software abstractions

Daniel Jackson · CSAIL
SDM Alumni Conference
MIT · October 18, 2006
premise #1:
abstractions are the essence of software development
what’s an abstraction

a conceptual structure
  › not a program component

discovered in the problem domain
  › eg, type face family

or invented by the designer
  › eg, drawing layer

or something in between
  › eg, spreadsheet
core abstractions: examples

Microsoft Word
- paragraph
- style
- stylesheet
- comment ...

Adobe Indesign
- frame
- text flow
- paragraph
- style ...

Omnigraffle
- object
- group
- magnet
- connector ...

CTAS
- aircraft
- flight plan
- conflict
- runway slot ...

BPTC
- patient id
- room
- beam setting
- orientation ...
brittle abstractions

typography
  › character indexing
  › font families

most word processors
  › columns, sections
  › figures and captions
  › indexing and numbering

email clients
  › message threads
  › deleted messages mailbox

web
  › location & provenance
abstractions determine quality

<table>
<thead>
<tr>
<th></th>
<th>abstractions</th>
<th>for user</th>
<th>for developer</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>good software</strong></td>
<td>robust, flexible</td>
<td>clear model</td>
<td>clean interfaces</td>
</tr>
<tr>
<td><strong>typical software</strong></td>
<td>weak, broken</td>
<td>complex model</td>
<td>messy interfaces</td>
</tr>
<tr>
<td><strong>bad software</strong></td>
<td>non-existent</td>
<td>no model</td>
<td>coupling hell</td>
</tr>
</tbody>
</table>
example: hotel locking
hotel locking

reocodable 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

from US patent 4511946; many other similar schemes

card & lock have two keys
if both match, door opens

if first card key matches second door key, door opens and lock is recoded
suppose you’re designing a hotel locking system
what are the abstractions?

<table>
<thead>
<tr>
<th>sets</th>
<th>relations</th>
<th>events</th>
</tr>
</thead>
<tbody>
<tr>
<td>Room</td>
<td>current: Room -&gt; Key</td>
<td>Checkin</td>
</tr>
<tr>
<td>Guest</td>
<td>prev: Room -&gt; Key</td>
<td>Checkout</td>
</tr>
<tr>
<td>Key</td>
<td>holds: Guest -&gt; Key</td>
<td>NormalEnter</td>
</tr>
<tr>
<td></td>
<td>occ: Room -&gt; Guest</td>
<td>RecodeEnter</td>
</tr>
<tr>
<td></td>
<td>issued: set Key</td>
<td></td>
</tr>
</tbody>
</table>
premise #2: abstractions must be articulated formally
informal & formal notations

why not English?
› wishful thinking: prose hides confusions
› ambiguity, especially when not evident

why not informal diagrams?
› often say little
› postpone hard questions

formal notations
› diagrammatic or textual
› unambiguous and precise
› amendable to machine analysis
› easier to see patterns
alloy

lightweight
› very small syntax and semantics
› a few, flexible operators

expressive
› navigation expressions
› arbitrary first order constraints

succinct
› can express partial constraints

example: nobody is his own grandpa
  \textbf{no} p: Person \mid p \textbf{in} p.(father+mother).father
warning!

complete Alloy model follows

why?
› to show you that it’s small
› to point out some salient features
keys, rooms, guests

**sig** Key {
    opens: Room
}
declares *Key*, a set of keys and *opens*, a field mapping each key to a room $k\text{.opens}$ is set of rooms that key $k$ opens

**sig** Room {
    current, prev: Key -> Time
}

**sig** Guest {
    holds: Key -> Time
}
$g\text{.holds.t}$ is set of keys held by guest $g$ at time $t$

one **sig** Desk {
    occ: Room -> Guest -> Time, issued: Key -> Time
}

**sig** Time {}
abstract hotel events

abstract sig Event {
    before, after: Time
}

declares Event, a set of abstract events
    event begins at e.before and ends at e.after

abstract sig HotelEvent extends Event {
    guest: Guest,
    room: Room
}

abstract sig KeyEvent extends HotelEvent {
    key: Key
}
enter events

**abstract sig** Enter **extends** KeyEvent {}

{  
key in guest.holds.before  
key.opens = room  
}

**sig** NormalEnter **extends** Enter {}

{  
key = room.current.before  
}

**sig** RecodeEnter **extends** Enter {}

{  
key not in room.prev.before  
prev.after = prev.before + room -> key  
current.after = current.before ++ room -> key  
}

characterizes *Enter* events by constraints:  
key is held by guest before event  
key opens room

constrains state before and after:  
before: key not previously been used  
after: key previously used  
room’s current key updated
checkin and checkout

**sig** Checkin **extends** KeyEvent {}

```
{
  holds.after = holds.before + guest->key

  no Desk.occ.before [room]

  Desk.occ.after = Desk.occ.before + room->guest

  Desk.issued.after = Desk.issued.before + key

  key not in Desk.issued.before

  key.opens = room
}
```

**pick any key not issued before that opens the room**

**sig** Checkout **extends** HotelEvent {}

```
{
  room -> guest in Desk.occ.before

  Desk.occ.after = Desk.occ.before - room->guest
}
```
from events to traces

open util/ordering[Time] as traces

pred init (t: Time) {
    no (holds + prev + current).t
    no issued.t and no occ.t
}

fact Traces {
    first[].init
    all e: Event {
        e.noChange [current] and e.noChange [prev] or e in RecodeEnter
        e.noChange [holds] and e.noChange [issued] or e in Checkin
        e.noChange [occ] or e in Checkin + Checkout
    }
    all t: Time - trace/last[] |
    one e: Event | e.before = t and e.after = t.next
}
premise #3: automatic analysis is essential
without analysis

- wishful thinking gets you
- what you write doesn’t mean what you think it means
- even if it does, the implications aren’t what you thought

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

--Richard P. Feynman
alloy’s analysis

simulation
  › find an instance
  › state, event, trace
    that satisfies an additional property

checking
  › find a counterexample
  › state, event, trace
    that violates an additional property

all analyses are in finite scope: search in bounded universe
instances and counterexamples are arbitrarily selected
simulating a configuration

```plaintext
sig Key {opens: Room}
sig Room {
    current, prev: Key -> Time
}
sig Guest {holds: Key -> Time}
sig Time {}

run {}
```

Executing "run *1*
Solver=BerkMin Bitwidth=4 Symmetry=OFF
283 vars. 102 primary vars. 358 clauses. 53ms.
Instance found: predicate is consistent. 1132ms.
simulating an event

abstract sig Event {before, after: Time}
abstract sig HotelEvent extends Event {
guest: Guest, room: Room
}

abstract sig KeyEvent extends HotelEvent
key: Key
}

abstract sig Enter extends KeyEvent {}
{
key in guest.holds.before
key.opens = room
}

sig NormalEnter extends Enter {}
{
key = room.current.before
}

run{} for 2 but 1 Event

Executing "run *1* for 2 but 1 Event"
Solver=BerkMin Bitwidth=4 Symmetry=OFF
246 vars. 63 primary vars. 371 clauses. 18ms.
Instance found: It is consistent. 1388ms.
checking safety

formulate a safety assertion:
  whenever there’s an enter event,
  the guest is registered as an occupant of the room

assert Safe {
  all e: Enter | e.guest in Desk.occ.(e.before) [e.room]
}
counterexample!

Guest checks in to Room and receives Key
Guest recode-enters Room with Key
Guest checks out
Guest reenters Room with Key
a fix

doesn’t matter if guest enters empty room

safety condition should have said:
  whenever there’s an enter event,
  if the room has registered occupants,
  the guest is one of them

assert Safe {
  all e: Enter | let occs = Desk.occ.(e.before) [e.room] | 
  some occs implies e.guest in occs
}
Guest1 checks in to Room1 and receives Key1
Guest1 immediately checks out of Room1
Guest0 checks in to Room1 and receives Key0
Guest1 recode-enters Room1 with Key1
an unrealistic fix

make guest hand back all keys at checkout:

```
sig Checkout extends HotelEvent {}
{
  room -> guest in Desk.occ.before
  Desk.occ.after = Desk.occ.before - room->guest
  holds.after = holds.before - guest->Key
}
```

Executing "check Safe2 for 5 but 2 Room, 2 Guest"
Solver=BerkMin Bitwidth=4 Symmetry=ON
7575 vars. 354 primary vars. 18731 clauses. 323ms.
No counterexample found: assertion may be valid. 4407ms.
more on analysis
analysis

automatic & deep
› shallow analyses don’t justify cost of modelling
› non-automatic analyses are impractical

problem
› logic is **undecidable**
› so sound and complete analysis does **not exist**
scope-complete analysis

observations about analyzing designs
  › most assertions are wrong
  › most flaws have small counterexamples

testing: a few cases of arbitrary size
scope-complete: all cases within small scope
how big is the space?

for hotel problem
  › space is 354 bits wide ~ \(2^{354}\) (or \(10^{107}\)) states
  › can’t solve by explicit enumeration!

Alloy Analyzer approach
  › reduces problem to SAT
  › uses off-the-shelf SAT solvers
alloy analyzer architecture

- scope
- alloy formula
- translate formula
- mapping
- alloy analyzer
- translate instance
- boolean formula
- SAT solver
- boolean instance
- alloy instance
performance: SAT solvers

size of solvable constraint in #boolean variables

from Sharad Malik
performance: moore’s law

speed of main processor offering in MHz

from intel.com
analysis opportunities

- Environment Assumptions
- Requirements
- Specification
- Testcases
- Code

- check
- generate
- verify
- test
experiences
alloy studies by MIT students

many small case studies
› intentional naming [Balakrishnan+]
› Chord peer-to-peer lookup [Kaashoek+]
› Unison file sync [Pierce+]
› distributed key management
› beam scheduling for proton therapy

typically
› 100-1000 lines of Alloy
› analysis in 10 secs - 1 hour
› 3-20 person-days of work
some alloy applications

in industry
› animating requirements (Venkatesh, Tata)
› military simulation (Hashii, Northtrop Grumman)
› role-based access control (Zao, BBN)
› generating network configurations (Narain, Telcordia)

in research
› exploring design of switching systems (Zave, AT&T)
› checking semantic web ontologies (Dong, Singapore)
› enterprise modelling (Wegmann, EPFL)
› checking refinements (Bolton, Oxford)
› security policies (Fisler, WPI and Krishnamurthi, Brown)
› Mondex electronic purse (Ramananadadro, EPS)
alloy in education

courses using Alloy at Michigan State (Laura Dillon), Imperial College (Michael Huth), National University of Singapore (Jin Song Dong), University of Iowa (Cesare Tinelli), Queen's University (Juergen Dingel), University of Waterloo (Joanne Atlee), Worcester Polytechnic (Kathi Fisler), University of Wisconsin (Somesh Jha), University of California at Irvine (David Rosenblum), Kansas State University (John Hatcliff and Matt Dwyer), University of Southern California (Nenad Medvidovic), Georgia Tech (Colin Potts), Politecnico di Milano (Carlo Ghezzi), Rochester Institute of Technology (Michael Lutz), University of Auckland (John Hamer, Jing Sun), Stevens Institute (David Naumann), Trinity College (Arthur Hughes), USC (David Wilczynski)
conclusions
how to be safe in a hotel

don’t let the bellboy open your door!
› must open it yourself
acknowledgments

current students & collaborators who’ve worked on Alloy
Greg Dennis
Derek Rayside
Robert Seater
Mana Taghdiri
Emina Torlak
Jonathan Edwards
Vincent Yeung

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

alloy.mit.edu
› downloads, papers, tutorial

alloy@mit.edu
› questions about Alloy
› send us a challenge

dnj@mit.edu
› happy to hear from you!

Software Abstractions
› MIT Press, March 2006