```
MODULE ConsensusUpdate
EXTENDS Integers, Sequences, TLC
CONSTANTS
    Names,  a set
    Participants,  an array of participants, in their order in the state channel
    NULL
ASSUME
  ∧ Len(Participants) > 1

NumParticipants ≜ Len(Participants)
Types ≜ [
    WAITING ↦ "WAITING",
    SENT    ↦ "SENT",
    SUCCESS ↦ "SUCCESS",
    FAILURE ↦ "FAILURE"
]
Status ≜ [
  OK       ↦ "OK",
  ABORT    ↦ "ABORT",
  SUCCESS  ↦ "SUCCESS"
]
Range(f) ≜ {f[x] : x ∈ DOMAIN f}
Running(state) ≜ state.type ∈ {Types.WAITING, Types.SENT}
Terminated(state) ≜ ¬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) ≜ 1 + ((turnNumber − 1)%NumParticipants)
safeToSend(state)   ≜
    ∧ state.type = Types.WAITING
    ∧ ∨ state.ourIndex = mover(state.turnNumber)
      ∨ ∧ msg ≠ NULL
        ∧ msg.status = Status.OK
        ∧ state.ourIndex = mover(msg.turnNumber)
target(turnNumber) ≜ 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 ;
```

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 $consensusUpdate \in$ DOMAIN $Participants$
variables
  $state = [$
    $turnNumber \mapsto 1,$
    $ourIndex \mapsto self,$
    $type \mapsto Types.WAITING$
  $],$
  $me = Participants[self]$

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 $safeToSend(state) \land msg = NULL$ then
            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 ;
         else
            await $msg \neq NULL \land msg.to = me$ ;
              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 $returnFailure(state.turnNumber)$
              In this case, the commitment was valid.
            or if $msg.status = Status.OK$ then
                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 $vote(msg.turnNumber + 1, msg.votesRequire$
                         else assert FALSE ;
                        end if ;
                     else waitForUpdate(msg.turnNumber)
                    end if ;
                end if ;
            elsif msg.status = Status.ABORT    then returnFailure(state.turnNumber)
            elsif msg.status = Status.SUCCESS then returnSuccess()
           end if ; end either ;
       end if ;
   end while ;
end process ;

end algorithm  ;
```

```
 BEGIN TRANSLATION
VARIABLES msg, pc

 define statement
mover(turnNumber) ≜ 1 + ((turnNumber − 1)%NumParticipants)
safeToSend(state)    ≜
     ∧ state.type = Types.WAITING
     ∧ ∨ state.ourIndex = mover(state.turnNumber)
        ∨ ∧ msg ≠ NULL
           ∧ msg.status = Status.OK
           ∧ state.ourIndex = mover(msg.turnNumber)
target(turnNumber) ≜ Participants[mover(turnNumber)]

VARIABLES state, me

vars ≜ ⟨msg, pc, state, me⟩

ProcSet ≜ (DOMAIN Participants)

Init ≜  Global variables
         ∧ msg = NULL
          Process consensusUpdate
         ∧ state = [self ∈ DOMAIN Participants ↦          [
                                                     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 ∧ IF state[self].type = Types.WAITING
                                                   THEN ∧ IF (NumParticipants − 1) = 0
```

```
                    THEN  ∧ state′ = [state EXCEPT ![self] =
                          ∧ msg′ =         [
                                      to      ↦ target(state
                                      status ↦ Status.SUC
                                   ]
                    ELSE  ∧ Assert((NumParticipants − 1)
                                   “Failure of assertion at li
                          ∧ state′ = [state EXCEPT ![self] =



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



                          ∧ msg′ =         [
                                      to ↦ target(state′[se
                                      status ↦ Status.ABC
                                   ]
                    ELSE  ∧ Assert(FALSE,
                                   “Failure of assertion at li
                          ∧ UNCHANGED ⟨msg,
                                       state⟩
ELSE   ∧ msg ≠ NULL ∧ msg.to = me[self]
       ∧ ∨ ∧ state′ = [state EXCEPT ![self] =            [
                                                type ↦ Types.F
                                                turnNumber ↦
                                           ] @@ state[self]]
           ∧ msg′ =         [
                       to ↦ target(state′[self].ourIndex + 1),
                       status ↦ Status.ABORT
                    ]
         ∨ ∧ IF msg.status = Status.OK
                THEN  ∧ IF msg.turnNumber > state[self].turnN
                           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 ![s



                                                                        ∧ msg′ =         [
                                                                                       to ↦ target(sta
                                                                                       status ↦ Status
                                                                                  ]
                                                                  ELSE  ∧ IF msg.status = Status.SU
                                                                           THEN  ∧ state′ = [state
                                                                                 ∧ msg′ =
                                                                                                to
                                                                                                sta
                                                                                           ]
                                                                           ELSE  ∧ TRUE
                                                                                 ∧ UNCHANGED


                                       ∧ 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}
  ]
```

$\quad \cup \{NULL\}$
$\quad \cup [$
$\qquad to : Names,$
$\qquad status : \{Status.ABORT, Status.SUCCESS\}$
$\quad]$

$States \triangleq \{\}$
$\quad \cup \quad [turnNumber : Nat, ourIndex : \textsc{domain}\ Participants, type : Range(Types)]$

Safety properties

$TypeOK \triangleq$

The following two conditions specify the format of each message and participant state.

$\quad \land state \in [\textsc{domain}\ Participants \rightarrow States]$
$\quad \land msg \in AllowedMessages$

TODO : Get TurnNumberDoesNotDecrease and StaysTerminated
For some reason, $state[p].turnNumber$ is not valid

$TurnNumberDoesNotDecrease \triangleq$
$\quad \land \forall p \in \textsc{domain}\ Participants : state[p].turnNumber' \geq state[p].turnNumber$

Once a process has terminated, its state does not change.

$StaysTerminated \triangleq \forall p \in \textsc{domain}\ Participants : (Terminated(state[p]) \Rightarrow (state'[p] = state[p]))$

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*.

$ProtocolTerminates \triangleq$
$\qquad \lor \land (\forall p \in \textsc{domain}\ Participants : \Diamond\Box(state[p].type = Types.SUCCESS))$
$\qquad\quad \land \text{TRUE}$ *TODO*: In this case, should we specify that they reach the same turn number?
$\qquad \lor (\forall p \in \textsc{domain}\ Participants : \Diamond\Box(state[p].type = Types.FAILURE))$

The value of *msg* should eventually always be *NULL*

$MessagesAreRead \triangleq \Diamond\Box(msg = NULL)$

* Modification History
* Last modified *Mon Aug* 12 23:19:29 *MDT* 2019 by *andrewstewart*
* Created *Tue Aug* 06 14:38:11 *MDT* 2019 by *andrewstewart*