MODULE *ConsensusPlusCal*

EXTENDS *Integers*, *Sequences*, *TLC*

CONSTANTS

Names, a set

Participants, an array of participants, in their order in the state channel

NULL

ASSUME

$\land Len(Participants) > 1$

$NumParticipants \triangleq Len(Participants)$

$Types \triangleq [$
$\quad WAITING \mapsto$ "WAITING",
$\quad SENT \mapsto$ "SENT",
$\quad SUCCESS \mapsto$ "SUCCESS",
$\quad FAILURE \mapsto$ "FAILURE"
$]$

$Status \triangleq [$
$\quad OK \mapsto$ "OK",
$\quad ABORT \mapsto$ "ABORT",
$\quad SUCCESS \mapsto$ "SUCCESS"
$]$

$Range(f) \triangleq \{f[x] : x \in \text{DOMAIN } f\}$

$Running(state) \triangleq state.type \in \{Types.WAITING, Types.SENT\}$

$Terminated(state) \triangleq \neg Running(state)$

--algorithm *consensus_update*

For the moment, we assume that participants only send commitments forward.
Since a read message is then discarded, it's enough to just store one.

variables $msg = NULL$

define

Arrays are 1-indexed, while the % operator returns a number between 0 and *NumParticipants*.
This explains the following slightly complicated expression

$mover(turnNumber) \triangleq 1 + ((turnNumber - 1)\%NumParticipants)$

$safeToSend(state) \triangleq$
$\quad \land state.type = Types.WAITING$
$\quad \land \lor state.ourIndex = state.turnNumber\%NumParticipants$
$\quad\quad \lor \land msg \neq NULL$
$\quad\quad\quad \land msg.status = Status.OK$
$\quad\quad\quad \land state.ourIndex = mover(msg.turnNumber)$

$target(turnNumber) \triangleq Participants[mover(turnNumber)]$

end define ;

macro *sendVote*(*turnNumber*, *votesRequired*)
begin
assert $votesRequired > 0$ **;**

```
state := [
    type ↦ Types.SENT,
    turnNumber ↦ turnNumber,
    ourIndex    ↦ state.ourIndex
];
msg := [
    to ↦ target(state.turnNumber),
    turnNumber      ↦ state.turnNumber,
    votesRequired   ↦ votesRequired,
    status          ↦ Status.OK
]
end macro ;

macro returnSuccess()
begin
state := [type ↦ Types.SUCCESS] @@ state ;
msg := [
    to      ↦ target(state.turnNumber),
    status ↦ Status.SUCCESS
]
end macro ;

macro returnFailure(turnNumber)
begin
state := [
    type ↦ Types.FAILURE,
    turnNumber ↦ turnNumber
] @@ state ;
msg := [
    to ↦ target(state.ourIndex + 1),
    status ↦ Status.ABORT
];
end macro ;

macro vote(turnNumber, votesRequired)
begin
if votesRequired = 0 then returnSuccess()
 else sendVote(turnNumber, votesRequired)
end if ; end macro ;

macro waitForUpdate(turnNumber)
begin
state := [
    turnNumber ↦ turnNumber,
    type          ↦ Types.WAITING,
    ourIndex    ↦ state.ourIndex
```

```
];
msg := NULL;
end macro ;

macro voteOrreturnFailure(turnNumber, votesRequired)
begin
  If the participant agrees with the allocation, they vote
either vote(turnNumber, votesRequired)
  Otherwise, they return FAILURE
or returnFailure(turnNumber)
end either ; end macro ;
```

Calling this a fair process prevents the process from stuttering forever. It's always considered to be valid to take a step where your state variables don't change, which could be the case if some unrelated protocols end up in an infinite loop, for instance. However, we want to check that IF A: wallets always eventually take some valid action THEN B: wallets always eventually terminate the consensus-update protocol Calling the process fair ensures that A is true, and therefore the model checks that under the assumption A, B is also true.

```
fair process consensusUpdate ∈ DOMAIN Participants
variables
  state = [
    turnNumber ↦ 1,
    ourIndex ↦ self,
    type ↦ Types.WAITING
  ],
  me = Participants[self]

begin
  Each participant either sends a message if it's currently safe to do so, or else it reads a message for the participant, updates their state accordingly, and sends a message if it's then safe. These actions are currently assumed to be atomic, and are therefore assigned to a single label, ReachConsensus
  ReachConsensus:
    while Running(state) do
      if safeToSend(state) ∧ msg = NULL then
        either returnFailure(state.turnNumber)  If the commitment is not valid
        or
          if   state.type  = Types.WAITING then vote(state.turnNumber + 1, NumParticipants − 1) ;
           elsif state.type = Types.SENT      then returnFailure(state.turnNumber) ;
           else assert FALSE
          end if  ;
        end either ;
       else
        await msg ≠ NULL ∧ msg.to = me ;
        if msg.status = Status.OK then
            If the commitment receved is not valid, return FAILURE
```

```
                TODO : Is this the actual behaviour we want?
                In the readme, we say this is what works, but the reducer does not
                work this way
              either returnFailure(state.turnNumber)
              or if msg.turnNumber > state.turnNumber then
                    First, update our state based on the incoming message
                 if msg.votesRequired = 0 then returnSuccess()
                  elsif safeToSend(state) then
                     if    state.type  = Types.SENT then returnFailure(msg.turnNumber)
                      elsif state.type = Types.WAITING then voteOrreturnFailure(msg.turnNumber + 1, n
                      else assert FALSE ;
                     end if ;
                  else waitForUpdate(msg.turnNumber)
                 end if ;
              end if ; end either ;
            elsif msg.status = Status.ABORT then returnFailure(state.turnNumber)
            elsif msg.status = Status.SUCCESS then returnSuccess()
            else assert FALSE
           end if ;
        end if ;
    end while ;
end process ;

end algorithm ;
```

BEGIN TRANSLATION

VARIABLES msg, pc

define statement

$mover(turnNumber) \triangleq 1 + ((turnNumber - 1)\%NumParticipants)$

$safeToSend(state) \triangleq$

$\land\ state.type = Types.WAITING$

$\land\ \lor\ state.ourIndex = state.turnNumber\%NumParticipants$

$\lor\ \land\ msg \neq NULL$

$\land\ msg.status = Status.OK$

$\land\ state.ourIndex = mover(msg.turnNumber)$

$target(turnNumber) \triangleq Participants[mover(turnNumber)]$

VARIABLES $state$, me

$vars \triangleq \langle msg, pc, state, me \rangle$

$ProcSet \triangleq (\text{DOMAIN } Participants)$

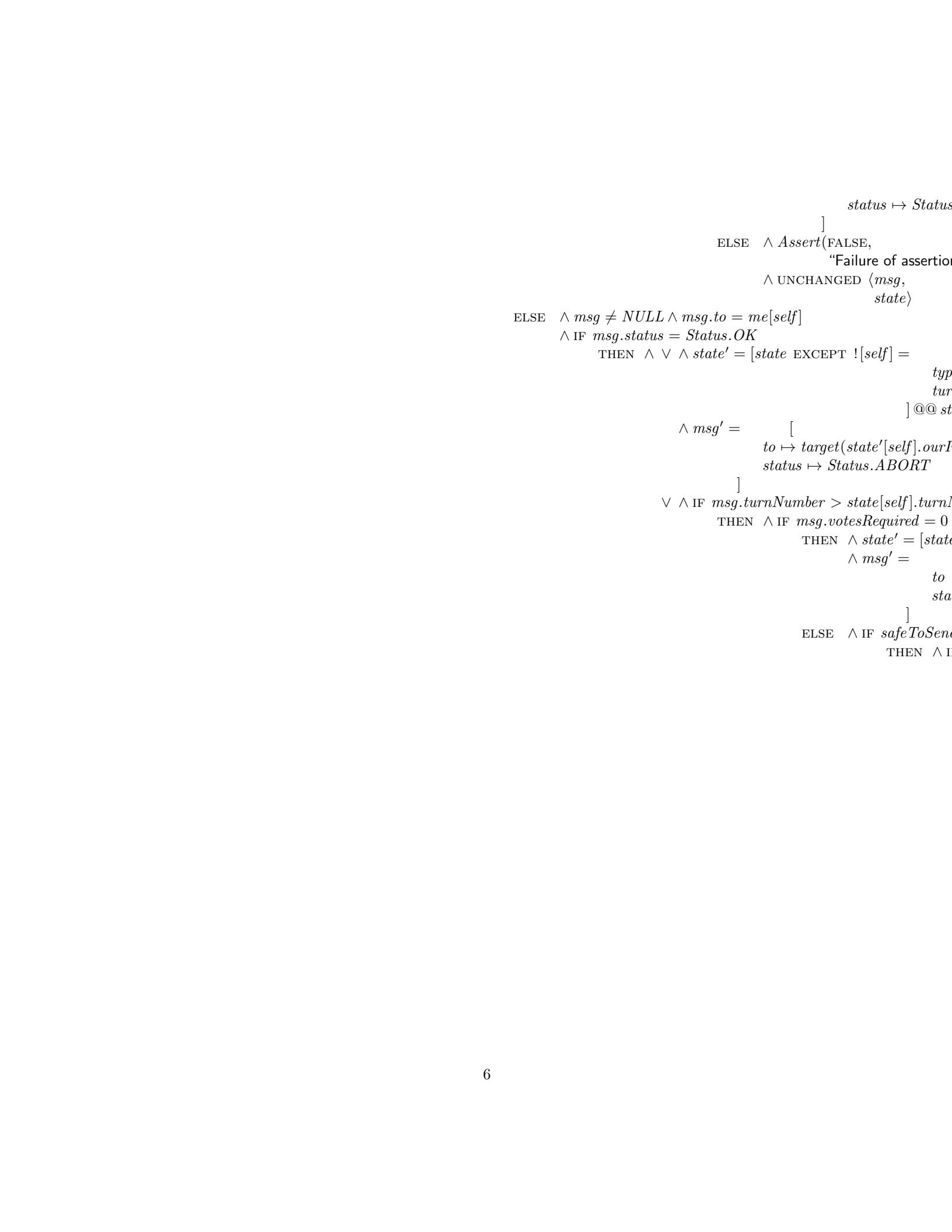
$Init \triangleq$ *Global variables*

$\land\ msg = NULL$

Process consensusUpdate

$\land\ state = [self \in \text{DOMAIN } Participants \mapsto [$

```
                                                    turnNumber ↦ 1,
                                                    ourIndex ↦ self,
                                                    type ↦ Types.WAITING
                                                ]]
    ∧ me = [self ∈ DOMAIN Participants ↦ Participants[self]]
    ∧ pc  = [self ∈ ProcSet ↦ "ReachConsensus"]

ReachConsensus(self) ≜  ∧ pc[self] = "ReachConsensus"
                        ∧ IF Running(state[self])
                             THEN  ∧ IF safeToSend(state[self]) ∧ msg = NULL
                                        THEN  ∧ ∨ ∧ state' = [state EXCEPT ![self] =  [
                                                                                        type ↦ Types.F
                                                                                        turnNumber ↦
                                                                                    ] @@ state[self]]
                                                  ∧ msg' =  [
                                                              to ↦ target(state'[self].ourIndex + 1),
                                                              status ↦ Status.ABORT
                                                            ]
                                              ∨ ∧ IF state[self].type = Types.WAITING
                                                     THEN  ∧ IF (NumParticipants − 1) = 0
                                                                THEN  ∧ state' = [state EXCEPT ![s
                                                                      ∧ msg' =  [
                                                                                  to      ↦ target
                                                                                  status ↦ Status
                                                                                ]
                                                                ELSE  ∧ Assert((NumParticipants
                                                                               "Failure of assertio
                                                                      ∧ state' = [state EXCEPT ![s




                                                                      ∧ msg' =  [
                                                                                  to ↦ target(sta
                                                                                  turnNumber
                                                                                  votesRequired
                                                                                  status
                                                                                ]
                                                     ELSE  ∧ IF state[self].type = Types.SENT
                                                                THEN  ∧ state' = [state EXCEPT ![s



                                                                      ∧ msg' =  [
                                                                                  to ↦ target(sta
```


status ↦ Status
]
ELSE ∧ Assert(FALSE,
"Failure of assertio
∧ UNCHANGED ⟨msg,
state⟩
ELSE ∧ msg ≠ NULL ∧ msg.to = me[self]
∧ IF msg.status = Status.OK
THEN ∧ ∨ ∧ state′ = [state EXCEPT ![self] =
typ
tur
] @@ st
∧ msg′ = [
to ↦ target(state′[self].ourI
status ↦ Status.ABORT
]
∨ ∧ IF msg.turnNumber > state[self].turnI
THEN ∧ IF msg.votesRequired = 0
THEN ∧ state′ = [state
∧ msg′ =
to
sta
]
ELSE ∧ IF safeToSend
THEN ∧ I

ELSE ∧ s

∧ r

ELSE ∧ TRUE
∧ UNCHANGED ⟨msg,
state⟩

ELSE ∧ IF msg.status = Status.ABORT
THEN ∧ state′ = [state EXCEPT ![self] =

∧ msg′ = [
to ↦ target(state′[se
status ↦ Status.ABO
]

ELSE ∧ IF msg.status = Status.SUCCE
THEN ∧ state′ = [state EXC
∧ msg′ = [
to ↦
status ↦
]
ELSE ∧ Assert(FALSE,

```
                                                                                    "Failure of
                                                                                ∧ UNCHANGED ⟨msg
                                                                                            state
                                        ∧ pc' = [pc EXCEPT ![self] = "ReachConsensus"]
                                ELSE    ∧ pc' = [pc EXCEPT ![self] = "Done"]
                                        ∧ UNCHANGED ⟨msg, state⟩
                            ∧ me' = me

consensusUpdate(self) ≜ ReachConsensus(self)

Allow infinite stuttering to prevent deadlock on termination.
Terminating ≜ ∧ ∀ self ∈ ProcSet : pc[self] = "Done"
              ∧ UNCHANGED vars

Next ≜ (∃ self ∈ DOMAIN Participants : consensusUpdate(self))
         ∨ Terminating

Spec ≜ ∧ Init ∧ □[Next]_vars
       ∧ ∀ self ∈ DOMAIN Participants : WF_vars(consensusUpdate(self))

Termination ≜ ◇(∀ self ∈ ProcSet : pc[self] = "Done")

END TRANSLATION

AllowedMessages ≜
 [
   turnNumber : Nat,
   votesRequired : 0 .. (NumParticipants − 1),
   to : Names,
   status : {Status.OK}
 ]
 ∪ {NULL}
 ∪ [
   to : Names,
   status : {Status.ABORT, Status.SUCCESS}
 ]

States ≜ {}
  ∪  [turnNumber : Nat, ourIndex : DOMAIN Participants, type : Range(Types)]

Safety properties

TypeOK ≜
   The following two conditions specify the format of each message and
   participant state.
  ∧ state ∈ [DOMAIN Participants → States]
  ∧ msg ∈ AllowedMessages
```

TODO : *Get TurnNumberDoesNotDecrease and StaysTerminated*
For some reason, *state*[*p*].*turnNumber* is not valid
$TurnNumberDoesNotDecrease \triangleq$
$\quad \land \forall\, p \in \text{DOMAIN}\ Participants : state[p].turnNumber' \geq state[p].turnNumber$

Once a process has terminated, its state does not change.
$StaysTerminated \triangleq \forall\, p \in \text{DOMAIN}\ Participants : (Terminated(state[p]) \Rightarrow (state'[p] = state[p]))$

Liveness properties

The protocol always terminates consistently across all processes.
TODO: Is this actually feasible, or actually what we want?
For example, perhaps the last person to vote agrees, and sends a message reaching consensus.
Their process terminates in the *SUCCESS* state, but for whatever reason their
commitment was invalid, and the other processes therefore terminate in *FAILURE*.
$ProtocolTerminates \triangleq$
$\quad \lor\ \land (\forall\, p \in \text{DOMAIN}\ Participants : \Diamond\Box(state[p].type = Types.SUCCESS))$
$\qquad \land \text{TRUE}$ *TODO*: In this case, should we specify that they reach the same turn number?
$\quad \lor (\forall\, p \in \text{DOMAIN}\ Participants : \Diamond\Box(state[p].type = Types.FAILURE))$

The value of *msg* should eventually always be *NULL*
$MessagesAreRead \triangleq \Diamond\Box(msg = NULL)$

* Modification History
* Last modified *Fri Aug* 09 12:11:18 *MDT* 2019 by *andrewstewart*
* Created *Tue Aug* 06 14:38:11 *MDT* 2019 by *andrewstewart*