Some Applications of the Coq Proof Assistant

Karl Palmskog

01/30/18

Languages in Coq: Regular Expressions

```
Section RegExp.

Variable char : Type.

Inductive re : Type :=
 | re_zero : re
 | re_unit : re
 | re_char (c:char)
 | re_plus (r:re) (r':re)
 | re_times (r:re) (r':re)
 | re_star (r:re).

Definition s : Type := list char.
End RegExp.
```

Specifications in Coq: Regular Expression Matching

$\boxed{s \in L(r)}$ string in regexp language

$$\frac{}{\epsilon \in L(1)}\ \text{S\_IN\_REGEXP\_LANG\_UNIT}$$

$$\frac{}{c \in L(c)}\ \text{S\_IN\_REGEXP\_LANG\_CHAR}$$

$$\frac{s \in L(r_1)}{s \in L(r_1 + r_2)}\ \text{S\_IN\_REGEXP\_LANG\_PLUS\_1}$$

$$\frac{s \in L(r_2)}{s \in L(r_1 + r_2)}\ \text{S\_IN\_REGEXP\_LANG\_PLUS\_2}$$

$$\frac{s \in L(r_1) \quad s' \in L(r_2)}{s\,s' \in L(r_1\ r_2)}\ \text{S\_IN\_REGEXP\_LANG\_TIMES}$$

$$\frac{}{\epsilon \in L(r^*)}\ \text{S\_IN\_REGEXP\_LANG\_STAR\_1}$$

$$\frac{s \in L(r) \quad s' \in L(r^*)}{s\,s' \in L(r^*)}\ \text{S\_IN\_REGEXP\_LANG\_STAR\_2}$$

Specifications in Coq: Regular Expression Matching

```
Inductive s_in_regexp_lang : s → re → Prop :=
| s_in_regexp_lang_unit : s_in_regexp_lang [] re_unit
| s_in_regexp_lang_char : forall (c:char),
   s_in_regexp_lang (c :: []) (re_char c)
| s_in_regexp_lang_plus_1 : forall (s5:s) (r1 r2:re),
   s_in_regexp_lang s5 r1 →
   s_in_regexp_lang s5 (re_plus r1 r2)
| s_in_regexp_lang_plus_2 : forall (s5:s) (r1 r2:re),
   s_in_regexp_lang s5 r2 →
   s_in_regexp_lang s5 (re_plus r1 r2)
| s_in_regexp_lang_times : forall (s5 s':s) (r1 r2:re),
   s_in_regexp_lang s5 r1 → s_in_regexp_lang s' r2 →
   s_in_regexp_lang (s5 ++ s') (re_times r1 r2)
| s_in_regexp_lang_star_1 : forall (r:re),
   s_in_regexp_lang [] (re_star r)
| s_in_regexp_lang_star_2 : forall (s5 s':s) (r:re),
   s_in_regexp_lang s5 r → s_in_regexp_lang s' (re_star r) →
   s_in_regexp_lang (s5 ++ s') (re_star r).
```

Specification Validation

How do we know this relation captures "real" regular expression matching?

- Encode the theory of regular languages and prove, e.g., pumping lemma [hard]
- Find an encoding of the theory of regular languages and prove consistency [easier]

Specifications in Coq: Regular Expression Matching

Via Doczkal and Smolka: "Regular Language Representations in the Constructive Type Theory of Coq":

```
Lemma regexp_re_in : forall (r : re char) (w : seq char),
  s_in_regexp_lang _ w r → w \in re_lang (re2regexp r).
Proof.
...
Qed.

Lemma re_regexp_in : forall (r : regexp char) (w : seq char),
  s_in_regexp_lang _ w (regexp2re r) → w \in re_lang r.
Proof.
...
Qed.
```

Other Results by Doczkal and Smolka

- DFAs, NFAs, equivalence, minimization
- regular languages, pumping lemma

```
Section LangNFA.
Variable char : finType.

Definition nfa_void : nfa char :=
{| nfa_s := ·set0 char; nfa_fin := set0; nfa_trans p a q := false |}.

Lemma nfa_void_correct: nfa_lang (nfa_void) =i pred0.
Proof.
...
Qed.

Lemma plus_nfa_void : forall N : nfa char,
  nfa_lang (nfa_plus N nfa_void) =i nfa_lang N.
Proof.
..
Qed.
End LangNFA.
```

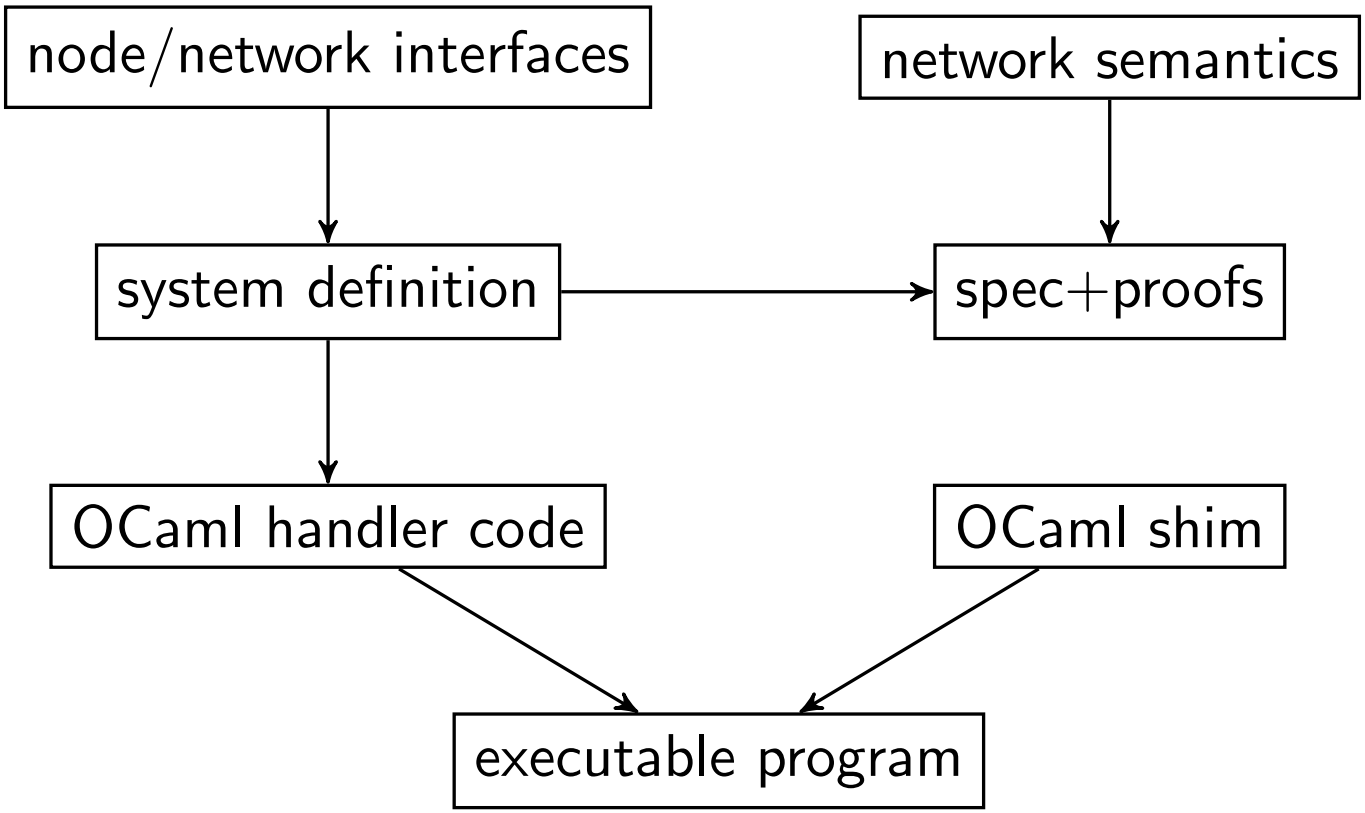
Verdi

- general Coq framework for distributed systems verification
- case study: Raft-replicated KV store [CPP '16]
- executable code via extraction to OCaml
- `https://github.com/uwplse/verdi`

What does Verdi offer?

- interfaces for (a) node data/input/output, (b) network messages and node event handlers
- lots of different operational semantics for network behavior (unordered with duplication, ordered, ...)
- helpful general lemmas (proof decomposition, etc.)
- verified system transformers (sequence numbering, Raft)
- proof automation libraries ("structural tactics")
- liveness support via coinduction and shallow LTL embedding
- several OCaml shims for extracted code (UDP or TCP)

Verdi Components



Verdi's Trusted Base

- network semantics matches physical network and chosen shim
- Coq's proof checker (small due to de Bruijn principle)
- OCaml extraction preserves meaning of Coq's Gallina language
- OCaml compiler, runtime, libraries, OS, hardware
- liveness: fair scheduling of event loop in shim

Example: Dynamic Ordered Network Configurations

(N, F, Σ, P) where:

- N: set of active node names
- F: set of failed node names
- Σ: **partial** function from names to node state
- P: function from pair of node names to message list

```
Record dynamic_ordered_network := {
 nwNodes : list name ;
 nwFailed : list name ;
 nwState : name → option data ;
 nwPackets : name → name → list msg
}.
```

Example Network Semantics: Starting a Node

$$\frac{h \notin N \qquad H_{init}(h) = \sigma \qquad P' = P +\!\!+ [(a,\ \mathsf{New}) \mid a \in N \setminus F,\ h \sim a]}{(N,\ F,\ \Sigma,\ P) \rightsquigarrow (\{h\} \cup N,\ F,\ \Sigma[h \mapsto \sigma],\ P')}\ \textsc{Start}$$

Example Network Semantics: Node Failure

$$\frac{h \in N \setminus F \qquad P' = P +\!\!+ [(a,\ \mathsf{Fail}) \mid a \in N \setminus F,\ h \sim a]}{(N,\ F,\ \Sigma,\ P) \rightsquigarrow (N,\ \{h\} \cup F,\ \Sigma,\ P')}\ \textsc{Fail}$$

Example Network Semantics: Message Delivery

$$\frac{\begin{array}{c} to \in N \setminus F \qquad P[from,\ to] = m :: ms \\ H_{net}(to,\ from,\ m,\ \Sigma[to]) = (\sigma',\ l) \\ P' = P[from,\ to \mapsto ms] +\!\!+ l \end{array}}{(N,\ F,\ \Sigma,\ P) \rightsquigarrow (N,\ F,\ \Sigma[to \mapsto \sigma'],\ P')}\ \textsc{Deliver}$$

Case Study: KV store using Raft

- Raft does strongly consistent state machine replication using leader election and logs
- Verdi approach:
  1. define and prove correct single-node KV store
  2. apply Raft transformer and get certified replicated store
- single-node KV store definition+correctness takes $\approx$ 350 LOC
- Raft transformer+correctness takes $\approx$ 50,000 LOC
- assumes crash failures/recoveries, packet drops and reordering
- only covers safety (election safety, linearization)

Node in singleton cluster never becomes leader #40

Open **tschottdorf** opened this issue 24 days ago · 5 comments

tschottdorf commented 24 days ago

I'm trying to run the benchmarks against a single-node system:

```
$ ./vard.native -dbpath /tmp/vard-8000 -port 8000 -me 0 -node 0,127.0.0.1:9000 -debug
unordered shim running setup for VarD
unordered shim ready for action
client 115512982 connected on 127.0.0.1:49446
client 115512982 disconnected: client closed socket
```

The client logged above is the following invocation:

```
python2 bench/setup.py --service vard --keys 50 --cluster 127.0.0.1:8000
Traceback (most recent call last):
  File "bench/setup.py", line 34, in <module>
    main()
  File "bench/setup.py", line 27, in main
    host, port = Client.find_leader(args.cluster)
  File "/Users/tschottdorf/tla/verdi-raft/extraction/vard/bench/vard.py", line 27, in find_leader
    raise cls.NoLeader
vard.NoLeader
```

I haven't dug deeper but I did verify that I can run the benchmarks against a three-node cluster (everything running on the same machine). So, perhaps I'm silly or there is a problem with the edge case of a single-node system.

palmskog commented 23 days ago Member

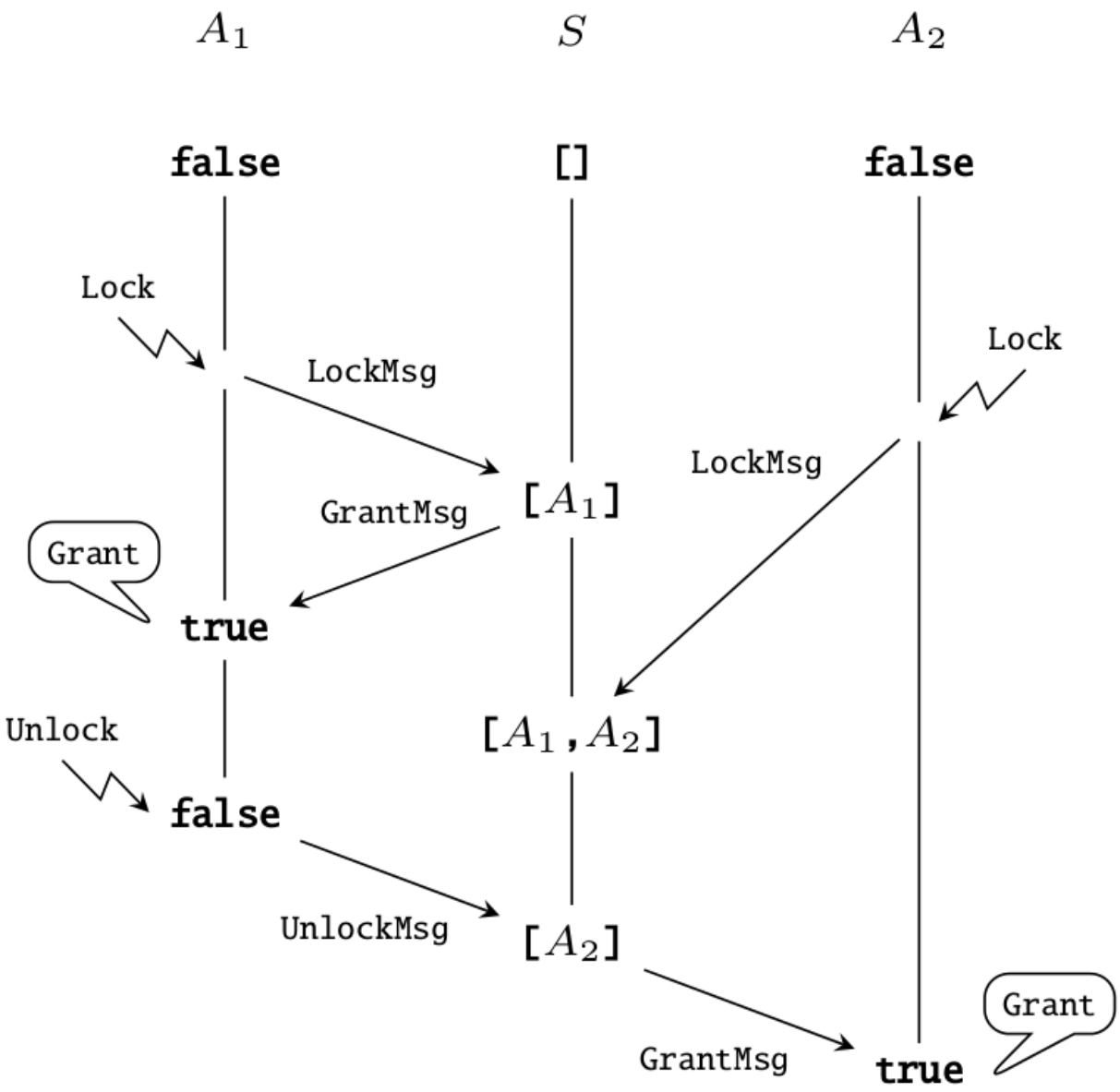
I'm pretty sure this is a liveness bug (and thus an issue outside the scope of election safety, which is guaranteed). What happens is that the singleton node never manages to elect itself leader - it waits forever for a `RequestVoteReply` message.

The `tryToBecomeLeader` function in `raft/Raft.v` is called when a timeout occurs. However, `tryToBecomeLeader` does not immediately check whether the candidate wins the vote. This is only done once a `RequestVoteReply` message is received, using a call to `wonElection`.

The original Go implementation of Raft uses a general loop for the Candidate state that first sends all necessary `RequestVote` messages and then immediately checks whether it has enough votes (and becomes leader if possible). The bug could be fixed by adding a similar check to `tryToBecomeLeader`, but I'm not sure how much that would mess with the proofs. Arguably, there is no point in running Raft in a singleton node cluster anyway - it's enough to run a system that directly uses the underlying state machine (`VarD`).

Case Study: Lock Server with Certified Mutual Exclusion

- `github.com/DistributedComponents/verdi-lockserv`
- a server node maintains a queue of agent nodes, initially empty, where the head of the queue is the agent node currently holding the lock and the rest of the queue are agents waiting to acquire the lock
- each agent maintains a boolean, initially false, which is true exactly when that agent holds the lock
- reordering OK, but no message drops or crashes


A_1
S
A_2
false
[]
false
Lock
LockMsg
Lock
LockMsg
GrantMsg
$[A_1]$
Grant
true
Unlock
$[A_1, A_2]$
false
UnlockMsg
$[A_2]$
Grant
GrantMsg
true

Lock Server Pseudocode

```
Name := Server | Agent(int)

Input := Lock | Unlock
Out := Granted

Msg := LockMsg | UnlockMsg | GrantedMsg

State Server := list Name (* head holds lock, tail waits *)

State (Agent _) := bool (* true iff client holds lock *)

InitState Server := []
InitState (Agent _) := false
```

Lock Server Pseudocode, continued

```
Definition HandleInp (n: Name) (s: State n) (inp: Inp) :=
match n with
| Server ⇒ nop (* server performs no external IO *)
| Agent _ ⇒
  match inp with
  | Lock ⇒
    send (Server, LockMsg)
  | Unlock ⇒
    if s == true then s := false ;
    send (Server, UnlockMsg)
```

Lock Server Pseudocode, continued

```
Definition HandleMsg (n: Name) (s: State n) (src: Name) (msg: Msg) :=
match n with
| Server ⇒
  match msg with
  | LockMsg ⇒
    (* if lock not held, immediately grant *)
    if s == [] then send (src, GrantedMsg) ;
    (* add requestor to end of queue *)
    s := s ++ [src]
  | UnlockMsg ⇒
    (* head of queue no longer holds lock *)
    s := tail s ;
    (* grant lock to next waiting agent, if any *)
    if s != [] then send (head s, GrantedMsg)
  | _ ⇒ nop (* never happens *)
| Agent _ ⇒
  match msg with
  | GrantedMsg ⇒
    s := true ;
    output Granted
  | _ ⇒ nop (* never happens *)
```

Lock Server Coq definitions

```
Inductive Name :=
| Client : Client_index → Name
| Server : Name.

Inductive Msg :
| Lock : Msg
| Unlock : Msg
| Locked : Msg.

Definition Input := Msg.
Definition Output := Msg.

Record Data := mkData { queue : list Client_index ; held : bool }.

Definition init_data (n : Name) : Data := mkData [] false.
```

Lock Server Coq definitions

```
Definition ServerIOHandler (m : Msg) : Handler Data := nop.

Definition ClIOHandler (i : Client_index) (m : Msg) : Handler Data :=
  d ← get ;;
  match m with
    | Lock ⇒ send (Server, Lock)
    | Unlock ⇒
      when (held d) (put {[ d with held := false ]} ;;
      send (Server, Unlock))
    | _ ⇒ nop
  end.

Definition ClNetHandler (i : Client_index) (m : Msg) : Handler Data :=
  d ← get ;;
  match m with
    | Locked ⇒
      put {[ d with held := true ]} ;;
      write_output Locked
    | _ ⇒ nop
  end.
```

Lock Server Coq definitions

```
Definition ServerNetHandler (src : Name) (m : Msg) : Handler Data :=
  d ← get ;;
  match m with
    | Lock ⇒
      match src with
        | Server ⇒ nop
        | Client c ⇒
          when (null (queue d)) (send (src, Locked)) ;;
          put {[ d with queue := queue d ++ [c] ]}
      end
    | Unlock ⇒
      match queue d with
        | _ :: c :: xs ⇒
          put {[ d with queue := c :: xs ]} ;;
          send (Client c, Locked)
        | _ ⇒
          put {[ d with queue := [] ]}
      end
    | _ ⇒ nop
  end.
```

Lock Server Coq definitions

```
Instance LockServ_BaseParams : BaseParams :=
  {
    data := Data ; input := Input ; output := Output
  }.
Instance LockServ_MultiParams : MultiParams LockServ_BaseParams :=
  {
    name := Name ;
    msg := Msg ;
    msg_eq_dec := Msg_eq_dec ;
    name_eq_dec := Name_eq_dec ;
    nodes := Nodes ;
    all_names_nodes := In_n_Nodes ;
    no_dup_nodes := nodup ;
    init_handlers := init_data ;
    net_handlers := fun dst src msg s ⇒
      runGenHandler_ignore s (NetHandler dst src msg) ;
    input_handlers := fun nm msg s ⇒
      runGenHandler_ignore s (InputHandler nm msg)
  }.
```

Lock Server Correctness in Coq

```
Definition mutual_exclusion (sigma : name → data) : Prop :=
  forall m n, held (sigma (Client m)) = true →
        held (sigma (Client n)) = true → m = n.

Theorem LockServ_mutual_exclusion : forall net tr,
  step_async_star step_async_init net tr →
  mutual_exclusion (nwState net).
Proof.
 ...
Qed.

Lemma LockServ_mutual_exclusion_trace : forall net tr,
step_async_star step_async_init net tr →
trace_mutual_exclusion tr ∧
(forall n, last_holder tr = Some n → held (nwState net (Client n)) = t
(forall n, held (nwState net (Client n)) = true → last_holder tr = Som
Proof.
 ...
Qed.
```

Lock Server OCaml Glue Code

```
module type Params = sig val debug : bool val num_clients : int end
module LockServArrangement (P : Params) = struct
  type name = LockServ.name
  type state = LockServ.data0
  type input = LockServ.msg
  type output = LockServ.msg
  type msg = LockServ.msg
  type res = (output list * state) * ((name * msg) list)
  let systemName = "Lock Server"
  let serializeName = LockServSerialization.serializeName
  let deserializeName = LockServSerialization.deserializeName

  let init = fun n ->
   let open LockServ in
   Obj.magic ((lockServ_MultiParams P.num_clients).init_handlers (Obj.magic n))

  let handleNet = fun dst src m s ->
   let open LockServ in
   Obj.magic ((lockServ_MultiParams P.num_clients).net_handlers (Obj.magic dst)
     (Obj.magic src) (Obj.magic m) (Obj.magic s))
...
```