AN ADVANCED INTRODUCTION TO ALLOY

Daniel Jackson · ASE’07 · Atlanta · Nov 8, 2007
introduction
premises

software development needs

- simple, expressive and precise notations
- deep and automatic analyses

... especially in early stages

The first principle is that you must not fool yourself, and you are the easiest person to fool.

--Richard P. Feynman
what motivated Alloy?

an expressive, natural syntax [from object modelling]
  • classification hierarchies, multiplicities, navigations

a simple semantics [from Z, conceptual data modelling, Tarski]
  • relations and sets, behaviour as constraints

a powerful analysis [from model checking]
  • full automation, sound counterexamples
transatlantic alloy
transatlantic alloy

Oxford, home of Z
transatlantic alloy

Oxford, home of Z

Pittsburgh, home of SMV
<table>
<thead>
<tr>
<th>Feature</th>
<th>Limitation</th>
</tr>
</thead>
<tbody>
<tr>
<td>relational structures, not just trees</td>
<td>sequences are second-class citizens</td>
</tr>
<tr>
<td>first-order logic with join &amp; closure</td>
<td>limited higher-order logic</td>
</tr>
<tr>
<td>supports fully declarative specification</td>
<td>no loop construct; frame conditions</td>
</tr>
<tr>
<td>subtypes, union types, overloading</td>
<td>module system a bit clunky</td>
</tr>
<tr>
<td>symbolic model finding</td>
<td>state space limited to a few 100 bits</td>
</tr>
<tr>
<td>coverage analysis</td>
<td>experimental feature</td>
</tr>
<tr>
<td>basic arithmetic constraints</td>
<td>limited scalability</td>
</tr>
<tr>
<td>refinement checks, temporal checks</td>
<td>no temporal logic</td>
</tr>
</tbody>
</table>
what’s new in Alloy (since 2006)

Kodkod, a new engine*

unsat core: a new coverage analysis
improved visualizer: new interface, magic layout
language: better overloading, sequences, syntactic shorthands
supported API: for use of Alloy as a backend

*http://web.mit.edu/emina/www/kodkod.html
plan for today’s tutorial

Alloy by example
  • some comments on formal semantics only at the end

3 styles of modelling
  • static object modelling (cf. OCL)
  • state/operation modelling (cf. Z)
  • event modelling

essence of Alloy
  • what makes Alloy tick?
static object modelling
problem: EXIF filtering

design a scheme for

• storing EXIF data for photos
• displaying photos matching a filter
objects & classification

our first Alloy model!

**sig** Photo { file: File, tags: set Tag }
**sig** Tag { key: Key, val: Value }
**sig** File {}

**abstract sig** Key {}
**one sig** Aperture, FocalLength, ShutterSpeed **extends** Key {}

**abstract sig** Value {}
**sig** Number, String, Date **extends** Value {}

**sig**: declares set of objects

**field**: relation, association

**extends**: subset

**abstract**: subsets exhaust
sig Photo { file: File, tags: set Tag }
sig Tag { key: Key, val: Value }
sig File {}  
abstract sig Key {}
one sig Aperture, FocalLength, ShutterSpeed extends Key {}
abstract sig Value {}
sig Number, String, Date extends Value {/}
an example

example, instance: automatically generated by solver, in an arbitrary order

```
sig Photo { file: File, tags: set Tag }
sig Tag { key: Key, val: Value }
sig File {}
abstract sig Key {}
one sig Aperture, FocalLength, ShutterSpeed extends Key {}
abstract sig Value {}
sig Number, String, Date extends Value {}
run{}
```
let's say photos don't share files or tags

```
fact NoSharing {
    no disj p, p': Photo |
    p.file = p'.file or some p.tags & p'.tags
}
```

and that tags are typed

```
fact TypedTags {
    all t: Tag | t.key in Aperture implies t.val in Number
...
}
```
two ways to say no tag sharing:

\[
\begin{align*}
\text{pred NoTagShare} &\{ \\
&\quad \text{no disj } p, p': \text{Photo} | \\
&\quad \text{some } p.\text{tags} \cap p'.\text{tags} \\
\}
\end{align*}
\]

\[
\begin{align*}
\text{pred NoTagShare'} &\{ \\
&\quad \text{all } t: \text{Tag} | \#t.\sim\text{tags} > 1 \\
\}
\end{align*}
\]

ask analyzer if they’re the same:

\text{check } \{\text{NoTagShare iff NoTagShare'}\}

\begin{itemize}
\item \textbf{t.~tags:} backwards navigation
\item \textbf{#e:} cardinality
\item \textbf{check:} asks solver to generate counterexample to \textbf{assertion}
\end{itemize}
two ways to say that tags are typed:

pred TypedTag {\textbf{all} t: Tag | t.key in Aperture \textbf{implies} t.val in Number}  
pred TypedTag' { Aperture.\sim\text{key}.val in Number}

\textbf{Aperture.\sim\text{key}.value}: multistep navigation; notice no set/scalar conversions
inherited fields

basic and compound filters

abstract sig Filter {matches: set Photo}

abstract sig BasicFilter extends Filter {key: Key, range: set Value}
sig NumFilter extends BasicFilter {range in Number}
sig StringFilter extends BasicFilter {range in String}
sig DateFilter extends BasicFilter {range in Date}

abstract sig CompoundFilter extends Filter {filters: some Filter}
sig AndFilter extends CompoundFilter {}
sig OrFilter extends CompoundFilter {}

key: Key in sig decl of BasicFilter means b.key is a scalar, for every b in BasicFilter

some: multiplicity symbol

signature facts: implicitly quantified over all objects in sig, just like receivers in Java
abstract sig Filter {matches: set Photo}

abstract sig BasicFilter extends Filter {key: Key, range: set Value}
sig NumFilter extends BasicFilter {}
{range in Number}
sig StringFilter extends BasicFilter {}
{range in String}
sig DateFilter extends BasicFilter {}
{range in Date}

abstract sig CompoundFilter extends Filter {filters: some Filter}
sig AndFilter extends CompoundFilter {}
sig OrFilter extends CompoundFilter {}

all f: Filter | some f.key implies f in BasicFilter

all b: BasicFilter | some b.filters

no false alarms: and type system consistent with modelling

Warning #1
The join operation here always yields an empty set.
Left type = {this/BasicFilter}
Right type = {this/CompoundFilter-> this/Filter}
overloading resolution

```plaintext
sig Tag {
    key: Key,
    val: Value
}
abstract sig BasicFilter extends Filter {
    key: Key,
    range: set Value
}

resolvent is determined automatically from context

all disj t, t': Tag | t.~tags = t'.~tags implies t.key != t'.key

unlike Java, not limited to resolution based on argument to left of dot
```

all k: Key | some k.~key

all k: Key | some k.~key & Tag
recursive definitions

**fact** Matching {
    all f: Filter | f.matches = (
        if f in BasicFilter then
            {p: Photo | all t: p.tags | t.key = f.key \implies t.val in f.range}
        else f in AndFilter then {p: Photo | all cf: f.filters | p in cf.matches}
        else f in OrFilter then {p: Photo | some cf: f.filters | p in cf.matches}
        else none)
    )
}
assert MatchMonotonic {
  all b, b': BasicFilter |
  b.range in b'.range
  implies b.matches in b'.matches
}
check MatchMonotone for 10 but 2 Filter

**assert**: just declares assertion

**check**: instructs solver to check assertion in given ‘scope’
counterexample!

Executing "Check MatchMonotone for 10 but 2 Filter"
Solver=minisat(jni) Bitwidth=4 MaxSeq=7 Symmetry=20
6049 vars. 460 primary vars. 12956 clauses. 113ms.
Counterexample found. Assertion is invalid. 50ms.
checking a theorem, again

```plaintext
assert MatchMonotonic {
  all b, b': BasicFilter |
  b.range in b'.range and b.key = b'.key
  implies b.matches in b'.matches
}
check MatchMonotone for 10 but 2 Filter

assert: just declares assertion
check: instructs solver to check assertion in given ‘scope’
```

Executing "Check MatchMonotone for 10 but 2 Filter"
Solver=minisat(jni) Bitwidth=4 MaxSeq=7 Symmetry=20
6073 vars. 460 primary vars. 13022 clauses. 139ms.
No counterexample found. Assertion may be valid. 249ms.
state/operation modelling
simple memory

\[
\text{sig Memory} \{
\begin{align*}
\text{data: Addr} & \rightarrow \text{lone Data} \\
\end{align*}
\}
\]

\text{pred init (m: Memory) \{no m.data\}}

\text{pred write (m, m’: Memory, a: Addr, d: Data) \{}
\begin{align*}
m’.data &= m.data ++ a \rightarrow d \\
\end{align*}
\}

\text{pred read (m: Memory, a: Addr, d: Data) \{}
\begin{align*}
\text{some m.data [a] implies d = m.data [a]} \\
\end{align*}
\}

\text{predicate: a parameterized constraint}

\text{++: relational override}

\text{a->d: the tuple/relation that maps a to d}

\text{m.data: a relation}

\text{m.data[a]: [] is lookup}
sig CacheSystem {
    main, cache: Addr -> lone Data
}
pred init (c: CacheSystem) {no c.main + c.cache}
pred write (c, c': CacheSystem, a: Addr, d: Data) {
    c'.main = c.main
    c'.cache = c.cache ++ a -> d
}
pred CacheSystem.read (a: Addr, d: Data) {
    some d and d = this.cache [a]
}
load and flush

**pred** load \([c, c': \text{CacheSystem}]\) \{ 
  **some** entries: \(c\.main\) | 
  \(c'\.cache = c\.cache + \text{entries}\) 
  \(c'\.main = c\.main\) 
\} 

**pred** flush \([c, c': \text{CacheSystem}]\) \{ 
  **some** entries: \(c\.cache\) \{ 
    \(c'\.main = c\.main ++ \text{entries}\) 
    \(c'\.cache = c\.cache - \text{entries}\) 
  \} 
\} 

**higher-order**: OK if existential: 
*entries* ranges over relational subsets of **c.main**

**declarative spec**: try saying this in an operational notation
fun alpha (c: CacheSystem): Memory {
    {m: Memory | m.data = c.main ++ c.cache}
}

ReadOK: check {
    all c: CacheSystem, a: Addr, d: Data |
    c.read[a, d] implies c.alpha.read[ a, d]
} for 2 but 10 Addr, 10 Data

FlushOK: check {
    all c, c': CacheSystem |
    flush[c, c'] implies c.alpha = c'.alpha
} for 2 but 10 Addr, 10 Data

abstraction function: maps cache system to memory
refinement check: not a special feature; just finding counterexample of assertion
event modelling
hotel locking

**recodeable locks (since 1980)**
- new guest gets a different key
- lock is ‘recoded’ to new key
- last guest can no longer enter

**how does it work?**
- locks are standalone, not wired
a recodable locking scheme
a recodable locking scheme

card has two keys
if first matches lock, recode with second
a recodable locking scheme

card has two keys
if first matches lock, recode with second
a recodable locking scheme

card has two keys
if first matches lock, recode with second

if second matches, just open
a recodable locking scheme

card has two keys
if first matches lock, recode with second

if second matches, just open
module events

open util/ordering[Time] as time

sig Time {}

abstract sig Event {
  pre, post: Time
}

fact Traces {
  all t: Time - last | one e: Event | e.pre = t and e.post = t.next
}

pred Event.unchanged (field: univ -> Time) {
  field.(this.pre) = field.(this.post)
}
module hotel

open events

sig Key {}

sig Card {k1, k2: Key}
-- c.k1 is first key of card c

sig Guest {
    holds: Card -> Time
}
-- g.holds.t is set of cards g holds at time t

sig Room {
    key: Key one -> Time,
    prev: Key lone -> Time,
    occ: Guest -> Time
}
-- r.key.t is key of room r at time t

\textbf{r.s}: denotes set of objects that r maps to members of set s

\textbf{projection}: model diagram is projected over \textbf{Time}
local state

module hotel
open events

sig Key {}

sig Card {k1, k2: Key}
-- c.k1 is first key of card c

sig Guest {
    holds: Card -> Time
}
-- g.holds.t is set of cards g holds at time t

sig Room {
    key: Key one -> Time,
    prev: Key lone -> Time,
    occ: Guest -> Time
}
-- r.key.t is key of room r at time t

projection: model diagram is projected over Time

r.s: denotes set of objects that r maps to members of set s
events as objects

abstract sig Event {
    pre, post: Time
}

abstract sig HotelEvent extends Event {
    guest: Guest
}

sig Checkout extends HotelEvent {}

abstract sig RoomCardEvent extends HotelEvent {
    room: Room,
    card: Card
}

sig Checkin extends RoomCardEvent {}

abstract sig Enter extends RoomCardEvent {}

sig NormalEnter extends Enter {}

sig RecodeEnter extends Enter { }
events as objects

abstract sig Event {
  pre, post: Time
}

abstract sig HotelEvent extends Event {
  guest: Guest
}

csig Checkout extends HotelEvent {}

abstract sig RoomCardEvent extends HotelEvent {
  room: Room,
  card: Card
}

csig Checkin extends RoomCardEvent {}

abstract sig Enter extends RoomCardEvent {}

csig NormalEnter extends Enter {}

csig RecodeEnter extends Enter { }
constraining events

def sig Room {
  key: Key one --> Time
  ...
}

def abstract sig Enter extends RoomCardEvent { }
  {
    card in guest.holds.pre
  }

def sig RecodeEnter extends Enter { }
  {
    card.k1 = room.key.pre
    key.post = key.pre ++ room --> card.k2
  }

event occurrences constrained by signature facts; as in Z, no explicit preconditions
frame conditions

**sig** RecodeEnter **extends** Enter { }
{  
card.k1 = room.key.pre  
key.post = key.pre ++ room -> card.k2
}

prev.unchanged
holds.unchanged
occ.unchanged

**pred** Event.unchanged (field: univ -> Time) {  
field.(this.pre) = field.(this.post)  
}
reiter-style frame conditions

standard scheme
  · for each operation, say which state components are unchanged

Ray Reiter’s explanation closure axioms
  · if x changed, e happened

Borgida et al’s application to specs
  · no frame conditions in operations
  · instead, a global frame condition

frame conditions, Reiter-style

```plaintext
sig Room {
  key: Key one -> Time,
  prev: Key lone -> Time,
  occ: Guest -> Time
}

Checkin.modifies [prev]
(Checkin + Checkout).modifies [occ]
RecodeEnter.modifies [key]

pred modifies (es: set Event, field: univ -> Time) {
  all e: Event - es | field.(e.pre) = field.(e.post)
}
```

note that an event sig name (eg, Checkin) just denotes a set of events
is it safe?

safety condition

\* if an enter event occurs, and the room is occupied, then the guest who enters is an occupant

assert NoBadEntry {
    all e: Enter | let occs = occ.(e.pre) [e.room] |
    some occs => e.guest in occs
}

check NoBadEntry for 5
intruder!
one solution

environmental assumption

> no events intervene between checking in and entering room

```plaintext
fact NoIntervening {
  all c: Checkin - pre.last |
  some e: Enter |
    e.pre = c.post and e.room = c.room and e.guest = c.guest
}
```

Executing "Check NoBadEntry for 5"
Solver=minisat(jni) Bitwidth=4 MaxSeq=5 Symmetry=20
19163 vars. 720 primary vars. 50682 clauses. 280ms.
No counterexample found. Assertion may be valid. 591ms.
coverage

when no counterexample found
• can ask analyzer to show ‘coverage’
• highlights constraints used in proof

in this case
• only events shown: init, Checkout
  • Checkin can’t happen?
  • the culprit, also highlighted
  • omitted disj keyword
essence of Alloy
what makes Alloy tick?

all values are relations
  · not just functions and sequences, but sets and scalars too

relations are flat
  · dot is just generalized relational join

analysis is model finding
  · result of analysis is always one instance
  · visualizer’s projection turns into cartoon
relations from Z to A

- sequence
  - function
    - relation
      - set

- scalar
  - tuple
    - binding

Alloy
everything’s a relation

a **relation** is a set of tuples
   
   \{(a_0, d_0), (a_0, d_1), (a_1, d_1)\}

a **function** is a relation that’s functional
   
   \{(a_0, d_0), (a_1, d_1)\}

a **sequence** is a function over a prefix of the integers
   
   \{(0, d_0), (1, d_0), (2, d_1)\}

a **tuple** is a relation with one tuple
   
   \{(a_0, d_0)\}

a **set** is a relation containing only one-tuples
   
   \{(a_0), (a_1)\}

a **scalar** is a singleton set
   
   \{(a_0)\}
arrow product

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

when \( s \) and \( t \) are sets

\begin{itemize}
  \item \( s \rightarrow t \) is their cartesian product
  \item \( r: s \rightarrow t \) says \( r \) maps atoms in \( s \) to atoms in \( t \)
    
    (in conventional notation, Alloy says \( r \subseteq A \times B \) not \( r \in 2^{(A \times B)} \))
\end{itemize}

when \( x \) and \( y \) are scalars

\begin{itemize}
  \item \( x \rightarrow y \) is a tuple
dot join

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

\[ q \cdot r [p] = p.(q \cdot r) \]

when \( p \) and \( q \) are binary relations

\( p \cdot q \) is standard relational composition

when \( r \) is a binary relation and \( s \) is a set

\( s \cdot r \) is relational image of \( s \) under \( r \) (‘navigation’) \n
\( r \cdot s \) is relational image of \( s \) under \( \sim r \) (‘backwards navigation’)

when \( f \) is a function and \( x \) is a scalar

\( x \cdot f \) or \( f[x] \) is application of \( f \) to \( x \)
consequences

mostly preserve traditional syntax

\[ r: s \to t \]

simple semantics: no undefined terms

\[ \text{all } p: \text{Person} \mid p \neq p.\text{wife} \quad \text{-- ill-typed in OCL} \]
\[ \text{some } i: I \mid a[i] = e \quad \text{-- undefined in Z} \]

‘dereferencing’ is just join

\[ m.\text{data}[a] \quad \text{-- two applications of join} \]

simple syntax: no set/scalar conversions

\[ \text{all } p: \text{Person} \mid \text{p.name.last} \text{ in } p.\text{parents.name.last} \quad \text{-- even though } p \text{ is a scalar and } p.\text{parents} \text{ is a set} \]
small example

recall from our EXIF example

\[
\text{sig } \text{Photo } \{ \text{file: File, tags: set Tag} \}
\]

\[
\text{no disj } p, p': \text{Photo } | \ p.\text{file} = p'.\text{file} \text{ or some } p.\text{tags} & p'.\text{tags}
\]

can rewrite more uniformly, despite set/scalar difference

\[
\text{no disj } p, p': \text{Photo } | \ \text{some } p.\text{file} & p'.\text{file} \text{ or some } p.\text{tags} & p'.\text{tags}
\]

(and can actually write more easily like this)

\[
\text{file in Photo lone } \rightarrow \text{ one File}
\]

\[
\text{tags in Photo lone } \rightarrow \text{ Tag}
\]
uniform counting symbols

**formula quantifiers**

\[
\text{all } p: \text{Photo} \mid \text{one } f: \text{File} \mid p.\text{file} = f
\]

**expression quantifiers**

\[
\text{all } p: \text{Photo} \mid \text{one } p.\text{file}
\]

**multiplicity markings**

\[
\text{file in Photo} \rightarrow \text{one } \text{File}
\]
alloy architecture

alloy command

visual output

alloy front end

elaborate

typecheck

visualize

alloy formula

bounds

alloy instance

kodkod engine

translate formula

mapping

translate instance

API

SAT solver

boolean formula

boolean instance

API
cartoon is one instance

tree output

visualization, projected on Time

textual output
acknowledgments

current students & collaborators who’ve contributed to Alloy

Felix Chang
lead developer, A4

Emina Torlak
developer, Kodkod

Greg Dennis
Derek Rayside
Robert Seater
Mana Taghdiri
Jonathan Edwards
Vincent Yeung

former students who’ve contributed to Alloy

Ilya Shlyakhter
Manu Sridharan
Sarfraz Khurshid
Mandana Vaziri
Andrew Yip
Sam Daitch
Ning Song
Edmond Lau
Jesse Pavel
Ian Schechter
Li-kuo Lin
Joseph Cohen
Uriel Schafer
Arturo Arizpe
for more info

http://alloy.mit.edu
  • downloads, tutorial

http://softwareabstractions.org
  • sample chapters, model repository

http://tech.groups.yahoo.com/group/alloy-discuss/
  • discussion group, 560 members

alloy@mit.edu
  • tool support and comments

dnj@mit.edu
  • happy to hear from you!