Chapter 1

Library relayserver

Author : Frederic Peschanski – LIP6 – Sorbonne University

1.1 Introduction

This is a (for now) partial "port" of TLA+ relayserver example in Coq... It is not to say that "we should use coq" but it is the tool I am most familiar with, and my idea is to see if the limitation you encounter with TLA+ can be lifted using Coq... For now I am trying to find a proper way to model the same thing, overally...

You are constructing a state-machine, and the formal method I am most familiar with when it comes to state machines is Event-B ... However I don't want to use Event-B because like TLA+ it is untyped, it has strong expressivity limitations, etc. I have already implemented the main ideas of Event-B, this is my inspiration for formulating the TLA+ spec in Coq.

In Coq there is no methodological support. It is very difficult to find limitations in terms of expressivity (you have all mathematics reachable with type theory) but then you lose the underlying methodology and many automation tools (but Coq has also support for automation, cf. the one liner "firstorder" proofs below).

In a way Coq is a general purpose "specification language" (like general purpose programming language) whereas TLA+ (and EventB) are domain-specific languages (something like SQL ? ;-)

1.2 Basic definitions

```
Require Import Nat. (*  use Nat for rounds ? *)
Require Import subset.
```

These are just basic enumerations (sum types)

```
Inductive ServerState : Set :=
```

```
| init : ServerState
| running : ServerState.

Inductive PartyState : Set :=
| idle : PartyState
| ready : PartyState
| assigned : PartyState
| aborted : PartyState.
```

The following is not really needed but then a party record has something in it

```
Variable IDENT : Set.

Record Party : Set :=
  mkParty {
       id : IDENT;
       state : PartyState
    }.

Definition changePartyState (p : Party) (st : PartyState) :=
  mkParty p.(id) st.
```

Message contents can be formalized, I used nats for the rounds but a dedicated type can be defined if required

```
Definition ROUND := nat.

Inductive Message : Set :=
  abort | start : Message
  | readyMsg : Party → Message
  | assign : Party → Message
  | abortReq : Party → Message
  | P2P : Party → Party → ROUND → Message
  | relayP2P : Party → Party → ROUND → Message
  | broadcast : Party → ROUND → Message
  | relayBroadcast : Party → ROUND → Message.
```

Modeling sets in type theory and Coq is not trivial, here I use the "Subtype" approach which defines a set as a function from a type to propositions, i.e. for a set S of elements of type T a value e of type T is element of S iff (S e) holds.

This is typed set theory (better than set theory IMHO ;-)

If we really want to model finite sets (e.g. to reason on cardinality) then we must proceed differently...

I wrote a minimal set library to support this, it is very easy to understand (cf. subset.v in the archive).

The following in the main definition of the system state.

```
Record System : Type :=
  mkSys {
       parties : SET Party;
```

```
    serverState : ServerState;
    network : SET Message;
  }.
```

Remark: the system record has no invariant, but it is a common (and encouraged) practice to define the constraint such that the entity is consistent.

Suggestion : we should discuss invariant(s) ;-).

1.3 Init Event

In the EventB terminology the description of a possible transition in the state machine is an "event"... It is defined by preconditions (or guards, there is a distinction if we are interested in refinement) and the event itself... Sometimes events are non-deterministic but this is not the case in this development (this is a good thing).

```
Definition InitPartyPrecond (p : Party) :=
  p.(state) = idle.

Definition InitSysPrecond (parties : Party → Prop) :=
  ∀ p : Party, p ∈ parties → InitPartyPrecond p.
```

Here we can define "the" initial state of the system in a deterministic way (i.e. as a function). This is often called an action on the pre-state.

We could just define a postcondition relating the pre-state with the post-state (cf. below)

```
Definition InitSysEvent (parties : Party → Prop) :=
  mkSys parties init ∅.
```

An example lemma.

Assuming the precondition, all the parties in the initialized system are idle

It's obvious and Coq knows it so the "auto" tactic works perfectly... This kind of easy lemma can be useful in further proofs...

```
Lemma InitSys_all_parties_idle:
  ∀ ps : SET Party,
    InitSysPrecond ps
    → let sys := InitSysEvent ps
      in
      ∀ p : Party, p ∈ (sys.(parties)) → p.(state) = idle.
Proof.
  auto.
Qed.

Lemma InitSys_no_message:
  ∀ ps : SET Party,
    ∀ m : Message, m ∉ ((InitSysEvent ps).(network)).
Proof.
  auto.
Qed.
```

1.4 PartyReady event

```
Definition PartyReadyPrecond (s : System) (p : Party):=
  s.(serverState) = init
  ∧ p ∈ (s.(parties))
  ∧ p.(state) ≠ ready.

Definition setPartyState (parties : SET Party) (p : Party) (st : PartyState) :=
  swap p (changePartyState p st) parties.

(*  This covers many UNCHANGED constraints *)
Lemma UnchangedOtherParties:
  ∀ parties : SET Party, ∀ p p' : Party, ∀ st : PartyState,
    p' ≠ p → p' ∈ parties → p' ∈ (setPartyState parties p st).
Proof.
  firstorder.
Qed.

Definition PartyReadyEvent (s : System) (p : Party) :=
  mkSys (setPartyState s.(parties) p ready)
        init
        ((readyMsg p) # (s.(network))).

(*  This is an examples of an UNCHANGED requireent
 and it is not an axiom, it is a proof  *)
Lemma partyReadyOtherParties:
  ∀ sys : System, ∀ p : Party,
      PartyReadyPrecond sys p
      → let sys' := PartyReadyEvent sys p
          in ∀ p' : Party,
               p' ∈ (sys.(parties))
               → p' ≠ p
               → p' ∈ (sys'.(parties)).
Proof.
  firstorder. (*  one liner ! *)
Qed.
```

1.5 Assign event

```
Definition AssignPrecond (sys : System) (p : Party) :=
  sys.(serverState) = init
  ∧ p ∈ (sys.(parties))
  ∧ (readyMsg p) ∈ (sys.(network)) (*  Question: this message is not consumed ? *)
```

```
  ∧ (assign p) ∉ (sys.(network)).
Definition AssignEvent (sys : System) (p : Party) :=
  mkSys (setPartyState sys.(parties) p assigned)
        init
        ((assign p) # (sys.(network))).
```

1.6 Start event

```
Definition StartPrecond (sys : System) :=
  sys.(serverState) = init
  ∧ ∀ p : Party, p ∈ (sys.(parties)) → p.(state) = assigned.
Definition StartEvent (sys : System) :=
  mkSys (sys.(parties))
        running
        (start # (sys.(network))).
```

1.7 PartyAbort event

```
Definition PartyAbortPrecond (sys : System) (p : Party) :=
  sys.(serverState) = running
  ∧ p ∈ (sys.(parties))
  ∧ p.(state) = assigned
  ∧ (abortReq p) ∈ (sys.(network)).
Definition PartyAbortEvent (sys : System) (p : Party) :=
  mkSys (setPartyState sys.(parties) p aborted)
        init
        { abort }.
```

1.8 Abort event

```
Definition AbortPrecond (sys : System) :=
  sys.(serverState) = init
  ∧ abort ∈ (sys.(network)).
Definition AbortEvent (sys : System) :=
  mkSys ∅ init ∅.
```

1.9 Event ReqToBroadcast

```
Definition ReqToBroadcastPrecond (sys : System) (p : Party) (r : ROUND) :=
  sys.(serverState) = running
  ∧ p ∈ (sys.(parties))
  ∧ (∀ p : Party, p.(state) = assigned)
  ∧ (broadcast p r) ∉ (sys.(network)).

Definition ReqToBroadcastEvent (sys : System) (p : Party) (r : ROUND) :=
  mkSys (sys.(parties)) (sys.(serverState))
        ((broadcast p r) # (sys.(network))).
```

1.10 Event RelayBroadcast

```
Definition RelayBroadcastPrecond (sys : System) (p : Party) (r : ROUND) :=
  sys.(serverState) = running
  ∧ p ∈ (sys.(parties))
  ∧ (∀ p : Party, p.(state) = assigned)
  ∧ (broadcast p r) ∈ (sys.(network))
  ∧ (relayBroadcast p r) ∉ (sys.(network)).

Definition RelayBroadcastEvent (sys : System) (p : Party) (r : ROUND) :=
  mkSys (sys.(parties)) (sys.(serverState))
        ((relayBroadcast p r) # (sys.(network))).
```

1.11 Event ReqToP2P

```
Definition ReqToP2PPrecond (sys : System) (p1 p2 : Party) (r : ROUND) :=
  sys.(serverState) = running
  ∧ p1 ∈ (sys.(parties))
  ∧ p2 ∈ (sys.(parties))
  ∧ (∀ p : Party, p.(state) = assigned)
  ∧ (P2P p1 p2 r) ∉ (sys.(network)).

Definition ReqToP2PEvent (sys : System) (p1 p2 : Party) (r : ROUND) :=
  mkSys (sys.(parties)) (sys.(serverState))
        ((P2P p1 p2 r) # (sys.(network))).
```

1.12 Event RelayP2P

```
Definition RelayP2PPrecond (sys : System) (p1 p2 : Party) (r : ROUND) :=
  sys.(serverState) = running
```

```
  ∧ p1 ∈ (sys.(parties))
  ∧ p2 ∈ (sys.(parties))
  ∧ (∀ p : Party, p.(state) = assigned)
  ∧ (P2P p1 p2 r) ∈ (sys.(network))
  ∧ (relayP2P p1 p2 r) ∉ (sys.(network)).

Definition RelayP2PEvent (sys : System) (p1 p2 : Party) (r : ROUND) :=
  mkSys (sys.(parties)) (sys.(serverState))
        ((relayP2P p1 p2 r) # (sys.(network))).

(*  * Transition system
  *)
```

Transitions (of state machines) are what appear to me the closest the next operator of TLA+

```
Inductive Trans (sys : System) : System → Prop :=
| start_t: StartPrecond sys → Trans sys (StartEvent sys)
| abort_t: AbortPrecond sys → Trans sys (AbortEvent sys)
| party_abort_t: ∀ p : Party,
    PartyAbortPrecond sys p → Trans sys (PartyAbortEvent sys p)
| party_ready_t: ∀ p : Party,
    PartyReadyPrecond sys p → Trans sys (PartyReadyEvent sys p)
| assign_t: ∀ p : Party,
    AssignPrecond sys p → Trans sys (AssignEvent sys p)
| req_broadcast_t: ∀ p : Party, ∀ r : ROUND,
      ReqToBroadcastPrecond sys p r → Trans sys (ReqToBroadcastEvent sys p r)
| relay_broadcast_t: ∀ p : Party, ∀ r : ROUND,
      RelayBroadcastPrecond sys p r → Trans sys (RelayBroadcastEvent sys p r)
| req_p2p_t: ∀ p1 p2 : Party, ∀ r : ROUND,
      ReqToP2PPrecond sys p1 p2 r → Trans sys (ReqToP2PEvent sys p1 p2 r)
| relay_p2p_t: ∀ p1 p2 : Party, ∀ r : ROUND,
      RelayP2PPrecond sys p1 p2 r → Trans sys (RelayP2PEvent sys p1 p2 r).
```

The "always", or invariant, predicate can be modeled as a kind of transitive closure of transitions.

```
Inductive Behavior : System → Prop :=
| beh_init: ∀ parties : SET Party,
    InitSysPrecond parties → Behavior (InitSysEvent parties)
| beh_trans: ∀ sys sys' : System,
    Behavior sys → Trans sys sys' → Behavior sys'.
```

The next question is : what it is that we want to prove ? It is useless to prove any type constraints because we are in type theory and everything is typed by construction.

That's all for now ... I'll continue and then discuss withyou (Omer) the issues you raised.