MODULE *PConProof*

This is a specification of a variant of the classic *Paxos* consensus algorithm described in

> *AUTHOR* = "*Leslie Lamport*", *TITLE* = "The Part-Time Parliament", journal = *ACM* Transactions on Computing Systems,
> volume = 16,
> Number = 2, Month = may, Year = 1998, pages = "133–169"

This algorithm was also described without proof in *Brian Oki*'s *Ph.D.* thesis.

It describes the actions that can be performed by leaders, but does not introduce explicit leader processes. More precisely, the specification is written as if there were a separate leader for each ballot.

This variant of the classic *Paxos* algorithm is an abstraction of an algorithm that is used in

> *AUTHOR* = "*Leslie Lamport* and Dahlia *Malkhi* and *Lidong Zhou* ", *TITLE* = "Vertical *Paxos* and Primary-Backup Replication", Journal = "*ACM SIGACT News* (Distributed Computing Column)",
> editor = {*Srikanta Tirthapura and Lorenzo Alvisi*},
> *booktitle* = {*PODC*},
> publisher = {*ACM*}, *YEAR* = 2009, *PAGES* = "312–313"

and in

> Cheap paxos United States Patent 7249280 Inventors: *Lamport*, *Leslie B*.
> Massa, *Michael T*.
> Filing Date:06/18/2004

In the classic *Paxos* algorithm, the leader sends a phase $2a$ message for a ballot b and value v that instructs acceptors to vote for v in ballot b. In terms of implementing the voting algorithm of module *VoteProof*, that $2a$ message serves two functions:

- It asserts that value v is safe at ballot b, so the acceptor can vote for it without violating invariant *VInv*2
- It tells the acceptors which single safe value they can vote for in ballot b, so they can vote for that value without violating *VInv*3.

The variant of the algorithm we specify here introduces phase $1c$ messages that perform the first function. The phase $2a$ message serves only the first function, being sent only if a $1c$ message had been sent for the value.

This variant of the algorithm is useful when reconfiguration is performed by using different sets of acceptors for different ballots. The leader propagates knowledge of what values are safe at ballot b so that the acceptors in the current configuration are no longer needed to determine that information. If the ballot b leader determines that all values are safe at b, then it sends a $1c$ message for every value and sends a phase $2a$ message only when it has a value to propose. The presence of the $1c$ messages removes dependency on the acceptors of ballots numbered b or lower for progress. (If the leader determines that only a single value is safe at b, then it sends the $1c$ and $2a$ messages together.)

In the algorithm described here, we do not include reconfiguration. Therefore, the sending of a $1c$ message serves only as a precondition for the sending of a $2a$ message with that value.

Classic *Paxos* and its variants maintain consensus in the presence of omission faults–faults in which a process fails to perform some enabled action or a message that is sent fails to be received. The safety specification, which is given by the *PlusCal* code, does not require that any action need ever be performed. A process need not execute an enabled action. Receipt of a message is modeled by a process performing the action enabled by that message having been sent, so message loss is also represented by a process not performing an enabled action. Thus, failures are never mentioned in the description of the algorithm.

$$\textsc{extends}\ \mathit{Integers},\ \mathit{TLAPS}$$

The constant parameters and the set Ballots are the same as in the voting algorithm.

$$\textsc{constant}\ \mathit{Value},\ \mathit{Acceptor},\ \mathit{Quorum}$$

$$\begin{aligned}\textsc{assume}\ \mathit{QA} \triangleq\ & \land \forall Q \in \mathit{Quorum} : Q \subseteq \mathit{Acceptor}\\ & \land \forall Q1,\ Q2 \in \mathit{Quorum} : Q1 \cap Q2 \neq \{\}\end{aligned}$$

$$\mathit{Ballot} \triangleq \mathit{Nat}$$

We are going to have a leader process for each ballot and an acceptor process for each acceptor. So we can use the ballot numbers and the acceptors themselves as the identifiers for these processes, we assume that the set of ballots and the set of acceptors are disjoint. For good measure, we also assume that -1 is not an acceptor, although that is probably not necessary.

$$\textsc{assume}\ \mathit{BallotAssump} \triangleq (\mathit{Ballot} \cup \{-1\}) \cap \mathit{Acceptor} = \{\}$$

We define *None* to be an unspecified value that is not in the set *Value*.

$$\mathit{None} \triangleq \textsc{choose}\ v : v \notin \mathit{Value}$$

This is a message-passing algorithm, so we begin by defining the set Message of all possible messages. The messages are explained below with the actions that send them. A message m with $m.type =$ "1a" is called a $1a$ message, and similarly for the other message types.

$$\begin{aligned}\mathit{Message} \triangleq\ & [\mathit{type} : \{\text{“1a”}\},\ \mathit{bal} : \mathit{Ballot}]\\ \cup\ & [\mathit{type} : \{\text{“1b”}\},\ \mathit{acc} : \mathit{Acceptor},\ \mathit{bal} : \mathit{Ballot},\\ & \ \ \mathit{mbal} : \mathit{Ballot} \cup \{-1\},\ \mathit{mval} : \mathit{Value} \cup \{\mathit{None}\}]\\ \cup\ & [\mathit{type} : \{\text{“1c”}\},\ \mathit{bal} : \mathit{Ballot},\ \mathit{val} : \mathit{Value}]\\ \cup\ & [\mathit{type} : \{\text{“2a”}\},\ \mathit{bal} : \mathit{Ballot},\ \mathit{val} : \mathit{Value}]\\ \cup\ & [\mathit{type} : \{\text{“2b”}\},\ \mathit{acc} : \mathit{Acceptor},\ \mathit{bal} : \mathit{Ballot},\ \mathit{val} : \mathit{Value}]\end{aligned}$$

The algorithm is easiest to understand in terms of the set *msgs* of all messages that have ever been sent. A more accurate model would use one or more variables to represent the messages actually in transit, and it would include actions representing message loss and duplication as well as message receipt.

In the current spec, there is no need to model message loss explicitly. The safety part of the spec says only what messages may be received and does not assert that any message actually is received. Thus, there is no difference between a lost message and one that is never received. The liveness property of the spec will make it clear what messages must be received (and hence either not lost or successfully retransmitted if lost) to guarantee progress.

Another advantage of maintaining the set of all messages that have ever been sent is that it allows us to define the state function *votes* that implements the variable of the same name in the voting algorithm without having to introduce a history variable.

****

In addition to the variable *msgs*, the algorithm uses four variables whose values are arrays indexed by acceptor, where for any acceptor a :

maxBal[a] The largest ballot number in which a has participated

maxVBal[a] The largest ballot number in which a has voted, or -1 if it has never voted.

maxVVal[a] If a has voted, then this is the value it voted for in ballot *maxVBal*; otherwise it equals *None*.

As in the voting algorithm, an execution of the algorithm consists of an execution of zero or more ballots. Different ballots may be in progress concurrently, and ballots may not complete (and need not even start). A ballot b consists of the following actions (which need not all occur in the indicated order).

*Phase*1a : The leader sends a 1a message for ballot b

*Phase*1b : If $maxBal[a] < b$, an acceptor a responds to the 1a message by setting *maxBal*[a] to b and sending a 1b message to the leader containing the values of *maxVBal*[a] and *maxVVal*[a].

*Phase*1c : When the leader has received ballot-b 1b messages from a quorum, it determines some set of values that are safe at b and sends 1c messages for them.

*Phase*2a : The leader sends a 2a message for some value for which it has already sent a ballot-b 1c message.

*Phase*2b : Upon receipt of the 2a message, if $maxBal[a] \leq b$, an acceptor a sets *maxBal*[a] and *maxVBal*[a] to b, sets *maxVVal*[a] to the value in the 2a message, and votes for that value in ballot b by sending the appropriate 2b message.

Here is the *PlusCal* code for the algorithm, which we call *PCon*.

```
--algorithm PCon{
  variables maxBal  = [a ∈ Acceptor ↦ −1],
            maxVBal = [a ∈ Acceptor ↦ −1],
            maxVVal = [a ∈ Acceptor ↦ None],
            msgs = {}
  define {
    sentMsgs(t, b) ≜ {m ∈ msgs : (m.type = t) ∧ (m.bal = b)}
```

We define *ShowsSafeAt* so that *ShowsSafeAt*(Q, b, v) is true for a quorum Q iff *msgs* contain ballot-b 1b messages from the acceptors in Q show that v is safe at b.

```
    ShowsSafeAt(Q, b, v) ≜
      LET Q1b ≜ {m ∈ sentMsgs("1b", b) : m.acc ∈ Q}
      IN  ∧ ∀ a ∈ Q : ∃ m ∈ Q1b : m.acc = a
          ∧ ∨ ∀ m ∈ Q1b : m.mbal = −1
            ∨ ∃ m1c ∈ msgs :
                 ∧ m1c = [type ↦ "1c", bal ↦ m1c.bal, val ↦ v]
                 ∧ ∀ m ∈ Q1b : ∧ m1c.bal ≥ m.mbal
                               ∧ (m1c.bal = m.mbal) ⇒ (m.mval = v)

    }
```

The Actions

As before, we describe each action as a macro.

The leader for process *self* can execute a $Phase1a()$ action, which sends the ballot *self* $1a$ message.

```
macro Phase1a(){msgs := msgs ∪ {[type ↦ "1a", bal ↦ self]} ; }
```

Acceptor *self* can perform a $Phase1b(b)$ action, which is enabled iff $b > maxBal[self]$. The action sets $maxBal[self]$ to b and sends a phase $1b$ message to the leader containing the values of $maxVBal[self]$ and $maxVVal[self]$.

```
macro Phase1b(b){
  when (b > maxBal[self]) ∧ (sentMsgs("1a", b) ≠ {}) ;
  maxBal[self] := b ;
  msgs := msgs ∪ {[type  ↦ "1b", acc ↦ self, bal ↦ b,
                   mbal ↦ maxVBal[self], mval ↦ maxVVal[self]]} ;
 }
```

The ballot *self* leader can perform a $Phase1c(S)$ action, which sends a set S of $1c$ messages indicating that the value in the *val* field of each of them is safe at ballot b. In practice, S will either contain a single message, or else will have a message for each possible value, indicating that all values are safe. In the first case, the leader will immediately send a $2a$ message with the value contained in that single message. (Both logical messages will be sent in the same physical message.) In the latter case, the leader is informing the acceptors that all values are safe. (All those logical messages will, of course, be encoded in a single physical message.)

```
macro Phase1c(S){
  when ∀ v ∈ S : ∃ Q ∈ Quorum : ShowsSafeAt(Q, self, v) ;
  msgs := msgs ∪ {[type ↦ "1c", bal ↦ self, val ↦ v] : v ∈ S}
 }
```

The ballot *self* leader can perform a $Phase2a(v)$ action, sending a $2a$ message for value v, if it has not already sent a $2a$ message (for this ballot) and it has sent a ballot *self* $1c$ message with *val* field v.

```
macro Phase2a(v){
  when  ∧ sentMsgs("2a", self) = {}
        ∧ [type ↦ "1c", bal ↦ self, val ↦ v] ∈ msgs ;
  msgs := msgs ∪ {[type ↦ "2a", bal ↦ self, val ↦ v]}
 }
```

The $Phase2b(b)$ action is executed by acceptor *self* in response to a ballot-b $2a$ message. Note this action can be executed multiple times by the acceptor, but after the first one, all subsequent executions are stuttering steps that do not change the value of any variable.

```
macro Phase2b(b){
  when b ≥ maxBal[self] ;
  with (m ∈ sentMsgs("2a", b)){
    maxBal[self]  := b ;
    maxVBal[self] := b ;
    maxVVal[self] := m.val ;
```

```
      msgs := msgs ∪ {[type ↦ "2b", acc ↦ self,
                                 bal ↦ b, val ↦ m.val]}
    }
  }
```

An acceptor performs the body of its *while* loop as a single atomic action by nondeterministically choosing a ballot in which its *Phase*1*b* or *Phase*2*b* action is enabled and executing that enabled action. If no such action is enabled, the acceptor does nothing.

```
process (acceptor ∈ Acceptor){
  acc: while (TRUE){
         with (b ∈ Ballot){either Phase1b(b) or Phase2b(b)
         }
  }
}
```

The leader of a ballot nondeterministically chooses one of its actions that is enabled (and the argument for which it is enabled) and performs it atomically. It does nothing if none of its actions is enabled.

```
process (leader ∈ Ballot){
  ldr: while (TRUE){
        either Phase1a()
        or     with (S ∈ SUBSET Value){Phase1c(S)}
        or     with (v ∈ Value){Phase2a(v)}
       }
  }

}
```

The translator produces the following TLA+ specification of the algorithm. Some blank lines have been deleted.
****

```
BEGIN TRANSLATION
VARIABLES maxBal, maxVBal, maxVVal, msgs

define statement
sentMsgs(t, b) ≜ {m ∈ msgs : (m.type = t) ∧ (m.bal = b)}

ShowsSafeAt(Q, b, v) ≜
  LET Q1b ≜ {m ∈ sentMsgs("1b", b) : m.acc ∈ Q}
  IN  ∧ ∀ a ∈ Q : ∃ m ∈ Q1b : m.acc = a
      ∧ ∨ ∀ m ∈ Q1b : m.mbal = −1
        ∨ ∃ m1c ∈ msgs :
             ∧ m1c = [type ↦ "1c", bal ↦ m1c.bal, val ↦ v]
             ∧ ∀ m ∈ Q1b : ∧ m1c.bal ≥ m.mbal
                           ∧ (m1c.bal = m.mbal) ⇒ (m.mval = v)

vars ≜ ⟨maxBal, maxVBal, maxVVal, msgs⟩
```

$$ProcSet \triangleq (Acceptor) \cup (Ballot)$$

$$Init \triangleq \quad \text{Global variables}$$
$$\begin{array}{l}
\land\ maxBal = [a \in Acceptor \mapsto -1] \\
\land\ maxVBal = [a \in Acceptor \mapsto -1] \\
\land\ maxVVal = [a \in Acceptor \mapsto None] \\
\land\ msgs = \{\}
\end{array}$$

$$\begin{array}{l}
acceptor(self) \triangleq \exists\, b \in Ballot : \\
\quad \lor\ \land\ (b > maxBal[self]) \land (sentMsgs(\text{“1a”}, b) \neq \{\}) \\
\quad \ \land\ maxBal' = [maxBal \text{ EXCEPT } ![self] = b] \\
\quad \ \land\ msgs' = (msgs \cup \{[type \mapsto \text{“1b”}, acc \mapsto self, bal \mapsto b, \\
\quad \ \qquad\qquad\qquad mbal \mapsto maxVBal[self], mval \mapsto maxVVal[self]]\}) \\
\quad \ \land\ \text{UNCHANGED } \langle maxVBal, maxVVal \rangle \\
\quad \lor\ \land\ b \geq maxBal[self] \\
\quad \ \land\ \exists\, m \in sentMsgs(\text{“2a”}, b) : \\
\quad \ \qquad \land\ maxBal' = [maxBal \text{ EXCEPT } ![self] = b] \\
\quad \ \qquad \land\ maxVBal' = [maxVBal \text{ EXCEPT } ![self] = b] \\
\quad \ \qquad \land\ maxVVal' = [maxVVal \text{ EXCEPT } ![self] = m.val] \\
\quad \ \qquad \land\ msgs' = (msgs \cup \{[type \mapsto \text{“2b”}, acc \mapsto self, \\
\quad \ \qquad\qquad\qquad bal \mapsto b, val \mapsto m.val]\})
\end{array}$$

$$\begin{array}{l}
leader(self) \triangleq \land\ \lor\ \land\ msgs' = (msgs \cup \{[type \mapsto \text{“1a”}, bal \mapsto self]\}) \\
\qquad\qquad \lor\ \land\ \exists\, S \in \text{SUBSET } Value : \\
\qquad\qquad\qquad \land\ \forall\, v \in S : \exists\, Q \in Quorum : ShowsSafeAt(Q, self, v) \\
\qquad\qquad\qquad \land\ msgs' = (msgs \cup \{[type \mapsto \text{“1c”}, bal \mapsto self, val \mapsto v] : v \in S\}) \\
\qquad\qquad \lor\ \land\ \exists\, v \in Value : \\
\qquad\qquad\qquad \land\ \land\ sentMsgs(\text{“2a”}, self) = \{\} \\
\qquad\qquad\qquad \ \land\ [type \mapsto \text{“1c”}, bal \mapsto self, val \mapsto v] \in msgs \\
\qquad\qquad\qquad \land\ msgs' = (msgs \cup \{[type \mapsto \text{“2a”}, bal \mapsto self, val \mapsto v]\}) \\
\qquad\quad \land\ \text{UNCHANGED } \langle maxBal, maxVBal, maxVVal \rangle
\end{array}$$

$$\begin{array}{l}
Next \triangleq (\exists\, self \in Acceptor : acceptor(self)) \\
\qquad\quad \lor (\exists\, self \in Ballot : leader(self))
\end{array}$$

$$Spec \triangleq Init \land \Box[Next]_{vars}$$

END TRANSLATION

We now rewrite the next-state relation in a way that makes it easier to use in a proof. We start by defining the formulas representing the individual actions. We then use them to define the formula *TLANext*, which is the next-state relation we would have written had we specified the algorithm directly in TLA+ rather than in *PlusCal*.

$$\begin{array}{l}
Phase1a(self) \triangleq \\
\quad \land\ msgs' = (msgs \cup \{[type \mapsto \text{“1a”}, bal \mapsto self]\}) \\
\quad \land\ \text{UNCHANGED } \langle maxBal, maxVBal, maxVVal \rangle
\end{array}$$

$$Phase1c(self, S) \triangleq$$

$$\begin{array}{l}
\quad \land \forall v \in S : \exists Q \in Quorum : ShowsSafeAt(Q, self, v) \\
\quad \land msgs' = (msgs \cup \{[type \mapsto \text{“1c”}, bal \mapsto self, val \mapsto v] : v \in S\}) \\
\quad \land \textsc{unchanged}\ \langle maxBal, maxVBal, maxVVal \rangle
\end{array}$$

$$\begin{array}{l}
Phase2a(self, v) \triangleq \\
\quad \land sentMsgs(\text{“2a”}, self) = \{\} \\
\quad \land [type \mapsto \text{“1c”}, bal \mapsto self, val \mapsto v] \in msgs \\
\quad \land msgs' = (msgs \cup \{[type \mapsto \text{“2a”}, bal \mapsto self, val \mapsto v]\}) \\
\quad \land \textsc{unchanged}\ \langle maxBal, maxVBal, maxVVal \rangle
\end{array}$$

$$\begin{array}{l}
Phase1b(self, b) \triangleq \\
\quad \land b > maxBal[self] \\
\quad \land sentMsgs(\text{“1a”}, b) \neq \{\} \\
\quad \land maxBal' = [maxBal\ \textsc{except}\ ![self] = b] \\
\quad \land msgs' = msgs \cup \{[type \mapsto \text{“1b”}, acc \mapsto self, bal \mapsto b, \\
\qquad\qquad\qquad\qquad mbal \mapsto maxVBal[self], mval \mapsto maxVVal[self]]\} \\
\quad \land \textsc{unchanged}\ \langle maxVBal, maxVVal \rangle
\end{array}$$

$$\begin{array}{l}
Phase2b(self, b) \triangleq \\
\quad \land b \geq maxBal[self] \\
\quad \land \exists m \in sentMsgs(\text{“2a”}, b) : \\
\qquad \land maxBal' = [maxBal\ \textsc{except}\ ![self] = b] \\
\qquad \land maxVBal' = [maxVBal\ \textsc{except}\ ![self] = b] \\
\qquad \land maxVVal' = [maxVVal\ \textsc{except}\ ![self] = m.val] \\
\qquad \land msgs' = (msgs \cup \{[type \mapsto \text{“2b”}, acc \mapsto self, \\
\qquad\qquad\qquad\qquad bal \mapsto b, val \mapsto m.val]\})
\end{array}$$

$$\begin{array}{l}
TLANext \triangleq \\
\quad \lor \exists self \in Acceptor : \\
\qquad \exists b \in Ballot : \lor Phase1b(self, b) \\
\qquad\qquad\qquad\quad\ \lor Phase2b(self, b) \\
\quad \lor \exists self \in Ballot : \\
\qquad \lor Phase1a(self) \\
\qquad \lor \exists S \in \textsc{subset}\ Value : Phase1c(self, S) \\
\qquad \lor \exists v \in Value : Phase2a(self, v)
\end{array}$$

The following theorem specifies the relation between the next-state relation *Next* obtained by translating the *PlusCal* code and the next-state relation *TLANext*.

$$\begin{array}{l}
\textsc{theorem}\ NextDef \triangleq (Next \equiv TLANext) \\
\langle 1 \rangle 2.\ \textsc{assume new}\ self \in Acceptor \\
\qquad \textsc{prove}\ \ acceptor(self) \equiv TLANext!1!(self) \\
\quad \textsc{by}\ \langle 1 \rangle 2, BallotAssump\ \textsc{def}\ acceptor, ProcSet, Phase1b, Phase2b \\
\langle 1 \rangle 3.\ \textsc{assume new}\ self \in Ballot \\
\qquad \textsc{prove}\ \ leader(self) \equiv TLANext!2!(self) \\
\quad \textsc{by}\ \langle 1 \rangle 3, BallotAssump\ \textsc{def}\ leader, ProcSet, Phase1a, Phase1c, Phase2a \\
\langle 1 \rangle 4.\ \textsc{qed}
\end{array}$$

BY $\langle 1 \rangle 2$, $\langle 1 \rangle 3$ DEF *Next*, *TLANext*

The type invariant.

$$
\begin{aligned}
TypeOK \triangleq\ & \land\ maxBal \in [Acceptor \rightarrow Ballot \cup \{-1\}] \\
& \land\ maxVBal \in [Acceptor \rightarrow Ballot \cup \{-1\}] \\
& \land\ maxVVal \in [Acceptor \rightarrow Value \cup \{None\}] \\
& \land\ msgs \subseteq Message
\end{aligned}
$$

Here is the definition of the state-function *chosen* that implements the state-function of the same name in the voting algorithm.

$$
\begin{aligned}
chosen \triangleq\ & \{v \in Value : \exists\, Q \in Quorum,\ b \in Ballot : \\
& \qquad \forall\, a \in Q : \exists\, m \in msgs : \land\ m.type = \text{“2b”} \\
& \qquad\qquad\qquad\qquad\qquad\qquad \land\ m.acc = a \\
& \qquad\qquad\qquad\qquad\qquad\qquad \land\ m.bal = b \\
& \qquad\qquad\qquad\qquad\qquad\qquad \land\ m.val = v\}
\end{aligned}
$$

We now define the refinement mapping under which this algorithm implements the specification in module *Voting*.

As we observed, votes are registered by sending phase $2b$ messages. So the array *votes* describing the votes cast by the acceptors is defined as follows.

$$
\begin{aligned}
votes \triangleq\ & [a \in Acceptor \mapsto \\
& \quad \{\langle m.bal,\ m.val \rangle : m \in \{mm \in msgs : \land\ mm.type = \text{“2b”} \\
& \qquad\qquad\qquad\qquad\qquad\qquad\qquad \land\ mm.acc = a\}\}]
\end{aligned}
$$

We now instantiate module *Voting*, substituting:

- The constants *Value*, *Acceptor*, and *Quorum* declared in this module for the corresponding constants of that module *Voting*.
- The variable *maxBal* and the defined state function *votes* for the correspondingly-named variables of module *Voting*.

$V \triangleq$ INSTANCE *VoteProof*

We now define *PInv* to be what I believe to be an inductive invariant and assert the theorems for proving that this algorithm implements the voting algorithm under the refinement mapping specified by the INSTANCE statement. Whether *PInv* really is an inductive invariant will be determined only by a rigorous proof.

$$
\begin{aligned}
PAccInv \triangleq\ & \forall\, a \in Acceptor : \\
& \quad \land\ maxBal[a] \geq maxVBal[a] \\
& \quad \land\ \forall\, b \in (maxVBal[a] + 1) \,..\, (maxBal[a] - 1) : V!DidNotVoteIn(a,\ b) \\
& \quad \land\ (maxVBal[a] \neq -1) \Rightarrow V!VotedFor(a,\ maxVBal[a],\ maxVVal[a])
\end{aligned}
$$

$$
\begin{aligned}
P1bInv \triangleq\ & \forall\, m \in msgs : \\
& \quad (m.type = \text{“1b”}) \Rightarrow \\
& \qquad \land\ (maxBal[m.acc] \geq m.bal) \land (m.bal > m.mbal) \\
& \qquad \land\ \forall\, b \in (m.mbal + 1) \,..\, (m.bal - 1) : V!DidNotVoteIn(m.acc,\ b)
\end{aligned}
$$

$P1cInv \triangleq \forall m \in msgs : (m.type = \text{“1c”}) \Rightarrow V!SafeAt(m.bal, m.val)$

$$
\begin{array}{l}
P2aInv \triangleq \forall m \in msgs : \\
\qquad (m.type = \text{“2a”}) \Rightarrow \exists m1c \in msgs : \begin{array}[t]{l} \land m1c.type = \text{“1c”} \\ \land m1c.bal = m.bal \\ \land m1c.val = m.val \end{array}
\end{array}
$$

The following theorem is interesting in its own right. It essentially asserts the correctness of the definition of *ShowsSafeAt*.

$$
\begin{array}{l}
\text{THEOREM } PT1 \triangleq TypeOK \land P1bInv \land P1cInv \Rightarrow \\
\qquad \forall Q \in Quorum, b \in Ballot, v \in Value : \\
\qquad\quad ShowsSafeAt(Q, b, v) \Rightarrow V!SafeAt(b, v)
\end{array}
$$

$PInv \triangleq TypeOK \land PAccInv \land P1bInv \land P1cInv \land P2aInv$

THEOREM $Invariance \triangleq Spec \Rightarrow \Box PInv$

THEOREM $Implementation \triangleq Spec \Rightarrow V!Spec$

The following result shows that our definition of *chosen* is the correct one, because it implements the state-function *chosen* of the voting algorithm.

THEOREM $Spec \Rightarrow \Box(chosen = V!chosen)$

The four theorems above have been checked by *TLC* for a model with 3 acceptors, 2 values, and 3 ballot numbers. Theorem $PT1$ was checked as an invariant, therefore checking only that it is true for all reachable states. This model is large enough that it would most likely have revealed any “coding” errors in the algorithm. We believe that the algorithm is well-enough understood that it is unlikely to contain any fundamental errors.

* Modification History
* Last modified Sat *Nov* 16 22:19:39 *CST* 2019 by *hengxin*
* Last modified *Tue Feb* 08 12:09:41 *PST* 2011 by lamport