idioms
-frame conditions

-how do you modularize the change?

-mutation

-how do you describe change in a logic?

-state change

-issues
the relational model

ideas

view behaviour as a relation on states

represent as a simple formula

non-deterministic: some pre-states have >1 post-state

partial: some pre-states have no post-states

this procedure is

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

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

example

distinct variables for pre- and post-state
What does partiality mean?

- **Guard**: won’t allow invocation in bad state
- **Precondition**: havoc if invoked in bad state

A disclaimer: havoc if invoked in bad state
the VDM/Larch approach

the VDM/Larch approach

make precondition explicit

do implementability check

\[ \forall s: \text{State} \quad \text{pre} (s) \Rightarrow \exists s': \text{State} \quad \text{post} (s',s) \]

\[ \text{op Find} (t: \text{Index} \to \text{Elt}, e: \text{Elt}) \Rightarrow \text{Index} \]

\[ \text{post t[result]} = e \]

\[ \text{pre e in Index \cdot t \cdot map} \]

\[ \text{do implementability check} \]

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

the Z approach

precondition is implicit

pre (Find) = true ??

and check properties

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

compute precondition

Find = \[ t : Index \rightarrow ? Elt, e? : Elt, i! : Index | t[i!] = e? \]

precondition is implicit
the Alloy approach

- can interpret as guard or precondition
- can combine in different ways
- can split pre/post into distinct functions
- roll your own!
Frame conditions

are these constrained enough?

\{
  all i, j : elt | i->j in Index: elt = t[i]|->t[j] in Elt
  Index' = Index
\}

Run Sort (t : Index ->? Elt)

procedure that sorts array without dupls

\{
  \#Index' = #t
  Index' = Index
\}

Run removeDupls (t : Index ->? Elt)

procedure that removes dupls
summary so far

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

3 approaches

- Global state machine
- Local state machines
- Object oriented
A static model of router tables

```prolog
run init: [t, t'] (r: r -> t)
run consistent ()
run inj : [r, t, t'] (r: r -> t)

module routing sig IP {} sig Link {from, to: Router} sig Router {ip: IP, table: IP ->? Link, nexts: set Router}

fun Consistent () { all r: Router, i: IP | r.table[i].to in i.~ip.*~nexts }

fun inj [t, t'] (r: r -> t)

fact inj (Router$ip)

{[no table[ip]
    nexts = table[ip].to
    table[ip].from in this
}

    sig Link [from: Router, to: Router]
    ()
    sig IP
}
```

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'] in r.table [s] && r.table [s'] = r.table [s] && r.x.to [s'] = r.x.to [s]
  } \[Link]
}

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

can you write NotTopologyChange?

assert PropagationOK

{ ( (consistent (s) && propagate (s, s') => consistent (s'), s', s) | all s', s: State )
  | assert PropagationOK
  }

assert Propagation

{ [ ]
  if Router.r.table [s] in r.table [s] + r~r.nexts.table [s]
  | { r, r': Router | r~r.nexts -> r' in Router$nexts
    } \[Link] in
  }

fun Propagate (s, s': State)
global state version

{ inj (ip) }
{ [table[Router][IP]]
  no table[Router][IP]
  nexts[Router] = table[Router][IP].to
  from in r
  all r : Router
    table[Router][IP].from in r
      from in r
      nexts[Router] = table[Router][IP].to
      inj (ip) }

sig NetworkState extends LinkState

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

{}
fun Consistent (s: NetworkState) {
  all r: Router, i: IP | s.table[r][i].to[s] in i.\(\text{ip}[s]\) \(\ast\) (nexts[s])
}

fun NoTopologyChange (s, s': NetworkState) {
  s'.from = s'.from
  s.to = s'.to
  s'.from = s'.from
}

fun Propagate (s, s': NetworkState) {
  all r: Router
  \{ [r], [s'.ip[r] = [s'.ip[r]]]
  s', ip[r] in s'.table[r] + [r(\text{nexts}[s]) \ast (\text{table}[s])]
  all r: Router, i: IP
  \{ [s]'(\text{table}[r][I][I] to s[I][I].to[s][I][I]) \ast [(\text{nexts}[s]) \ast (\text{ip}[s])]
  \}
}

fun Operations for Global Version

object oriented version

{ rob: RouterRef ->! Router
  rob: RouterRef ->! Router,
} sig State

{ } sig RouterRef
{ } sig LinkRef
{ } sig State
{fact (in: (Router$ip))
{ [ip] no table[ip]
  { [ip] no table[ip]
    { [ip] no table[ip]
      { [ip] no table[ip]
        { rob: RouterRef ->! Router, nexts: set RouterRef,
          rob: RouterRef ->! Router,
        }
        { rob: RouterRef ->! Router, nexts: set RouterRef,
          rob: RouterRef ->! Router,
        }
      }
    }
  }
}

{ rob: LinkRef -? Link
  rob: RouterRef -? Router,
} sig State

{ } sig RouterRef
{ } sig LinkRef
{ } sig State
{fact (in: (Router$ip))
{ [ip] no table[ip]
  { [ip] no table[ip]
    { [ip] no table[ip]
      { rob: LinkRef -? Link, table: IP, nexts: set RouterRef,
        rob: RouterRef ->! Router,
      }
    }
  }
}

object oriented version
router invariant is now recognized as ‘cross object’

\[
\text{fact} \{ \\
\text{all } s: \text{State}, r: \text{RouterRef} \{ \\
\quad r.(s.robj).table[IP](s.lobj).from \in r \\
\quad r.(s.robj).nexts = r.(s.robj).table[IP](s.lobj).to \\
\} \\
\} 
\]
operations in OO version

```haskell
run NotTopoChange (s, s' :: State)

can you write NotTopoChange?

{ let rnexts = (r, r' :: Router | r -> s -> r') in
  let
    rnexts = {r, r' :: Router | r -> s -> r'} in
  in Propagate (s, s' :: State)
}

run Consistent (s' :: Jail)

{ s' = s

run NotTopoChange (s, s' :: State)
```
didn’t get round to discussing this

how do the approaches compare?

• frame conditions
• how systematic
• degree of modularity
• ease of expression

comparison