Proving Safety of a Distributed Program

GEOFF MOORES
Way Ahead:

Program Orientation

Proof Structure + Methods

Lessons Learned + Takeaways

Code Inspection + Questions

Other Resources
To eat, a node must hold the fork. Hungry nodes (waiting to eat, without fork) send ‘R’ request and wait for fork. Thinking nodes must release fork on receipt of ‘R’.

**Desired Safety Property:** Two nodes should never ‘eat’ at the same time
Program Orientation

“Dining” Distributed Lock

Node: State, Fork, Req, Done

acquire () #eat
  if not Fork: State <- Hungry; send Req; Req <- False
  wait for Fork: State <- Eat

release ()
  state <- Thinking
  if Req: send Fork; Fork <- False

recv_msg()
  while not Done: msg <- FIFO channel
  msg = Fork: Fork <- True
  "  = Req: Req <- True ; if Thinking: send Fork; Fork <- False
  "  = End: Done <- True

Assume atomicity of these rules. All state changes in acq, rel, and recv_msg will happen without any other state changes interfering.
Proof Structure

Represent a World with two Nodes in Coq

Capture all relevant state

Define the possible state transitions via atomic rules

Identify a set of Invariant Assertions:
  - hold for every reachable state
  - imply Safety:
    - Invariant( ~ (Node A eating ∧ Node B eating) )
Proof Structure

Represent a World with two Nodes in Coq

Capture all relevant state

Define the possible state transitions via atomic rules

Identify a set of Invariant Assertions:
  o Number of forks (World) = 1
  o Node X Eating -> X has Fork
  o imply Safety:
    o Invariant( ~ (Node A eating \ Node B eating) )
Coq Methods

Definition initState (n: node) : nodeState :=
  match n with
  | a => { --> 10 } & { S --> 0 ; F --> 1 ; R --> 0 ; D --> 0 ; W --> 0 }
  | b => { --> 10 } & { S --> 0 ; F --> 0 ; R --> 1 ; D --> 0 ; W --> 0 }
end.

Inductive msg : Type :=
| req : msg
| fork : msg
| endm : msg
| nullm : msg.

Inductive input : Type :=
| acq : input
| rel : input
| endi : input
| nulli : input.

Inductive world : Type :=
Definition `processInput` (n : node)(i : input)(s : nodeState) : response :=
match (s W),(s D) with
| 1,_ => r(s, ([ ])) (* Accept / process no input if the node is waiting *)
| _,1 => r(s, ([ ])) (* Accept / process no input if the node is ended *)
| _,_ =>
match i with
| acq => match (s F) with
  | 1 => r((s & { S --> 2 }), ([ ]))
  | _ => r((s & { S --> 1 ; R --> 0 ; W --> 1 }), ( p((neighbor n),req) :: [ ] ))
end
| rel => match (s F),(s R) with
  | 0, _ => r(s, ([ ])) (* Do nothing, invalid user call (rel without fork) *)
  | 1, 1 => r((s & { F --> 0 ; S --> 0 }), ( p((neighbor n),fork) :: [ ] ))
  | 1, _ => r((s & { S --> 0 }), ([ ]))
  | _ , _ => r(s, ([ ])) (* Do nothing, invalid node state *)
end
| endi => r((s & { D --> 1 }), ( p((neighbor n),endm) :: [ ] ))
| nulli => r(s, ([ ])) (* nothing placeholder for our null input *)
end
end.
Definition `processMsg (n : node)(m : msg)(s : nodeState) : response :=` match m with
  | `req` => match `(s S),(s F)` with
    | `0,1` => `r( (s & { F --> 0 ; R --> 1 }), ( p((neighbor n),fork) :: [ ] ) )` (* comment *)
    | `_,_` => `r( (s & { R --> 1 }), ( [ ] ) )` (* optimization + proofing *)
  end
  | `fork` => `(processInput n acq (s & { F --> 1 ; W --> 0 }))`
  | `endm` => match `s D` with
    | `0` => `r((s & { D --> 1 }), ( [ ] ))`
    | `_` => `r(s, ( [ ] ))`
  end
  | `nullm` => `r(s, ( [ ] ))` (* nothing placeholder for our null message *)
end.
Coq Methods

Inductive reliable_step : world -> world -> Prop :=
| step_input : forall w i n st' ms,
  processInput n i ((localSt w) (key n)) = r(st', ms) ->
  reliable_step w
  ( W ( ((localSt w) & {(key n) --> st'}), ( ms ++ (inFlightMsgs w)),
      ( trace w ) ++ [e(n, i)] ))
| step_msg : forall w m n st' ms,
  nextMessage n (inFlightMsgs w) = (p(n,m)) ->
  processMsg n m ((localSt w) (key n)) = r(st', ms) ->
  reliable_step w
  ( W( ((localSt w) & {(key n) --> st'}), ( ms ++ ( pop (inFlightMsgs w) (p(n,m)) )),
     ( trace w ) ) ).

Definition reliable_step_star := clos_refl_trans_n1 _ reliable_step.

Definition reachable (w : world) : Prop := reliable_step_star initWorld w.
Coq Methods

Theorem bool_vars_bool : forall w, reachable w -> check_binvars_bin w bool_vars = true. (* Comment on setup of world state *)

Theorem one_fork : forall w, reachable w -> forks w = 1.

Theorem eating_imp_fork : forall w N, reachable w ->
N = A ∨ N = B ->
localSt w N S = 2 -> localSt w N F = 1.

Theorem one_eater : forall w, reachable w ->
~(localSt w A S = 2 ∧ localSt w B S = 2).
Code Inspection + Questions
Other Resources: Verdi

“framework from the University of Washington to implement and formally verify distributed systems”

http://verdi.uwplse.org/

Open source, nice blog intros

Verified System Transformers – prove safety under certain conditions, will transform an application into another which holds under a different environment.