This is a high-level consensus algorithm in which a set of processes called acceptors cooperatively choose a value. The algorithm uses numbered ballots, where a ballot is a round of voting. Acceptors cast votes in ballots, casting at most one vote per ballot. A value is chosen when a large enough set of acceptors, called a quorum, have all voted for the same value in the same ballot.

Ballots are not executed in order. Different acceptors may be concurrently performing actions for different ballots.

EXTENDS Integers, NaturalsInduction, FiniteSets, FiniteSetTheorems, WellFoundedInduction, TLC, TLAPS

CONSTANT Value, Acceptor, Quorum

As in module Consensus, the set of choosable values.
The set of all acceptors.
The set of all quorums.

The following assumption asserts that a quorum is a set of acceptors, and the fundamental assumption we make about quorums: any two quorums have a non-empty intersection.

ASSUME QA ≜ ∀Q ∈ Quorum : Q ⊆ Acceptor
∧ ∀Q1, Q2 ∈ Quorum : Q1 ∩ Q2 ≠ {}

THEOREM QuorumNonEmpty ≜ ∀Q ∈ Quorum : Q ≠ {}

Proof by QA

Ballot is the set of all ballot numbers. For simplicity, we let it be the set of natural numbers. However, we write Ballot for that set to make it clear what the function of those natural numbers are.

The algorithm and its refinements work with Ballot any set with minimal element 0, −1 not an element of Ballot, and a well-founded total order < on Ballot ∪ {−1} with minimal element −1, and 0 < b for all non-zero b in Ballot. In the proof, any set of the form i..j must be replaced by the set of all elements b in Ballot ∪ {−1} with i ≤ b ≤ j, and i.(j−1) by the set of such b with i ≤ b < j.

Ballot ≜ Nat

In the algorithm, each acceptor can cast one or more votes, where each vote cast by an acceptor has the form ⟨b, v⟩ indicating that the acceptor has voted for value v in ballot b. A value is chosen if a quorum of acceptors have voted for it in the same ballot.

The algorithm uses two variables, votes and maxBal, both arrays indexed by acceptor. Their meanings are:

votes[a] — The set of votes cast by acceptor a.

maxBal[a] — The number of the highest-numbered ballot in which a has cast a vote, or −1 if it has not yet voted.

The algorithm does not let acceptor a vote in any ballot less than maxBal[a].
We specify our algorithm by the following PlusCal algorithm. The specification \( \text{Spec} \) defined by this algorithm describes only the safety properties of the algorithm. In other words, it specifies what steps the algorithm may take. It does not require that any (non-stuttering) steps be taken. We prove that this specification \( \text{Spec} \) implements the specification \( \text{Spec} \) of module Consensus under a refinement mapping defined below. This shows that the safety properties of the voting algorithm (and hence the algorithm with additional liveness requirements) imply the safety properties of the Consensus specification. Liveness is discussed later.

\[\text{**********************************}\]

\[\text{----algorithm Voting}\{\text{variables votes } = [a \in \text{Acceptor} \mapsto \{\}], maxBal = [a \in \text{Acceptor} \mapsto -1]; \text{define }{\}
\]

We now define the operator \( \text{SafeAt} \) so \( \text{SafeAt}(b, v) \) is function of the state that equals \( \text{true} \) if no value other than \( v \) has been chosen or can ever be chosen in the future (because the values of the variables \( \text{votes} \) and \( \text{maxBal} \) are such that the algorithm does not allow enough acceptors to vote for it). We say that value \( v \) is safe at ballot number \( b \) iff \( \text{Safe}(b, v) \) is true. We define \( \text{Safe} \) in terms of the following two operators.

Note: This definition is weaker than would be necessary to allow a refinement of ordinary Paxos consensus, since it allows different quorums to “cooperate” in determining safety at \( b \). This is used in algorithms like Vertical Paxos that are designed to allow reconfiguration within a single consensus instance, but not in ordinary Paxos. See

\[\text{AUTHOR} = \text{“Leslie Lamport and Dahlia Malkhi and Lidong Zhou”},\]
\[\text{TITLE} = \text{“Vertical Paxos and Primary-Backup Replication”},\]
\[\text{Journal} = \text{“ACM SIGACT News (Distributed Computing Column)”},\]
\[\text{booktitle} = \{\text{Srikanta Tirthapura and Lorenzo Alvisi}\},\]
\[\text{publisher} = \{\text{ACM}\}, \text{YEAR} = 2009, \text{PAGES} = \text{“312–313”}\]

\[\text{VotedFor}(a, b, v) \triangleq \langle b, v \rangle \in \text{votes}[a]\]

\[\text{DidNotVoteIn}(a, b) \triangleq \forall v \in \text{Value} : \neg \text{VotedFor}(a, b, v)\]

We now define \( \text{SafeAt} \). We define it recursively. The nicest definition is

\[\text{RECURSIVE } \text{SafeAt}(..., ...)\]
\[\text{SafeAt}(b, v) \triangleq\]
\[\forall b = 0\]
\[\forall \exists Q \in \text{Quorum} :\]
\[\land \forall a \in Q : \text{maxBal}[a] > b\]
\[\land \exists c \in -1 \ldots (b-1) :\]
\[\land (c \neq -1) \Rightarrow \land \text{SafeAt}(c, v)\]
\[\land \forall a \in Q : \forall w \in \text{Value} :\]
\[\text{VotedFor}(a, c, w) \Rightarrow (w = v)\]
\[\land \forall d \in (c+1) \ldots (b-1), a \in Q : \text{DidNotVoteIn}(a, d)\]

However, TLAPS does not currently support recursive operator definitions. We therefore define it as follows using a recursive function definition.

\[\text{SafeAt}(b, v) \triangleq\]
\[\text{LET } \text{SA}[bb \in \text{Ballot}] \triangleq\]

2
This recursively defines $SA[bb]$ to equal $SafeAt(bb, v)$.

\[
\begin{align*}
\forall bb & = 0 \\
\forall \exists Q \in Quorum : \\
\quad & \forall a \in Q : maxBal[a] \geq bb \\
\quad & \exists c \in -1 \ldots (bb - 1) : \\
\quad & (c \neq -1) \Rightarrow \forall c \in Q : \\
\quad & \forall w \in Value : VotedFor(a, c, w) \Rightarrow (w = v) \\
\quad & \forall d \in (c + 1) \ldots (bb - 1), a \in Q : DidNotVoteIn(a, d)
\end{align*}
\]

There are two possible actions that an acceptor can perform, each defined by a macro. In these macros, self is the acceptor that is to perform the action. The first action, IncreaseMaxBal(b) allows acceptor self to set $maxBal[self]$ to $b$ if $b$ is greater than the current value of $maxBal[self]$.

macro IncreaseMaxBal( b ) {
  when $b > maxBal[self]$ ;
  $maxBal[self] := b$
}

Action VoteFor(b, v) allows acceptor self to vote for value $v$ in ballot $b$ if its when condition is satisfied.

macro VoteFor( b, v ) {
  when $\land maxBal[self] \leq b$
  $\land DidNotVoteIn(self, b)$ 
  $\land \forall p \in Acceptor \setminus \{self\} :$
  $\forall w \in Value : VotedFor(p, b, w) \Rightarrow (w = v)$  
  $\land SafeAt(b, v)$ ;
  $votes[self] := votes[self] \cup \{(b, v)\}$ ;
  $maxBal[self] := b$
}

The following process declaration asserts that every process self in the set Acceptor executes its body, which loops forever nondeterministically choosing a Ballot $b$ and executing either an IncreaseMaxBal(b) action or nondeterministically choosing a value $v$ and executing a VoteFor(b, v) action. The single label indicates that an entire execution of the body of the while loop is performed as a single atomic action.

From this intuitive description of the process declaration, one might think that a process could be deadlocked by choosing a ballot $b$ in which neither an IncreaseMaxBal(b) action nor any VoteFor(b, v) action is enabled. An examination of the TLA+ translation (and an elementary knowledge of the meaning of existential quantification) shows that this is not the case. You can think of all possible choices of $b$ and of $v$ being examined simultaneously, and one of the choices for which a step is possible being made.

process ( acceptor \in Acceptor ) { 
  acc : while ( TRUE ) { 
    with ( b \in Ballot ) { 

either IncreaseMaxBal(b) or with ( v ∈ Value ) { VoteFor(b, v) }

The following is the TLA+ specification produced by the translation. Blank lines, produced by the translation because of the comments, have been deleted.

BEGIN TRANSLATION

VARIABLES votes, maxBal
define statement
VotedFor(a, b, v) ∆= ⟨b, v⟩ ∈ votes[a]

DidNotVoteIn(a, b) ∆= ∀v ∈ Value : ¬VotedFor(a, b, v)

SafeAt(b, v) ∆=
LET SA[bb ∈ Ballot] ∆=
  ∨ bb = 0
  ∨ ∃ Q ∈ Quorum :
    ∧ ∀ a ∈ Q : maxBal[a] ≥ bb
    ∧ ∃ c ∈ −1..(bb − 1) :
      ∧ (c ≠ −1) ⇒ ∧ SA[c]
      ∧ ∀ a ∈ Q :
        ∀ w ∈ Value :
          VotedFor(a, c, w) ⇒ (w = v)
          ∧ ∀ d ∈ (c + 1)..(bb − 1), a ∈ Q : DidNotVoteIn(a, d)

in SA[b]

vars ∆= ⟨votes, maxBal⟩

ProcSet ∆= (Acceptor)

Init ∆= Global variables
∧ votes = [a ∈ Acceptor ↦ {}]
∧ maxBal = [a ∈ Acceptor ↦ −1]

acceptor(self) ∆= ∃ b ∈ Ballot :
  ∨ ∧ b > maxBal[self]
  ∧ maxBal’ = [maxBal except ![self] = b]
  ∧ UNCHANGED votes
  ∨ ∧ ∀ v ∈ Value :
    ∧ ∃ v ∈ Value :
      VotedFor(p, b, w) ⇒ (w = v)
\[ SafeAt(b, v) \wedge votes' = [votes \text{ EXCEPT } ![self] = votes \cup \{(b, v)\}] \wedge maxBal' = [maxBal \text{ EXCEPT } ![self] = b] \]

Next \(\triangleq (\exists \text{ self } \in \text{ Acceptor} : \text{ acceptor}(\text{self}))\)

Spec \(\triangleq \text{ Init} \wedge \square[Next]_{\text{vars}}\)

END TRANSLATION

To reason about a recursively-defined operator, one must prove a theorem about it. In particular, to reason about \(SafeAt\), we need to prove that \(SafeAt(b, v)\) equals the right-hand side of its definition, for \(b \in \text{Ballot}\) and \(v \in \text{Value}\). This is not automatically true for a recursive definition. For example, from the recursive definition

\[ \text{Silly}[n \in \text{Nat}] \triangleq \text{ choose } v : v \neq \text{Silly}[n] \]

we cannot deduce that

\[ \text{Silly}[42] = \text{ choose } v : v \neq \text{Silly}[42] \]

(From that, we could easily deduce \(\text{Silly}[42] \neq \text{Silly}[42]\).)

Here is the theorem that essentially asserts that \(SafeAt(b, v)\) equals the right-hand side of its definition.

**THEOREM** \(SafeAtProp \triangleq \forall b \in \text{Ballot}, v \in \text{Value} :\)

\[ SafeAt(b, v) \equiv \]

\[ \forall b = 0 \]

\[ \forall \exists Q \in \text{Quorum} : \]

\[ \wedge \forall a \in Q : \text{maxBal}[a] \geq b \]

\[ \wedge \exists c \in -1 \ldots (b - 1) : \]

\[ \wedge (c \neq -1) \Rightarrow \wedge \text{SafeAt}(c, v) \]

\[ \forall a \in Q : \]

\[ \forall w \in \text{Value} : \]

\[ \text{VotedFor}(a, c, w) \Rightarrow (w = v) \]

\[ \wedge \forall d \in (c + 1) \ldots (b - 1), a \in Q : \text{DidNotVoteIn}(a, d) \]

(1)\text{. SUFFICES ASSUME NEW } v \in \text{Value}

PROVE \(\forall b \in \text{Ballot} : SafeAtProp!(b, v)\)

BY Zenon

(1) USE \text{ DEF Ballot}

(1) DEFINE \(\text{Def}(SA, bb) \triangleq \)

\[ \forall bb = 0 \]

\[ \forall \exists Q \in \text{Quorum} : \]

\[ \wedge \forall a \in Q : \text{maxBal}[a] \geq bb \]

\[ \wedge \exists c \in -1 \ldots (bb - 1) : \]

\[ \wedge (c \neq -1) \Rightarrow \wedge SA[c] \]

\[ \forall a \in Q : \]

\[ \forall w \in \text{Value} : \]

END
\[ \text{VotedFor}(a, c, w) \Rightarrow (w = v) \]
\[ \land \forall d \in (c + 1) \ldots (bb - 1), a \in Q : \text{DidNotVoteIn}(a, d) \]
\[ \text{SA}[bb \in \text{Ballot}] \triangleq \text{Def}(\text{SA}, bb) \]
\[ \text{BY} \ \text{DEF} \ \text{SafeAt} \]
\[ \langle 1 \rangle 2. \ \forall b : \text{SafeAt}(b, v) = \text{SA}[b] \]
\[ \text{BY} \ \langle 1 \rangle 2. \ \text{SafeAtProp} \]
\[ \langle 1 \rangle 3. \ \text{ASSUME NEW} n \in \text{Nat}, \text{NEW} g, \text{NEW} h, \]
\[ \land \forall i \in 0 \ldots (n - 1) : g[i] = h[i] \]
\[ \text{PROVE} \ \text{Def}(g, n) = \text{Def}(h, n) \]
\[ \text{BY} \ \langle 1 \rangle 3. \ \text{RecursiveFcnOfNat}, \text{Isa} \]
\[ \langle 1 \rangle 4. \ \text{SA} = [b \in \text{Ballot} \mapsto \text{Def}((\text{SA}, b))] \]
\[ \text{HIDE} \ \text{Def} \]
\[ \text{BY} \ \langle 1 \rangle 4. \ \text{SafeAtProp} \]
\[ \langle 1 \rangle 5. \ \forall b \in \text{Ballot} : \text{SA}[b] = \text{Def}(\text{SA}, b) \]
\[ \text{HIDE} \ \text{Def} \]
\[ \text{BY} \ \langle 1 \rangle 5. \ \text{SafeAtProp} \]
\[ \langle 1 \rangle 6. \ \text{QED} \]
\[ \text{BY} \ \langle 1 \rangle 2, \langle 1 \rangle 5, \text{Zenon DEF SafeAt} \]

We now define \text{TypeOK} to be the type-correctness invariant.
\[ \text{TypeOK} \triangleq \land \ \text{votes} \in [\text{Accept} \rightarrow \text{SUBSET} (\text{Ballot} \times \text{Value})] \]
\[ \land \ \text{maxBal} \in [\text{Accept} \rightarrow \text{Ballot} \cup \{-1\}] \]

We now define \text{chosen} to be the state function so that the algorithm specified by formula \text{Spec} conjoined with the liveness requirements described below implements the algorithm of module \text{Consensus} (satisfies the specification \text{LiveSpec} of that module) under a refinement mapping that substitutes this state function \text{chosen} for the variable \text{chosen} of module \text{Consensus}. The definition uses the following one, which defines \text{ChosenIn}(b, v) to be true iff a quorum of acceptors have all voted for \text{v} in ballot \text{b}.

\[ \text{ChosenIn}(b, v) \triangleq \exists Q \in \text{Quorum} : \forall a \in Q : \text{VotedFor}(a, b, v) \]
\[ \text{chosen} \triangleq \{ v \in \text{Value} : \exists b \in \text{Ballot} : \text{ChosenIn}(b, v) \} \]

The following lemma is used for reasoning about the operator \text{SafeAt}. It is proved from \text{SafeAtProp} by induction.

\begin{lemma} \text{SafeLemma} \triangleq \text{TypeOK} \Rightarrow \]
\[ \forall b \in \text{Ballot} : \forall v \in \text{Value} : \text{SafeAt}(b, v) \Rightarrow \]
\[ \forall c \in 0 \ldots (b - 1) : \exists Q \in \text{Quorum} : \forall a \in Q : \land \text{maxBal}[a] \geq c \]
\[ \land \lor \text{DidNotVoteIn}(a, c) \]
\end{lemma}
\[ \forall \text{VotedFor}(a, c, v) \]

(1) SUFFICES ASSUME \text{TypeOK}
PROVE \text{SafeLemma!2}

OBSVIOUS
(1) DEFINE \( P(b) \triangleq \forall c \in 0..b : \text{SafeLemma!2}(c) \)
(1) USE \text{DEF Ballot}
(1)1. \( P(0) \)
OBSVIOUS
(1)2. ASSUME NEW \( b \in \text{Ballot} \), \( P(b) \)
PROVE \( P(b + 1) \)
(2)1. \( \land b + 1 \in \text{Ballot} \setminus \{0\} \)
\( \land (b + 1) - 1 = b \)
OBSVIOUS
(2)2. \( 0..(b + 1) = (0..b) \cup \{b + 1\} \)
OBSVIOUS
(2)3. SUFFICES ASSUME NEW \( v \in \text{Value} \),
\[ \text{SafeAt}(b + 1, v), \]
NEW \( c \in 0..b \)
PROVE \( \exists Q \in \text{Quorum} : \)
\( \forall a \in Q : \land \text{maxBal}[a] \geq c \)
\( \land \lor \text{DidNotVoteIn}(a, c) \)
\( \lor \text{VotedFor}(a, c, v) \)

BY (1)2
(2)4. PICK \( Q \in \text{Quorum} : \)
\( \land \forall a \in Q : \land \text{maxBal}[a] \geq (b + 1) \)
\( \land \exists cc \in -1..b : \)
\( \land (cc \neq -1) \Rightarrow \land \text{SafeAt}(cc, v) \)
\( \land \forall a \in Q : \)
\( \forall w \in \text{Value} : \)
\[ \text{VotedFor}(a, cc, v) \Rightarrow (w = v) \]
\( \land \forall d \in (cc + 1) .. b, a \in Q : \text{DidNotVoteIn}(a, d) \)

BY \text{SafeAtProp}, (2)3, (2)1, Zenon
(2)5. PICK \( cc \in -1..b : \)
\( \land (cc \neq -1) \Rightarrow \land \text{SafeAt}(cc, v) \)
\( \land \forall a \in Q : \)
\( \forall w \in \text{Value} : \)
\[ \text{VotedFor}(a, cc, v) \Rightarrow (w = v) \]
\( \land \forall d \in (cc + 1) .. b, a \in Q : \text{DidNotVoteIn}(a, d) \)

BY (2)4
(2)6.CASE \( c > cc \)
BY (2)4, (2)5, (2)6, QA DEF \text{TypeOK}
(2)7.CASE \( c = cc \)
(3)2. \( \forall a \in Q : \text{maxBal}[a] \in \text{Ballot} \cup \{-1\} \)
BY QA DEF \text{TypeOK}
(3)3. \( \forall a \in Q : \text{maxBal}[a] \geq c \)
We now define the invariant that is used to prove the correctness of our algorithm—meaning that specification Spec implements specification Spec of module Consensus under our refinement mapping. Correctness of the voting algorithm follows from the following three invariants:

VInv1: In any ballot, an acceptor can vote for at most one value.
VInv2: An acceptor can vote for a value v in ballot b iff v is safe at b.
VInv3: Two different acceptors cannot vote for different values in the same ballot.

Their precise definitions are as follows.

\[
\begin{align*}
VInv_1 \triangleq & \forall a \in \text{Acceptor}, b \in \text{Ballot}, v, w \in \text{Value} : \\
& \text{VotedFor}(a, b, v) \land \text{VotedFor}(a, b, w) \Rightarrow (v = w) \\
VInv_2 \triangleq & \forall a \in \text{Acceptor}, b \in \text{Ballot}, v \in \text{Value} : \\
& \text{VotedFor}(a, b, v) \Rightarrow \text{SafeAt}(b, v) \\
VInv_3 \triangleq & \forall a_1, a_2 \in \text{Acceptor}, b \in \text{Ballot}, v_1, v_2 \in \text{Value} : \\
& \text{VotedFor}(a_1, b, v_1) \land \text{VotedFor}(a_2, b, v_2) \Rightarrow (v_1 = v_2)
\end{align*}
\]

It is obvious that VInv3 implies VInv1—a fact that we now let TLAPS prove as a little check that we haven’t made a mistake in our definitions. (Actually, we used TLC to check everything before attempting any proofs.) We define VInv1 separately because VInv3 is not needed for proving safety, only for liveness.

THEOREM VInv3 \(\Rightarrow\) VInv1
BY DEF VInv1, VInv3

The following lemma proves that SafeAt(b, v) implies that no value other than v can have been chosen in any ballot numbered less than b. The fact that it also implies that no value other than v can ever be chosen in the future follows from this and the fact that SafeAt(b, v) is stable—meaning that once it becomes true, it remains true forever. The stability of SafeAt(b, v) is proved as step (1)6 of theorem InductiveInvariance below.

This lemma is used only in the proof of theorem VT1 below.

LEMMA VT0 \(\triangleq\) \(\land\) TypeOK
\[ \forall \! v, \, w \in \text{Value}, \, b, \, c \in \text{Ballot} : \]
\[ (b > c) \land \text{SafeAt}(b, \, v) \land \text{ChosenIn}(c, \, w) \Rightarrow (v = w) \]

\( \langle 1 \rangle \) **Suffices Assume** \( \text{TypeOK}, \, \text{Vin}1, \, \text{Vin}2, \)
\newl
\[ \text{NEW} \, v \in \text{Value}, \, \text{NEW} \, w \in \text{Value} \]
\[ \text{PROVE} \, \forall \, b, \, c \in \text{Ballot} : \]
\[ (b > c) \land \text{SafeAt}(b, \, v) \land \text{ChosenIn}(c, \, w) \Rightarrow (v = w) \]

\( \langle 2 \rangle \) **Obvious**

\( \langle 1 \rangle \) **P(0)**

\( \langle 1 \rangle \) **Use Def** \( \text{Ballot} \)

\( \langle 1 \rangle \) **1. Assume New** \( b \in \text{Ballot}, \, \forall \, i \in 0 \ldots (b - 1) : P(i) \)

\( \langle 2 \rangle \) **1. Case** \( b = 0 \)

\( \langle 2 \rangle \) **2. Case** \( b \neq 0 \)

\( \langle 3 \rangle \) **1. Suffices Assume New** \( c \in \text{Ballot}, \, b > c, \, \text{SafeAt}(b, \, v), \, \text{ChosenIn}(c, \, w) \)

\( \langle 3 \rangle \) **Obvious**

\( \langle 3 \rangle \) **2. Pick** \( Q \in \text{Quorum} : \forall \, a \in Q : \text{VotedFor}(a, \, c, \, w) \)

\( \langle 3 \rangle \) **1. Def** \( \text{ChosenIn} \)

\( \langle 3 \rangle \) **3. Pick** \( QQ \in \text{Quorum}, \)
\[ d \in -1 \ldots (b - 1) : \]
\[ \land \,(d \neq -1) \Rightarrow \land \text{SafeAt}(d, \, v) \]
\[ \land \forall \, a \in QQ : \]
\[ \forall \, x \in \text{Value} : \]
\[ \text{VotedFor}(a, \, d, \, x) \Rightarrow (x = v) \]

\( \langle 3 \rangle \) **3. Pick** \( \text{aa} \in QQ \cap Q : \text{TRUE} \)

\( \langle 3 \rangle \) **4. c \leq d \)

\( \langle 3 \rangle \) **5. Case** \( c = d \)

\( \langle 3 \rangle \) **6. Case** \( d > c \)

\( \langle 3 \rangle \) **7. QED \)

\( \langle 2 \rangle \) **QED** \( \langle 1 \rangle 2, \, 1 \)

\( \langle 1 \rangle 3. \, \forall \, b \in \text{Ballot} : P(b) \)
The following theorem asserts that the invariance of TypeOK, VInv1, and VInv2 implies that the algorithm satisfies the basic consensus property that at most one value is chosen (at any time). If you can prove it, then you understand why the Paxos consensus algorithm allows only a single value to be chosen. Note that VInv3 is not needed to prove this property.

**THEOREM VT1** \[ \triangleq \land TypeOK \land VInv1 \land VInv2 \Rightarrow \forall v, w : (v \in chosen) \land (w \in chosen) \Rightarrow (v = w) \]

**PROVE** \[ v = w \]

**OBVIOUS**

(1)2. \( v \in Value \land w \in Value \)

by (1)1 \ DEF chosen

(1)3. \text{PICK} \( b \in Ballot, c \in Ballot : ChosenIn(b, v) \land ChosenIn(c, w) \)

by (1)1 \ DEF chosen

(1)4. \text{PICK} \( Q \in Quorum, R \in Quorum : \land \forall a \in Q : VotedFor(a, b, v) \land \forall a \in R : VotedFor(a, c, w) \)

by (1)3 \ DEF ChosenIn

(1)5. \text{PICK} \( av \in Q, aw \in R : \land VotedFor(av, b, v) \land VotedFor(aw, c, w) \)

by (1)4, QuorumNonEmpty

(1)6. \text{SafeAt}(b, v) \land \text{SafeAt}(c, w)

by (1)1, (1)2, (1)5, QA \ DEF VInv2

(1)7. \text{CASE} \( b = c \)

(2) \text{PICK} \( a \in Q \cap R : \text{true} \)

by QA

(2)1. \( \land VotedFor(a, b, v) \land VotedFor(a, c, w) \)

by (1)4

(2)2. QED

by (1)1, (1)2, (1)7, (2)1, QA \ DEF VInv1

(1)8. \text{CASE} \( b > c \)

by (1)1, (1)6, (1)3, (1)8, VT0, (1)2

(1)9. \text{CASE} \( c > b \)

by (1)1, (1)6, (1)3, (1)9, VT0, (1)2

(1)10. QED

by (1)7, (1)8, (1)9 \ DEF Ballot
The rest of the proof uses only the primed version of $VT_1$—that is, the theorem whose statement is $VT_1'$. (Remember that $VT_1$ names the formula being asserted by the theorem we call $VT_1$.) The formula $VT_1'$ asserts that $VT_1$ is true in the second state of any transition (pair of states). We derive that theorem from $VT_1$ by simple temporal logic, and similarly for $VT_0$ and $SafeAtProp$.

**THEOREM SafeAtPropPrime**
\[
\forall b \in \text{Ballot}, v \in \text{Value} : \\
SafeAt(b, v)' \equiv \\
\forall b = 0 \\
\exists Q \in \text{Quorum} : \\
\forall a \in Q : \maxBal'[a] \geq b \\
\exists c \in -1..(b-1) : \\
(c \neq -1) \Rightarrow \forall a \in Q : \\
\forall w \in \text{Value} : \\
VotedFor(a, c, w)' \Rightarrow (w = v) \\
\langle 1 \rangle 1. \text{SafeAtProp'} \text{ BY SafeAtProp, PTL} \\
\langle 1 \rangle . \text{QED} \text{ BY } \langle 1 \rangle 1
\]

**LEMMA VT0Prime**
\[
\forall v, w \in \text{Value}, b, c \in \text{Ballot} : \\
(b > c) \land SafeAt(b, v)' \land ChosenIn(c, w)' \Rightarrow (v = w) \\
\langle 1 \rangle 1. \text{VT0'} \text{ BY VT0, PTL} \\
\langle 1 \rangle . \text{QED} \text{ BY } \langle 1 \rangle 1
\]

**THEOREM VT1Prime**
\[
\forall v, w : \\
(v \in \text{chosen'}) \land (w \in \text{chosen'}) \Rightarrow (v = w) \\
\langle 1 \rangle 1. \text{VT1'} \text{ BY VT1, PTL} \\
\langle 1 \rangle . \text{QED} \text{ BY } \langle 1 \rangle 1
\]

The invariance of $VT1'$ depends on $SafeAt(b, v)$ being stable, meaning that once it becomes true it remains true forever. Stability of $SafeAt(b, v)$ depends on the following invariant.

$VInv4 \equiv \forall a \in \text{Acceptor}, b \in \text{Ballot} : \\
\maxBal[a] < b \Rightarrow DidNotVoteIn(a, b)$

The inductive invariant that we use to prove correctness of this algorithm is $VInv$, defined as follows.

$VInv \equiv TypeOK \land VInv2 \land VInv3 \land VInv4$
To simplify reasoning about the next-state action $\text{Next}$, we want to express it in a more convenient form. This is done by lemma $\text{NextDef}$ below, which shows that $\text{Next}$ equals an action defined in terms of the following subactions.

$\text{IncreaseMaxBal}(self, b) \triangleq$
- $b > \text{maxBal}[self]$
- $\text{maxBal}' = \left[ \text{maxBal} \text{ EXCEPT } ![self] = b \right]$
- $\text{UNCHANGED votes}$

$\text{VoteFor}(self, b, v) \triangleq$
- $\text{maxBal}[self] \leq b$
- $\text{DidNotVoteIn}(self, b)$
- $\forall p \in \text{Acceptor} \setminus \{self\}$:
  - $\forall w \in \text{Value} : \text{VotedFor}(p, b, w) \Rightarrow (w = v)$
- $\text{SafeAt}(b, v)$
- $\text{votes}' = \left[ \text{votes} \text{ EXCEPT } ![self] = \text{votes}[self] \cup \{(b, v)\} \right]$
- $\text{maxBal}' = \left[ \text{maxBal} \text{ EXCEPT } ![self] = b \right]$

$\text{BallotAction}(self, b) \triangleq$
- $\lor \text{IncreaseMaxBal}(self, b)$
- $\exists v \in \text{Value} : \text{VoteFor}(self, b, v)$

When proving lemma $\text{NextDef}$, we were surprised to discover that it required the assumption that the set of acceptors is non-empty. This assumption isn’t necessary for safety, since if there are no acceptors there can be no quorums (see theorem $\text{QuorumNonEmpty}$ above) so no value is ever chosen and the Consensus specification is trivially implemented under our refinement mapping. However, the assumption is necessary for liveness and it allows us to lemma $\text{NextDef}$ for the safety proof as well, so we assert it now.

$\text{assume } \text{AcceptorNonempty} \triangleq \text{Acceptor} \neq \{\}$

The proof of the lemma itself is quite simple.

**Lemma NextDef**: $\text{TypeOK} \Rightarrow\ (\text{Next} = \exists self \in \text{Acceptor} : \exists b \in \text{Ballot} : \text{BallotAction}(self, b))$

1. **Have** $\text{TypeOK}$
2. $\text{Next} = \exists self \in \text{Acceptor} : \text{acceptor}(self)$
   - **by** $\text{AcceptorNonempty}$
   - **DEF** $\text{Next}, \text{ProcSet}$
   - **1)3. @ = NextDef!2!2**
   - **by** $\text{DEF Next, BallotAction, IncreaseMaxBal, VoteFor, ProcSet, acceptor}$
3. **QED**
   - **by** 1)2, 1)3

We now come to the proof that $\text{VInv}$ is an invariant of the specification. This follows from the following result, which asserts that it is an inductive invariant of the next-state action. This fact is used in the liveness proof as well.

**Theorem InductiveInvariance** $\triangleq \text{VInv} \land [\text{Next}]_{vars} \Rightarrow \text{VInv}'$

1. $\text{VInv} \land \text{vars}' = \text{vars} \Rightarrow \text{VInv}'$
BY Isa

DEF VInv, vars, TypeOK, VInv2, VotedFor, SafeAt, DidNotVoteIn, VInv3, VInv4

(1) SUFFICES ASSUME VInv,
    NEW self ∈ Acceptor,
    NEW b ∈ Ballot,
    BallotAction(self, b)
    PROVE VInv'

BY (1)1, NextDef DEF VInv

(1)2. TypeOK'
   (2)1.CASE IncreaseMaxBal(self, b)
       BY (2)1 DEF IncreaseMaxBal, VInv, TypeOK
   (2)2.CASE ∃ v ∈ Value : VotedFor(self, b, v)
       BY (2)2 DEF VInv, TypeOK, VotedFor
   (2)3. QED
       BY (2)1, (2)2 DEF BallotAction

(1)3. ASSUME NEW a ∈ Acceptor, NEW c ∈ Ballot, NEW w ∈ Value,
    VotedFor(a, c, w)
    PROVE VotedFor(a, c, w)'

(2)1.CASE IncreaseMaxBal(self, b)
   BY (2)1, (1)3 DEF IncreaseMaxBal, VotedFor
   (2)2.CASE ∃ v ∈ Value : VotedFor(self, b, v)
      (3)1. PICK v ∈ Value : VotedFor(self, b, v)
      BY (2)2
      (3)2.CASE a = self
         (4)1. votes'[a] = votes[a] ∪ {(b, v)}
            BY (3)1, (3)2 DEF VotedFor, VInv, TypeOK
         (4)2. QED
            BY (1)3, (4)1 DEF VotedFor
      (3)3.CASE a ≠ self
         (4)1. votes[a] = votes'[a]
            BY (3)1, (3)3 DEF VotedFor, VInv, TypeOK
         (4)2. QED
            BY (1)3, (4)1 DEF VotedFor
      (3)4. QED
            BY (3)2, (3)3 DEF VotedFor
   (2)3. QED
       BY (2)1, (2)2 DEF BallotAction

(1)4. ASSUME NEW a ∈ Acceptor, NEW c ∈ Ballot, NEW w ∈ Value,
    ¬VotedFor(a, c, w), VotedFor(a, c, w)'
    PROVE (a = self) ∧ (c = b) ∧ VotedFor(self, b, w)

(2)1.CASE IncreaseMaxBal(self, b)
   BY (2)1, (1)4 DEF IncreaseMaxBal, VInv, TypeOK, VotedFor
   (2)2.CASE ∃ v ∈ Value : VotedFor(self, b, v)
⟨3⟩1. Pick \( v \in \text{Value} : \text{VoteFor}(self, b, v) \)
   By (2)2
⟨3⟩2. \( a = self \)
   By (3)1, (1)4 DEF VoteFor, VInv, TypeOK, VotedFor
⟨3⟩3. \( \text{votes}'[a] = \text{votes}[a] \cup \{b, v\} \)
   By (3)1, (3)2 DEF VoteFor, VInv, TypeOK
⟨3⟩4. \( c = b \land v = w \)
   By (1)4, (3)3 DEF VotedFor
⟨3⟩5. QED
   By (3)1, (3)2, (3)4
(2)3. QED
   By (2)1, (2)2 DEF BallotAction

⟨1⟩5. Assume new \( a \in \text{Acceptor} \)
   Prove \( \land \text{maxBal}'[a] \in \text{Ballot} \cup \{-1\} \)
   \( \land \text{maxBal}''[a] \in \text{Ballot} \cup \{-1\} \)
   \( \land \text{maxBal}'''[a] \geq \text{maxBal}''[a] \)
   By DEF VInv, TypeOK, IncreaseMaxBal, VInv, VoteFor, BallotAction, DidNotVoteIn, VotedFor, Ballot

⟨1⟩6. Assume new \( c \in \text{Ballot}, w \in \text{Value}, SafeAt(c, w) \)
   Prove \( SafeAt(c, w)' \)
   (2) USE DEF Ballot
   (2) DEFINE \( P(i) \triangleq \forall j \in 0..i : SafeAt(j, w) \Rightarrow SafeAt(j, w)' \)
   (2)1. \( P(0) \)
      By SafeAtPropPrime, 0..0 = \{0\}, Zenon
   (2)2. Assume new \( d \in \text{Ballot}, P(d) \)
      PROVE \( P(d+1) \)
      ⟨3⟩1. SUFFICES Assume new \( e \in 0..(d+1) \), SafeAt(e, w)
         PROVE \( SafeAt(e, w)' \)
         OBVIOUS
      ⟨3⟩2. CASE \( e \in 0..d \)
         By (2)2, (3)1, (3)2
      ⟨3⟩3. CASE \( e = d + 1 \)
         ⟨4⟩. \( e \in \text{Ballot} \setminus \{0\} \)
            By (3)3
         ⟨4⟩1. Pick \( Q \in \text{Quorum} : SafeAtProp!(e, w)!2!2!(Q) \)
            By (3)1, SafeAtProp, Zenon
         ⟨4⟩2. \( \forall aa \in Q : \text{maxBal}''[aa] \geq e \)
            By (1)5, (4)1, QA
         ⟨4⟩3. \( \exists cc \in -1..(e-1) : \)
            \( \land (cc \neq -1) \Rightarrow \land SafeAt(cc, w)' \)
            \( \land \forall ax \in Q : \)
            \( \forall z \in \text{Value} : \)
\[VotedFor(ax, cc, z)' \Rightarrow (z = w)\]
\[\wedge \forall dd \in (cc + 1) \ldots (e - 1), ax \in Q : DidNotVoteIn(ax, dd)’\]

\(\langle 5 \rangle 1. \text{Assume new } cc \in 0 \ldots (e - 1),\]
\[\text{NEW } ax \in Q, \text{NEW } z \in Value, VotedFor(ax, cc, z)', \neg VotedFor(ax, cc, z)\]
\[\text{PROVE FALSE}\]

\(\langle 6 \rangle 1. (ax = self) \wedge (cc = b) \wedge VoteFor(self, b, z)\]
\[\text{BY } \langle 5 \rangle 1, \langle 1 \rangle 4, QA\]

\(\langle 6 \rangle 2. \wedge maxBal[ax] \geq e\]
\[\wedge maxBal[sf] \leq b\]
\[\text{BY } \langle 4 \rangle 1, \langle 6 \rangle 1 \text{ DEF VoteFor}\]

\(\langle 6 \rangle \text{ QED BY } \langle 3 \rangle 3, \langle 6 \rangle 1, \langle 6 \rangle 2 \text{ DEF VInv, TypeOK}\)

\(\langle 5 \rangle 2 . \text{Pick } cc \in -1 \ldots (e - 1) : SafeAtProp!(e, w)!2!2!(Q)!2!(cc)\]
\[\text{BY } \langle 4 \rangle 1\]

\(\langle 5 \rangle 3 . \text{Assume } cc \neq -1\]
\[\text{PROVE } \wedge SafeAt(cc, w)’\]
\[\wedge \forall ax \in Q : \forall z \in Value : VotedFor(ax, cc, z)' \Rightarrow (z = w)\]

\(\langle 6 \rangle 1. \wedge SafeAt(cc, w)’\]
\[\wedge \forall ax \in Q : \forall z \in Value : VotedFor(ax, cc, z) \Rightarrow (z = w)\]
\[\text{BY } \langle 5 \rangle 2, \langle 5 \rangle 3\]

\(\langle 6 \rangle 2. SafeAt(cc, w)’\]
\[\text{BY } \langle 6 \rangle 1, \langle 5 \rangle 3, \langle 3 \rangle 3, \langle 2 \rangle 2\]

\(\langle 6 \rangle 3. \text{Assume new } ax \in Q, \text{NEW } z \in Value, VotedFor(ax, cc, z)’\]
\[\text{PROVE } z = w\]

\(\langle 7 \rangle 1. \text{CASE VotedFor}(ax, cc, z)\]
\[\text{BY } \langle 6 \rangle 1, \langle 7 \rangle 1\]

\(\langle 7 \rangle 2. \text{CASE } \neg VotedFor(ax, cc, z)\]
\[\text{BY } \langle 7 \rangle 2, \langle 6 \rangle 3, \langle 5 \rangle 1, \langle 5 \rangle 3\]

\(\langle 7 \rangle 3. \text{QED}\]
\[\text{BY } \langle 7 \rangle 1, \langle 7 \rangle 2\]

\(\langle 6 \rangle 4. \text{QED}\]
\[\text{BY } \langle 6 \rangle 2, \langle 6 \rangle 3\]

\(\langle 5 \rangle 4. \text{Assume new } dd \in (cc + 1) \ldots (e - 1), \text{NEW } ax \in Q,\]
\[\neg DidNotVoteIn(ax, dd)’\]
\[\text{PROVE FALSE}\]
\[\text{BY } \langle 5 \rangle 2, \langle 5 \rangle 1, \langle 5 \rangle 4 \text{ DEF } DidNotVoteIn\]

\(\langle 5 \rangle 5. \text{QED}\]
\[\text{BY } \langle 5 \rangle 3, \langle 5 \rangle 4\]

\(\langle 4 \rangle 4. \forall e = 0\]
\[\forall \exists Q_1 \in Quorum : \]
\[\wedge \forall aa \in Q_1 : maxBal'[aa] \geq e \]
\[\wedge \exists c_1 \in -1 \ldots e - 1 : \]
\[\wedge c_1 \neq -1\]
⇒ ( ∧ SafeAt(c_1, w)
∧ ∀ aa ∈ Q_1 :
    ∀ w_1 ∈ Value :
        VotedFor(aa, c_1, w_1)′ ⇒ w_1 = w)
∧ ∀ d_1 ∈ c_1 + 1 . . . e - 1, aa ∈ Q_1 :
    DidNotVoteIn(aa, d_1)′

BY (4)2, (4)3, (3)3
(4)6. SafeAt(c, w)′ ≡ (4)4
    BY SafeAtPropPrime, (3)3, Zenon
(4)7. QED
    BY (4)2, (4)3, (4)6
(3)4. QED
    BY (3)2, (3)3
(2)3. ∀ d ∈ Ballot : P(d)
    BY (2)1, (2)2, NatInduction, Isa
(2)4. QED
    BY (2)3, (1)6

17. VInv2'
(2)1. SUFFICES ASSUME NEW a ∈ Acceptor, NEW c ∈ Ballot, NEW v ∈ Value,
    VotedFor(a, c, v)
    PROVE SafeAt(c, v)'
    BY DEF VInv2
(2)2. CASE VotedFor(a, c, v)
        (1)6, (2)2 DEF VInv, VInv2
(2)3. CASE ¬VotedFor(a, c, v)
        (1)6, (2)1, (2)3, (1)4 DEF VoteFor
(2)4. QED
    BY (2)2, (2)3

18. VInv3'
(2)1. ASSUME NEW a1 ∈ Acceptor, NEW a2 ∈ Acceptor,
    NEW c ∈ Ballot, NEW v1 ∈ Value, NEW v2 ∈ Value,
    VotedFor(a1, c, v1)',
    VotedFor(a2, c, v2)',
    VotedFor(a1, c, v1),
    VotedFor(a2, c, v2)
    PROVE v1 = v2
    BY (2)1 DEF VInv, VInv3
(2)2. ASSUME NEW a1 ∈ Acceptor, NEW a2 ∈ Acceptor,
    NEW c ∈ Ballot, NEW v1 ∈ Value, NEW v2 ∈ Value,
    VotedFor(a1, c, v1)',
    VotedFor(a2, c, v2)',
    ¬VotedFor(a1, c, v1)
    PROVE v1 = v2
\(3\). (a_1 = \text{self}) \land (c = b) \land \text{VoteFor}(\text{self}, b, v_1) \\
by (2)2, (1)4

\(3\). \text{case } a_2 = \text{self} \\
\(4\). \text{\lnot VoteFor}(\text{self}, b, v_2) \\
by (3)1 \text{ def VoteFor, DidNotVoteIn} \\
\(4\). \text{VoteFor}(\text{self}, b, v_2) \\
by (2)2, (3)1, (3)2, (4)1, (1)4

\(4\). \text{QED by (3)1, (4)2, (2)2 def VoteFor, VoteFor, VInv, TypeOK} \\
\(3\). \text{case } a_2 \neq \text{self} \\
by (3)1, (3)3, (2)2 \text{ def VoteFor, VoteFor, VInv, TypeOK} \\
\(3\). \text{QED} \\
by (3)2, (3)3

\(2\). \text{QED} \\
by (2)1, (2)2 \text{ def VInv3} \\
\(1\). \text{VInv4'} \\
\(2\). \text{suffices assume new } a \in \text{Acceptor}, \text{ new } c \in \text{Ballot}, \\
maxBal'[a] < c, \\
\text{\lnot DidNotVoteIn}(a, c) \\
prove \text{ false} \\
by \text{ def VInv4} \\
\(2\). \text{maxBal}[a] < c \\
by (1)5, (2)1 \text{ def Ballot} \\
\(2\). \text{DidNotVoteIn}(a, c) \\
by (2)2 \text{ def VInv, VInv4} \\
\(2\). \text{pick } v \in \text{Value : VotedFor}(a, c, v) \\
by (2)1 \text{ def DidNotVoteIn} \\
\(2\). (a = \text{self}) \land (c = b) \land \text{VoteFor}(\text{self}, b, v) \\
by (1)4, (2)1, (2)3, (2)4 \text{ def DidNotVoteIn} \\
\(2\). \text{maxBal'}[a] = c \\
by (2)5 \text{ def VoteFor, VInv, TypeOK} \\
\(2\). \text{QED} \\
by (2)1, (2)6 \text{ def Ballot} \\
\(1\). \text{QED} \\
by (1)2, (1)7, (1)8, (1)9 \text{ def VInv}

The invariance of VInv follows easily from theorem InductiveInvariance and the following result, which is easy to prove with TLAPS.

**Theorem** InitImpliesInv \(\triangleq\) Init \(\Rightarrow\) VInv \\
by \text{ def Init, VInv, TypeOK, ProcSet, VInv2, VInv3, VInv4, VotedFor, DidNotVoteIn}

The following theorem asserts that VInv is an invariant of Spec.

**Theorem** VT2 \(\triangleq\) Spec \(\Rightarrow\) \(\square\) VInv \\
by InitImpliesInv, InductiveInvariance, PTL def Spec
The following instance statement instantiates module `Consensus` with the following expressions substituted for the parameters (the constants and variables) of that module:

<table>
<thead>
<tr>
<th>Parameter of <code>Consensus</code></th>
<th>Expression (of this module)</th>
<th>Value</th>
<th>Value chosen</th>
<th>chosen</th>
</tr>
</thead>
</table>

(Note that if no substitution is specified for a parameter, the default is to substitute the parameter or defined operator of the same name.) More precisely, for each defined identifier `id` of module `Consensus`, this statement defines `C!id` to equal the value of `id` under these substitutions.

C ≜ INSTANCE `Consensus`

The following theorem asserts that the safety properties of the voting algorithm (specified by formula `Spec`) of this module implement the consensus safety specification `Spec` of module `Consensus` under the substitution (refinement mapping) of the instance statement.

THEOREM VT3 ≜ Spec ⇒ C!Spec

1. **Init ⇒ C!Init**
   - **(2) SUFFICES ASSUME Init**
   - **PROVE C!Init**
     **OBSVIOUS**
   - **(2)1. SUFFICES ASSUME NEW v ∈ chosen**
       **PROVE FALSE**
       **BY DEF C!Init**
   - **(2)2. PICK b ∈ Ballot, Q ∈ Quorum : ∀ a ∈ Q : VotedFor(a, b, v)**
     **BY (2)1 DEF chosen, ChosenIn**
   - **(2)3. PICK a ∈ Q : (b, v) ∈ votes[a]**
     **BY QuorumNonEmpty, (2)2 DEF VotedFor**
   - **(2)4. QED**
     **BY (2)3, QA DEF Init**

2. **VInv ∧ VInv′ ∧ [Next]vars ⇒ [C!Next]C!vars**
   - **(2) SUFFICES ASSUME VInv, VInv′, [Next]vars**
     **PROVE [C!Next]C!vars**
     **OBSVIOUS**
   - **(2)1. CASE vars′ = vars**
     **BY (2)1 DEF vars, C!vars, chosen, ChosenIn, VotedFor**
   - **(2)2. SUFFICES ASSUME NEW self ∈ Acceptor,**
     **NEW b ∈ Ballot,**
     **BallotAction(self, b)**
     **PROVE [C!Next]C!vars**
     **BY (2)1, NextDef DEF VInv**
   - **(2)3. ASSUME IncreaseMaxBal(self, b)**
     **PROVE C!vars′ = C!vars**
     **BY (2)3 DEF IncreaseMaxBal, C!vars, chosen, ChosenIn, VotedFor**
   - **(2)4. ASSUME NEW v ∈ Value,**
     **VoteFor(self, b, v)**
     **PROVE [C!Next]C!vars**
We now state the liveness property required of our voting algorithm and prove that it and the safety property imply specification \textit{LiveSpec} of module \textit{Consensus} under our refinement mapping.

We begin by stating two additional assumptions that are necessary for liveness. Liveness requires that some value eventually be chosen. This cannot hold with an infinite set of acceptors. More precisely, liveness requires the existence of a finite quorum. (Otherwise, it would be impossible for all acceptors of any quorum ever to have voted, so no value could ever be chosen.) Moreover, it is impossible to choose a value if there are no values. Hence, we make the following two assumptions.
ASSUME $\text{AcceptorFinite} \triangleq \text{IsFiniteSet}(\text{Acceptor})$

ASSUME $\text{ValueNonempty} \triangleq \text{Value} \neq \{\}$

**Lemma FiniteSetHasMax**

ASSUME NEW $S \in \text{subset Int}, \text{IsFiniteSet}(S), S \neq \{\}$

PROVE $\exists \max \in S : \forall x \in S : \max \geq x$

(1).DEFINE $P(T) \triangleq T \in \text{subset Int} \land T \neq \{\} \Rightarrow \exists \max \in T : \forall x \in T : \max \geq x$

(1)1. $P(\{\})$

OBSVIOUS

(1)2. ASSUME NEW $T, \text{new } x, P(T), x \notin T$

PROVE $P(T \cup \{x\})$

BY (1)2

(1)3. $\forall T : \text{IsFiniteSet}(T) \Rightarrow P(T)$

(2).HIDE DEF $P$

(2).QED BY (1)1, (1)2, FS_Induction, IsaM("blast")

(1).QED BY (1)3, Zenon

The following theorem implies that it is always possible to find a ballot number $b$ and a value $v$ safe at $b$ by choosing $b$ large enough and then having a quorum of acceptors perform $\text{IncreaseMaxBal}(b)$ actions. It will be used in the liveness proof. Observe that it is for liveness, not safety, that invariant $V\text{Inv}3$ is required.

**Theorem VT4**

$\Rightarrow \forall Q \in \text{Quorum}, b \in \text{Ballot} :$

$(\forall a \in Q : (\maxBal[a] \geq b)) \Rightarrow \exists v \in \text{Value} : \text{SafeAt}(b, v)$

Checked as an invariant by TLC with 3 acceptors, 3 ballots, 2 values

(1).USE DEF $\text{Ballot}$

(1)1. SUFFICES ASSUME $\text{TypeOK}, V\text{Inv}2, V\text{Inv}3,$

NEW $Q \in \text{Quorum}, \text{new } b \in \text{Ballot},$

$(\forall a \in Q : (\maxBal[a] \geq b))$

PROVE $\exists v \in \text{Value} : \text{SafeAt}(b, v)$

OBSVIOUS

(1)2. CASE $b = 0$

BY $\text{ValueNonempty}, (1)1, \text{SafeAtProp}, (1)2, \text{Zenon}$

(1)4. SUFFICES ASSUME $b \neq 0$

PROVE $\exists v \in \text{Value} :$

$\exists c \in -1 \ldots (b-1) :$

$\land (c \neq -1) \Rightarrow \land \text{SafeAt}(c, v)$

$\land \forall a \in Q :$

$\forall w \in \text{Value} :$

$\land \forall d \in (c+1) \ldots (b-1), a \in Q : \text{DidNotVoteIn}(a, d)$

BY (1)1, (1)2, SafeAtProp

(1)5. CASE $\forall a \in Q, c \in 0 \ldots (b-1) : \text{DidNotVoteIn}(a, c)$
BY $\langle 1 \rangle 5$, $ValueNonempty$

$\langle 1 \rangle 6$. CASE $\exists a \in Q, c \in 0 \ldots (b - 1) : \neg DidNotVoteIn(a, c)$

(2) 1. PICK $c \in 0 \ldots (b - 1) :$

\begin{align*}
\land \exists a & \quad \in Q : \neg DidNotVoteIn(a, c) \\
\land \forall d & \quad \in (c + 1) \ldots (b - 1), a \in Q : DidNotVoteIn(a, d)
\end{align*}

(3) DEFINE $S \triangleq \{ c \in 0 \ldots (b - 1) : \exists a \quad \in Q : \neg DidNotVoteIn(a, c) \}$

(3) 1. $S \neq \{ \}$

BY $\langle 1 \rangle 6$

(3) 2. PICK $c \in S : \forall d \in S : c \geq d$

(4) 2. IsFiniteSet($S$)

BY $FS_{\text{Interval}}, FS_{\text{Subset}}, 0 \in \text{Int}, b - 1 \in \text{Int}, Zenon$

(4) 3. QED

BY $\langle 3 \rangle 1, \langle 4 \rangle 2, FiniteSetHasMax$

(3) QED

BY $\langle 3 \rangle 2$ DEF $Ballot$

(2) 4. PICK $a0 \in Q, v \in Value : VotedFor(a0, c, v)$

BY $\langle 2 \rangle 1$ DEF $DidNotVoteIn$

(2) 5. $\forall a \in Q : \forall w \in Value :$

$VotedFor(a, c, w) \Rightarrow (w = v)$

BY $\langle 2 \rangle 4, QA, \langle 1 \rangle 1$ DEF $VInv3$

(2) 6. $SafeAt(c, v)$

BY $\langle 1 \rangle 1, \langle 2 \rangle 4, QA$ DEF $VInv2$

(2) 7. QED

BY $\langle 2 \rangle 1, \langle 2 \rangle 5, \langle 2 \rangle 6$

$\langle 1 \rangle 7$. QED

BY $\langle 1 \rangle 5, \langle 1 \rangle 6$

The progress property we require of the algorithm is that a quorum of acceptors, by themselves, can eventually choose a value $v$. This means that, for some quorum $Q$ and ballot $b$, the acceptors $a$ of $Q$ must make $SafeAt(b, v)$ true by executing $IncreaseMaxBal(a, b)$ and then must execute $VoteFor(a, b, v)$ to choose $v$. In order to be able to execute $VoteFor(a, b, v)$, acceptor $a$ must not execute a $Ballot(a, c)$ action for any $c > b$.

These considerations lead to the following liveness requirement $LiveAssumption$. The $WF$ condition requires that the acceptors $a$ in $Q$ eventually execute the necessary $BallotAction(a, b)$ actions if they are enabled, and the $\square \langle \ldots \rangle_{vars}$ condition ensures that they never perform $BallotAction$ actions for higher-numbered ballots, so the necessary $BallotAction(a, b)$ actions are enabled.

$\neg \diamond \langle \ldots \rangle_{vars}$

$LiveAssumption \triangleq$

$\exists Q \in Quorum, b \in Ballot :$

$\land \forall self \in Q : WF_{vars}(BallotAction(self, b))$

$\land \forall self \in Q : \forall c \in Ballot :$

$(c > b) \Rightarrow \neg BallotAction(self, c)_{vars}$

$LiveSpec \triangleq Spec \land LiveAssumption$
LiveAssumption is stronger than necessary. Instead of requiring that an acceptor in $Q$ never executes an action of a higher-numbered ballot than $b$, it suffices that it doesn’t execute such an action until unless it has voted in ballot $b$. However, the natural liveness requirement for a Paxos consensus algorithm implies condition LiveAssumption.

Condition LiveAssumption is a liveness property, constraining only what eventually happens. It is straightforward to replace “eventually happens” by “happens within some length of time” and convert LiveAssumption into a real-time condition. We have not done that for three reasons:

1. The real-time requirement and, we believe, the real-time reasoning will be more complicated, since temporal logic was developed to abstract away much of the complexity of reasoning about explicit times.
2. TLAPS does not yet support reasoning about real numbers.
3. Reasoning about real-time specifications consists entirely of safety reasoning, which is almost entirely action reasoning. We want to see how the TLA+ proof language and TLAPS do on temporal logic reasoning.

Here are two temporal-logic proof rules. Their validity is obvious when you understand what they mean.

**THEOREM AlwaysForall** $\triangleq$

\begin{align*}
\text{ASSUME NEW CONSTANT } S, \text{ NEW TEMPORAL } P(\_)
\hspace{1cm}
\text{PROVE } (\forall s \in S : \Box P(s)) \equiv \Box (\forall s \in S : P(s))
\end{align*}

OBLVIOUS

**LEMMA EventuallyAlwaysForall** $\triangleq$

\begin{align*}
\text{ASSUME NEW CONSTANT } S, \text{ IsFiniteSet}(S), \\
\text{NEW TEMPORAL } P(\_)
\hspace{1cm}
\text{PROVE } (\forall s \in S : \Diamond \Box P(s)) \Rightarrow \Diamond (\forall s \in S : P(s))
\end{align*}

\[\langle 1 \rangle \text{. DEFINE } A(x) \triangleq \Diamond \Box P(x)\]

\[L(T) \triangleq \forall s \in T : A(s)\]

\[R(T) \triangleq \forall s \in T : P(s)\]

\[Q(T) \triangleq L(T) \Rightarrow \Diamond \Box R(T)\]

\[\langle 1 \rangle 1. \text{ Q}()\]

\[\langle 1 \rangle 2. \text{ R}() \hspace{1cm} \text{OBLVIOUS}\]

\[\langle 1 \rangle 2. \text{ } \Diamond \Box \text{R}() \hspace{1cm} \text{BY } (1)1, \text{ PTL}\]

\[\langle 1 \rangle 2. \text{ QED } \hspace{1cm} \text{BY } (1)2\]

\[\langle 2 \rangle 1. \text{ ASSUME NEW } T, \text{ NEW } x\]

\[\text{PROVE } Q(T) \Rightarrow Q(T \cup \{x\})\]

\[\langle 3 \rangle . \text{ HIDE } \text{DEF A}\]

\[\langle 3 \rangle . \text{ QED } \hspace{1cm} \text{OBLVIOUS}\]

\[\langle 2 \rangle 2. \text{ L}(T \cup \{x\}) \land Q(T) \Rightarrow \Diamond \Box R(T)\]

\[\text{OBLVIOUS}\]

\[\langle 2 \rangle 3. \text{ } \Diamond \Box \text{R}(T) \land A(x) \Rightarrow \Diamond \Box (\text{R}(T) \land P(x))\]

\[\text{BY PTL}\]

\[\langle 2 \rangle 4. \text{ R}(T) \land P(x) \Rightarrow \text{R}(T \cup \{x\})\]

\[\text{OBLVIOUS}\]

\[\langle 2 \rangle 5. \text{ } \Diamond \Box (\text{R}(T) \land P(x)) \Rightarrow \Diamond \Box \text{R}(T \cup \{x\})\]
Here is our proof that \textit{LiveSpec} implements the specification \textit{LiveSpec} of module \textit{Consensus} under our refinement mapping.

\textbf{THEOREM \textit{Liveness} } \triangleq \ \textit{LiveSpec} \Rightarrow \textit{C!LiveSpec}

\textit{1.} SUFFICES ASSUME NEW \(Q \in \text{Quorum}, \ \text{NEW} \ b \in \text{Ballot}

\textit{PROVE} \ Spec \land \textit{LiveAssumption}!(Q, b) \Rightarrow \textit{C!LiveSpec}

\textbf{BY} Isa \ DEF \ \textit{LiveSpec}, \ \textit{LiveAssumption}

\textit{1a.} \ \textit{IsFiniteSet}(Q)

\textbf{BY} QA, \ \textit{AcceptorFinite}, \ \textit{FS\_Subset}

\textit{1b.} \ \textit{C!LiveSpec} \equiv \textit{C!Spec} \land (\Box \Diamond (C!Next)_c !\ vars \lor \Box \Diamond (\text{chosen} \neq \{}))

\textbf{BY} \ \textit{ValueNonempty}, \ \textit{C!LiveSpecEquals}

\textit{1c.} \ \textbf{DEFINE} \ \textit{LNext} \triangleq \exists \text{self} \in \text{Acceptor}, \ c \in \text{Ballot}:

\hspace{1cm} \land \text{BallotAction}(\text{self}, c)
\hspace{1cm} \land (\text{self} \in Q) \Rightarrow (c \leq b)

\textit{1d.} \ Spec \land \textit{LiveAssumption}!(Q, b) \Rightarrow \Box[LNext] vars

\textbf{BY} NextDef \ DEF \ \textit{LNext}, \ \textit{Ballot}

\textit{2.} \ Spec \land \textit{LiveAssumption}!(Q, b) \Rightarrow \Box[LNext] vars

\textbf{BY} (2)1, \ \textit{PTL}

\textit{2c.} \ \textbf{QED}

\textbf{BY} (2)2, \ VT2, \ Isa \ DEF \ \textit{Spec}, \ \textit{VInv}

\textit{1.} \ \textbf{DEFINE} \ \textit{LInv1} \triangleq \forall a \in Q : \text{maxBal}[a] \leq b
\hspace{1cm} \textit{LInv1} \triangleq \textit{VInv} \land \textit{LInv1}

\textit{1c.} \ \textit{LInv1} \land [LNext] vars \Rightarrow [LInv1]'

\textbf{BY} (2)1. \ SUFFICES ASSUME \textit{LInv1}, [LNext] vars

23
PROVE \( \text{LInv}' \)

\( (2) \). \( \text{VInv}' \)
  
  \( \text{BY (2)1, NextDef, InductiveInvariance def LInv1, VInv} \)

\( (2) \). \( \text{LNInv}' \)
  
  \( \text{BY (2)1, QA def BallotAction, IncreaseMaxBal, VoteFor, VInv, TypeOK, vars} \)

\( (2) \). QED
  
  \( \text{BY (2)2, (2)3} \)

\( (1) \). \( \forall a \in Q : VInv \land (\text{maxBal}[a] = b) \land [\text{LNext}]_{\text{vars}} \Rightarrow \text{VInv}' \land (\text{maxBal}'[a] = b) \)

\( (2) \). SUFFICES ASSUME NEW \( a \in Q \), \( \text{VInv}, \text{maxBal}[a] = b, [\text{LNext}]_{\text{vars}} \)
  
  \( \text{PROVE VInv} \land (\text{maxBal}'[a] = b) \)

\( (2) \). \( \text{VInv}' \)
  
  \( \text{BY (2)1, NextDef, InductiveInvariance def VInv} \)

\( (2) \). \( \text{maxBal}'[a] = b \)
  
  \( \text{BY (2)1, QA def BallotAction, IncreaseMaxBal, VoteFor, VInv, TypeOK, Ballot, vars} \)

\( (2) \). QED
  
  \( \text{BY (2)2, (2)3} \)

\( (1) \). \( \text{Spec} \land \text{LiveAssumption}!(Q, b) \Rightarrow \diamond \Box (\forall \text{self} \in Q : \text{maxBal}['\text{self}'] = b) \)

\( (2) \). SUFFICES ASSUME NEW \( \text{self} \in Q \)
  
  \( \text{PROVE Spec} \land \text{LiveAssumption}!(Q, b) \Rightarrow \diamond \Box (\text{maxBal}['\text{self}'] = b) \)

\( \text{BY (1) a, EventuallyAlwaysForall}\* \) doesn’t check, even when introducing definitions

\( \text{PROOF OMITTED} \)

\( (2) \). DEFINE \( P \triangleq \text{LInv1} \land \neg(\text{maxBal}'[\text{self}] = b) \)
  
  \( QQ \triangleq \text{LInv1} \land (\text{maxBal}'[\text{self}] = b) \)
  
  \( A \triangleq \text{BallotAction}(\text{self}, b) \)

\( (2) \). \( \Box [\text{LNext}]_{\text{vars}} \land \text{WF}_{\text{vars}}(A) \Rightarrow (\text{LInv1} \leadsto QQ) \)

\( (3) \). \( P \land [\text{LNext}]_{\text{vars}} \Rightarrow (P' \lor QQ') \)
  
  \( \text{BY (1)3} \)

\( (3) \). \( P \land \langle \text{LNext} \land A \rangle_{\text{vars}} \Rightarrow QQ' \)

\( (4) \). SUFFICES ASSUME \( \text{LInv1}, \text{LNext}, A \)
  
  \( \text{PROVE QQ'} \)

\( (4) \). \( \text{OFFIOUS} \)

\( (4) \). \( \text{LInv1}' \)
  
  \( \text{BY (4)1, (1)3} \)

\( (4) \). \( \text{CASE IncreaseMaxBal(} \text{self}, b) \)
  
  \( \text{BY (4)1, (4)2, (4)3, QA def IncreaseMaxBal, VInv, TypeOK} \)

\( (4) \). \( \text{CASE } \exists v \in \text{Value} : \text{VoteFor(} \text{self}, b, v) \)
  
  \( \text{BY (4)1, (4)2, (4)4, QA def VoteFor, VInv, TypeOK} \)

\( (4) \). QED
\text{BY (4)1, (4)3, (4)4} \text{ DEF BallotAction}

\langle 3 \rangle.3. \; P \Rightarrow \text{ENABLED } \langle A \rangle_{\text{vars}}

\langle 4 \rangle.1. \; (\text{ENABLED } \langle A \rangle_{\text{vars}}) \equiv

\exists \text{votesp, maxBalp}:
\begin{align*}
\land \land \land b > \text{maxBal}[self] \\
\land \text{maxBal} = [\text{maxBal EXCEPT } !\{self\} = b] \\
\land \text{votesp} = \text{votes}
\end{align*}
\lor \exists v \in \text{Value}:
\begin{align*}
\land \text{maxBal}[self] \leq b \\
\land \text{DidNotVoteIn}(self, b) \\
\land \forall p \in \text{Acceptors} \setminus \{self\}:
\begin{align*}
\forall w \in \text{Value} : \text{VotedFor}(p, b, w) \Rightarrow (w = v) \\
\land \text{SafeAt}(b, v) \\
\land \text{votesp} = [\text{votes EXCEPT } !\{self\} = \text{votes}[self] \\
\cup \{(b, v)\}] \\
\land \text{maxBal} = [\text{maxBal EXCEPT } !\{self\} = b]
\end{align*}
\land (\text{votesp, maxBalp}) \neq (\text{votes, maxBal})
\end{align*}

\text{PROOF OMMITTED}

\langle 4 \rangle. \text{SUFFICES \textbf{ASSUME} } P

\text{PROVE } \exists \text{votesp, maxBalp}:
\begin{align*}
\land b > \text{maxBal}[self] \\
\land \text{maxBal} = [\text{maxBal EXCEPT } !\{self\} = b] \\
\land \text{votesp} = \text{votes} \\
\land (\text{votesp, maxBalp}) \neq (\text{votes, maxBal})
\end{align*}

\text{BY (4)1}

\langle 4 \rangle. \text{WITNESS } \text{votes}, [\text{maxBal EXCEPT } !\{self\} = b]

\langle 4 \rangle. \text{QED} \text{ BY QA \textbf{DEF} VInv, TypeOK, Ballot}

\langle 3 \rangle. \text{QED} \text{ BY (3)1, (3)2, (3)3, PTL}

\langle 2 \rangle.3. \; QQ \land [LNext]_{\text{vars}} \Rightarrow \Box QQ

\langle 3 \rangle.1. \; QQ \land [LNext]_{\text{vars}} \Rightarrow QQ'

\text{BY (1)3, (1)4}

\langle 3 \rangle. \text{QED} \text{ BY (3)1, PTL}

\langle 2 \rangle.4. \Box QQ \Rightarrow \Box (\text{maxBal}[self] = b)

\text{BY PTL}

\langle 2 \rangle.5. \text{LiveAssumption} !Q, b \Rightarrow \text{WF}_{\text{vars}}(A)

\text{BY Isa}

\langle 2 \rangle.6. \; \text{Spec} \Rightarrow \text{LInv1}

\langle 3 \rangle.1. \; \text{Init} \Rightarrow \text{VInv}

\text{BY InitImpliesInv}

\langle 3 \rangle.2. \; \text{Init} \Rightarrow \text{LInv1}

\text{BY QA \textbf{DEF} Init, Ballot}

\langle 3 \rangle. \text{QED} \text{ BY (3)1, (3)2 \textbf{DEF} Spec}

\langle 2 \rangle. \text{QED}
BY (2)2, (2)3, (2)4, (2)5, (2)6, (1)2, PTL

(1) DEFINE \(LNInv \triangleq \forall a \in Q : \text{maxBal}[a] = b\) 
\(LInv \triangleq \text{VInv} \land LNInv\)

(1)6. \(LInv \land [LNext]_{vars} \Rightarrow LInv'\)
BY (1)4, QuorumNonEmpty

(1)7. \(Spec \land \text{LiveAssumption}!(Q, b) \Rightarrow \lozenge \lozenge \text{chosen} \neq \{\}\)
(2) DEFINE \(Voted(a) \triangleq \exists v \in \text{Value} : \text{VotedFor}(a, b, v)\)
(2)1. \(Spec \land \text{LiveAssumption}!(Q, b) \Rightarrow \lozenge \lozenge LInv\)
(3)1. \(Spec \land \text{LiveAssumption}!(Q, b) \Rightarrow \lozenge \lozenge LNInv\)
BY (1)5 \* doesn’t check

PROOF OMITTED

(3) QED BY (3)1, VT2, PTL

(2)2. \(LInv \land (\forall a \in Q : \text{VotedFor}(a)) \Rightarrow (\text{chosen} \neq \{\})\)
(3)1. SUFFICES ASSUME \(LInv\), 
\(\forall a \in Q : \text{Voted}(a)\)

PROVE \(\text{chosen} \neq \{\}\)

OBIVIOUS

(3)2. \(\exists v \in \text{Value} : \forall a \in Q : \text{VotedFor}(a, b, v)\)
(4)2. PICK \(a0 \in Q, v \in \text{Value} : \text{VotedFor}(a0, b, v)\)
BY (3)1, QuorumNonEmpty
(4)3. ASSUME NEW \(a \in Q\)

PROVE \(VotedFor(a, b, v)\)
BY (3)1, (4)2, QA DEF \(VInv, VInv3\)
(4)4. QED

BY (4)3

(3)3. QED
BY (3)2 DEF \(\text{chosen, ChosenIn}\)

(2)3. \(Spec \land \text{LiveAssumption}!(Q, b) \Rightarrow (\forall a \in Q : \lozenge \lozenge \text{VotedFor}(a))\)
(3)1. SUFFICES ASSUME NEW \(\text{self} \in Q\)

PROVE \(\text{Spec} \land \text{LiveAssumption}!(Q, b) \Rightarrow \lozenge \lozenge \text{VotedFor}(\text{self})\)

OBIVIOUS \* doesn’t check?!

PROOF OMITTED

(3)2. \(\text{Spec} \land \text{LiveAssumption}!(Q, b) \Rightarrow \lozenge \lozenge \text{VotedFor}(\text{self})\)
(4)2. \(\square [LNext]_{vars} \land \text{WF}_{vars}(\text{BallotAction}(\text{self}, b))\)
\(\Rightarrow ((\text{LInv}2 \land \neg \text{Voted}(\text{self})) \Rightarrow \text{LInv}2 \land \text{Voted}(\text{self}))\)
(5) DEFINE \(P \triangleq \text{LInv}2 \land \neg \text{Voted}(\text{self})\)
\(QQ \triangleq \text{LInv}2 \land \text{Voted}(\text{self})\)
\(A \triangleq \text{BallotAction}(\text{self}, b)\)
(5)1. \(P \land [LNext]_{vars} \Rightarrow (P' \lor QQ')\)
BY (1)6
(5)2. \(P \land (\text{LNext} \land A)_{vars} \Rightarrow QQ'\)
(6)1. SUFFICES ASSUME \(P, LNext,\}
\(P',\ldots\)
prove $QQ'$

(6.2) case $\exists v \in \text{Value} : \text{VoteFor}(\text{self}, b, v)$
  by (6.1), (6.2), (5.1), QA, Zenon def VoteFor, Voted, VotedFor, LInv2, VInv, TypeOK
(6.3) case $\text{IncreaseMaxBal}(\text{self}, b)$
  by (6.1), (6.3) def IncreaseMaxBal, Ballot
(6.4) QED
by (6.1), (6.2), (6.3) def BallotAction

(5.3) $P \Rightarrow \text{enabled} \langle A \rangle_{\text{vars}}$

(6.1) suffices assume $P$
prove $\text{enabled} \langle A \rangle_{\text{vars}}$

obvious

(6.2) $(\text{enabled} \langle A \rangle_{\text{vars}}) \equiv$
$\exists \text{votesp}, \text{maxBalp} :$
$\wedge \exists v \in \text{Value} : \text{VoteFor}(\text{self}, b, v)$
$\wedge \text{maxBal} = [\text{maxBal} \text{ except } ![\text{self}] = b]$
$\wedge \text{votesp} = \text{votes}$
$\vee \exists v \in \text{Value} :$
$\wedge \text{maxBal} = [\text{maxBal} \text{ except } ![\text{self}] = b]$
$\wedge \text{DidNotVoteIn}(\text{self}, b)$
$\wedge \forall p \in \text{Acceptor} \setminus \{\text{self}\} :$
$\forall w \in \text{Value} : \text{VotedFor}(p, b, w) \Rightarrow (w = v)$
$\wedge \text{SafeAt}(b, v)$
$\wedge \text{votesp} = [\text{votes} \text{ except } ![\text{self}] = \text{votes}[\text{self}]$
$\cup \{ (b, v) \}]$
$\wedge \text{maxBal} = [\text{maxBal} \text{ except } ![\text{self}] = b]$
$\wedge (\text{votesp}, \text{maxBalp}) \neq (\text{votes}, \text{maxBal})$

by def BallotAction, IncreaseMaxBal, VoteFor, vars, SafeAt, DidNotVoteIn, VotedFor

proof omitted

(6) suffices
$\exists \text{votesp}, \text{maxBal} :$
$\wedge \exists v \in \text{Value} :$
$\wedge \text{maxBal} = [\text{maxBal} \text{ except } ![\text{self}] = b]$
$\wedge \text{DidNotVoteIn}(\text{self}, b)$
$\wedge \forall p \in \text{Acceptor} \setminus \{\text{self}\} :$
$\forall w \in \text{Value} : \text{VotedFor}(p, b, w) \Rightarrow (w = v)$
$\wedge \text{SafeAt}(b, v)$
$\wedge \text{votesp} = [\text{votes} \text{ except } ![\text{self}] = \text{votes}[\text{self}]$
$\cup \{ (b, v) \}]$
$\wedge \text{maxBal} = [\text{maxBal} \text{ except } ![\text{self}] = b]$
$\wedge (\text{votesp}, \text{maxBal}) \neq (\text{votes}, \text{maxBal})$

by (6.2)

(6) define someVoted $\equiv \exists p \in \text{Acceptor} \setminus \{\text{self}\} :$
∃ w ∈ Value : VotedFor(p, b, w)

vp ≜ CHOOSE p ∈ Acceptor \ \{self\} :

∃ w ∈ Value : VotedFor(p, b, w)

v rval ≜ CHOOSE w ∈ Value : VotedFor(vp, b, w)

(6)3. someVoted ⇒ \wedge vp ∈ Acceptor

\wedge rval ∈ Value

\wedge VotedFor(vp, b, rval)

BY Zenon

(6) DEFINE v ≜ IF someVoted THEN rval

ELSE CHOOSE v ∈ Value : SafeAt(b, v)

(6)4. (v ∈ Value) ∧ SafeAt(b, v)

BY (6)1, (6)3, VT4 DEF VInv, VInv2, Ballot

(6) DEFINE votesp ≜ [votes EXCEPT ![self] = votes[self] ∪ \{⟨b, v⟩\}]

maxBalp ≜ [maxBal EXCEPT ![self] = b]

(6) WITNESS votesp, maxBalp

(6) SUFFICES maxBal[self] ≤ b

\wedge DidNotVoteIn(self, b)

\wedge \forall p ∈ Acceptor \ \{self\} :

\forall w ∈ Value : VotedFor(p, b, w) ⇒ (w = v)

\wedge votesp ≠ votes

BY (6)4, Zenon

(6)5. maxBal[self] ≤ b

BY (6)1 DEF Ballot

(6)6. DidNotVoteIn(self, b)

BY (6)1 DEF Voted, DidNotVoteIn

(6)7. ASSUME NEW p ∈ Acceptor \ \{self\},

NEW w ∈ Value,

VotedFor(p, b, w)

PROVE w = v

BY (6)7, (6)3, (6)1 DEF VInv, VInv3

(6)8. votesp ≠ votes

(7)1. votesp[self] = votes[self] ∪ \{⟨b, v⟩\}

BY (6)1, QA DEF LInv2, VInv, TypeOK

(7)2. \forall w ∈ Value : ⟨b, w⟩ /∈ votes[self]

BY (6)6 DEF DidNotVoteIn, VotedFor

(7)3. QED

BY (7)1, (7)2, (6)4, Zenon

(6)9. QED

BY (6)5, (6)6, (6)7, (6)8, Zenon

(5)4. QED

BY (5)1, (5)2, (5)3, PTL

(4)3. □LInv2 ∧ ((LInv2 ∧ ¬Voted(self)) ⇒ LInv2 ∧ Voted(self))

⇒ □Voted(self)

BY PTL

(4)4. LiveAssumption!(Q, b) ⇒ WF_vars(BallotAction(self, b))
BY Isa

(4).QED

BY (1), (2), (4), (3), (4), PTL

〈3〉 Spec \Rightarrow \Box (Voted(self) \Rightarrow \Box Voted(self))

〈4〉. (VInv \land Voted(self)) \land ([Next]_{\text{vars}} \Rightarrow (VInv \land Voted(self))')

〈5〉 SUFFICES ASSUME VInv, Voted(self), [Next]_{\text{vars}}

prove VInv' \land Voted(self)'

OBSVIOUS

〈5〉1. VInv'

BY InductiveInvariance

〈5〉2. Voted(self)'

〈6〉 CASE vars' = vars

BY def vars, Voted, VotedFor

〈6〉 CASE Next

〈7〉2. PICK a \in Acceptor, c \in Ballot : BallotAction(a, c)

BY NextDef def VInv

〈7〉3. CASE IncreaseMaxBal(a, c)

BY (7), DEF IncreaseMaxBal, Voted, VotedFor

〈7〉4. CASE \exists v \in Value : VoteFor(a, c, v)

BY (7), QA DEF VInv, TypeOK, VoteFor, Voted, VotedFor

〈7〉5. QED

BY (7), (7), (7) DEF BallotAction

〈6〉 QED

OBSVIOUS

〈5〉3. QED

BY (5), (5), (5), (5)

〈4〉3. QED

BY (4), VT2, PTL DEF Spec

〈3〉4. QED

BY (3), (3), (3), PTL

〈2〉4. (\forall a \in Q : \Diamond \Box Voted(a)) \Rightarrow \Diamond \Box (\forall a \in Q : Voted(a))

BY (1), EventuallyAlwaysForall \* doesn’t check

PROOF OMITTED

〈2〉QED

BY (2), VT2, (2), (2), (2), (2), (2), PTL

〈1〉QED

〈2〉1. Spec \land LiveAssumption!(Q, b) \Rightarrow C!Spec \land \Box (chosen \neq \{\})

BY VT3, (1), Isa

〈2〉2. Spec \land LiveAssumption!(Q, b) \Rightarrow C!Spec \land \Box (chosen \neq \{\})

BY (2), PTL

〈2〉QED

BY (2), (1), Isa