February 13, 2002
6898: Advanced Topics in Software Design
MIT Lab for Computer Science
Daniel Jackson

tims idioms
issues

- state change
  - how do you describe change in a logic?
  - how do you modularize the change?
  - how do you control unwanted changes?
- mutation
- frame conditions
  - how do you control unwanted changes?
the relational model

non-deterministic: some pre-states have >1 post-state
partial: some pre-states have no post-states

this procedure is

example

function Find (t: Index => Elt; e: Elt): Index

a procedure that returns the index of an element in an array

distinct variables for pre- and post-state

represent as a simple formula

view behaviour as a relation on states

ideas
What does partiality mean?

Guard:

Preconditions:

Disclaimer: havoc if invoked in bad state

Precondition:

A control: won’t allow invocation in bad state
the VDM/Larch approach

\( \text{do implementability check} \)

\[ \text{post} \left[ \text{result} = e \right. \]

\[ \text{pre} \ e \in \text{Index}.\text{(t,map)} \]

\[ \text{op FInd (t: Index -> Elt; e: Elt): Index} \]

\[ \text{make precondition explicit} \]
the Z approach

Pre (Find) = true

and check properties

Pre (Find) = some s : State | Find (s, s')

compute precondition

Pre (Find) = Pre (Find (s, s'))

Preccondition is implicit

Find = [ t : Index -> Elt, e : Elt | i : Index | t[i] = e ]
the Alloy approach

{ FindPre (t, e) => FindPost (t, result, e) }

fun Find (t: Index ->? Elt, e: Elt): Index
{ e = [result] t[result] } Index

fun FindPost (t: Index ->? Elt, e: Elt): Index
{ FindPre (t: Index ->? Elt, e: Elt) e in Index t }

- can interpret as guard or precondition
- can combine in different ways
- can split pre/post into distinct functions
- roll your own!
are these constrained enough?

\[
\{ \\
\text{all } i, j : \text{ t.\text{El}t} | i \rightarrow j \text{ in Index.\text{El}t} \Rightarrow \text{t}[i] \rightarrow \text{t}[j] \text{ in Elt}\text{El}t} \\
\text{Index}_{\text{t}} = \text{Index}_{\text{t}'}
\}
\]

func Sort (t : Index \rightarrow \text{El}t) \
\{ \text{ procedure that sorts array without dup\-}
\}

\{
\text{Index}_{\text{t}'} = \#t'
\text{Index}_{\text{t}'} = \text{Index}_{\text{t}}
\}
\]

\{ \text{ procedure that removes dup\-}
\}

Frame conditions
Summary so far

- Declarative style is powerful
- Declarative style is dangerous
- Important stylistic issue, even when not subtle
- Subtle in OO programs
- Subtle for non-deterministic operations
- Important stylistic issue, even when not subtle
- Frame conditions
- Underconstrained
- Implicit preconditions
- Declarative style is dangerous
- Separation of concerns
- Spec by conjunction
- Declarative style is powerful
modularizing change

3 approaches
- object oriented
- local state machines
- global state machine

modularizing change
module routingsig IP {}
sig Link {from, to: Router} sig Router {ip: IP, table: IP ->? Link, nexts: set Router} ... Consistent () {all r: Router, i: IP | r.table[i].to in i.~ip.*~nexts}

fun inj 
(t:t) (r:t->t') {all x: t, r | sole r.x}
{
  fun Consistent () {
    fun inj (Router$ip)
    {
      [no table[ip]
       nexts = table[ip].to
      table[ip].from in this
      }
    }
  }
  sig Router {ip: IP, table: IP ->? Link, nexts: set Router}
  sig Link {from: Router, to: Router}
  fun inj (Router, IP)
  module routing

a static model of router tables
a dynamic model
fun Propagate (s, s': State) {
  let rnexts = {r, r': Router | r->s->r' in Router$nexts} |
    all r: Router | r.table[s'] = x.from[s'] x.to[s]
  in NoTopologyChange (s, s': State)
}

fun NoTopologyChange (s, s': State) {
  all x: Link { x.from[s] = x.from[s'] && x.to[s] = x.to[s']}
  in NoTopologyChange (s, s': State)
}

assert PropagationOK

{ (s, s': State) => Consistent (s', s) }

assert PropagationOK

{ [s | all r: Router | r.table[s] + r.nexts.tab[| s [s + r.nexts.tab | { let inexts = {r', r: Router | r->s->r' in Router$nexts} |
              in Propagate (s', s: State) }

Propagation
{ (ip) in
  { [ip][r].ip = nil
    nexts[r] = table[r][ip].to
    table[r][ip].from in r,
      all r : Router
    } { r : Router -> nexts: Router = r -> ? link,
        table: Router -> IP -> ? link,
        ip: Router -> IP
      } sig NetworkState extends LinkState
  }

{ from, to : link -> Router
  } sig LinkState

{} sig IP () sig Router () sig Link ()

global state version
operations for global version

```haskell
can you write `NoTopologyChange`?

```
Object oriented version

```object oriented
{
  robj: RouterRef ->? Router,
  robj: RouterRef ->? Router,
}

sig State
{
  robj: RouterRef,
  robj: LinkRef,
}

fact {inj (Router$ip)}
{
  robj: RouterRef,
  robj: LinkRef,
}{
  [no table
  [ip]
}

sig RouterRef
{
  robj: RouterRef,
  robj: LinkRef,
  nexts: set RouterRef
}

sig LinkRef
{
  robj: RouterRef,
  table: IP ->? LinkRef,
}

sig IP
{
  from: RouterRef,
  to: RouterRef,
}
```
invariants

router invariant is now recognized as ‘cross object’

\[
\text{all s: State, r: RouterRef}
\]

\[
\begin{align*}
\text{fact}\{ \\
\text{r.(s.robj).nextr = r.(s.robj).table(IP)(s.robj).from in r} \\
\text{r.(s.robj).table(IP)(s.robj).from in r} \\
\}
\end{align*}
\]
can you write NoTopologyChange?

\[
\begin{align*}
\text{run Propagate} & \ (s, \ s': \ \text{State}) \ \\
\{ \ 
\text{propagate} & \ (s', \ s) \ \text{to} \ s' \ \\
& \ . \ \text{let} \ r \ \text{in} \ \{ \ r \ \text{~} \ \\
& \ \text{let} \ r \ \text{in} \ \{ \ r \ \text{~} \ \\
& \ \text{let} \ r \ \text{in} \ \{ \ r \ \text{~} \ \\
& \ \text{let} \ r \ \text{in} \ \{ \ r \ \text{~} \ \\
\}
\end{align*}
\]

operations in O0 Version
didn't get round to discussing this

- Frame conditions
- How systematic
- Degree of modularity
- Ease of expression

how do the approaches compare?