micromodels of software
declarative modelling and analysis with Alloy

lecture 4: a case study

Daniel Jackson
MIT Lab for Computer Science
Marktoberdorf, August 2002

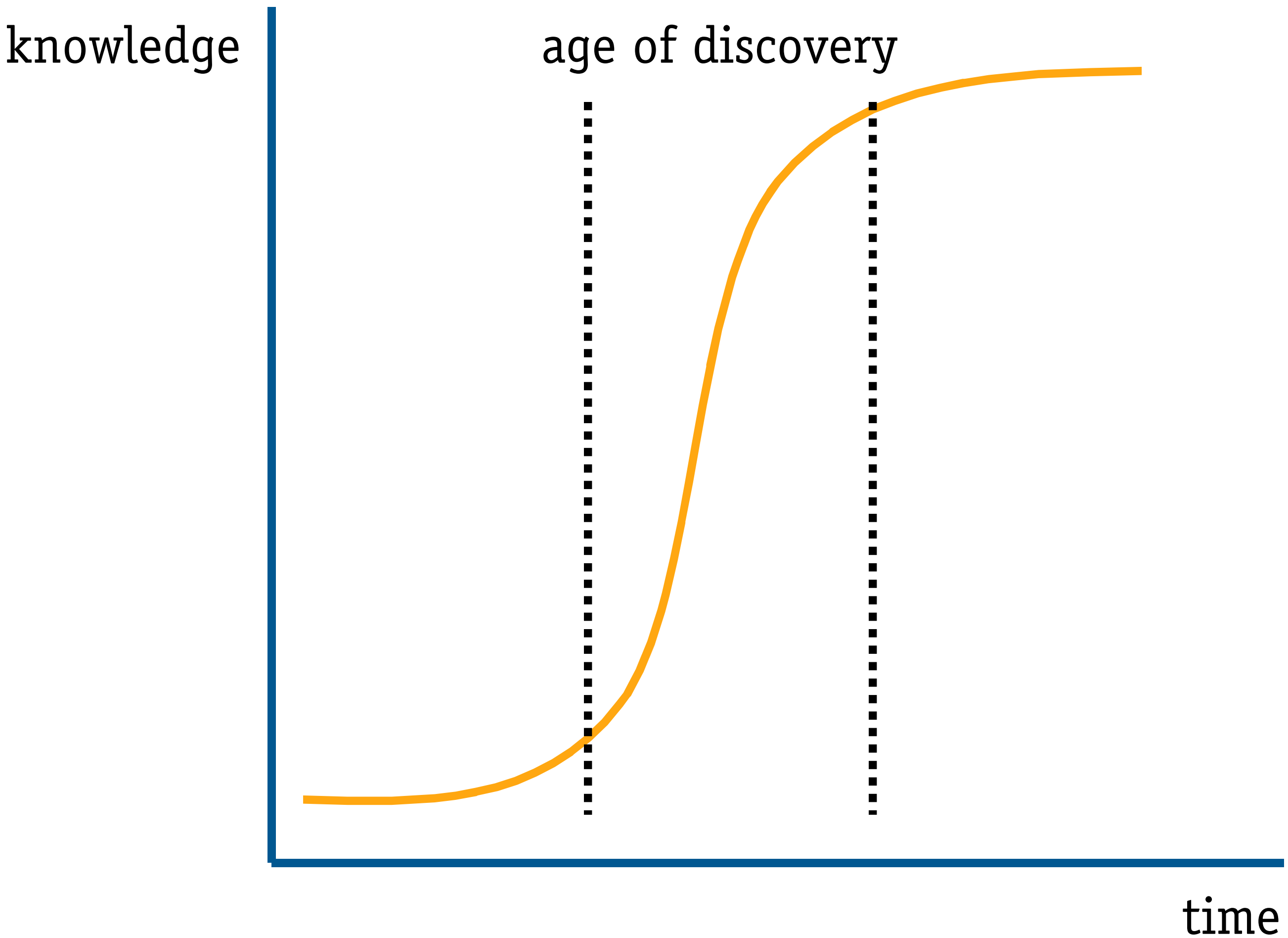
on research strategy

I have grown more and more aware that success in science … comes not so much to the most gifted, nor the most skillful, nor the most knowledgeable, but rather to the superior strategist and tactician. -- Jack Oliver

To make an important discovery, you must study an important problem. -- Peter Medawar

Know your secret weapon. -- Herb Simon

know where you are



collection of aphorisms at

theory.lcs.mit.edu/~dnj/6898/lecture-notes.html

Session 20: Hints on Research Strategy

bakery algorithm

why this example?

› curiosity -- I hadn’t done it before
› familiarity -- you can compare to Rushby
› illustrate aspects of Alloy modelling

aspects

› no commitment to fixed topology in model itself
› can easily encode traces in the logic
› both invariant reasoning & trace analysis
› can mitigate effects of finite bounds

general observations

Rushby on PVS

› nothing's easy, but everything's possible

Jackson on Alloy

› everything's easy, but nothing's possible

not quite…

› it's not always so easy
› more is possible than you might have guessed

signatures

```
module bakery
open std/ord

sig Process {}
sig Ticket {}

sig State {
  ticket: Process ->? Ticket,
  part idle, trying, critical: set Process
  }
```

process in critical phase
holds **no ticket**: hand in ticket
when you're being served

safety condition

at most one process is in the critical phase:

```
sig State {
  ticket: Process ->? Ticket,
  part idle, trying, critical: set Process
  }
fun Safe (s: State) {
  sole s.critical
  }
```

transition relation

```
fun Trans (s, s': State, p: Process) {
  let otherTickets = s.ticket[Process-p],
    next = Ord[Ticket].next |
    {
    p in s.trying
    otherTickets in s.ticket[p].^next
    p in s'.critical && no s'.ticket[p]
    }
    or ...
  }
```

precondition:
p is in trying phase
and all other tickets
follow its ticket

postcondition:
p is in critical phase
after, and holds
no ticket

other cases

```
fun Trans (s, s': State, p: Process) {
  let otherTickets = s.ticket[Process-p], next = Ord[Ticket].next |
    ...
or
    {p in s.critical
    p in s'.idle
    s'.ticket[p] = s.ticket[p]}
or
    {p in s.idle
    p in s'.trying
    some s'.ticket[p] & otherTickets.^next}
  }
```

frame condition

define a condition saying that a process *p* doesn't change:

```
fun NoChange  (s, s': State, p: Process) {
  s.ticket[p] = s'.ticket[p]
  p in s.idle => p in s'.idle
  p in s.trying => p in s'.trying
  p in s.critical => p in s'.critical
  }
```

initial condition

```
fun Init (s: State) {
  Safe (s)
  }
```

putting things together

```
fun Interleaving () {
  Init (Ord[State].first)
  all s: State - Ord[State].last, s': Ord[State].next[s] |
    some p: Process {
      Trans (s,s',p)
      all x: Process - p | NoChange(s,s',x)
      }
  }
```

use of ordering:
instantiation imposes
a total order on
the set State

allowing simultaneous actions

```
fun Simultaneity () {
  Init (Ord[State].first)
  all s: State - Ord[State].last, s': Ord[State].next[s] |
    all p: Process | Trans (s,s',p) or NoChange(s,s',p)
  }
```

checking a conjecture

```
assert InterleavingSafe {
  Interleaving () => all s: State | Safe (s)
  }

check InterleavingSafe for 4 but 2 Process
```

counterexamples…

how much assurance?

analysis within bounded scope:

```
check InterleavingSafe for 4 but 2 Process
```

2 processes … seems reasonable

- we've learned something about a real scenario

4 tickets? 4 states? … not at all reasonable

- running out of tickets is a poor approximation
- not considering all states may miss bugs

when is a trace long enough?

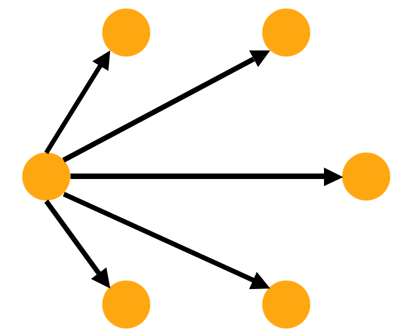
for safety properties, check all traces
- but how long? ie, what is scope of State?

idea: bound the diameter
- if all states reached in path ≤ k
- enough to consider only traces ≤ k

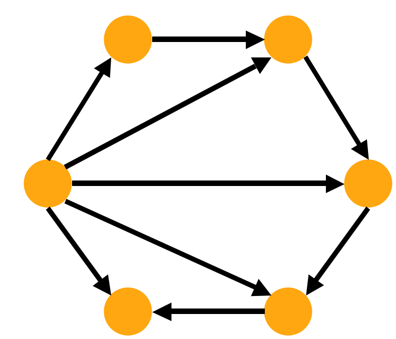
strategy
- ask for loopless trace of length k+1

  if none, then k is a bound
- tighter bounds possible: eg, no shortcuts

like bounded model checking
- but can express conditions directly



diameter = 1
max loopless = 1



diameter = 1
max loopless = 5

finding the diameter

```
fun NoRepetitionsI () {
  Interleaving ()
  no disj s, s': State | Equiv (s,s')
  }
fun Equiv (s, s': State) {
  s.ticket = s'.ticket
  s.idle = s'.idle && s.critical = s'.critical
  }
run NoRepetitionsI for 3 but 2 Process, 8 State
```

can we fix the tickets in the same way?

what we want to do

- bound the ticket scope for fast analysis
- but know that we never run out of tickets

one idea

- find diameter of machine
- ensure enough tickets for longest trace

a better idea

- ticket allocations with same process order are equivalent
- so find diameter with respect to ticket ordering
- and show not all tickets are used

defining the order

introduce process ordering as a new field

```
sig StateWithOrder extends State {
  precedes: Process -> Process
  }{
  all p, p': Process |
    p->p' in precedes iff
      ticket[p'] in ^(Ord[Ticket].next)[ticket[p]]
  }
fact {State = StateWithOrder}
```

defining state equivalence

define equivalence modulo ordering

```
fun EquivProcessOrder (s, s': State) {
  s.precedes = s'.precedes
  s.idle = s'.idle && s.critical = s'.critical
  }
```

define no repetition constraint

```
fun NoRepetitionsUnderOrderI () {
  Interleaving ()
  no disj s, s': State | EquivProcessOrder (s,s')
  }
```

finding the bounds

find a diameter

```
run NoRepetitionsUnderOrderI for 7 but 3 Process, 13 State
```

check that tickets not all used

```
assert EnoughTicketsI {
  Interleaving () => Ord[Ticket].last !in State.ticket [Process]
  }
check EnoughTicketsI for 7 but 3 Process, 12 State
```

so now we know

- for 3 processes, 12 states and 7 tickets is fully general

getting full coverage

finally, we check this

```
check InterleavingSafe for 7 but 3 Process, 12 State
```

if no counterexample

- we have a ‘proof’ for 3 processes

what we did

unbounded model of bakery
› no fixed number of processes or tickets

analysis in small finite scope
› may miss counterexamples

established diameter
› for 3 processes, 12 states and 7 tickets is enough

full analysis for bounded topology
› all scenarios for 3 processes

summary of Alloy

a simple language

› relational first-order logic
› signatures for structuring: global relations
› description is set of constraints

an effective analysis

› simulation & checking are instance-finding
› user provides scope, distinct from model
› tool reduces Alloy to SAT

applications

› a variety of case studies
› used for teaching in ~15 universities

challenges: better analysis

improving analysis

- exploiting equalities?
- eliminating irrelevant constraints?
- choosing symmetry predicates?

mitigating effects of scope

- data independence: scope of 3 enough?
- decision procedure for subset?

analyzing inconsistency

- what when no instances are found?
- might have shown

```
false => property
```

- tool might show which constraints used

challenges: applications

finding bugs in code

- extract formula from procedure

  ```
  p(s,s0,s1,…,s')
  ```

- check the conjecture

  ```
  pre(s) && p(s,s0,s1,…,s') => post(s,s')
  ```

- counterexample is trace

build veneers on Alloy

- on API, or as macro language
- eg, role-based access control
- eg, semantic web design

challenges: case studies

source code control

› model CVS at multiple levels
› is it correct?

meta modelling

› check consistency of UML metamodel
› check theorems of Unified Theory?

dynamic topology algorithms

› reverse path forwarding, eg


thank you!
dnj@mit.edu