_m) \in [q]\} \]

puns
  for scalars a, b, sets S, T and relations p, q
  a -> b is a tuple; S -> T is a relation
  S.p is image; p.q is join
formulas

e in e’

not F

F and G

F or G

F => G

{ F G }

all x: X | F

some x: X | F

one x: X | F

sole x: X | F

no x: X | F

no e

some e

sole e

e is a subset of e’

implicit conjunction

there is exactly one x such that F

there is at most one x such that F

there is no tuple in e; e is empty

there is some tuple in e; e is non-empty

there is at most one tuple in e
fields

module email

sig Name, Addr {}

sig AddrBook {
  map: Name -> Addr
}

fun add (b, b': AddrBook, n: Name, a: Addr) {
  b'.map = b.map + n->a
}

run add

an instance
map = {AB2->N2->A2}
b = AB2, b' = AB2
a = A2, n = N2
projection (a visualization technique)

to show ternary relations
  › index the arcs
  › or project a type
some conjectures

module email

sig Name, Addr {}

sig AddrBook {map: Name -> Addr}

fun add (b, b': AddrBook, n: Name, a: Addr) {b'.map = b.map + n->a}

fun del (b, b': AddrBook, n: Name) {b'.map = b.map - n->Addr}

fun lookup (b: AddrBook, n: Name): set Addr {result = b.map[n]}

assert delUndoesAdd {all b,b',b'": AddrBook, n: Name, a: Addr | add (b,b',n,a) and del (b',b",n) => b.map = b".map }

assert addIdempotent {all b,b',b"": AddrBook, n: Name, a: Addr | add (b,b',n,a) and add (b',b",n,a) => b'.map = b".map }

assert addLocal {all b,b': AddrBook, n,n": Name, a: Addr | add (b,b',n,a) and n != n' => lookup (b,n') = lookup (b',n') }
a counterexample

```
assert delUndoesAdd {all b,b',b'": AddrBook, n: Name, a: Addr |
add (b,b',n,a) and del (b',b'',n) => b.map = b''.map }
```
subsignatures

module email

sig Target

part sig Addr, Name extends Target {}
part sig Alias, Group extends Name {}

sig AddrBook {
    map: Name -> Target
}

fun add (b, b’: AddrBook, n: Name, t: Target) {b’.map = b.map + n->t}
fun lookup (b: AddrBook, n: Name): set Addr {
    result = n.^ (b.map) & Addr }
counterexample

```assert addLocal {all b,b': AddrBook, n,n': Name, a: Addr |
  add (b,b',n,a) and n != n' => lookup (b,n') = lookup (b',n') }```
fields of subsignatures

module email
sig Host, Target {}
disj sig Name extends Target {}
disj sig Addr extends Target {host: Host}
part sig Alias, Group extends Name {}
sig AddrBook {
  map: Name -> Target
}{all a: Alias | sole map[a]}
fun getHosts (b: AddrBook, n: Name): set Hosts {
  result = n.(b.map).host
}

no ...

› partial functions, undefinedness, third logical value
› type casts
flexible declarations

module email
sig Target {}
part sig Addr, Name extends Target {}
part sig Alias, Group extends Name {}
sig AddrBook {names: set Name, map: names ->+ Target}

fun lookup (b: AddrBook, n: Name): set Addr {
    result = n.^(b.map) & Addr }
fun add (b, b': AddrBook, n: Name, a: Target) {
    a in Addr or some lookup(b,a)
    b'.map = b.map + n->a}
fun del (b, b': AddrBook, n: Name) {b'.map = b.map - n->Addr}
module email
open std/ord
sig Target {}...
sig AddrBook {names: set Name, map: names ->+ Target}
fun lookup (b: AddrBook, n: Name): set Addr {...}
fun add (b, b': AddrBook, n: Name, a: Target) {...}
fun del (b, b': AddrBook, n: Name) {...}
fun init (b: AddrBook) {no b.map}

fact traces {
 all b: AddrBook - Ord[AddrBook].last | let b' = OrdNext(b) |
 some n: Name, a: Target | add (b, b', n, a) or del (b, b', n)
 init (Ord[AddrBook].first) }

assert lookupYields {all b: AddrBook, n: b.names | some lookup(b,n)}
counterexample
what you’ve seen

language
› first-order encoding
  \[ r : A \rightarrow B \text{ looks like } r \circ \mathcal{P}(A\rightarrow B) \text{ but means } r \circ \mathcal{A}\rightarrow \mathcal{B} \]
  instead of \( \text{AddrBook} = \mathcal{P}(\mathcal{P}(\text{Name}\rightarrow \text{Addr})) \)
  define \textbf{map: AddrBook} \rightarrow \text{Name} \rightarrow \text{Addr}
› simple and uniform syntax
  navigational dot, rich declarations
  explicit parameterization

analysis
› executable and declarative
› no ad hoc constraint on language
› no test cases
not seen: modelling idioms

› schema extension
  \textbf{sig} AddrBook' \textbf{extends} AddrBook \{cache: Name -> Addr\}

› object-oriented heap
  \textbf{sig} State \{obj: Ref -> Obj\}

› asynchronous processes
  \textbf{sig} Process \{state: Time ->! State\}

› explicit events
  \textbf{sig} Event \{t: Time\}
  \textbf{sig} AddEvent \textbf{extends} Event \{n: Name, a: Addr\}
not seen: analysis idioms

› refactoring
  \[\textbf{fun} \ \text{lookup} \ (b: \text{AddrBook}, \ n: \text{Name}): \textbf{set} \ \text{Target} \ \{\ldots\}\]
  \[\textbf{fun} \ \text{lookup}' \ (b: \text{AddrBook}, \ n: \text{Name}): \textbf{set} \ \text{Target} \ \{\ldots\}\]
  \[\textbf{assert} \ \text{same} \ \{\text{all} \ b: \text{AddrBook}, \ n: \text{Name} \ | \ \text{lookup}(b,n) = \text{lookup}(b',n)\}\]

› abstraction
  \[\textbf{fun} \ \text{abstract} \ \{c: \text{ConcreteState}, \ a: \text{AbstractState}\} \ \{\ldots\}\]
  \[\textbf{fun} \ \text{opC} \ (c, \ c': \text{ConcreteState}) \ \{\ldots\}\]
  \[\textbf{fun} \ \text{opA} \ (a, \ a': \text{AbstractState}) \ \{\ldots\}\]
  \[\textbf{assert} \ \text{refines} \ \{\text{all} \ a, \ a': \text{AbstractState}, \ c, \ c': \text{ConcreteState} \ | \ \text{opC}(c,c') \ \textbf{and} \ \text{abstract}(c,a) \ \textbf{and} \ \text{abstract}(c',a') \Rightarrow \text{opA}(a,a')\}\]

› machine diameter
  \[\textbf{fun} \ \text{noRepeats} \ \{\textbf{no disj} \ b, \ b': \text{AddrBook} \ | \ b.\text{map} = b'.\text{map}\}\]
  -- when noRepeats is unsatisfiable, trace is long enough
how analyzer works

space is huge
› in scope of 5, each relation has $2^{25}$ possible values
› 10 relations gives $2^{250}$ possible assignments

SAT to the rescue
› 1971: satisfiability problem to be shown NP-complete
› 1990’s: shown to be easy in practice
› fastest solvers (Chaff, Berkmin) can handle
  thousands of boolean variables, millions of clauses

translating to SAT
› an instance is a graph
› for space of instances,
  label arcs with boolean variables
analyzer architecture

- **Scope**: Translate formula → Mapping → Translate model
- **Alloy**: Formula → Instance
- **Boolean**: Formula → SAT Solver → Instance
- **Symmetry breaking, template detection, optimizations**

**Customized Visualization**
experience: general

amazing number of flaws
  › blatant and subtle
  › in every model

good things
  › raises the bar
  › sense of confidence
  › compelling and fun

bad things
  › encourages hacking
  › over confidence
experience: design analyses

about 20 small case studies completed
  › Key management (Taghdiri)
  › Chord peer-to-peer storage (Wee)
  › Firewire leader election (Jackson)
  › Intentional Naming (Khurshid)
  › Query Interface in COM (Sullivan)
  › Unison file synchronizer (Nolte)
  › Cellular automata (Sridharan)
  › Role-based access control (Schaad et al)
  › Ideal Address Translation (Seater & Dennis)

typically
  › a few hundred lines of Alloy
  › longest analysis time: 10 mins to 1 hour
experience: education

helps teach modelling
  › abstract descriptions, concrete cases
  › very close to standard first-order logic

major part of a course
  › Imperial, U. Iowa, Kansas State

taught, usually with project
  › CMU, Waterloo, Wisconsin, Rochester, Irvine, Georgia Tech, Queen’s, Michigan State, Colorado State, Twente, WPI, USC, MIT

how long?
  › undergraduate with no formal methods background can build small models in 2 weeks
applications: code analysis

- procedure specification
- procedure source code
- unroll loops, bound heap
- alloy formula instance is execution trace
- alloy formula instance is counter trace

applied to small, complex algorithms
- Schorr-Waite garbage collection
- red-black trees

Mandana Vaziri’s doctoral thesis
applications: test case generation

why?
› easier to write invariant than test cases
› all test cases within scope give better coverage
› symmetry breaking gives good quality quite

applied to Galileo, a NASA fault tree tool
› generated about 50,000 input trees, each less than 5 nodes
› found unknown subtle flaws

Sarfraz Khurshid’s doctoral thesis
research challenges

scalability: dancing around the intractability tarpit
  › circuit minimization

overconstraint: the dark side of declarative models
  › unsat core prototype
  › highlights contradicting formulas

new type system: real subtypes
  › makes semantics fully untyped
  › still no casts, down or up
  › catches more errors, more flexible, better performance

model extraction
  › looking at how to extract models from code
for more information …

alloy.mit.edu
  › downloads for windows, unix, macintosh
  › courses, talks, case studies, papers