MODULE *cbccasper_binary*

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

The first four are parameters of the protocol, and the rest are defined for purposes of model checking
- *validators*: a set specifying the names (consecutive integers) of the *validators*
- *validator_weights*: a tuple assigning a weight (integer) to each *validator*
- *byzantine_threshold*: a number less than the sum of the weights of all the *validators*
- *consensus_values*: the set of values the *validators* decide on; the binary values 0 and 1 in this model
- *validators_initial_values*: the estimates without receiving other messages
- *message_ids*: a number used to limit the number of messages sent in the model
- *byzantine_fault_nodes*: define the equivocating nodes

CONSTANTS

validators,
validator_weights,
byzantine_threshold,
consensus_values,
validators_initial_values,
message_ids,
byzantine_fault_nodes

$Even \triangleq \{i \in Integers : i \bmod 2 = 0\}$

The following is written in *PlusCal*, which will be transpiled to TLA+.
The transpiled TLA+ code is appended to the *PlusCal* code, and the *PlusCal* code will appear as comments.

--algorithm *algo*

variables are updated as the model checker runs:
- *all_msg*: a set of all messages ids (integers)
- *equivocating_msg*: a set specifying all the equivocating messages
  (not all the *validators* receive that same message)
- *msg_sender*: a tuple specifying the sender of the message with the given id
- *msg_estimate*: a tuple specifying the estimate of the message with the given id
- *msg_justification*: a tuple specifying the justification of the message
  (set of messages used to calculate the estimate) with the given id,
- *cur_msg_id*: the id of the current message being sent; incremented after every message
- *validator_init_done*: a tuple indicating whether the *validator* has sent the initial message
  (1 is done; all the *validators* are initialized to 0 in the beginning)
- *equiv_msg_receivers*: a tuple specifying a subset of all *validators* receiving a certain equivocating message;
  if the message is not equivocating append the empty set
- *cur_subset*: a 'temp' variable used to store the set of *validators* receiving the current equivocating message

variables

$all\_msg = \{\}$,
$equivocating\_msg = \{\}$,
$msg\_sender = \langle\rangle$,
$msg\_estimate = \langle\rangle$,
$msg\_justification = \langle\rangle$,

$cur\_msg\_id = 1,$
$validator\_init\_done = \langle 0, 0, 0, 0, 0, 0, 0, 0, 0, 0 \rangle,$
$equiv\_msg\_receivers = \langle \rangle,$
cur_subset

define

The dependencies of a message $m1$ are the messages in the justification of $m1$ and in the justifications of the justifications of $m1$ and so on. The set is generated using a recursive function until the base case is reached - the only justification of a message is itself.

$$
\begin{array}{l}
dependencies(message) \triangleq \\
\quad \text{LET} \\
\qquad \text{RECURSIVE } dep(\_) \\
\qquad dep(msg) \triangleq \\
\qquad\quad \text{IF } Cardinality(msg\_justification[msg]) = 1 \land msg \in msg\_justification[msg] \\
\qquad\quad \text{THEN } \{msg\} \\
\qquad\quad \text{ELSE UNION } \{dep(msg2) : msg2 \in msg\_justification[msg]\} \\
\quad \text{IN } \quad dep(message)
\end{array}
$$

Gets the set of dependencies of all the messages in a set of messages.

$$dep\_set(messages) \triangleq messages \cup \text{UNION } \{dependencies(m) : m \in messages\}$$

The latest message of a *validator* in an observed set of messages is the message for which no other messages sent by the same *validator* justifies it.

$$
\begin{array}{l}
latest\_message(validator, messages) \triangleq \\
\quad \{msg \in messages : \\
\qquad \land\, msg\_sender[msg] = validator \\
\qquad \land\, \neg\exists\, msg2 \in messages : \\
\qquad\quad \land\, msg\_sender[msg2] = validator \\
\qquad\quad \land\, msg \neq msg2 \\
\qquad\quad \land\, msg \in dependencies(msg2) \\
\quad \}
\end{array}
$$

Defines the *Pick* operator used to choose an arbitrary element from a set.

$$Pick(S) \triangleq \text{CHOOSE } s \in S : \text{TRUE}$$

Defines the *Sum* operator used to find the sum of all the elements in a set.

$$
\begin{array}{l}
\text{RECURSIVE } SetReduce(\_, \_, \_) \\
\quad SetReduce(Op(\_, \_), S, value) \triangleq \\
\qquad \text{IF } S = \{\} \text{ THEN } value \\
\qquad \text{ELSE LET } s \triangleq Pick(S) \text{IN} \quad SetReduce(Op, S \setminus \{s\}, Op(s, value)) \\
\\
Sum(S) \triangleq \text{LET } \_op(a, b) \triangleq a + b \text{IN} \quad SetReduce(\_op, S, 0)
\end{array}
$$

The following defines the estimator used in the binary consesnsus protocol.
The score of an estimate is the sum of the weights of all the *validators* having the given estimate in its latest message in a set of observed messages.

The estimator returns the estimate with the larger score. If there's a tie,
in this example, the value 0 is used.

$$
\begin{array}{l}
score(estimate,\ messages) \triangleq \\
\quad \text{LET } ss \triangleq \\
\qquad \{v \in validators : \\
\qquad\quad \land\ Cardinality(latest\_message(v,\ messages)) = 1 \\
\qquad\quad \land \exists\, m \in latest\_message(v,\ messages) : \\
\qquad\qquad msg\_estimate[m] = estimate\} \\
\qquad ss2 \triangleq \{validator\_weights[v] : v \in ss\} \\
\quad \text{IN } \ Sum(ss2) \\
binary\_estimator(messages) \triangleq \\
\quad \text{IF } score(1,\ messages) > score(0,\ messages) \\
\quad \text{THEN } 1 \\
\quad \text{ELSE } 0
\end{array}
$$

Two messages are equivocating if they have the same sender but do not justify each other.

$$
\begin{array}{l}
equivocation(m1,\ m2) \triangleq \\
\quad \land\ m1 \neq m2 \\
\quad \land\ msg\_sender[m1] = msg\_sender[m2] \\
\quad \land\ m1 \notin dependencies(m2) \\
\quad \land\ m2 \notin dependencies(m1)
\end{array}
$$

A *validator* is byzantine if it sends equivocating messages.

$$
\begin{array}{l}
faulty\_node(validator,\ messages) \triangleq \\
\quad \land \exists\, m1 \in dep\_set(messages) : \\
\qquad \land \exists\, m2 \in dep\_set(messages) : \\
\qquad\quad \land\ validator = msg\_sender[m1] \\
\qquad\quad \land\ equivocation(m1,\ m2)
\end{array}
$$

Set of byzantine *validators* in an observed set of messages.

$$faulty\_nodes(messages) \triangleq \{v \in validators : faulty\_node(v,\ messages)\}$$

Returns the total weight of all byzatine *validators*.

$$
\begin{array}{l}
fault\_weight(messages) \triangleq \\
\quad \text{LET } byz \triangleq faulty\_nodes(messages) \\
\quad \text{IN } \ Sum(\{validator\_weights[v] : v \in byz\})
\end{array}
$$

For a given fault tolerance threshold t, a protocol state is the set of messages
with fault weight less than t.
A state transition is possible if one state is a subset of another.

$$protocol\_states(messages,\ t) \triangleq \{s \in \text{SUBSET } (messages) : fault\_weight(s) < t\}$$

$$protocol\_executions(state1,\ state2) \triangleq state1 \subseteq state2$$

Two *validators* $v1$ and $v2$ are agreeing with each other if:

- $v1$ has exactly one latest message in messages
- $v2$ has exactly one latest message in the justification of $v1$'s latest message

- the latest messages have estimates that agree with each other

$validators\_agreeing(v1,\ v2,\ estimate,\ messages) \triangleq$
$\land\ Cardinality(latest\_message(v1,\ messages)) = 1$
$\land$ LET $v1\_latest\_msg \triangleq Pick(latest\_message(v1,\ messages))$
IN $Cardinality(latest\_message(v2,\ msg\_justification[v1\_latest\_msg])) = 1$
$\land$ LET $v2\_latest\_msg \triangleq Pick(latest\_message(v2,\ msg\_justification[v1\_latest\_msg]))$
IN $estimate = msg\_estimate[v2\_latest\_msg]$

Two *validators* are disagreeing with each other if:
– $v1$ has exactly one latest message in messages
– $v2$ has exactly one latest message in the justification of $v1$'s latest message
– $v2$ has a new latest message that doens't agree with the estimate
$validators\_disagreeing(v1,\ v2,\ estimate,\ messages) \triangleq$
$\land\ Cardinality(latest\_message(v1,\ messages)) = 1$
$\land$ LET $v1\_latest\_msg \triangleq$ CHOOSE $x \in latest\_message(v1,\ messages)$: TRUE
IN $Cardinality(latest\_message(v2,\ msg\_justification[v1\_latest\_msg])) = 1$
$\land$ LET $v2\_latest\_msg \triangleq$ CHOOSE $x \in latest\_message(v2,\ msg\_justification[v1\_latest\_msg])$:
IN $\exists\, m \in messages : v2\_latest\_msg \in dependencies(m)$
$\land\ estimate \neq msg\_estimate[m]$

An "e-clique" is a group of non-byzantine nodes in a set of observed messages such that :
- they mutually see each other agreeing with estimate in messages
- they mutually cannot see each other disagreeing with estimate in messages
$e\_clique(estimate,\ messages) \triangleq \{$
$ss \in$ SUBSET $(validators)$:
$\land\ \forall\, v1 \in ss$:
$\land\ \forall\, v2 \in ss$:
IF $v1 = v2$
THEN TRUE
ELSE
$\land\ validators\_agreeing(v1,\ v2,\ estimate,\ messages)$
$\land\ \neg validators\_disagreeing(v1,\ v2,\ estimate,\ messages)$
isn't this just $\lor$?
$\}$
Finds the existence of an e-clique
$e\_clique\_estimate\_safety(estimate,\ messages) \triangleq$
$\land\ \exists\, ss \in e\_clique(estimate,\ messages)$:
$\land\ 2 * Sum(\{validator\_weights[v] : v \in ss\}) > Sum(\{validator\_weights[v] : v \in validators\}) + b$

Gets the set of messges received by a particular valiadtor
$validator\_received\_msg(validator) \triangleq$
$(all\_msg \setminus equivocating\_msg) \cup$
$\{x \in equivocating\_msg : validator \in equiv\_msg\_receivers[x]\}$

Returns a subset of received messages for an equivocating *validator* to generate potentially equivocating messages
$get\_equivocation\_subset\_msg(validator) \triangleq$ CHOOSE $x \in$ SUBSET $(validator\_received\_msg(validator))$:

Returns a subset of *validators* receiving a particular equivocating message

$get\_equiv\_receivers(validator) \triangleq \text{CHOOSE } x \in (\text{SUBSET } (validators \setminus \{validator\})) : x \neq \{\}$

A temporal property checking that finality can eventually be reached in a binary consensus protocol

$check\_safety\_with\_oracle \triangleq$

$\quad \text{LET } v \triangleq \text{CHOOSE } v \in (validators \setminus byzantine\_fault\_nodes) : \text{TRUE}$

$\quad \text{IN} \quad \Diamond(e\_clique\_estimate\_safety(0, validator\_received\_msg(v)) \vee e\_clique\_estimate\_safety(1, valida$

end define ;

A message from a non-byzantine *validator* is received by all the *validators*

macro *make_message*(*validator*, *estimate*, *justification*)**begin**

equiv_msg_receivers := *Append*(*equiv_msg_receivers*, {}) ;
msg_sender := *Append*(*msg_sender*, *validator*) ;
msg_estimate := *Append*(*msg_estimate*, *estimate*) ;
msg_justification := *Append*(*msg_justification*, *justification*) ;
all_msg := *all_msg* ∪ {*cur_msg_id*} ;
cur_msg_id := *cur_msg_id* + 1 ;

end macro ;

A *validators* sends an initial message without receiving information from other *validators*
An initial message is only justified by itself

macro *init_validator*(*validator*)**begin**

make_message(*validator*, *validators_initial_values*[*validator*], {*cur_msg_id*}) ;
validator_init_done[*validator*] := 1 ;

end macro ;

An equivocating *validator* takes different subsets of its received messages to generate
different estimates and sends the different messages to different subsets of *validators*.

macro *make_equivocating_messages*(*validator*)**begin**

equiv_msg_receivers := *Append*(*equiv_msg_receivers*, *get_equiv_receivers*(*validator*)) ;
cur_subset := *get_equivocation_subset_msg*(*validator*) ;
equivocating_msg := *equivocating_msg* ∪ {*cur_msg_id*} ;
msg_sender := *Append*(*msg_sender*, *validator*) ;
msg_estimate := *Append*(*msg_estimate*, *binary_estimator*(*cur_subset*)) ;
msg_justification := *Append*(*msg_justification*, *cur_subset*) ;
all_msg := *all_msg* ∪ {*cur_msg_id*} ;
cur_msg_id := *cur_msg_id* + 1 ;

end macro ;

A general macro for sending messages
Non-equivocating and equivocating *validators* behave differently
No honest *validator* will send the same message multiple times consecutively
Equivocating *validators* may send different messages to different subsets of *validators* consecutively.

macro *send_message*(*validator*)**begin**

if (*cur_msg_id* > 1 ∧ *msg_sender*[*cur_msg_id* − 1] ≠ *validator*) ∨ (*validator* ∈ *byzantine_fault_nodes*)
if *validator* ∉ *byzantine_fault_nodes* **then**
make_message(*validator*, *binary_estimator*(*validator_received_msg*(*validator*)), *validator_receiv*

```
            else
                make_equivocating_messages(validator) ;
            end if ;
        else
                skip ;
        end if ;
    end macro ;

    Each process is an individual validator
    Validators send messages in random orders
    Validators keep on sending messages until the maximum limit is reached
    fair process v ∈ validators begin
        Validate:
        while cur_msg_id ≤ message_ids do
            if validator_init_done[self] = 0 ∧ self ∉ byzantine_fault_nodes then
                init_validator(self) ;
            else
                send_message(self) ;
            end if ;
        end while ;
    end process ;

end algorithm ; ***
```

```
BEGIN TRANSLATION
CONSTANT defaultInitValue
VARIABLES all_msg, equivocating_msg, msg_sender, msg_estimate,
          msg_justification, cur_msg_id, validator_init_done,
          equiv_msg_receivers, cur_subset, pc

define statement
dependencies(message) ≜
    LET
        RECURSIVE dep(_)
        dep(msg) ≜
            IF Cardinality(msg_justification[msg]) = 1 ∧ msg ∈ msg_justification[msg]
            THEN {msg}
            ELSE UNION {dep(msg2) : msg2 ∈ msg_justification[msg]}
    IN  dep(message)

dep_set(messages) ≜
    messages ∪ UNION {dependencies(m) : m ∈ messages}

latest_message(validator, messages) ≜
    {msg ∈ messages :
```

```
        ∧ msg_sender[msg] = validator
        ∧ ¬∃ msg2 ∈ messages :
            ∧ msg_sender[msg2] = validator
            ∧ msg ≠ msg2
            ∧ msg ∈ dependencies(msg2)
    }

Pick(S) ≜ CHOOSE s ∈ S : TRUE
RECURSIVE SetReduce(_, _, _)
    SetReduce(Op(_, _), S, value) ≜
        IF S = {} THEN value
        ELSE LET s ≜ Pick(S)IN  SetReduce(Op, S \ {s}, Op(s, value))

    Sum(S) ≜ LET _op(a, b) ≜ a + bIN  SetReduce(_op, S, 0)

score(estimate, messages) ≜
    LET ss ≜
        {v ∈ validators :
            ∧ Cardinality(latest_message(v, messages)) = 1
            ∧ ∃ m ∈ latest_message(v, messages) :
                msg_estimate[m] = estimate}
        ss2 ≜ {validator_weights[v] : v ∈ ss}
    IN  Sum(ss2)

binary_estimator(messages) ≜
    IF score(1, messages) > score(0, messages)
     THEN 1
     ELSE 0

equivocation(m1, m2) ≜
    ∧ m1 ≠ m2
    ∧ msg_sender[m1] = msg_sender[m2]
    ∧ m1 ∉ dependencies(m2)
    ∧ m2 ∉ dependencies(m1)

faulty_node(validator, messages) ≜
    ∧ ∃ m1 ∈ dep_set(messages) :
        ∧ ∃ m2 ∈ dep_set(messages) :
            ∧ validator = msg_sender[m1]
            ∧ equivocation(m1, m2)

faulty_nodes(messages) ≜
    {v ∈ validators : faulty_node(v, messages)}
```

```
fault_weight(messages) ≜
    LET byz ≜ faulty_nodes(messages)
    IN  Sum({validator_weights[v] : v ∈ byz})

protocol_states(messages, t) ≜ {ss ∈ SUBSET (messages) : fault_weight(ss) < t}
protocol_executions(state1, state2) ≜ state1 ⊆ state2

validators_agreeing(v1, v2, estimate, messages) ≜
    ∧ Cardinality(latest_message(v1, messages)) = 1
    ∧ LET v1_latest_msg ≜ CHOOSE x ∈ latest_message(v1, messages) : TRUE
      IN  Cardinality(latest_message(v2, msg_justification[v1_latest_msg])) = 1
      ∧   LET v2_latest_msg ≜ CHOOSE x ∈ latest_message(v2, msg_justification[v1_latest_msg]) : TRUE
          IN  estimate = msg_estimate[v2_latest_msg]

validators_disagreeing(v1, v2, estimate, messages) ≜
    ∧ Cardinality(latest_message(v1, messages)) = 1
    ∧ LET v1_latest_msg ≜ CHOOSE x ∈ latest_message(v1, messages) : TRUE
      IN  Cardinality(latest_message(v2, msg_justification[v1_latest_msg])) = 1
      ∧   LET v2_latest_msg ≜ CHOOSE x ∈ latest_message(v2, msg_justification[v1_latest_msg]) : TRUE
          IN  ∃ m ∈ messages : v2_latest_msg ∈ dependencies(m)
              ∧ estimate ≠ msg_estimate[m]

e_clique(estimate, messages) ≜ {
    ss ∈ SUBSET (validators) :
        ∧ ∀ v1 ∈ ss :
            ∧ ∀ v2 ∈ ss :
                IF v1 = v2
                THEN TRUE
                ELSE
                    ∧ validators_agreeing(v1, v2, estimate, messages)
                    ∧ ¬validators_disagreeing(v1, v2, estimate, messages)
    }

e_clique_estimate_safety(estimate, messages) ≜
    ∧ ∃ ss ∈ e_clique(estimate, messages) :
        ∧ 2 * Sum({validator_weights[v] : v ∈ ss}) > Sum({validator_weights[v] : v ∈ validators}) + byzantine

validator_received_msg(validator) ≜
    (all_msg \ equivocating_msg) ∪
    {x ∈ equivocating_msg : validator ∈ equiv_msg_receivers[x]}

get_equivocation_subset_msg(validator) ≜ CHOOSE x ∈ SUBSET (validator_received_msg(validator)) : TRUE
```

$get\_equiv\_receivers(validator) \triangleq \text{CHOOSE } x \in (\text{SUBSET } (validators \setminus \{validator\})) : x \neq \{\}$

$check\_safety\_with\_oracle \triangleq$
  LET $v \triangleq \text{CHOOSE } v \in (validators \setminus byzantine\_fault\_nodes) : \text{TRUE}$
  IN $\Diamond(e\_clique\_estimate\_safety(0, validator\_received\_msg(v)) \lor e\_clique\_estimate\_safety(1, validator\_rec$

$vars \triangleq \langle all\_msg, equivocating\_msg, msg\_sender, msg\_estimate,$
$msg\_justification, cur\_msg\_id, validator\_init\_done,$
$equiv\_msg\_receivers, cur\_subset, pc\rangle$

$ProcSet \triangleq (validators)$

$Init \triangleq$ Global variables
$\land all\_msg = \{\}$
$\land equivocating\_msg = \{\}$
$\land msg\_sender = \langle\rangle$
$\land msg\_estimate = \langle\rangle$
$\land msg\_justification = \langle\rangle$
$\land cur\_msg\_id = 1$
$\land validator\_init\_done = \langle 0, 0, 0, 0, 0, 0, 0, 0, 0, 0\rangle$
$\land equiv\_msg\_receivers = \langle\rangle$
$\land cur\_subset = defaultInitValue$
$\land pc = [self \in ProcSet \mapsto \text{“Validate”}]$

$Validate(self) \triangleq \land pc[self] = \text{“Validate”}$
$\land \text{IF } cur\_msg\_id \leq message\_ids$
$\text{THEN } \land \text{IF } validator\_init\_done[self] = 0 \land self \notin byzantine\_fault\_nodes$
$\text{THEN } \land equiv\_msg\_receivers' = Append(equiv\_msg\_receivers, \{\})$
$\land msg\_sender' = Append(msg\_sender, self)$
$\land msg\_estimate' = Append(msg\_estimate, (validators\_initial\_valu$
$\land msg\_justification' = Append(msg\_justification, (\{cur\_msg\_id\}))$
$\land all\_msg' = (all\_msg \cup \{cur\_msg\_id\})$
$\land cur\_msg\_id' = cur\_msg\_id + 1$
$\land validator\_init\_done' = [validator\_init\_done \text{ EXCEPT } ![self] = 1]$
$\land \text{UNCHANGED } \langle equivocating\_msg,$
$cur\_subset\rangle$
$\text{ELSE } \land \text{IF } (cur\_msg\_id > 1 \land msg\_sender[cur\_msg\_id - 1] \neq self) \lor (se$
$\text{THEN } \land \text{IF } self \notin byzantine\_fault\_nodes$
$\text{THEN } \land equiv\_msg\_receivers' = Append(equiv\_$
$\land msg\_sender' = Append(msg\_sender, se$
$\land msg\_estimate' = Append(msg\_estimate$
$\land msg\_justification' = Append(msg\_justi$
$\land all\_msg' = (all\_msg \cup \{cur\_msg\_id\})$
$\land cur\_msg\_id' = cur\_msg\_id + 1$
$\land \text{UNCHANGED } \langle equivocating\_msg,$

$cur\_subset\rangle$

ELSE $\land$ $equiv\_msg\_receivers' = Append(equiv\_$
$\land$ $cur\_subset' = get\_equivocation\_subset\_$
$\land$ $equivocating\_msg' = (equivocating\_msg$
$\land$ $msg\_sender' = Append(msg\_sender, se$
$\land$ $msg\_estimate' = Append(msg\_estimate$
$\land$ $msg\_justification' = Append(msg\_justi$
$\land$ $all\_msg' = (all\_msg \cup \{cur\_msg\_id\})$
$\land$ $cur\_msg\_id' = cur\_msg\_id + 1$

ELSE $\land$ TRUE
$\land$ UNCHANGED $\langle all\_msg,$
$equivocating\_msg,$
$msg\_sender,$
$msg\_estimate,$
$msg\_justification,$
$cur\_msg\_id,$
$equiv\_msg\_receivers,$
$cur\_subset\rangle$

$\land$ UNCHANGED $validator\_init\_done$

$\land$ $pc' = [pc$ EXCEPT $![self] =$ "Validate"$]$

ELSE $\land$ $pc' = [pc$ EXCEPT $![self] =$ "Done"$]$
$\land$ UNCHANGED $\langle all\_msg,\ equivocating\_msg,$
$msg\_sender,\ msg\_estimate,$
$msg\_justification,\ cur\_msg\_id,$
$validator\_init\_done,$
$equiv\_msg\_receivers,\ cur\_subset\rangle$

$v(self) \triangleq Validate(self)$

$Next \triangleq (\exists\, self \in validators : v(self))$
$\lor$ Disjunct to prevent deadlock on termination
$((\forall\, self \in ProcSet : pc[self] =$ "Done"$) \land$ UNCHANGED $vars)$

$Spec \triangleq \land Init$
$\land \Box[Next]_{vars} \triangleq \Box(Next \lor$ UNCHANGED $vars)$
$\land \forall\, self \in validators : \mathrm{WF}_{vars}(v(self))$

$Termination \triangleq \Diamond(\forall\, self \in ProcSet : pc[self] =$ "Done"$)$

END TRANSLATION

\ * Modification History
\ * Last modified *Fri Nov* 22 11:21:27 *EST* 2019 by *isaac*
\ * Created *Tue Nov* 19 11:24:16 *EST* 2019 by *isaac*