Correctness of Paxos with Replica-Set-Specific Views

MSR-TR-2004-45

Jon Howell, Jacob Lorch, and John Douceur
Microsoft Research
{howell,lorch,johndo}@microsoft.com

Abstract

We present a specification and proof of correctness for the Paxos replicated state machine consensus protocol in which replica-set-change is implemented with replica-set-specific views.

1 Introduction

We present a specification and proof of correctness for the Paxos replicated state machine consensus protocol. This technical report assumes that the reader is familiar with Paxos [2, 3], with the TLA+ specification language [4], and with our extensions to Paxos to implement replica-set change using replica-set-specific views [5].

The proof is rigorous in that it involves a high degree of detail. It is not formal in that it is not machine-checkable, and in fact not all lemmas are proven in the same degree of detail. The proofs follow a hierarchical style as recommended by Lamport [1] so that the reader can read as much or as little detail as she likes. This document provides an informal overview of the rigorous proof, outlining its structure and identifying the most important and interesting lemmas. The reader is encouraged to start by reading the highest-level statements of the interesting lemmas, and then drill down one level at a time into those statements that capture her interest or raise suspicion.

Section 2 provides a review of why Paxos works, and why our replica-set-change protocol works, in slightly more detail than the OSDI submission [5]. Section 3 is a guide to reading the specification. Understanding the structure of the specification facilitates refering to it while examining the proof. Section 4 is a guide to reading the proof proper.

2 The Argument

This section provides an overview of the basic argument behind Paxos and behind our replica-set-change extension. Knowing the argument will help the reader understand the protocol specification and the proof.

2.1 Why Paxos Works

The purpose of Paxos' agreement protocol is to determine a sequence of operations to feed to a deterministic state machine. If all cohorts agree on the sequence, then the cohorts will drive their state machines identically. We call the indices of the sequence *slots*. The goal of the protocol is for the cohorts to agree on a unique operation for each slot.

In normal operation, a distinguished cohort called the primary proposes operations for slots. If a single cohort were always the primary, it could trivially guarantee uniqueness by never proposing for the same slot twice; in fact, Paxos relies on exactly this property for the term in which a single primary serves, called a *view*. To tolerate failures, of course, the protocol cannot rely on a single primary. When a primary fails, the group can replace it by executing a *view change*.

Paxos relies on quorums to guarantee unique decisions in the presence of view changes. Every proposed operation is always prepared by a quorum of cohorts before it is chosen. Every cohort that prepares an operation promises to remember (that is, commits to stable storage before sending its Prepared message) the proposal. In the event of a view change, another quorum of cohorts elects a new primary, and conveys to that primary the list of preparations they have made in earlier views.

It is this use of quorums and relaying of prior preparations that guarantees unique decisions. If two conflicting operations were proposed for the same slot in different views, some quorum must have prepared the operation in the first view, and a second quorum must have elected the primary in the second view. Since some cohort is in both quorums, that cohort must relay the preparation from the earlier view to the primary in the later view, preventing the conflict.

2.2 Why Replica-Set Change Works

Our contribution to Paxos is to define replica-set change using replica-set-specific views. This definition makes it fairly straightforward to extend the reasoning above to handle changing replica sets.

Changing replica sets complicates the argument above, for we must consider the possibility that the preparing quorum involved members of a replica set entirely disjoint from the one that elected the later view. We resolve this quandary by assigning a well-defined replica set to decide each slot. The preparing quorum and the electing quorum will both be quorums of the same replica set.

Recall that we use replica sets that are entirely disjoint. In typical use, one might want to make less drastic changes to the set of machines participating in a consensus group. That is why we use the specific term *cohort*: a cohort is a logical entity defined as a $\langle \text{machine}, \text{epoch} \rangle$ pair. Thus every physical machine has an infinite supply of cohort identities. Whenever we change the set of machines participating, we increment the epoch, so the new replica set contains only cohorts we have never used before.

The execution of the state machine at slot $n - \alpha$ determines the replica set responsible for deciding slot n. If we make the proof invariants coinductive, we can show that all cohorts that have executed slot $n - \alpha$ agree on the replica set responsible for slot n. Because the Proposed, Prepared, and Committed messages refer to a specific slot, we know that the quorum that prepares the operation belongs to the unique replica set for that slot.

Unlike the preparation messages, the primary election messages InitiateViewChange, VcAck, and DesignatePrimary do not mention any specific slot. With replica-set-specific views, the system avoids ambiguity by using epochs to assign each new replica set a set of cohorts disjoint from all other replica sets. All of the cohorts involved in a view election therefore belong to the same replica set, and the designated primary belongs to the same replica set, as well. Since the primary only proposes for slots for which its replica set is responsible, it ensures that the electing replica set is the same as any replica set that prepares operations for the slot.

3 The Specification

The TLA+ modules that specify the system are arranged in four categories, as shown in Figure 1.

3.1 Environment

The Environment modules define the context in which the system works. PhysicalComponents assumes a set of Clients that will interact with the service. The MachineParameter module introduces the assumption of an abstract state machine $\mathsf{AbState}$ representing the desired service. The ClientIfc describes the messages comprising the communication protocol between clients and the service. Clients see the same interface regardless of whether the service is provided by a central implementation of the state machine or a replicated state machine.

DistributedComponents introduces the set of hosts from which replica sets may be constructed. MembershipMachineParameter extends the service interface to allow the service to request a replica set change, indicating the new set of hosts.

The Messenger represents the network that interconnects the clients and the replica hosts. The messenger simply records a set of all messages that have been sent in the behavior of the system; once a message has been sent, it may be received at any time thereafter. The messenger assumes a broadcast model, rather than delivering messages to particular hosts; this model is simple and

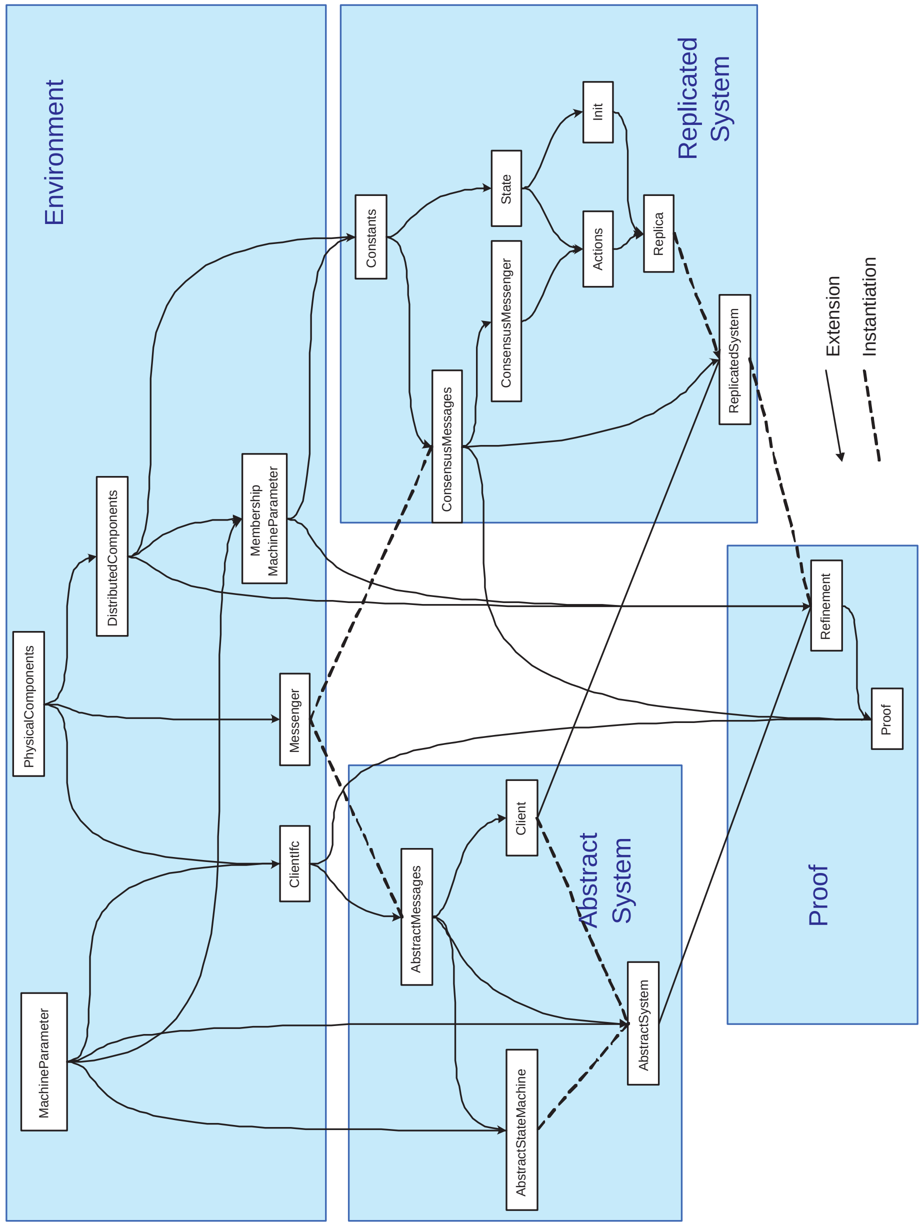


Figure 1: EXTEND and INSTANCE relationships among the specification modules

adequate for our purposes. The model allows for duplicate delivery (the ReceiveMessage action is forever enabled), out-of-order delivery (all sent messages are ready for receipt at the same time), and message drops (our specification is silent regarding liveness).

3.2 Abstract System

The Abstract System provides a reference for what the replicated state machine is trying to achieve. We wire together the set of clients and a single copy of the abstract state machine. The abstract state machine has a single action that receives network messages, processes them, and sends the reply to the client.

3.3 Replicated System

The replicated system modules form the heart of the specification.

The Constants module defines behavior-independent operators; we separate this module from the others so that it may be extended directly by the proof, and its definitions accessed without reference to a particular replica. The most important definitions build a state machine, CsState, as an extension of AbState with replica set information.

The Consensus Messages module defines the set of protocol messages exchanged among replicas. It also defines only constants, so that the message definitions can be directly referenced by the proof. The Consensus Messenger instantiates the environment Messenger to carry both the protocol messages and the client interface messages.

The State module introduces the state variables each replica maintains to participate in the protocol. These variables comprise a copy of the CsState extended state machine and the protocol control variables. The State module also defines a cohort's local idea of which replica set it is participating in. The Init module defines the initial values for the state variables in every behavior.

The Actions module defines the activity of the protocol proper. It defines four agreement actions Propose, Prepare, Commit, and Execute. It defines four view management actions InitiateViewChange, VcAck, DesignatePrimary, and BecomePrimary. It defines a Crash action that clears a cohort's volatile state.

The Replicated System module instantiates one Replica per cohort and all of the Clients, and interconnects them with the ConsensusMessenger.

Note that in the OSDI paper, we describe cohorts colocated on the same machine as sharing a common Execution Module. For simplicity, our specification assumes that each cohort has its own Execution Module. While the proof shows that no two cohorts will produce divergent executions, the simplification obscures the fact that a shared EM can make one cohort's state machine magically jump beyond the operations that cohort knows of. This property leads to correctness requirements in the implementation; a more detailed specification would model shared EMs.

3.4 Proof

The Refinement module instantiates one abstract system and one replicated system. The Proof module introduces some system-wide definitions, and follows them with a series of invariants and theorems. Its content is the focus of Section 4.

4 The Proof

In this section, we prepare the reader to read the proof.

4.1 Terminology

We have refined our terminology over time for pedagogical purposes. Our presentation of the protocol [5] uses the most recent, clearest terminology. The specification and proof use slighty older terms that match those used in the implementation. Here is a dictionary:

spec	presentation	meaning
cohort	AM	A logical replica, eligible to participate in only one replica set. A single machine may host several cohorts, distinguished by their epoch number.
opn	slot	An index into the sequence of inputs to be executed by the state machine.
membership	replica set	A set of cohorts responsible for deciding some slot.
quora	quorums	The plural of quorum, both of which sound pretty poor.
Committed	CHOSEN	The action a primary takes to announce that a slot's operation has been chosen.

4.2 Hierarchical organization

The proof is a collection of sixty-some lemmas. Section 4.4 organizes those lemmas into six general areas. Section 4.5 describes the idioms used in the detailed proofs, and points the reader at the most interesting lemmas. The proofs of the lemmas vary from a few lines to several pages.

In a conventional proof, the author must decide how much detail to present. Less detail may leave one reader wanting, but other readers may not enjoy slogging through greater detail. Our proofs are structured in a hierarchical style as recommended by Lamport [1], so that you may understand the high-level structure of each proof before diving into the details of any particular part. We recommend that you *avoid* reading the proofs linearly. Instead, read each top-level statement (Step 1, Step 2, Step 3), understand how they connect to justify the statement of the lemma, and then delve into any particular substep as you see fit, again reading breadth-first to manage detail in the substep.

4.3 Omissions

The Abstract State Machine specification does not handle client requests correctly: it treats duplicate network messages as duplicate client requests, rather than supressing them. The state machine has a client-request timestamping mechanism to prevent this problem, but we have not specified it yet because it may be reasonably omitted until proving refinement.

The replicated system specification refers to truncation points and a variable called CsStateSnapshot. We specified the system to include log truncation, but decided that it was orthogonal to our proof and hence needlessly complicated. Because the log-truncating operations are elided from our Next disjunction, any behavior admitted by the present specification has rather dull Crash actions that simply reinitialize the cohort's state. The reader may skip any references to log truncation.

We omit the proofs of the base cases for the inductive proofs of the invariants because they are trivial. Our specification has a simple Init condition, and the invariants are generally ovbiously true in the initial condition. For example, most of the invariants have as an antecedent that some message has been sent, and in the initial state, no messages have been sent; therefore, any such invariant is vacuously true.

The Refinement module should define a refinement mapping that maps states of the replicated system onto states of the abstract system. This mapping would take the AbState field of a state in KnownStates onto the AbState variable in the abstract system. Our proof would then show that the refinement holds; that is, the refinement mapping takes every behavior of the replicated system to a legal behavior of the abstract system. Once refinement is shown, we can see that the clients cannot actually distinguish whether they are attached to the replicated system or the abstract system. For sake of time, we proved only the key theorem needed for the refinement, but not the refinement itself.

There is a typesetting problem with the detailed proof: often a Reasoning block does not appear at the correct indentation level matching the step to which it applies.

4.4 The map

This technical report includes a map of the structure of the proof, broken logically into six *continents* of related lemmas. This section describes each of the six continents.

Page 14 is an overiew of the six continents, showing how they relate to one another. Pages 15–20 show each continent in detail, one continent per page.

If you can print 11 × 17 sheets (or have fantastic eyesight), you may prefer to fetch and assemble the one-page map (two sheets of 11 × 17 paper joined). The one-page map shows both the detail and context at the same time. Yellow regions on the one-page map delineate the continents.

The map should greatly assist navigating the full collection of lemmas in the same way that the hierarchical style helps navigate a single lemma. For example, one can infer the important conclusions ("outputs") of a continent by examining the dependency edges entering the continent. In Section 4.5.1, we describe the meanings of each symbol on the map.

4.4.1 State consonance

The primary goal of the proof appears in the continent labeled *state consonance.* The proof defines a notion of the *Known State* that collects the sequence of operations that have been committed by the system, and computes from that the history of the state machine's execution up through the last consecutively-available decided operation. The invariant shows that every cohort's local state agrees with some point in the history of the Known State. Typically, we expect most of the cohorts in the active replica set to have state near the most-recent available.

4.4.2 Nonconflicting decisions

The definition of globally-known state uses a CHOOSE statement (Hilbert's epsilon). Our proof strategy requires first proving that these sets are always singletons, making the choice unambiguous. Hence we must ensure that no two different operations are committed for the same slot. This statement reduces to showing that preparation by a quorum in an earlier view prevents any conflicting proposal in a later view. The latter lemma contains the primary contradiction proof underlying Paxos' view change described in Section 2.1.

4.4.3 Primaries behave well

Paxos is a practical consensus algorithm because it does minimal work in the common case, when everything is working correctly; it reserves most of its complexity for view changes, which handle failures. As a result, the good behavior of the common-case work of the protocol, proposal and preparation, is a fairly small part of the proof. The continent labeled *Primaries behave well* shows how a primary never proposes different operations for a single slot in the view it is responsible for. Even simpler, the statement Prepared Implies Proposed shows that preparers behave correctly: cohorts only prepare in response to proposals.

4.4.4 VcAcks relay information about prior prepares

When a view change does occur, it is crucial that the each cohort correctly relays information about previous operations it has prepared. Lemmas on this continent relates the Prepared Ops information in each VcAck message to the operations prepared in preceding views.

4.4.5 Elections and designation

The *Elections and designation* continent traces each view change election through from the quorum of VcAcks that ratify it to the designation of the primary, which should transmit to the primary the Prepared Op information from the election quorum.

A warning: the proof does not reason about the actual quorum involved in an election. This choice is an artifact of the exclusive use of sent messages to observe history (see Section 4.5.3): No message records the actual set of VcAcks that the view initiator considered in designating the primary for the view. Instead, we simply define a Plausible Election Quorum as any quorum whose VcAcks together justify the primary designation. Note that any Plausible Election Quorum witness differs from the actual quorum at most by the presence or absence of cohorts whose VcAck message was completely redundant with other participants in the election.

4.4.6 Replica-set change

If the system had a constant replica set, the proof would be complete. When we introduce replica-set change, however, we must be careful that the quorums in the proof of Quorum Preparation Prevents Conflicting Proposal in fact intersect. We do so by showing that they are quorums of the same replica set.

Most of the theorems are concerned with the complexity of identifying the replica set associated with a view. Recall that each message in the proposal phase of the protocol identifies a slot, which maps directly to a replica set: on cohorts through the CsState.membershipMap, and in the proof through KnownState[opn $-$ α].membershipMap.

The view change phase of the protocol is more subtle: a cohort will only initiate a view if the cohort knows that it belongs to some replica set. The Nonconflicting View Memberships theorem says that if a replica set has been established of which the view initiator is a member, then that is the only replica set associated with that view.

The key theorem Quorum Preparation Prevents Conflicting Proposal uses Proposed Implies Electing Quorum to find an election quorum in the view replica set; then it uses Proposed Constrains View Membership to ensure that the view replica set is the same as the replica set assigned to the slot under consideration.

4.5 Detailed Proofs

This section prepares the reader to dig into the detailed proofs. Sections 4.5.1 through 4.5.3 describe the idioms used in the detailed proofs. Section 4.5.4 points the reader at good places to start reading the proof.

4.5.1 Types of lemmas

Each lemma in the proof is labeled either a basic *Theorem* or an *Invariant* theorem. Invariant theorems prove the inductive step of some invariant: if R then R'.

A *nontemporal Theorem* is one whose statement has no primed expressions, and hence refers only to a single state. Such a statement's antecedent typically incorporates some invariant by reference. For convenience, any lemma may incorporate the antecedents ("Assume" statements) of a Theorem by reference. This lets us use the same name for the proof of a statement and the statement itself. To save space, we do not repeat the incorporated hypotheses in the detailed presentation.

A *temporal Theorem* is one whose statement has a primed expression, and hence relates two consecutive states of a behavior of the specification. Most of these depend on no invariants, and are simply statements of monotonicity: In any state, be it reachable in a behavior accepted by the specification or not, the specification will preserve some property in the following state. More specifically, many such theorems say that once a message has been sent, it stays sent; these follow easily from the way the Messenger always expands the SentMessages set. The monotonicity lemmas are sufficiently dull that they do not warrant inclusion in the map.

A *basic Invariant theorem* is one of the form $R \implies R'$. The proof assumes R, and proves R', providing the inductive step of a proof that all behaviors satisfying the specification hold R true at every step. Another lemma may incorporate the statement R of an invariant by name.

An *implication Invariant theorem* is one where the invariant R is of the form $P \implies Q$, so that the statement of the inductive step is $(P \implies Q) \implies (P' \implies Q')$. The inductive proofs are written as

$$\begin{array}{l} P \implies Q \\ P' \\ \hline Q' \end{array}$$

because it avoids a repetitive layer of tedious logic. The invariant itself, however, is still just $P \implies Q$, and that is the statement that is incorporated when the invariant hypothesis is refered to by name, *not* $(P \implies Q) \wedge P'$.

Each blue oval on the map is a nontemporal statement of an invariant property, the R of the invariant theorem with the corresponding name. Each green parallelogram is the corresponding statement $R \implies R'$ showing the inductive step of the proof of the invariant. Each white rectangle is a basic theorem.

Each edge represents a dependency. For example, the proof of an invariant inductive step (Proposeds In Same View Do Not Conflict) may rely on the validity of a theorem (Unique Primary Designated), which itself may rely on the assumption of an invariant property (Unique Primary Designation Message Property). When a theorem relies on an invariant inductive step (as Prepareds in Same View Do Not Conflict relies on Prepared Implies Proposed), it is because the theorem uses the inductive step to show the invariant statement true in the primed state. Red dashed edges distinguish the induction hypotheses, where an induction step relies on its invariant statement being true in the unprimed state.

Although not explicitly stated in the detailed proof, there is a temporal statement $\Box R_1 \wedge R_2 \wedge \cdots$, proven by induction. The proof assumes $R_1 \wedge R_2 \wedge \cdots$, and simply applies each invariant inductive step to prove each of $R_1', R_2', \cdots$.

4.5.2 Proof strategies

Per Lamport's hierarchical proof style, each *Step* inside a lemma is itself a little numbered but unnamed lemma. A step may refer to any step preceding it at the same scope, or to any step that its parent may refer to, recursively up to the root lemma.

Every step or lemma begins with an assertion of what is to be proved, followed by the proof itself. For example,

Introduce	$x \in S$
Assume	$P(x)$
Definition	$Q(x) \triangleq x < 7$
Assume	$Q(x)$
Prove	$R(x)$

proves the logical formula:

$$\forall x \in S : \text{ LET } Q(x) \triangleq x < 7 \text{ IN } P(x) \land Q(x) \implies R(x).$$

Variables and definitions introduced in the assertion are visible in the argument for the step. The argument itself is a series of steps. Definitions may intersperse the steps; like the statement proven by a step, that definition is visible to all of the remaining steps in the argument and their descendents. The end of an argument (and in some cases the entire argument) is a *Reasoning* block that explains how the substeps together prove the assertion.

To prove a statement by contradiction, we introduce a substep that assumes the contradiction hypothesis, and then prove FALSE.

To prove a statement by case analysis, we introduce as many substeps as we have cases. Each substep has as its assertion simply

Case P

Such a case step has the same goal statement as the parent step, but introduces the additional assumption P. A step whose assertion is *DefaultCase* assumes the negation of the disjunction of all preceding Case statements. When a step is proven by case substeps, it should be clear that the cases are exhaustive, so that this proof rule applies:

$$\begin{array}{l} P \implies R \\ Q \implies R \\ P \lor Q \\ \hline R \end{array}$$

When a DefaultCase step is present, exhaustion is automatic; in other cases, exhaustion may be obvious and left unsaid. Most proofs by case analysis break cases up according to which action has occurred.

4.5.3 Examining history through SentMessages

Many of the lemmas in the proof prove that certain bad things can never have occurred; for example, never will two commit messages be sent for the same slot and different operations. That is, in no state of any accepted behavior will one see a SentMessages set containing two conflicting commit messages.

Inspecting the SentMessages set is the only way the proof examines history. It is a sufficient historical record because in our Messenger model, once sent, a message never disappears. Lemmas in the system fall into two categories: First are *external* invariants that constrain history by relating the messages in the SentMessages set, such as the example in the previous paragraph. Second are *local* invariants that constrain a cohort's local state, perhaps with respect to SentMessages.

We commonly prove an external invariant from a local one. The local invariant may be insufficient, for example, if its antecedent makes it useful only while a cohort remains in a certain view. But we may show the inductive step of an external invariant by reference to the local one: no cohort sends the disallowed message because its local state prevents it. The external statement regards history and thus remains true forever, making it valuable for use in later lemmas.

4.5.4 Where to start

The goal statement of the proof is Local State Consonant With Known State. Read the statements of that invariant theorem, and each theorem on the path down through the map to Quroum Preparation Prevents Conflicting Proposal, the key theorem.

Dive into the substatements of Quroum Preparation Prevents Conflicting Proposal. It has links into each of the remaining continents on the map; when you see a Reasoning reference to another theorem, you can find it on a map and decide if it is an interesting direction to pursue.

5 Summary

The text and figures of this report provide a guide to the bulk of the report, a formal specification and rigorous hierarchical proof of the correctness of Paxos with replica-set-specific views.

References

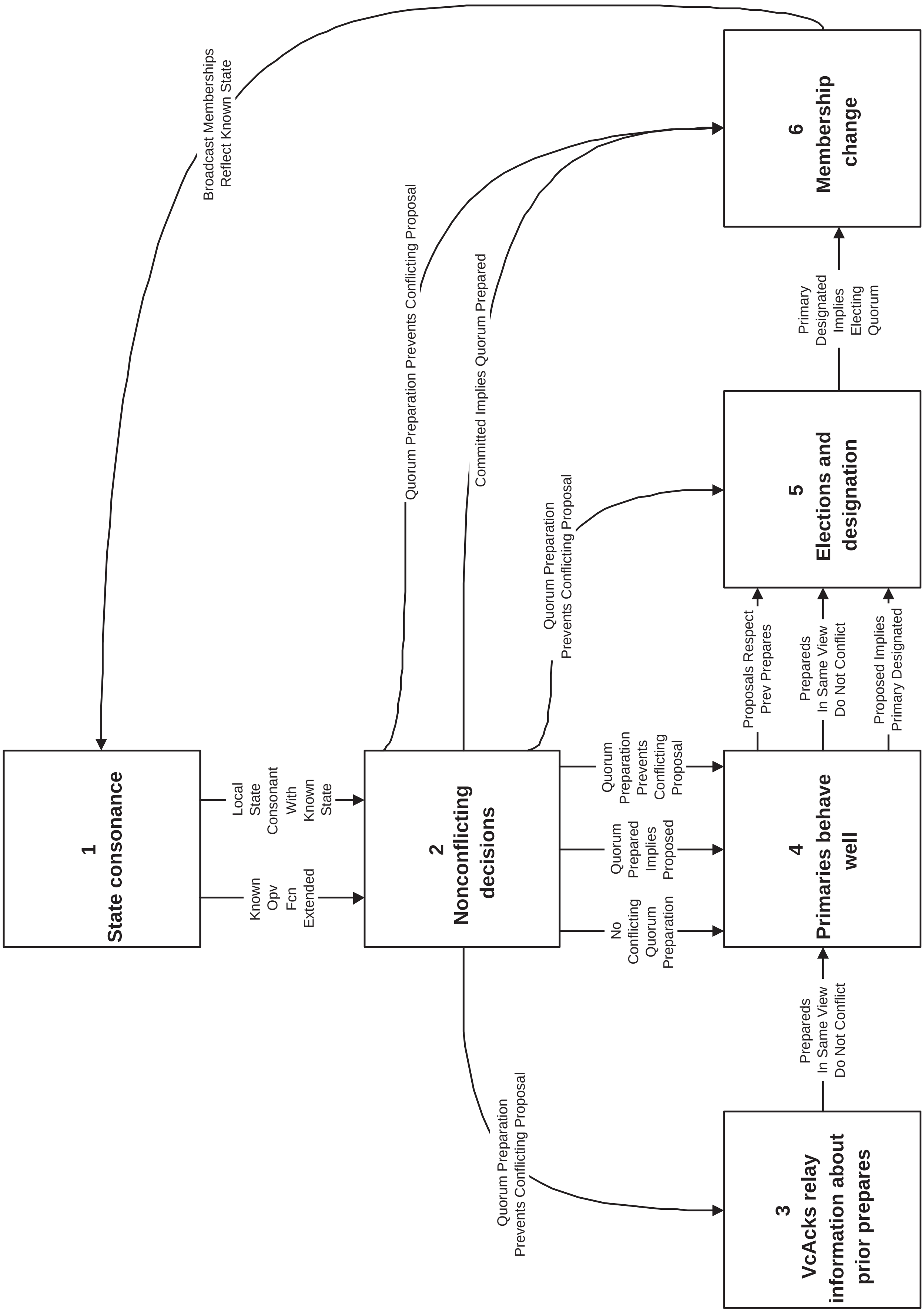
[1] L. Lamport. How to write a proof. *American Mathematical Monthly*, 102(7):600–608, August–September 1993.

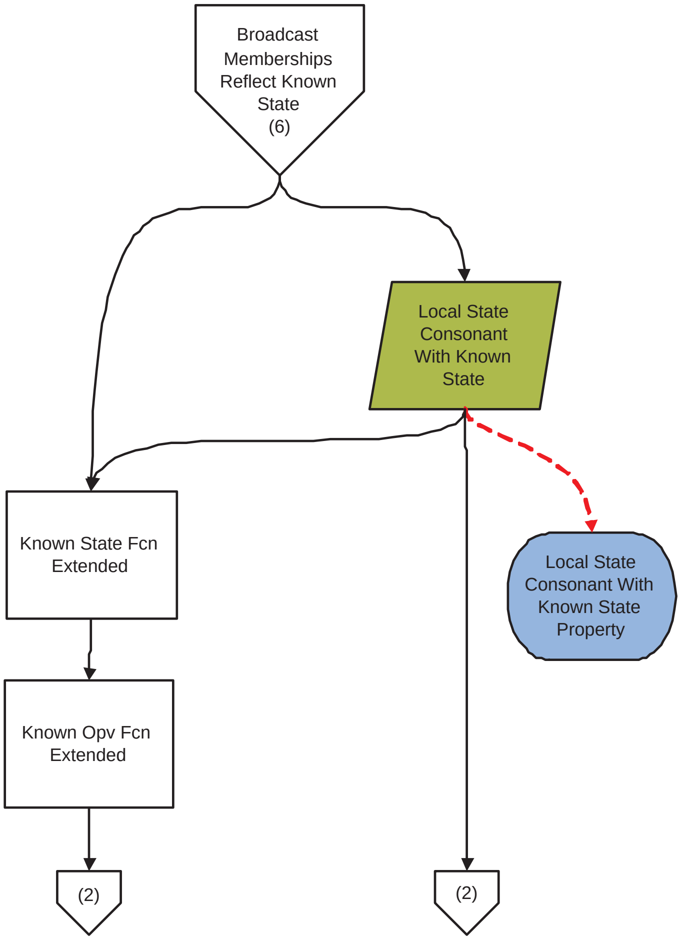
[2] L. Lamport. The part-time parliament. *ACM Transactions on Computer Systems*, 16(2):133–169, May 1998.

[3] L. Lamport. Paxos made simple. *ACM SIGACT News*, 32(4):18–25, Dec. 2001.

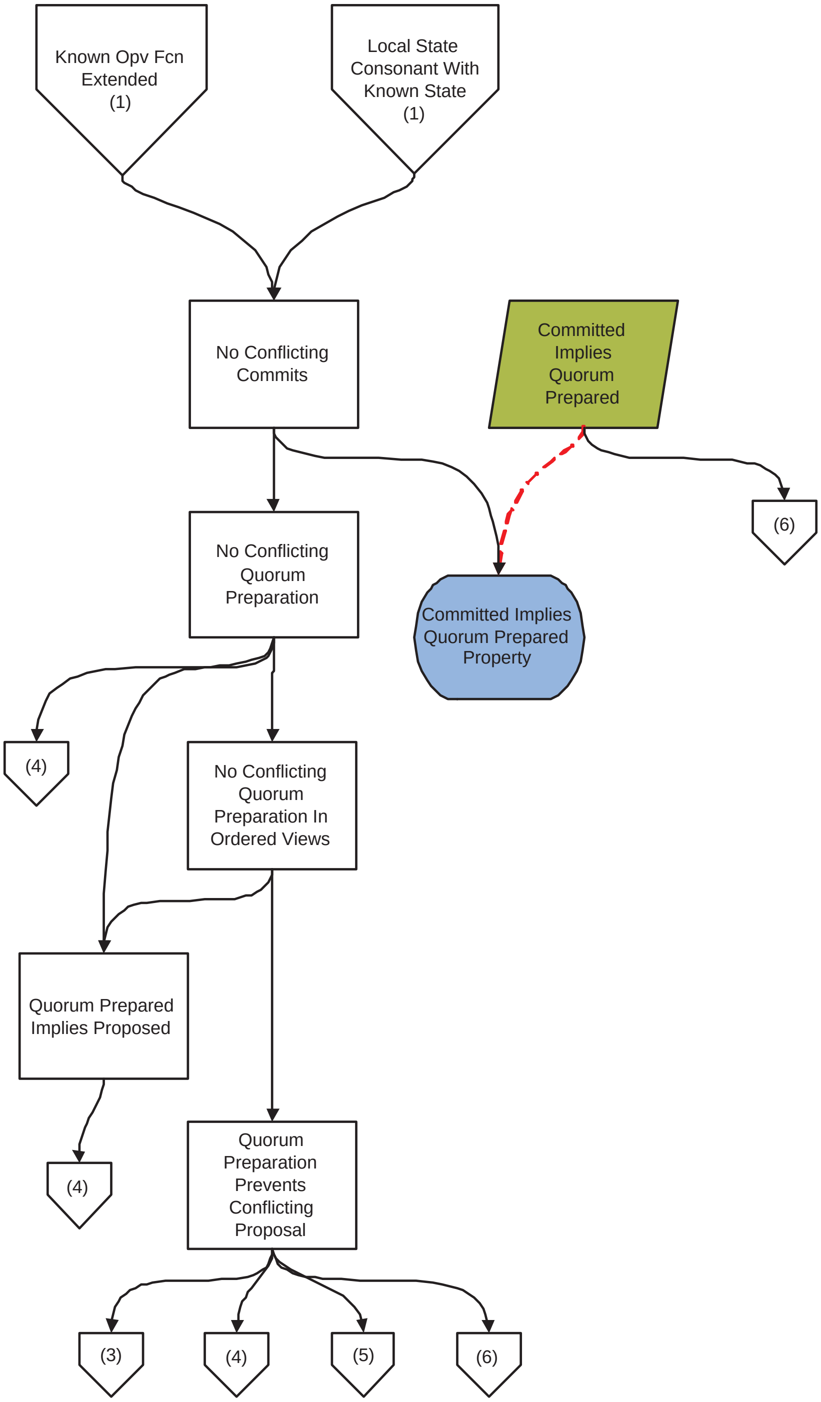
[4] L. Lamport. *Specifying Systems: The TLA+ Language and Tools for Hardware and Software Engineers.* Addison Wesley, 2003.

[5] J. Lorch, A. Adya, W. Bolosky, R. Chaiken, J. Douceur, and J. Howell. Practical state machine replication. In *submission to Sixth OSDI*, San Francisco, California, Dec. 2004.

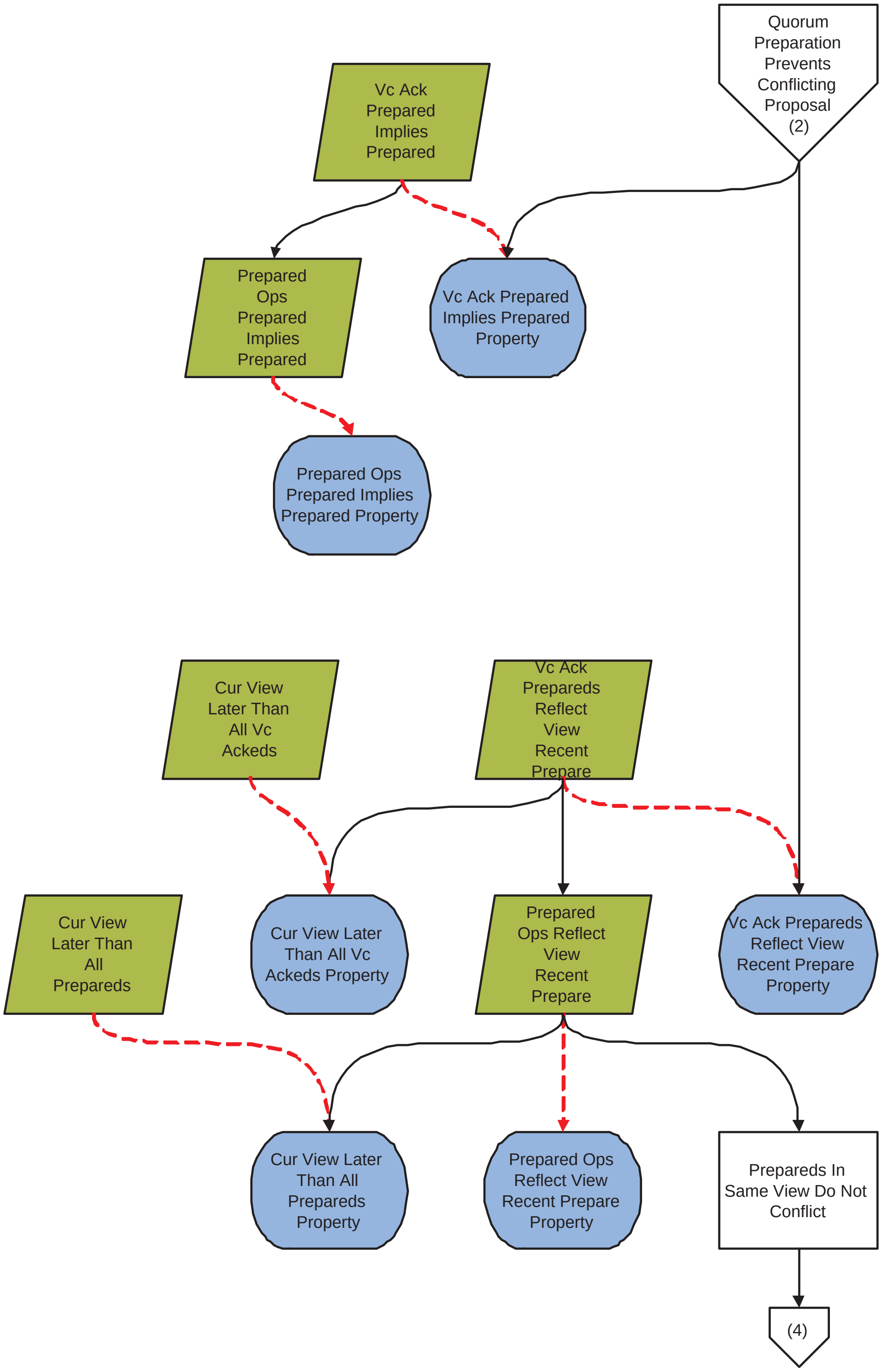

1
State consonance
2
Nonconflicting decisions
3
VcAcks relay information about prior prepares
4
Primaries behave well
5
Elections and designation
6
Membership change
Known Opv Fcn Extended
Local State Consonant With Known State
No Conflicting Quorum Preparation
Quorum Prepared Implies Proposed
Quorum Preparation Prevents Conflicting Proposal
Quorum Preparation Prevents Conflicting Proposal
Prepareds In Same View Do Not Conflict
Proposals Respect Prev Prepares
Prepareds In Same View Do Not Conflict
Proposed Implies Primary Designated
Quorum Preparation Prevents Conflicting Proposal
Primary Designated Implies Electing Quorum
Committed Implies Quorum Prepared
Quorum Preparation Prevents Conflicting Proposal
Broadcast Memberships Reflect Known State


Broadcast Memberships Reflect Known State (6)
Local State Consonant With Known State
Known State Fcn Extended
Local State Consonant With Known State Property
Known Opv Fcn Extended
(2)
(2)

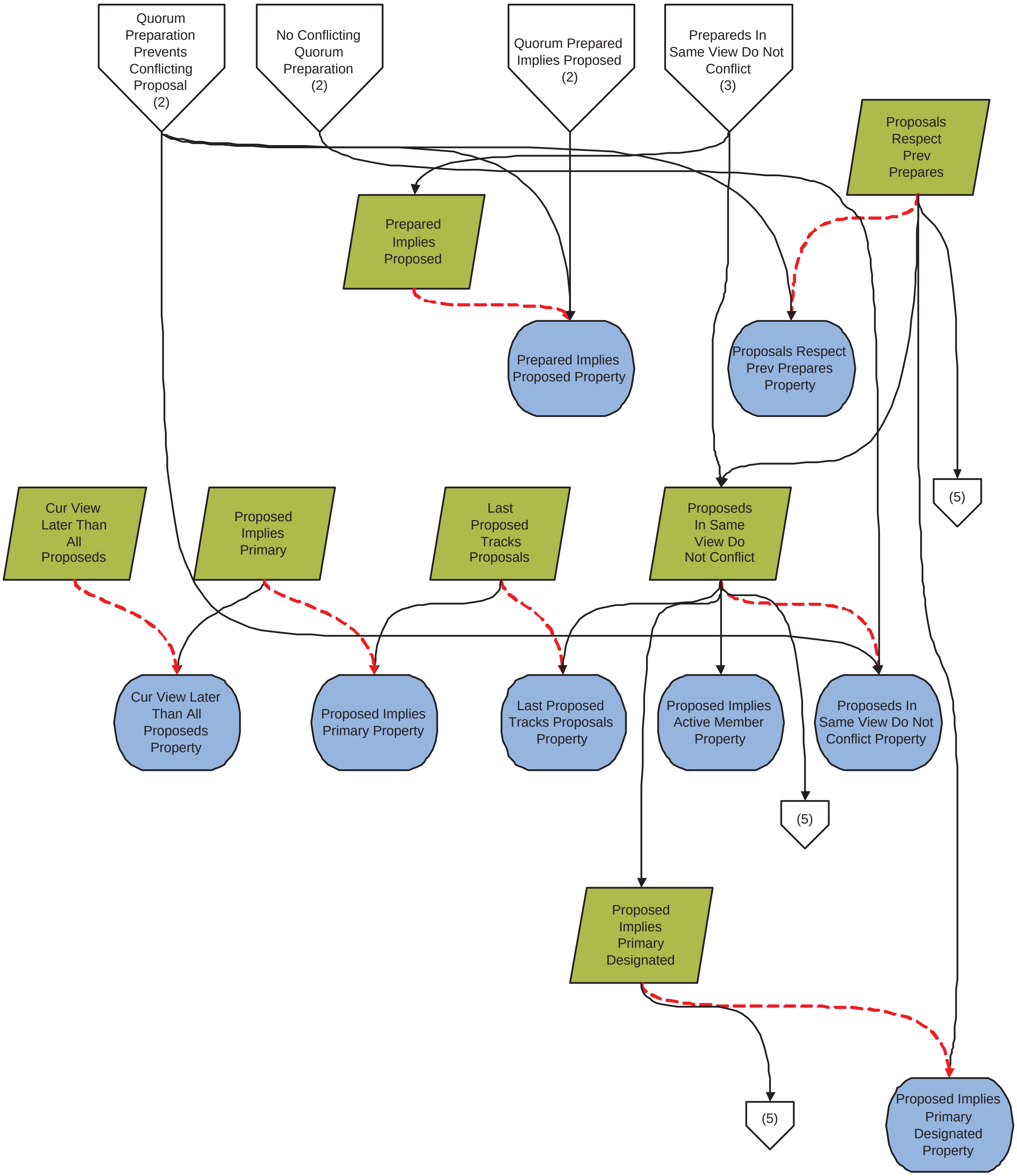
2. Nonconflicting decisions

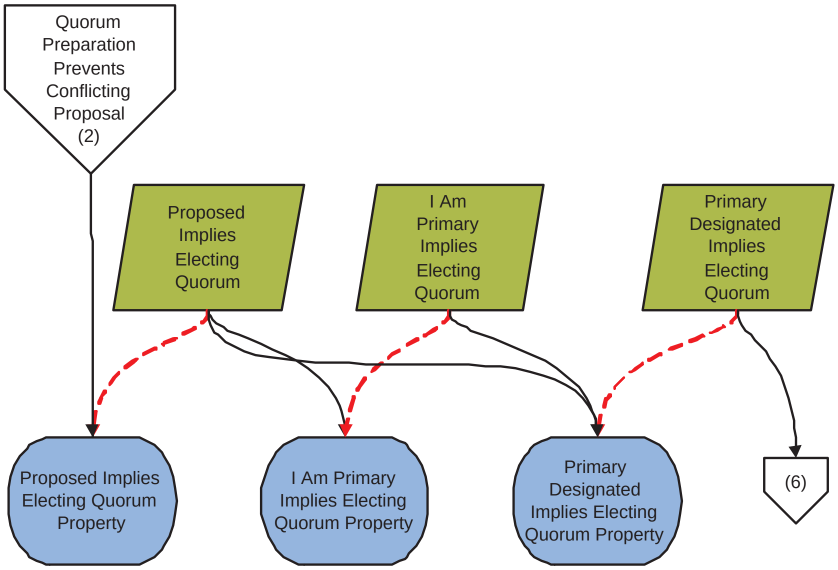


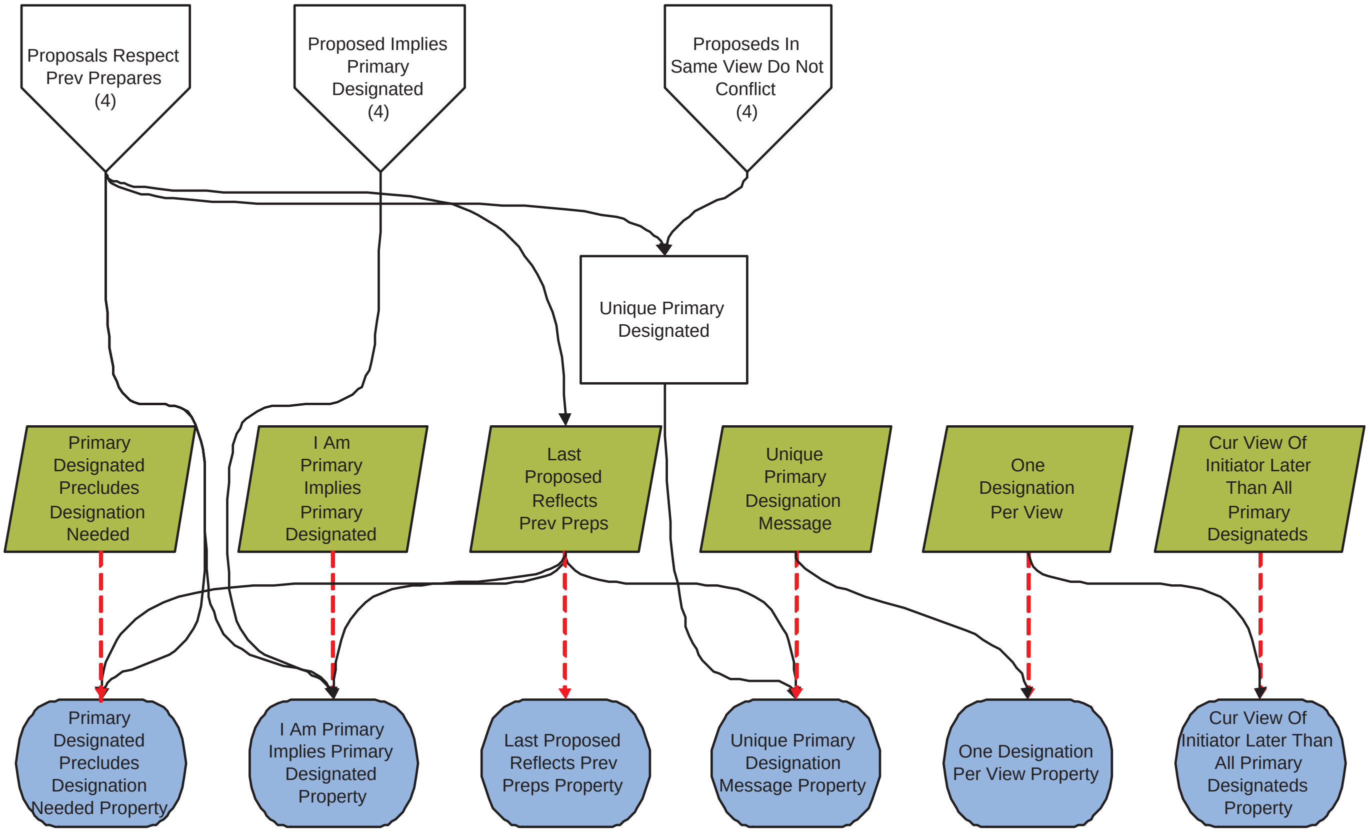
3. VcAcks relay information about prior prepares



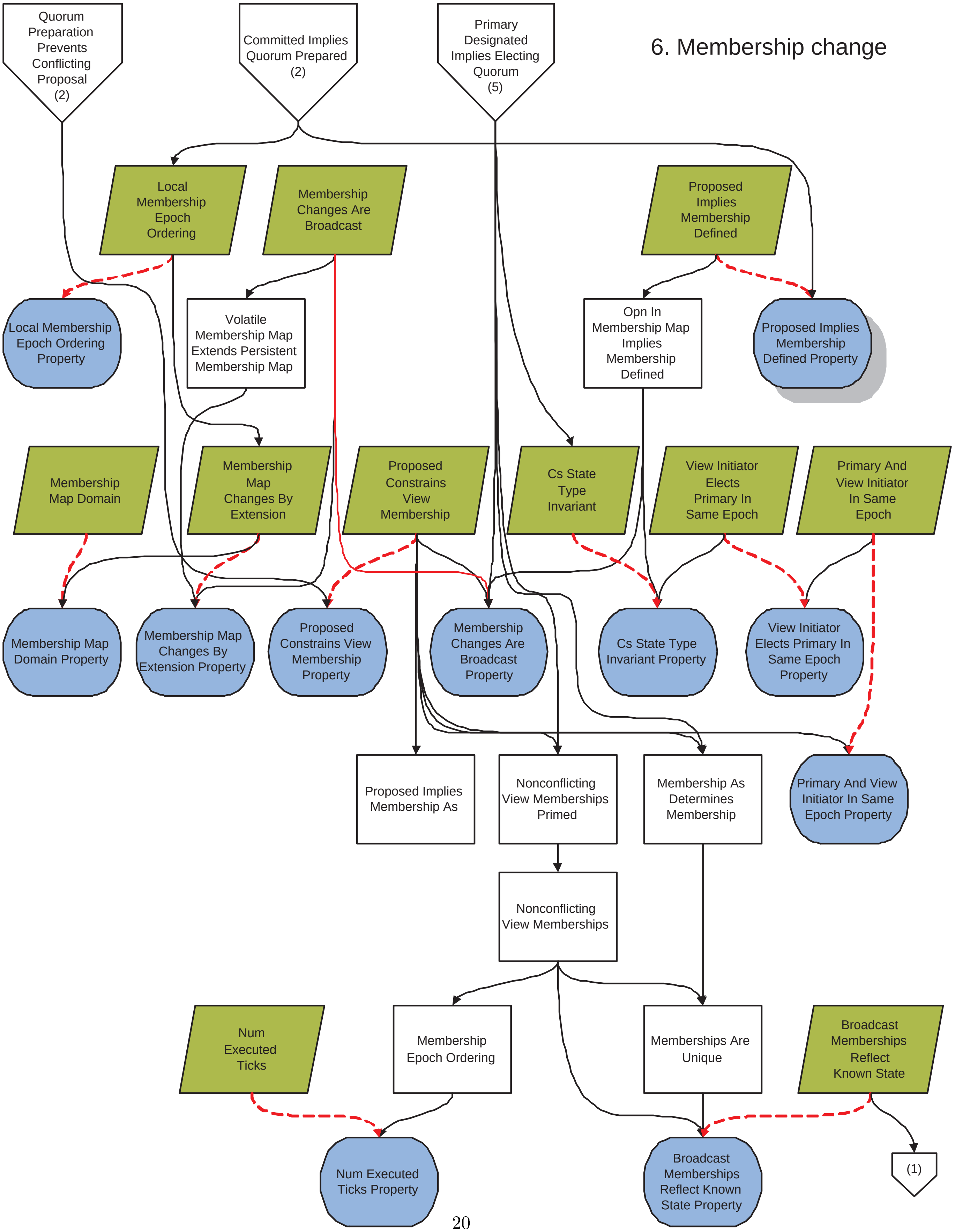
4. Primaries behave well




Quorum Preparation Prevents Conflicting Proposal (2)
Proposed Implies Electing Quorum
I Am Primary Implies Electing Quorum
Primary Designated Implies Electing Quorum
Proposed Implies Electing Quorum Property
I Am Primary Implies Electing Quorum Property
Primary Designated Implies Electing Quorum Property
(6)


Proposals Respect Prev Prepares (4)
Proposed Implies Primary Designated (4)
Proposeds In Same View Do Not Conflict (4)
Unique Primary Designated
Primary Designated Precludes Designation Needed
I Am Primary Implies Primary Designated
Last Proposed Reflects Prev Preps
Unique Primary Designation Message
One Designation Per View
Cur View Of Initiator Later Than All Primary Designateds
Primary Designated Precludes Designation Needed Property
I Am Primary Implies Primary Designated Property
Last Proposed Reflects Prev Preps Property
Unique Primary Designation Message Property
One Designation Per View Property
Cur View Of Initiator Later Than All Primary Designateds Property

6. Membership change



Contents

```
──────────── MODULE PaxosPhysicalComponents ────────────

EXTENDS FiniteSets, Naturals
Defn      Opns ≜ Nat

CONSTANT Clients

Defn    Timestamp ≜ Nat

CONSTANT ClientEndpoint
────────────────────────────────────────────────────────
```

MODULE *PaxosMachineParameter*

EXTENDS *Stubs*
CONSTANT *AbStates*

CONSTANT *AbOps*

CONSTANT *AbReplies*

CONSTANT *AbTx*

ASSUME $\mathit{AbTx} \in [\mathit{AbStates} \times \mathit{AbOps} \rightarrow [\mathit{state} : \mathit{AbStates}, \mathit{reply} : \mathit{AbReplies}]]$

CONSTANT *AbStateInit*

```
──────────────── MODULE PaxosClientIfc ────────────────

EXTENDS Stubs, Naturals, PaxosPhysicalComponents, PaxosMachineParameter
  For definition of Clients

   We only really need AbOps and AbReplies, but they're presently packaged together with the
          rest of the machine.
Defn     MTRequest ≜ "MTRequest"
Defn     MTReply ≜ "MTReply"
Defn     ClientMessageType ≜ {MTRequest, MTReply}

Defn     RequestMessage ≜
  [type : {MTRequest}, client : Clients, timestamp : Timestamp, op : AbOps]
Defn      MakeRequestMessage(i_client, i_timestamp, i_op) ≜
  [type ↦ MTRequest, op ↦ i_op]

Defn     ReplyMessage ≜
  [type : {MTReply}, client : Clients, timestamp : Timestamp, reply : AbReplies]
Defn      MakeReplyMessage(i_client, i_timestamp, i_reply) ≜
  [type ↦ MTReply, reply ↦ i_reply]
Defn     ClientMessage ≜ UNION {RequestMessage, ReplyMessage}
───────────────────────────────────────────────────────
```

```
MODULE PaxosDistributedComponents

EXTENDS PaxosPhysicalComponents
CONSTANT Hosts

CONSTANT InitialHosts

CONSTANT Alpha

Defn    Epochs ≜ Nat

Defn    Cohorts ≜ [host : Hosts, epoch : Epochs]
```

MODULE *PaxosMembershipMachineParameter*

EXTENDS *PaxosMachineParameter*, *PaxosDistributedComponents*

We assume that the "abstract" state machine also has a function on the side to specify membership changes. This doesn't belong in the abstract state machine per se, because it's aware of the distributed nature of the system. But it appears before the *Cs* state machine (the extended machine run by the distributed consensus group cohorts), because it's a parameter to the system.

Notes: 1. The membership changes specified by *AbMembership* are considered "advisory": the implementation is allowed to ignore membership change requests it doesn't care to implement.

2. The abstract interface is that the machine specifies a set of hosts to implement the group. The consensus group converts hosts into "cohorts" (\langle host, epoch \rangle - pairs), but that's a detail that the abstract interface shouldn't be aware of.

CONSTANT *AbMembership*

ASSUME $AbMembership \in [AbStates \times AbOps \rightarrow \text{SUBSET } Hosts]$

> Proof doesn't depend on how *AcceptMembershipChange* works, as long as all cohorts agree on its value (which we enforce by only supplying *CsState*, an already-agreed-upon value). One reasonable function would be "TRUE" (accept all changes).

CONSTANT $AcceptMembershipChange(\_)$

MODULE *PaxosMessenger*

EXTENDS *Util*, *PaxosPhysicalComponents*

2004.04.05.03

CONSTANT *Messages*

CONSTANT *Cohorts*

Defn $Endpoints \triangleq Cohorts \cup ClientEndpoint$

VARIABLE *SentMessages*

Defn $SentMessagesType \triangleq \text{SUBSET } Messages$

Defn $ReceiveMessageSet(messages) \triangleq messages \subseteq SentMessages$

A fine point about specification in TLA+:

Note that *SendMessageSet* includes an enabling condition: you're never allowed to re-send a message you've already sent. This condition is reasonable in this spec because: (a) resending identical messages is unnecessary for our protocol, since this Messenger can redeliver messages at any time, and (b) it ensures that any particular action only happens once. For example, once a primary proposes an operation for a slot in a view, it is no longer enabled to perform exactly that action again. Therefore, if we're in a case analysis in the proof, and we say that a Propose action relates the unprimed and primed states, we know that the action really is happening now (the message hadn't been sent before).

Without this condition, we'd have to restate all of the cases as "a message m, with the following properties, is in $SentMessages'$ and not in *SentMessages*." From that, we'd then conclude that the action must relate the two states. It's clumsier, and besides, it seems odd to leave an action "enabled" once it has occurred, when the only effect in can have is as a synonym for Stutter.

Defn $SendMessageSet(messages) \triangleq$

$\land\ messages \cap SentMessages = \{\}$

$\land\ (SentMessages') = SentMessages \cup messages$

Defn $NoMessageTraffic \triangleq SendMessageSet(\{\})$

Defn $SendMessage(m) \triangleq SendMessageSet(\{m\})$

Defn $ReceiveMessage(m) \triangleq ReceiveMessageSet(\{m\})$

MODULE *PaxosAbstractMessages*

EXTENDS *PaxosClientIfc*
VARIABLE *SentMessages*

Defn $\quad CentralCohort \triangleq$ "CentralCohort"

CONSTANT *Cohorts*

$Msgr \triangleq$
  INSTANCE *PaxosMessenger* WITH
    $Messages \leftarrow ClientMessage,\ ClientEndpoint \leftarrow ClientEndpoint$

Defn $\quad SendMessageSet(m) \triangleq Msgr!SendMessageSet(m)$

Defn $\quad ReceiveMessageSet(m) \triangleq Msgr!ReceiveMessageSet(m)$

Defn $\quad SendMessage(m) \triangleq Msgr!SendMessage(m)$

Defn $\quad ReceiveMessage(m) \triangleq Msgr!ReceiveMessage(m)$

MODULE *PaxosAbstractStateMachine*

EXTENDS $\mathit{PaxosAbstractMessages}, \mathit{PaxosMachineParameter}$
VARIABLE $\mathit{AbState}$

Defn $\mathit{Init} \triangleq \mathit{AbState} = \mathit{AbStateInit}$

Defn $\mathit{Next} \triangleq$
$\quad \exists\, m \in \mathit{RequestMessage} :$
$\qquad \land\ \mathit{ReceiveMessage}(m)$
$\qquad \land\ (\mathit{AbState}') = \mathit{AbTx}[\mathit{AbState},\ m.\mathit{op}].\mathit{state}$
$\qquad \land\ \mathit{SendMessage}(\mathit{MakeReplyMessage}(m.\mathit{client},\ m.\mathit{timestamp},\ \mathit{AbTx}[\mathit{AbState},\ m.\mathit{op}].\mathit{reply}))$

Defn $\mathit{Stutter} \triangleq$ UNCHANGED $\mathit{AbState}$

```
MODULE PaxosClient

EXTENDS PaxosClientIfc, PaxosAbstractMessages
CONSTANT ThisClient

Defn   ThisClientType ≜ Clients

VARIABLE LastTimestamp

Defn   LastTimestampType ≜ Timestamp

Defn   None ≜ "None"

VARIABLE OutstandingRequest

Defn   OutstandingRequestType ≜
  [timestamp : Timestamp, request : AbOps] ∪ {None}

Defn   Init ≜
   ∧ LastTimestamp = 0
   ∧ OutstandingRequest = None

Defn   SendRequest ≜
  ∃ newTimestamp ∈ Timestamp, newRequest ∈ AbOps :
     ∧ newTimestamp > LastTimestamp
     ∧ OutstandingRequest = None
     ∧ (OutstandingRequest′) = [timestamp ↦ newTimestamp, request ↦ newRequest]
     ∧ (LastTimestamp′) = newTimestamp
     ∧ SendMessage(MakeRequestMessage(ThisClient, newTimestamp, newRequest))

Defn   Crash ≜
   ∧ UNCHANGED LastTimestamp
   ∧ (OutstandingRequest′) = None

Defn   ReceiveReply ≜
  ∃ m ∈ ReplyMessage :
     ∧ ReceiveMessage(m)
     ∧ OutstandingRequest ≠ None
     ∧ m.client = ThisClient
     ∧ m.timestamp = OutstandingRequest.timestamp
     ∧ (OutstandingRequest′) = None

Defn   Next ≜
   ∨ SendRequest
   ∨ ReceiveReply
   ∨ Crash

Defn   Stutter ≜
```

$\land$ UNCHANGED *OutstandingRequest*
$\land$ UNCHANGED *OutstandingRequest*

```
──────────────── MODULE PaxosAbstractSystem ────────────────

EXTENDS PaxosAbstractMessages, PaxosMachineParameter
VARIABLE ClientState

Client(client) ≜
  INSTANCE PaxosClient WITH
    ThisClient ← client,
    LastTimestamp ← ClientState.lastTimestamp,
    OutstandingRequest ← ClientState.outstandingRequest

VARIABLE AbState

Server ≜ INSTANCE PaxosAbstractStateMachine

Defn        Init ≜
  ∧ (∀ client ∈ Clients : Client(client)!Init)
  ∧ Server!Init

Defn     Next ≜
  ∨ (∃ client ∈ Clients :
        ∧ Client(client)!Next
        ∧ (∀ oc ∈ Clients : oc ≠ client ⇒ Client(oc)!Stutter)
        ∧ Server!Stutter)
  ∨ ( ∧ (∀ client ∈ Clients : Client(client)!Stutter)
      ∧ Server!Next)
────────────────────────────────────────────────────────────
```

```
┌──────────────── MODULE PaxosConstants ────────────────┐

EXTENDS
  Util,
  PaxosMachineParameter,
  PaxosMembershipMachineParameter,
  PaxosDistributedComponents
Defn   Memberships ≜
  {cohortSet ∈ (SUBSET Cohorts) \ {} :
    (∃ epoch ∈ Epochs : (∀ cohort ∈ cohortSet : cohort.epoch = epoch))
  }

Defn   MakeMembership(hosts, epoch) ≜
  {cohort ∈ Cohorts :
    ( ∧ cohort.host  ∈ hosts
      ∧ cohort.epoch = epoch)
  }

Defn   EpochOf(membership) ≜
  LET
    Defn   arbitraryCohort ≜ CHOOSE cohort ∈ membership : TRUE
  IN
    arbitraryCohort.epoch

Defn   MembershipMap ≜ {[1 . . endOpn → Memberships] : endOpn ∈ Opns}

Defn   QuoraOfMembership(membership) ≜
  {memberSet ∈ SUBSET membership :
    (Cardinality(memberSet) > Cardinality(membership) ÷ 2)
  }

Defn   TimestampXReplies ≜ [timestamp : Timestamp, reply : AbReplies]

Defn   LastClientTimestampMap ≜ [Clients → TimestampXReplies]

  Consensus state machine parameters (wraps abstract state machine)

Defn   CsStates ≜
  [
    ab : AbStates,
    membershipMap : MembershipMap,
    numExecuted : Opns,
    lastClientTimestampMap : LastClientTimestampMap
  ]

Defn   NoOp ≜ [type ↦ "NoOp"]
```

```
ASSUME NoOp ∉ AbOps

ASSUME ∀ csState ∈ CsStates : AcceptMembershipChange(csState) ∈ Boolean

      An example of the definition Jay's implementation uses: allow a change only if there's not
      already one pending.
Defn    AcceptMembershipChange_NoConcurrency(csState) ≜
  ∃ membership ∈ Memberships :
    (∀ opn ∈ csState.numExecuted .. (csState.numExecuted + Alpha) :
      csState.membershipMap[opn] = membership)

Defn  CsOps ≜ AbOps ∪ {NoOp}

Defn  CsTx ≜
  [st ∈ CsStates, op ∈ CsOps ↦
    IF op = NoOp
     THEN
      st
     ELSE
      LET
        Defn    oldMembership ≜ st.membershipMap[(st.numExecuted + Alpha)]
        Defn    newMembership ≜
          IF AcceptMembershipChange(st)
           THEN
            MakeMembership(AbMembership[st.ab, op], EpochOf(oldMembership) + 1)
           ELSE
            oldMembership
        Defn    newMembershipMap ≜
          [opn ∈ 1 .. ((st.numExecuted + Alpha) + 1) ↦
            IF opn = (st.numExecuted + Alpha) + 1 THEN newMembership ELSE st.membershipMap[opn]
          ]
      IN
        [
        ab ↦ AbTx[st.ab, op],
        membershipMap ↦ newMembershipMap,
        numExecuted ↦ st.numExecuted + 1,
        lastClientTimestampMap ↦ TODO
        ]
  ]

Defn    CsTxType ≜ [CsStates × CsOps → CsStates]

Defn    CsStateInit ≜
  [
  ab ↦ AbStateInit,
  membershipMap ↦ [opn ∈ 1 .. Alpha ↦ MakeMembership(InitialHosts, 1)]
  ]
```

Defn $\mathit{CsStateInitType} \triangleq \mathit{CsStates}$

Defn $\mathit{ViewNumbers} \triangleq \mathit{Nat}$

Defn $\mathit{ViewIds} \triangleq [\mathit{viewNumber} : \mathit{ViewNumbers}, \mathit{viewInitiator} : \mathit{Cohorts}]$

Defn $\mathit{PreparedOpInfo} \triangleq [\mathit{view} : \mathit{ViewIds}, \mathit{opv} : \mathit{CsOps}]$

Defn $\mathit{PreparedOpZero} \triangleq [\mathit{view} \mapsto 0]$

Defn $\mathit{PreparedOpInfoWithZero} \triangleq \mathit{PreparedOpInfo} \cup \{\mathit{PreparedOpZero}\}$

Defn $\mathit{PreparedOpsType} \triangleq \{[\mathit{opnSet} \rightarrow \mathit{PreparedOpInfo}] : \mathit{opnSet} \in \textsc{subset}\ \mathit{Opns}\}$

Defn $\mathit{PreparedOpsWithZeroType} \triangleq$
$\{[\mathit{opnSet} \rightarrow \mathit{PreparedOpInfoWithZero}] : \mathit{opnSet} \in \textsc{subset}\ \mathit{Opns}\}$

These auxiallary operators are part of *BecomePrimary*.
They were once defined in a LET -IN , but they're factored out into global scope here so that the proof can refer to them.

Defn $\mathit{MaxPreparedOpn}(m) \triangleq$
$\mathit{Maximum}(\textsc{domain}\ (m.\mathit{prevPrepares} \cup \{m.\mathit{maxTruncationPoint}\}))$

Defn $\mathit{NotPrevPrepared}(m) \triangleq$
$((m.\mathit{maxTruncationPoint} + 1) \,..\, \mathit{MaxPreparedOpn}(m)) \setminus \textsc{domain}\ m.\mathit{prevPrepares}$

```
┌────────────── MODULE PaxosConsensusMessages ──────────────┐

EXTENDS PaxosConstants
 [2004.03.31.02]

Defn    MTProposed ≜ "MTProposed"
Defn    MTPrepared ≜ "MTPrepared"
Defn    MTCommitted ≜ "MTCommitted"
Defn    MTMembership ≜ "MTMembership"
Defn    MTVcInitted ≜ "MTVcInitted"
Defn    MTVcAcked ≜ "MTVcAcked"
Defn    MTPrimaryDesignated ≜ "MTPrimaryDesignated"
Defn    MTPersisted ≜ "MTPersisted"
Defn    MTSnapshot ≜ "MTSnapshot"
Defn    MessageType ≜
  {
  MTProposed,
  MTPrepared,
  MTCommitted,
  MTMembership,
  MTVcInitted,
  MTVcAcked,
  MTPrimaryDesignated,
  MTPersisted,
  MTSnapshot
  }

Defn    ProposedMsg ≜
  [type : {MTProposed}, sender : Cohorts, view : ViewIds, opn : Opns, opv : CsOps]
Defn     MakeProposedMsg(i_sender, i_view, i_opn, i_opv) ≜
  [type ↦ MTProposed, opn ↦ i_opn, opv ↦ i_opv]

Defn    PreparedMsg ≜
  [type : {MTPrepared}, sender : Cohorts, view : ViewIds, opn : Opns, opv : CsOps]
Defn     MakePreparedMsg(i_sender, i_view, i_opn, i_opv) ≜
  [type ↦ MTPrepared, opn ↦ i_opn, opv ↦ i_opv]
Defn    VcInittedMsg ≜ [type : {MTVcInitted}, sender : Cohorts, view : ViewIds]
Defn    MakeVcInittedMsg(i_sender, i_view) ≜ [type ↦ MTVcInitted]

Defn    VcAckedMsg ≜
  [
    type : {MTVcAcked},
    sender : Cohorts,
    view : ViewIds,
    logTruncationPoint : Opns,
```

$\mathit{preparedOps} : \mathit{PreparedOpsType}$

$]$

Defn $\mathit{MakeVcAckedMsg}(\mathit{i\_sender}, \mathit{i\_view}, \mathit{i\_logTruncationPoint}, \mathit{i\_preparedOps}) \triangleq$

$[$

$\mathit{type} \mapsto \mathit{MTVcAcked},$

$\mathit{logTruncationPoint} \mapsto \mathit{i\_logTruncationPoint},$

$\mathit{preparedOps} \mapsto \mathit{i\_preparedOps}$

$]$

Defn $\mathit{PrimaryDesignatedMsg} \triangleq$

$[$

$\mathit{type} : \{\mathit{MTPrimaryDesignated}\},$

$\mathit{sender} : \mathit{Cohorts},$

$\mathit{view} : \mathit{ViewIds},$

$\mathit{newPrimary} : \mathit{Cohorts},$

$\mathit{maxTruncationPoint} : \mathit{Opns},$

$\mathit{prevPrepares} : \mathit{PreparedOpsType}$

$]$

Defn

$\mathit{MakePrimaryDesignatedMsg}($

$\mathit{i\_sender}, \mathit{i\_view}, \mathit{i\_newPrimary}, \mathit{i\_maxTruncationPoint}, \mathit{i\_prevPrepares}) \triangleq$

$[$

$\mathit{type} \mapsto \mathit{MTPrimaryDesignated},$

$\mathit{newPrimary} \mapsto \mathit{i\_newPrimary},$

$\mathit{maxTruncationPoint} \mapsto \mathit{i\_maxTruncationPoint},$

$\mathit{prevPrepares} \mapsto \mathit{i\_prevPrepares}$

$]$

Defn $\mathit{ViewMessage} \triangleq$

$\text{UNION } \{\mathit{ProposedMsg}, \mathit{PreparedMsg}, \mathit{VcInittedMsg}, \mathit{VcAckedMsg}, \mathit{PrimaryDesignatedMsg}\}$

Defn $\mathit{CommittedMsg} \triangleq [\mathit{type} : \{\mathit{MTCommitted}\}, \mathit{sender} : \mathit{Cohorts}, \mathit{opn} : \mathit{Opns}, \mathit{opv} : \mathit{CsOps}]$

Defn $\mathit{MakeCommittedMsg}(\mathit{i\_sender}, \mathit{i\_opn}, \mathit{i\_opv}) \triangleq$

$[\mathit{type} \mapsto \mathit{MTCommitted}, \mathit{opn} \mapsto \mathit{i\_opn}, \mathit{opv} \mapsto \mathit{i\_opv}]$

Defn $\mathit{MembershipMsg} \triangleq$

$[\mathit{type} : \{\mathit{MTMembership}\}, \mathit{sender} : \mathit{Cohorts}, \mathit{opn} : \mathit{Opns}, \mathit{membership} : \mathit{Memberships}]$

Defn $\mathit{MakeMembershipMsg}(\mathit{i\_sender}, \mathit{i\_opn}, \mathit{i\_membership}) \triangleq$

$[\mathit{type} \mapsto \mathit{MTMembership}, \mathit{opn} \mapsto \mathit{i\_opn}, \mathit{membership} \mapsto \mathit{i\_membership}]$

Defn $\mathit{PersistedMsg} \triangleq [\mathit{type} : \{\mathit{MTPersisted}\}, \mathit{sender} : \mathit{Cohorts}, \mathit{opn} : \mathit{Opns}]$

Defn $\mathit{MakePersistedMsg}(\mathit{i\_sender}, \mathit{i\_opn}) \triangleq [\mathit{type} \mapsto \mathit{MTPersisted}, \mathit{opn} \mapsto \mathit{i\_opn}]$

Defn $\mathit{SnapshotMsg} \triangleq$

$[\mathit{type} : \{\mathit{MTSnapshot}\}, \mathit{sender} : \mathit{Cohorts}, \mathit{opn} : \mathit{Opns}, \mathit{snapshot} : \mathit{CsStates}]$

Defn $\mathit{MakeSnapshotMsg}(\mathit{i\_sender}, \mathit{i\_opn}, \mathit{i\_snapshot}) \triangleq$

$[\mathit{type} \mapsto \mathit{MTSnapshot}, \mathit{opn} \mapsto \mathit{i\_opn}, \mathit{snapshot} \mapsto \mathit{i\_snapshot}]$

```
Defn   ConsensusMessage ≜
  UNION {ViewMessage, CommittedMsg, MembershipMsg, PersistedMsg, SnapshotMsg}

Defn   PreparedOpInfoFromPreparedOps(preparedOps, opn) ≜
  IF opn ∈ DOMAIN preparedOps THEN preparedOps[opn] ELSE PreparedOpZero

Defn   MaxTruncationPoint(msgs) ≜ Maximum({m.logTruncationPoint : m ∈ msgs})

Defn   AggregatePreparedOps(msgs) ≜
  LET
    Defn   senders ≜ {m.sender : m ∈ msgs}
    Defn   PrevPrepDom ≜
      UNION ({DOMAIN m.preparedOps : m ∈ msgs} \ (1 .. MaxTruncationPoint(msgs)))
    Defn   Msg(cohort) ≜ CHOOSE m ∈ msgs : m.sender = cohort
    Defn   CohortPreparedOp(opn, cohort) ≜
      PreparedOpInfoFromPreparedOps(Msg(cohort).preparedOps, opn)
    Defn   PreparedOp(opn) ≜
      LET
        Defn   maxView ≜
          Maximum({CohortPreparedOp(opn, cohort).view : cohort ∈ senders})
        Defn   maxCohort ≜
          CHOOSE cohort ∈ senders : CohortPreparedOp(opn, cohort) = maxView
      IN
        CohortPreparedOp(opn, maxCohort).opv
  IN
    [opn ∈ PrevPrepDom ↦ PreparedOp(opn)]
```

Here we define the set of 'configuration records' that describe the set of legitimate *DesignatePrimary* actions. This definition is here (not in *PaxosActions*) because we use this definition in the proof.

TODO move this general note to the right place: Some modules (*PaxosConstants*, *PaxosConsensusMessages*) make no reference to state (they have no Variables), and so we can include (EXTEND) them in both the spec and the proof. By including them in the proof, we can refer to them without a (needless) reference to a specific cohort's instantiation of the constant definition. Thus we sometimes promote constant definitions up into a "constant module."

The *DesignationConfigurations.msgs* specifically disallows the empty set of messages to facilitate the proof. If we didn't, we'd need to prove that the message set is nonempty, which would require chasing around an invariant that the quorum size is always nonzero, which we'd have to chase all the way through to the *AcceptMembershipChange* predicate. Yikes! Instead, we simply disable the *DesignatePrimary* action for empty message sets. That's fine, because any system with zero members would wedge (er, "fail liveness") at view change initiation, before reaching this point.

The *DesignationConfigurations* contain redundant information: the *.view* field constrains the views of the messages in *.msgs*. So we construct a larger set of records first, and then enforce the (redundant) condition by removing those records that disobey it.

```
Defn   BasicDesignationConfigurations ≜
```

$$
\begin{array}{l}
[\\
\quad \mathit{designator} : \mathit{Cohorts},\\
\quad \mathit{view}\ : \mathit{ViewIds},\\
\quad \mathit{msgs} : (\textsc{subset}\ \mathit{VcAckedMsg}) \setminus \{\},\\
\quad \mathit{quorum} : \mathit{Memberships},\\
\quad \mathit{newPrimary} : \mathit{Cohorts}\\
]
\end{array}
$$

Defn $\mathit{DesignationConfigurations} \triangleq$

$$
\begin{array}{l}
\{\mathit{dc} \in \mathit{BasicDesignationConfigurations} :\\
\quad (\land (\forall\, m \in \mathit{dc.msgs} : \mathit{m.view} = \mathit{dc.view})\\
\quad\ \ \land\ \mathit{dc.newPrimary} \in \mathit{dc.quorum})\\
\}
\end{array}
$$

MODULE *PaxosConsensusMessenger*

$\textsc{extends}\ \mathit{PaxosConsensusMessages},\ \mathit{PaxosClientIfc}$
$\textsc{variable}\ \mathit{SentMessages}$

$$
\begin{aligned}
&\mathit{Msgr}\ \triangleq\\
&\quad \textsc{instance}\ \mathit{PaxosMessenger}\ \textsc{with}\\
&\qquad \mathit{Messages} \leftarrow (\mathit{ConsensusMessage} \cup \mathit{ClientMessage})
\end{aligned}
$$

Defn $\mathit{SendMessageSet}(m)\ \triangleq\ \mathit{Msgr}!\mathit{SendMessageSet}(m)$

Defn $\mathit{ReceiveMessageSet}(m)\ \triangleq\ \mathit{Msgr}!\mathit{ReceiveMessageSet}(m)$

Defn $\mathit{SendMessage}(m)\ \triangleq\ \mathit{Msgr}!\mathit{SendMessage}(m)$

Defn $\mathit{ReceiveMessage}(m)\ \triangleq\ \mathit{Msgr}!\mathit{ReceiveMessage}(m)$

Defn $\mathit{MessagesMatchPrototype}(\mathit{msgs},\ \mathit{proto})\ \triangleq$

$$\land\ (\forall\, m \in \mathit{msgs} : (\forall\, \mathit{field} \in \textsc{domain}\ \mathit{proto} : \mathit{field} \neq \text{“}\mathsf{sender}\text{”} \Rightarrow m[\mathit{field}] = \mathit{proto}[\mathit{field}]))$$

Defn $\mathit{EachCohortSentAMessage}(\mathit{cohorts},\ \mathit{msgs})\ \triangleq$

$$\land\ (\forall\, \mathit{cohort} \in \mathit{cohorts} : (\exists\, m \in \mathit{msgs} : m.\mathit{sender} = \mathit{cohort}))$$

Defn $\mathit{ReceiveFromQuorum}(\mathit{msg},\ \mathit{quora})\ \triangleq$

$$
\begin{aligned}
&\exists\, \mathit{mSet} \in \textsc{subset}\ \mathit{ConsensusMessage},\ \mathit{quorum} \in \mathit{quora} :\\
&\quad \land\ \mathit{ReceiveMessageSet}(\mathit{mSet})\\
&\quad \land\ \mathit{MessagesMatchPrototype}(\mathit{mSet},\ \mathit{msg})\\
&\quad \land\ \mathit{EachCohortSentAMessage}(\mathit{quorum},\ \mathit{mSet})
\end{aligned}
$$

MODULE *PaxosState*

EXTENDS *Util*, *PaxosDistributedComponents*, *PaxosMachineParameter*, *PaxosConstants*
CONSTANT *ThisCohort*

ASSUME $ThisCohort \in Cohorts$

Variables

VARIABLE *IAmPrimary*

Defn $IAmPrimaryType \triangleq Boolean$

StaleView is set upon a crash, preventing a cohort from becoming primary until it enters a new view. This keeps a primary from crashing, losing track of its non-persistent *LastProposed* variable, and then deciding to become primary again in the same view, possibly making conflicting proposals.

This new variable arose when attempting to prove the spec correct uncovered a bug. Specifically, I was working on Theorem *LastProposedTracksProposals*. 2004.04.27

VARIABLE *StaleView*

Defn $StaleViewType \triangleq Boolean$

VARIABLE *LogTruncationPoint*

Defn $LogTruncationPointType \triangleq Opns$

VARIABLE *DesignationNeeded*

Defn $DesignationNeededType \triangleq Boolean$

VARIABLE *LastProposed*

Defn $LastProposedType \triangleq Opns$

VARIABLE *PreparedOps*

VARIABLE *CsState*

Defn $CsStateType \triangleq CsStates$

VARIABLE *CsStateSnapshot*

Defn $CsStateSnapshotType \triangleq CsStates$

VARIABLE *LocalStablePoint*

Defn $LocalStablePointType \triangleq Opns$

Defn $Membership \triangleq$
CHOOSE $membership \in Range(CsState.membershipMap) : ThisCohort \in membership$

Defn $\mathit{MembershipType} \triangleq \mathit{Memberships}$

Defn $\mathit{ActiveMember} \triangleq \mathit{ThisCohort} \in \mathit{Membership}$

ActiveMember \implies \box *ActiveMember*

Defn $\mathit{OpInEpoch}(\mathit{opn}) \triangleq \mathit{ThisCohort}.\mathit{epoch} = \mathit{EpochOf}(\mathit{CsState}.\mathit{membershipMap}[\mathit{opn}])$

VARIABLE $\mathit{KnownStablePoints}$

Defn $\mathit{KnownStablePointsType} \triangleq [\mathit{Membership} \rightarrow \mathit{Opns}]$

VARIABLE $\mathit{CurView}$

Defn $\mathit{CurViewType} \triangleq \mathit{ViewIds}$

Defn $\mathit{Quora} \triangleq \mathit{QuoraOfMembership}(\mathit{Membership})$

Helpful definitions

Defn $\mathit{CollectiveStablePoint} \triangleq$

$$\begin{array}{l}
\mathit{Maximum}(\\
\quad \{\mathit{opn} \in \mathit{Opns} : \\
\qquad (\exists\, \mathit{quorum} \in \mathit{Quora} : (\forall\, \mathit{cohort} \in \mathit{quorum} : \mathit{KnownStablePoints}[\mathit{cohort}] \geq \mathit{opn})) \\
\quad \})
\end{array}$$

```
───────────────────────── MODULE PaxosInit ─────────────────────────

EXTENDS PaxosState
 Initialization

Defn    FirstPrimary ≜ [host ↦ Minimum(InitialHosts), epoch ↦ 1]

Defn    Init ≜
   ∧ IAmPrimary = FALSE
   ∧ StaleView = TRUE
   ∧ LogTruncationPoint = 0
   ∧ KnownStablePoints = [cohort ∈ Membership ↦ 0]
   ∧ DesignationNeeded  = FALSE
   ∧ LastProposed = 0
   ∧ PreparedOps = [x ∈ {} ↦ 0]
   ∧ CsState   = CsStateInit
   ∧ CurView = [viewNumber ↦ 1, viewInitiator ↦ FirstPrimary]
   ∧ CsStateSnapshot = CsStateInit
   ∧ LocalStablePoint = 0
────────────────────────────────────────────────────────────────────
```

```
──────────────── MODULE PaxosActions ────────────────

EXTENDS PaxosState, PaxosConsensusMessenger
Defn    ActiveMember_Aux ≜ ActiveMember

 Actions

Defn    Propose(opv) ≜
  LET
    Defn    opn ≜ LastProposed + 1
  IN
    ∧ ActiveMember
    ∧ opn ∈ DOMAIN CsState.membershipMap
    ∧ ThisCohort ∈ CsState.membershipMap[opn]
    ∧ IAmPrimary
    ∧ SendMessage(MakeProposedMsg(ThisCohort, CurView, opn, opv))
    ∧ (LastProposed′) = opn
    ∧ UNCHANGED IAmPrimary
    ∧ UNCHANGED PreparedOps
    ∧ UNCHANGED CsState
    ∧ UNCHANGED CurView
    ∧ UNCHANGED DesignationNeeded
    ∧ UNCHANGED CsStateSnapshot
    ∧ UNCHANGED KnownStablePoints
    ∧ UNCHANGED LocalStablePoint
    ∧ UNCHANGED LogTruncationPoint
    ∧ UNCHANGED StaleView

Defn    ProposeAction(view, opn, opv) ≜
  ∧ Propose(opv)
  ∧ view = CurView
  ∧ opn = LastProposed + 1

Defn    Prepare(m) ≜
  ∧ ActiveMember_Aux
  ∧ ReceiveMessage(m)
  ∧ m.view = CurView
  ∧ SendMessage(MakePreparedMsg(ThisCohort, CurView, m.opn, m.opv))
  ∧ (PreparedOps′) =
    [x ∈ DOMAIN (PreparedOps ∪ {m.opn}) ↦
      IF x = m.opn THEN [view ↦ CurView, op ↦ m.opv] ELSE PreparedOps[x]
    ]
  ∧ UNCHANGED IAmPrimary
  ∧ UNCHANGED LastProposed
  ∧ UNCHANGED CsState
```

$$\begin{array}{l}
\quad \land \textsc{unchanged}\ \mathit{CurView} \\
\quad \land \textsc{unchanged}\ \mathit{DesignationNeeded} \\
\quad \land \textsc{unchanged}\ \mathit{CsStateSnapshot} \\
\quad \land \textsc{unchanged}\ \mathit{KnownStablePoints} \\
\quad \land \textsc{unchanged}\ \mathit{LocalStablePoint} \\
\quad \land \textsc{unchanged}\ \mathit{LogTruncationPoint} \\
\quad \land \textsc{unchanged}\ \mathit{StaleView}
\end{array}$$

Defn $\mathit{PrepareAction}(v,\ opn,\ opv) \triangleq$

$$\begin{array}{l}
\quad \exists\, m \in \mathit{ProposedMsg} : \\
\qquad \land \mathit{Prepare}(m) \\
\qquad \land m.view = v \\
\qquad \land m.opn = opn \\
\qquad \land m.opv = opv
\end{array}$$

Defn $\mathit{Commit}(m) \triangleq$

$$\begin{array}{l}
\quad \land \mathit{ActiveMember} \\
\quad \land \mathit{ReceiveFromQuorum}(m,\ \mathit{Quora}) \\
\quad \land m.view = \mathit{CurView} \\
\quad \land \mathit{IAmPrimary} \\
\quad \land \mathit{SendMessage}(\mathit{MakeCommittedMsg}(\mathit{ThisCohort},\ m.opn,\ m.opv)) \\
\quad \land \textsc{unchanged}\ \mathit{IAmPrimary} \\
\quad \land \textsc{unchanged}\ \mathit{LastProposed} \\
\quad \land \textsc{unchanged}\ \mathit{PreparedOps} \\
\quad \land \textsc{unchanged}\ \mathit{CsState} \\
\quad \land \textsc{unchanged}\ \mathit{CurView} \\
\quad \land \textsc{unchanged}\ \mathit{DesignationNeeded} \\
\quad \land \textsc{unchanged}\ \mathit{CsStateSnapshot} \\
\quad \land \textsc{unchanged}\ \mathit{KnownStablePoints} \\
\quad \land \textsc{unchanged}\ \mathit{LocalStablePoint} \\
\quad \land \textsc{unchanged}\ \mathit{LogTruncationPoint} \\
\quad \land \textsc{unchanged}\ \mathit{StaleView}
\end{array}$$

Defn $\mathit{CommitAction}(opn,\ opv) \triangleq$

$$\begin{array}{l}
\quad \exists\, m \in \mathit{PreparedMsg} : \\
\qquad \land \mathit{Commit}(m) \\
\qquad \land m.opn = opn \\
\qquad \land m.opv = opv
\end{array}$$

Defn $\mathit{Crash} \triangleq$

$$\begin{array}{l}
\quad \land (\mathit{StaleView}') = \textsc{true} \\
\quad \land (\mathit{IAmPrimary}') = \textsc{false} \\
\quad \land (\mathit{LastProposed}') = 0 \\
\quad \land (\mathit{KnownStablePoints}') = [\mathit{cohort} \in \mathit{Membership} \mapsto 0] \\
\quad \land (\mathit{DesignationNeeded}') = \textsc{false} \\
\quad \land (\mathit{CsState}') = \mathit{CsStateSnapshot}
\end{array}$$

```
   ∧ UNCHANGED PreparedOps
   ∧ UNCHANGED LogTruncationPoint
   ∧ UNCHANGED CurView
   ∧ UNCHANGED CsStateSnapshot
   ∧ UNCHANGED LocalStablePoint
   ∧ Msgr!NoMessageTraffic

Defn   Execute(m) ≜
  LET
    Defn   newState ≜ CsTx[CsState, m.opv]
  IN
     ∧ ReceiveMessage(m)
     ∧ m.view = CurView
     ∧ m.opn = CsState.numExecuted + 1
     ∧ (CsState') = newState
     ∧ SendMessage(
           MakeMembershipMsg(
              ThisCohort, m.opn + Alpha, newState.membershipMap[(m.opn + Alpha)]))
     ∧ UNCHANGED IAmPrimary
     ∧ UNCHANGED LastProposed
     ∧ UNCHANGED PreparedOps
     ∧ UNCHANGED CurView
     ∧ UNCHANGED DesignationNeeded
     ∧ UNCHANGED CsStateSnapshot
     ∧ UNCHANGED KnownStablePoints
     ∧ UNCHANGED LocalStablePoint
     ∧ UNCHANGED LogTruncationPoint
     ∧ UNCHANGED StaleView

Defn   InitiateViewChange ≜
  LET
    Defn   newView ≜
      [viewNumber ↦ CurView.viewNumber + 1, viewInitiator ↦ ThisCohort]
  IN
     ∧ ActiveMember
     ∧ SendMessage(MakeVcInittedMsg(ThisCohort, newView))
     ∧ UNCHANGED CsState
     ∧ UNCHANGED IAmPrimary
     ∧ UNCHANGED LastProposed
     ∧ UNCHANGED PreparedOps
     ∧ UNCHANGED CurView
     ∧ UNCHANGED DesignationNeeded
     ∧ UNCHANGED CsStateSnapshot
     ∧ UNCHANGED KnownStablePoints
     ∧ UNCHANGED LocalStablePoint
```

```
    ∧ UNCHANGED LogTruncationPoint
    ∧ UNCHANGED StaleView

Defn    VcAck(m) ≜
  ∧ ActiveMember_Aux
  ∧ ReceiveMessage(m)
  ∧ m.view > CurView
  ∧ (CurView′) = m.view
  ∧ (IAmPrimary′) = FALSE
  ∧ (StaleView′) = FALSE
  ∧ (DesignationNeeded′) = (m.view.viewInitiator = ThisCohort)
  ∧ SendMessage(
       MakeVcAckedMsg(ThisCohort, CurView, LogTruncationPoint, PreparedOps))
  ∧ UNCHANGED CsState
  ∧ UNCHANGED CsStateSnapshot
  ∧ UNCHANGED KnownStablePoints
  ∧ UNCHANGED LastProposed
  ∧ UNCHANGED LocalStablePoint
  ∧ UNCHANGED LogTruncationPoint
  ∧ UNCHANGED PreparedOps

Defn    VcAckAction(view, preparedOps) ≜
  ∃ m ∈ VcInittedMsg :
    ∧ VcAck(m)
    ∧ m.view = view
    ∧ preparedOps = PreparedOps

 TODO: comments not making it out to .tla as

Defn    DesignatePrimaryAction(msgs, quorum, newPrimary) ≜
  ∧ ActiveMember
  ∧ ReceiveMessageSet(msgs)
  ∧ quorum ∈ Quora
  ∧ EachCohortSentAMessage(quorum, msgs)
  ∧ (∀ m ∈ msgs : m.view = CurView)
  ∧ ThisCohort = CurView.viewInitiator
  ∧ DesignationNeeded
  ∧ (DesignationNeeded′) = FALSE
  ∧ SendMessage(
       MakePrimaryDesignatedMsg(
          CurView,
          ThisCohort,
          newPrimary,
          MaxTruncationPoint(msgs),
          AggregatePreparedOps(msgs)))
```

Only the three parameters of the preceding operator are actually relevant for the protocol. The *DesignationConfiguration* record type and the following definition are here to facilitate the proof construction; the *.designator* and *.view* fields are strictly redundant.

Defn $\mathit{DesignatePrimary}(\mathit{config}) \triangleq$
$\land\ \mathit{config.designator} = \mathit{ThisCohort}$
$\land\ \mathit{DesignatePrimaryAction}(\mathit{config.msgs},\ \mathit{config.quorum},\ \mathit{config.newPrimary})$
$\land\ \textsc{unchanged}\ \mathit{CsState}$
$\land\ \textsc{unchanged}\ \mathit{CsStateSnapshot}$
$\land\ \textsc{unchanged}\ \mathit{CurView}$
$\land\ \textsc{unchanged}\ \mathit{IAmPrimary}$
$\land\ \textsc{unchanged}\ \mathit{KnownStablePoints}$
$\land\ \textsc{unchanged}\ \mathit{LastProposed}$
$\land\ \textsc{unchanged}\ \mathit{LocalStablePoint}$
$\land\ \textsc{unchanged}\ \mathit{LogTruncationPoint}$
$\land\ \textsc{unchanged}\ \mathit{PreparedOps}$
$\land\ \textsc{unchanged}\ \mathit{StaleView}$

Defn $\mathit{BecomePrimary}(m) \triangleq$
$\land\ \mathit{ReceiveMessage}(m)$
$\land\ m.\mathit{view} = \mathit{CurView}$
$\land\ m.\mathit{newPrimary} = \mathit{ThisCohort}$
$\land\ (\neg \mathit{IAmPrimary})$
$\land\ (\neg \mathit{StaleView})$
$\land\ (\mathit{LastProposed}') = \mathit{MaxPreparedOpn}(m)$
$\land\ (\mathit{IAmPrimary}') = \textsc{true}$
$\land\ (\mathit{PreparedOps}') = m.\mathit{prevPrepares}$
$\land\ \mathit{SendMessageSet}($
$\quad \{\mathit{MakeProposedMsg}(\mathit{ThisCohort},\ \mathit{CurView},\ \mathit{opn},\ m.\mathit{prevPrepares}[\mathit{opn}]) :$
$\qquad \mathit{opn} \in \textsc{domain}\ m.\mathit{prevPrepares}$
$\quad \}$
$\quad \cup$
$\quad \{\mathit{MakeProposedMsg}(\mathit{ThisCohort},\ \mathit{CurView},\ \mathit{opn},\ \mathit{NoOp}) : \mathit{opn} \in \mathit{NotPrevPrepared}(m)\})$
$\land\ \textsc{unchanged}\ \mathit{CsState}$
$\land\ \textsc{unchanged}\ \mathit{CsStateSnapshot}$
$\land\ \textsc{unchanged}\ \mathit{CurView}$
$\land\ \textsc{unchanged}\ \mathit{DesignationNeeded}$
$\land\ \textsc{unchanged}\ \mathit{KnownStablePoints}$
$\land\ \textsc{unchanged}\ \mathit{LocalStablePoint}$
$\land\ \textsc{unchanged}\ \mathit{LogTruncationPoint}$
$\land\ \textsc{unchanged}\ \mathit{StaleView}$

Defn $\mathit{Persist} \triangleq$
$\land\ (\mathit{CsStateSnapshot}') = \mathit{CsState}$
$\land\ \mathit{SendMessage}(\mathit{MakePersistedMsg}(\mathit{ThisCohort},\ \mathit{CsState.numExecuted}))$

Defn $\mathit{Transmit} \triangleq$

$\mathit{SendMessage}(\mathit{MakeSnapshotMsg}(\mathit{ThisCohort},\ \mathit{LocalStablePoint},\ \mathit{CsStateSnapshot}))$

Defn $\mathit{Transfer} \triangleq$

$$\begin{array}{l}
\exists\, m \in \mathit{SnapshotMsg} : \\
\quad \land\ \mathit{ReceiveMessage}(m) \\
\quad \land\ m.\mathit{snapshot}.\mathit{numExecuted} > \mathit{CsState}.\mathit{numExecuted} \\
\quad \land\ (\mathit{CsState}') = m.\mathit{snapshot}
\end{array}$$

Whenever a cohort discovers (by way of a *Membership* message) that it has been elected to an upcoming membership, it requests (by some as-yet-undefined message type) that an existing cohort Transmit its state to the new electee.

That chain of actions is only needed for liveness, to ensure that new cohorts get a state transfer and find their *ActiveMember* predicate true. Since we're proving nothing about liveness, we don't bother specifying the extra action and message.

Defn $\mathit{UpdateStablePoints} \triangleq$

$$\begin{array}{l}
\exists\, m \in \mathit{PersistedMsg} : \\
\quad \land\ \mathit{ReceiveMessage}(m) \\
\quad \land\ m.\mathit{opn} > \mathit{KnownStablePoints}[m.\mathit{sender}] \\
\quad \land\ (\mathit{KnownStablePoints}') = [\mathit{KnownStablePoints}\ \textsc{except}\ ![m.\mathit{sender}] = m.\mathit{opn}]
\end{array}$$

Defn $\mathit{Truncate}(\mathit{collectiveStablePoint}) \triangleq$

$$\begin{array}{l}
\land\ \mathit{collectiveStablePoint} \quad > \mathit{LogTruncationPoint} \\
\land\ (\mathit{LogTruncationPoint}') = \mathit{collectiveStablePoint} \\
\land\ (\mathit{PreparedOps}') = \\
\quad [i \in \textsc{domain}\ (\mathit{PreparedOps} \setminus \{1 \,..\, \mathit{collectiveStablePoint}\}) \mapsto \mathit{PreparedOps}[i]]
\end{array}$$

Defn $\mathit{Truncate1} \triangleq$

$$\begin{array}{l}
\land\ \mathit{ActiveMember} \\
\land\ \mathit{Truncate}(\mathit{CollectiveStablePoint})
\end{array}$$

Defn $\mathit{Next} \triangleq$

$$\begin{array}{l}
\lor\ (\exists\, \mathit{opv} \in \mathit{CsOps} : \mathit{Propose}(\mathit{opv})) \\
\lor\ (\exists\, m \in \mathit{ProposedMsg} : \mathit{Prepare}(m)) \\
\lor\ (\exists\, m \in \mathit{PreparedMsg} : \mathit{Commit}(m)) \\
\lor\ (\exists\, m \in \mathit{CommittedMsg} : \mathit{Execute}(m)) \\
\lor\ \mathit{Crash} \\
\lor\ \mathit{InitiateViewChange} \\
\lor\ (\exists\, m \in \mathit{VcInittedMsg} : \mathit{VcAck}(m)) \\
\lor\ (\exists\, \mathit{config} \in \mathit{DesignationConfigurations} : \mathit{DesignatePrimary}(\mathit{config})) \\
\lor\ (\exists\, m \in \mathit{PrimaryDesignatedMsg} : \mathit{BecomePrimary}(m))
\end{array}$$

Defn $\mathit{Stutter} \triangleq$

$$\begin{array}{l}
\land\ \textsc{unchanged}\ \mathit{IAmPrimary} \\
\land\ \textsc{unchanged}\ \mathit{LogTruncationPoint} \\
\land\ \textsc{unchanged}\ \mathit{DesignationNeeded} \\
\land\ \textsc{unchanged}\ \mathit{LastProposed} \\
\land\ \textsc{unchanged}\ \mathit{PreparedOps}
\end{array}$$

$\land \textsc{unchanged}\ \mathit{CsState}$
$\land \textsc{unchanged}\ \mathit{CsStateSnapshot}$
$\land \textsc{unchanged}\ \mathit{LocalStablePoint}$
$\land \textsc{unchanged}\ \mathit{KnownStablePoints}$
$\land \textsc{unchanged}\ \mathit{CurView}$

MODULE *PaxosReplica*

EXTENDS *PaxosInit*, *PaxosActions*

```
──────────────── MODULE PaxosReplicatedSystem ────────────────

EXTENDS
  PaxosPhysicalComponents,
  PaxosDistributedComponents,
  PaxosMachineParameter,
  PaxosConsensusMessenger
VARIABLE replicaState

Replica(cohort) ≜
  INSTANCE PaxosReplica WITH
    ThisCohort   ← cohort,
    IAmPrimary ← replicaState[cohort].IAmPrimary,
    StaleView ← replicaState[cohort].StaleView,
    LogTruncationPoint ← replicaState[cohort].LogTruncationPoint,
    DesignationNeeded ← replicaState[cohort].DesignationNeeded,
    LastProposed ← replicaState[cohort].LastProposed,
    PreparedOps ← replicaState[cohort].LastProposed,
    CsState ← replicaState[cohort].CsState,
    CsStateSnapshot ← replicaState[cohort].CsStateSnapshot,
    LocalStablePoint ← replicaState[cohort].LocalStablePoint,
    KnownStablePoints ← replicaState[cohort].KnownStablePoints,
    CurView ← replicaState[cohort].CurView

VARIABLE clientState

Client(client) ≜
  INSTANCE PaxosClient WITH
    ThisClient ← client,
    OutstandingRequest ← clientState[client].OutstandingRequest,
    LastTimestamp ← clientState[client].LastTimestamp

Defn    Init  ≜ ∀ c ∈ Cohorts : Replica(c)!Init

Defn    Next ≜
  ∧ DOMAIN replicaState = Cohorts
  ∧ DOMAIN clientState = Clients
  ∧ ( ∨ (∃ cohort ∈ Cohorts :
              ∧ Replica(cohort)!Next
              ∧ (∀ other ∈ Cohorts \ {cohort} : Replica(other)!Stutter)
              ∧ (∀ client ∈ Clients : Client(client)!Stutter))
      ∨ (∃ client ∈ Clients :
              ∧ Client(client)!Next
              ∧ (∀ other ∈ Clients \ {client} : Client(other)!Stutter)
              ∧ (∀ cohort ∈ Cohorts : Replica(cohort)!Stutter)))
──────────────────────────────────────────────────────────────
```

```
---------------------------- MODULE PaxosRefinement ----------------------------

EXTENDS
  PaxosMachineParameter,
  PaxosPhysicalComponents,
  PaxosDistributedComponents,
  PaxosMembershipMachineParameter
VARIABLE hlState

HL ≜
   INSTANCE PaxosAbstractSystem WITH
     SentMessages ← hlState.SentMessages,
     AbState ← hlState.AbState,
     ClientState ← hlState.ClientState,
     Cohorts ← {"DummyCohort"}

VARIABLE llState

LL ≜
   INSTANCE PaxosReplicatedSystem WITH
     SentMessages ← llState.SentMessages,
     replicaState    ← llState.replicaState,
     clientState ← llState.clientState
=================================================================================
```

MODULE *PaxosProof*

```
EXTENDS PaxosRefinement, PaxosClientIfc, PaxosConsensusMessages

Defn    SentMessages ≜ LL!Msgr!SentMessages

Defn    SentMessagesMatching(sender, mtype) ≜
  {m ∈ SentMessages ∩ mtype : (m.sender = sender)}

Defn    VcAcked(v, c, preparedOps) ≜
  ∃ m ∈ SentMessages ∩ VcAckedMsg :
    ∧ m.sender = c
    ∧ m.view = v
    ∧ m.preparedOps = preparedOps

Defn  VcAckedView(v, c) ≜
  ∃ preparedOps ∈ PreparedOpsType : VcAcked(v, c, preparedOps)

Defn  VcAckPreparedOpAs(v, c, opn, preparedOpInfo) ≜
  ∃ preparedOps ∈ PreparedOpsType :
    ∧ VcAcked(v, c, preparedOps)
    ∧ preparedOpInfo = PreparedOpInfoFromPreparedOps(preparedOps, opn)

Defn  ChooseVcAckPreparedOpInfo(v, c, opn) ≜
  CHOOSE preparedOpInfo ∈ PreparedOpInfo :
    VcAckPreparedOpAs(v, c, opn, preparedOpInfo)

Defn   PrimaryDesignatedAs(view, primary) ≜
  ∃ m ∈ SentMessages ∩ PrimaryDesignatedMsg :
    ∧ m.view = view
    ∧ m.newPrimary = primary

Defn  PrimaryDesignated(view) ≜
  ∃ primary ∈ Cohorts : PrimaryDesignatedAs(view, primary)

Defn  ProposedAs(v, c, opn, opv) ≜
  ∃ m ∈ SentMessages ∩ ProposedMsg :
    ∧ m.sender = c
    ∧ m.view = v
    ∧ m.opn = opn
    ∧ m.opv = opv

Defn  Proposed(v, c, opn) ≜ ∃ opv ∈ CsOps : ProposedAs(v, c, opn, opv)

Defn  ProposedByAnyAs(v, opn, opv) ≜ ∃ c ∈ Cohorts : ProposedAs(v, c, opn, opv)

Defn  ProposedByAny(v, opn) ≜ ∃ opv ∈ CsOps : ProposedByAnyAs(v, opn, opv)
```

Defn $\mathit{PreparedAs}(v,\ c,\ \mathit{opn},\ \mathit{opv}) \triangleq$

$$\begin{array}{l}
\exists\, m \in \mathit{SentMessages} \cap \mathit{PreparedMsg} : \\
\quad \land\ m.\mathit{sender} = c \\
\quad \land\ m.\mathit{view} = v \\
\quad \land\ m.\mathit{opn} = \mathit{opn} \\
\quad \land\ m.\mathit{opv} = \mathit{opv}
\end{array}$$

Defn $\mathit{Prepared}(v,\ c,\ \mathit{opn}) \triangleq \exists\, \mathit{opv} \in \mathit{CsOps} : \mathit{PreparedAs}(v,\ c,\ \mathit{opn},\ \mathit{opv})$

Defn $\mathit{DesignationReflectsVcAcks}(\mathit{view},\ \mathit{cohortSet}) \triangleq$

$$\begin{array}{l}
\exists\, \mathit{designationMsg} \in \mathit{LL!SentMessages} \cap \mathit{PrimaryDesignatedMsg}, \\
\quad \mathit{vcAckMsgSet} \in \textsc{subset}\ (\mathit{LL!SentMessages} \cap \mathit{VcAckedMsg}) \\
\quad : \\
\quad \land\ (\forall\, \mathit{vcAckMsg} \in \mathit{vcAckMsgSet} : \\
\qquad\quad \land\ \mathit{vcAckMsg}.\mathit{sender} \in \mathit{cohortSet} \\
\qquad\quad \land\ \mathit{vcAckMsg}.\mathit{view} = \mathit{view}) \\
\quad \land\ \mathit{designationMsg}.\mathit{view} = \mathit{view} \\
\quad \land\ \mathit{designationMsg}.\mathit{prevPrepares} = \mathit{AggregatePreparedOps}(\mathit{vcAckMsgSet})
\end{array}$$

A constant (level − 0) predicate that defines whether a given *SentMessage* set defines the membership of *opn* as 'membership'.

Defn $\mathit{MembershipAs}(\mathit{opn},\ \mathit{membership},\ \mathit{sentMessages}) \triangleq$

$$\begin{array}{l}
\textsc{if}\ \mathit{opn} \leq \mathit{Alpha} \\
\ \ \textsc{then} \\
\ \ \ \mathit{membership} = \mathit{MakeMembership}(\mathit{InitialHosts},\ 1) \\
\ \ \textsc{else} \\
\ \ \ \exists\, \mathit{msg} \in \mathit{sentMessages} \cap \mathit{MembershipMsg} : \\
\qquad \land\ \mathit{msg}.\mathit{opn} = \mathit{opn} \\
\qquad \land\ \mathit{msg}.\mathit{membership} = \mathit{membership}
\end{array}$$

A level − 1 (state-sensitive) expression that extracts a (the) membership declared for *opn* in the current state.

Defn $\mathit{Membership}(\mathit{opn}) \triangleq$

$$\begin{array}{l}
\textsc{choose}\ \mathit{membership} \in \mathit{Memberships} : \\
\quad \mathit{MembershipAs}(\mathit{opn},\ \mathit{membership},\ \mathit{LL!SentMessages})
\end{array}$$

Defn $\mathit{Quora}(\mathit{opn}) \triangleq \mathit{QuoraOfMembership}(\mathit{Membership}(\mathit{opn}))$

The *ViewMembership* is the membership that contains the cohort that initiated the specified view

Defn $\mathit{ViewMembership}(\mathit{view}) \triangleq$

$$\begin{array}{l}
\textsc{choose}\ \mathit{membership} \in \mathit{Memberships} : \\
\quad \land\ (\exists\, \mathit{opn} \in \mathit{Opns} : \mathit{MembershipAs}(\mathit{opn},\ \mathit{membership},\ \mathit{LL!SentMessages})) \\
\quad \land\ \mathit{EpochOf}(\mathit{membership}) = \mathit{view}.\mathit{viewInitiator}.\mathit{epoch}
\end{array}$$

Defn $\mathit{QuorumPreparedAs}(v,\ \mathit{opn},\ \mathit{opv}) \triangleq$

$$\exists\, \mathit{quorum} \in \mathit{Quora}(\mathit{opn}) : (\forall\, c \in \mathit{quorum} : \mathit{PreparedAs}(v,\ c,\ \mathit{opn},\ \mathit{opv}))$$

Defn $\mathit{MembershipDefined}(\mathit{opn}) \triangleq$
$\exists\, \mathit{membership} \in \mathit{Memberships} : \mathit{MembershipAs}(\mathit{opn}, \mathit{membership}, \mathit{LL}!\mathit{SentMessages})$

Defn $\mathit{QuorumPrepared}(v, \mathit{opn}) \triangleq \exists\, \mathit{opv} \in \mathit{CsOps} : \mathit{QuorumPreparedAs}(v, \mathit{opn}, \mathit{opv})$

Defn $\mathit{CommittedAs}(c, \mathit{opn}, \mathit{opv}) \triangleq$
$\exists\, m \in \mathit{SentMessages} \cap \mathit{CommittedMsg} :$
$\quad \land\, m.\mathit{sender} = c$
$\quad \land\, m.\mathit{opn} = \mathit{opn}$
$\quad \land\, m.\mathit{opv} = \mathit{opv}$

Defn $\mathit{CommittedByAnyAs}(\mathit{opn}, \mathit{opv}) \triangleq \exists\, c \in \mathit{Cohorts} : \mathit{CommittedAs}(c, \mathit{opn}, \mathit{opv})$

Defn $\mathit{CommittedByAny}(\mathit{opn}) \triangleq$
$\exists\, c \in \mathit{Cohorts}, \mathit{opv} \in \mathit{CsOps} : \mathit{CommittedAs}(c, \mathit{opn}, \mathit{opv})$

Defn $\mathit{PrimaryDesignatedPrevPrep}(\mathit{view}, \mathit{opn}, \mathit{opv}) \triangleq$
$\exists\, m \in \mathit{SentMessages} \cap \mathit{PrimaryDesignatedMsg} :$
$\quad \land\, m.\mathit{view} = \mathit{view}$
$\quad \land\, \mathit{opn} \in \textsc{domain}\ m.\mathit{prevPrepares}$
$\quad \land\, m.\mathit{prevPrepares}[\mathit{opn}] = \mathit{opv}$

The *PlausibleElectionQuorum* predicate is meaningful only when $\mathit{PrimaryDesignated}(\mathit{view})$. It is true when quorum is a set of cohorts that could reasonably be an election quorum for the view: they all *VcAcked* the view, and the primary designation reflects their input.

(Note that this predicate doesn't actually verify the quorumness of the supplied cohort set "quorum". In fact, it doesn't even know the *opn*.)

We fiddle with "Plausible" election quorums because we can't actually tell by looking at the message history which quorum the view-change initiator actually used. It may have used a large quorum that includes cohorts whose votes didn't actually matter. But then the results are identical to the case where a smaller quorum was used, and the proof works as well either way.

Defn $\mathit{PlausibleElectionQuorum}(\mathit{view}, \mathit{quorum}) \triangleq$
$\land\, (\forall\, \mathit{cohort} \in \mathit{quorum} : \mathit{VcAckedView}(\mathit{view}, \mathit{cohort}))$
$\land\, \mathit{DesignationReflectsVcAcks}(\mathit{view}, \mathit{quorum})$

A function $f2$ extends $f1$ if it simply defines values for new inputs, leaving all old ones as they were.

Defn $\mathit{FcnExtends}(f2, f1) \triangleq$
$\land\, \textsc{domain}\ f1 \subseteq \textsc{domain}\ f2$
$\land\, (\forall\, x \in \textsc{domain}\ f1 : f1[x] = f2[x])$

Defn $\mathit{MaxKnownOpn} \triangleq$
$\textsc{choose}\ \mathit{maxOpn} \in \mathit{Opns} :$
$\quad \land\, (\forall\, \mathit{opn} \in 1 \,..\, \mathit{maxOpn} : \mathit{CommittedByAny}(\mathit{opn}))$
$\quad \land\, (\lnot \mathit{CommittedByAny}(\mathit{maxOpn} + 1))$

Defn

$KnownOpv[$
$\quad opn \in 1 \,..\, MaxKnownOpn] \triangleq$
$\quad \textsc{choose}\ opv \in CsOps : CommittedByAnyAs(opn,\ opv)$

Defn
$KnownState[$
$\quad opn \in 0 \,..\, MaxKnownOpn] \triangleq$
$\quad \textsc{if}\ opn = 0\ \textsc{then}\ CsStateInit\ \textsc{else}\ \ CsTx[(KnownState[(opn - 1)]),\ (KnownOpv[opn])]$

Defn $\quad Consonant(state) \triangleq$
$\quad \land\ state.numExecuted \in \textsc{domain}\ KnownState$
$\quad \land\ state = KnownState[state.numExecuted]$

Defn $\quad KnownMembership(opn) \triangleq KnownState[(opn - Alpha)].membershipMap[opn]$

Defn $\quad ClientRequestIdentifier \triangleq [client : Clients,\ timestamp : Timestamp]$

Defn $\quad ClientRequestsSubmitted \triangleq$
$\quad \{cri \in ClientRequestIdentifier :$
$\qquad (\exists\, m \in SentMessages \cap RequestMessage :$
$\qquad\qquad \land\ m.client = cri.client$
$\qquad\qquad \land\ m.timestamp = cri.timestamp)$
$\quad \}$

Defn $\quad EpochsOrdered(map) \triangleq$
$\quad \forall\, opn1 \in \textsc{domain}\ map,\ opn2 \in \textsc{domain}\ map :$
$\qquad \land\ opn1 < opn2$
$\qquad \land\ EpochOf(map[opn1]) \leq EpochOf(map[opn2])$
$\qquad \land\ (EpochOf(map[opn1]) = EpochOf(map[opn2]) \Rightarrow map[opn1] = map[opn2])$

Theorem *SentMessagesMonotonic*
$SentMessages \subseteq (SentMessages')$

Reasoning: Every action includes a *SendMessageSet* partial action; Definition $\cup$

Theorem *PrimaryDesignatedMonotonic*
Introduce $\quad view \in ViewIds$
Assume $\quad PrimaryDesignated(view)$
Prove $\quad PrimaryDesignated(view)'$

Reasoning: *Ref*:*SentMessagesMonotonic* ; existential witness carries forward

Theorem *ProposedAsMonotonic*
Introduce $\quad v \in ViewIds$

Introduce $c \in \mathit{Cohorts}$
Introduce $opn \in \mathit{Opns}$
Introduce $opv \in \mathit{CsOps}$
Assume $\mathit{ProposedAs}(v, c, opn, opv)$
Prove $\mathit{ProposedAs}(v, c, opn, opv)'$

Reasoning: *Ref:SentMessagesMonotonic* ; existential witness carries forward

Theorem *PreparedAsMonotonic*
Introduce $v \in \mathit{ViewIds}$
Introduce $c \in \mathit{Cohorts}$
Introduce $opn \in \mathit{Opns}$
Introduce $opv \in \mathit{CsOps}$
Assume $\mathit{PreparedAs}(v, c, opn, opv)$
Prove $\mathit{PreparedAs}(v, c, opn, opv)'$

Reasoning: *Ref:SentMessagesMonotonic* ; existential witness carries forward

Theorem *VcAckedMonotonic*
Introduce $v \in \mathit{ViewIds}$
Introduce $c \in \mathit{Cohorts}$
Introduce $\mathit{preparedOps} \in \mathit{PreparedOpsType}$
Assume $\mathit{VcAcked}(v, c, \mathit{preparedOps})$
Prove $\mathit{VcAcked}(v, c, \mathit{preparedOps})'$

Reasoning: *Ref:SentMessagesMonotonic* ; existential witness carries forward

Theorem *VcAckedViewMonotonic*
Introduce $\mathit{view} \in \mathit{ViewIds}$
Introduce $\mathit{cohort} \in \mathit{Cohorts}$
Assume $\mathit{VcAckedView}(\mathit{view}, \mathit{cohort})$
Prove $\mathit{VcAckedView}(\mathit{view}, \mathit{cohort})'$

Reasoning: *Ref:VcAckedMonotonic* ; existential witness carries forward

Theorem *DesignationReflectsVcAcksMonotonic*
Introduce $\mathit{view} \in \mathit{ViewIds}$
Introduce $\mathit{cohortSet} \in \textsc{subset}\ \mathit{Cohorts}$

Assume $\mathit{DesignationReflectsVcAcks}(\mathit{view}, \mathit{cohortSet})$

Prove $\mathit{DesignationReflectsVcAcks}(\mathit{view}, \mathit{cohortSet})'$

Reasoning: *Ref:PrimaryDesignatedMonotonic* , *Ref:VcAckedMonotonic* ; existential witnesses carry forward

Theorem *MembershipDefinedMonotonic*

Introduce $\mathit{opn} \in \mathit{Opns}$

Assume $\mathit{MembershipDefined}(\mathit{opn})$

Prove $\mathit{MembershipDefined}(\mathit{opn})'$

Reasoning: *Ref:SentMessagesMonotonic* ; existential witnesses to *MembershipAs* and *MembershipDefined* carry forward

Invariant *ProposedImpliesActiveMember*

Introduce $\mathit{view} \in \mathit{ViewIds}$

Introduce $\mathit{cohort} \in \mathit{Cohorts}$

Introduce $\mathit{opn} \in \mathit{Opns}$

Assume

$\land\ \mathit{Proposed}(\mathit{view}, \mathit{cohort}, \mathit{opn})$
$\land\ \mathit{LL}!\mathit{Replica}(\mathit{cohort})!\mathit{CurView} = \mathit{view}$
$\Rightarrow$
$\land\ \mathit{LL}!\mathit{Replica}(\mathit{cohort})!\mathit{ActiveMember}$
$\land\ \mathit{opn} \in \textsc{domain}\ \mathit{LL}!\mathit{Replica}(\mathit{cohort})!\mathit{CsState.membershipMap}$

Assume

$(\land\ \mathit{Proposed}(\mathit{view}, \mathit{cohort}, \mathit{opn})$
$\ \land\ \mathit{LL}!\mathit{Replica}(\mathit{cohort})!\mathit{CurView} = \mathit{view})'$

Prove

$(\land\ \mathit{LL}!\mathit{Replica}(\mathit{cohort})!\mathit{ActiveMember}$
$\ \land\ \mathit{opn} \in \textsc{domain}\ \mathit{LL}!\mathit{Replica}(\mathit{cohort})!\mathit{CsState.membershipMap})'$

Reasoning:

Theorem *ProposedImpliesMembershipAs*

Hypotheses of $\mathit{ProposedImpliesMembershipDefined}$

Introduce $\mathit{view} \in \mathit{ViewIds}$

Introduce $\mathit{cohort} \in \mathit{Cohorts}$

Introduce $\mathit{opn} \in \mathit{Opns}$

Assume $\mathit{Proposed}(\mathit{view}, \mathit{cohort}, \mathit{opn})$

Prove $\mathit{MembershipAs}(\mathit{opn}, \mathit{Membership}(\mathit{opn}), \mathit{LL}!\mathit{SentMessages})$

Step 1. of 1

$\exists\, membership \in Memberships : MembershipAs(opn,\ membership,\ LL!SentMessages)$

Reasoning (1.):*Ref*:*ProposedImpliesMembershipDefined* ; *Defn MembershipDefined*

Reasoning: *Defn Membership*; CHOOSE axiom

Theorem *CommittedMonotonic*

Introduce $c \in Cohorts$

Introduce $opn \in Opns$

Introduce $opv \in CsOps$

Assume $CommittedAs(c,\ opn,\ opv)$

Prove $CommittedAs(c,\ opn,\ opv)'$

Reasoning: *Ref*:*SentMessagesMonotonic* ; existential witness carries forward

Invariant *PreparedImpliesProposed*

Introduce $view \in ViewIds$

Introduce $c \in Cohorts$

Introduce $opn \in Opns$

Introduce $opv \in CsOps$

Assume $PreparedAs(view,\ c,\ opn,\ opv) \Rightarrow ProposedByAnyAs(view,\ opn,\ opv)$

Assume $PreparedAs(view,\ c,\ opn,\ opv)'$

Prove $ProposedByAnyAs(view,\ opn,\ opv)'$

Step 1. of 1

Prove $ProposedByAnyAs(view,\ opn,\ opv)$

Case 1.1. of 2

$LL!Replica(c)!PrepareAction(view,\ opn,\ opv)$

Reasoning (1.1.): $ReceiveMessage(m)$ provides witness for *ProposedByAnyAs*

Case 1.2. of 2

$\neg LL!Replica(c)!PrepareAction(view,\ opn,\ opv)$

Step 1.2.1. of 1

$PreparedAs(view,\ c,\ opn,\ opv)$

Reasoning (1.2.1.): No prepare sent on this step

Reasoning (1.2.): induction hypothesis

Reasoning (1.): Case analysis

Reasoning: *Ref*:*ProposedAsMonotonic*

Theorem *QuorumPreparedAsMonotonic*

Introduce $v \in ViewIds$

Introduce $opn \in Opns$

Introduce $opv \in CsOps$

Assume $QuorumPreparedAs(v, opn, opv)$

Prove $QuorumPreparedAs(v, opn, opv)'$

Reasoning: Apply *Ref:PreparedAsMonotonic* on each member of the quorum that witnesses to the assumption

Theorem *CurViewsMonotonic*

Introduce $cohort \in Cohorts$

Prove $LL!Replica(cohort)!CurView \leq (LL!Replica(cohort)!CurView')$

Reasoning: Case analysis on actions; only *VcAck* changes, and its enabling condition is sufficient to prove this theorem.

Invariant *CurViewLaterThanAllPrepareds*

Introduce $view \in ViewIds$

Introduce $cohort \in Cohorts$

Introduce $opn \in Opns$

Assume $Prepared(view, cohort, opn) \Rightarrow LL!Replica(cohort)!CurView \geq view$

Assume $Prepared(view, cohort, opn)'$

Prove $(LL!Replica(cohort)!CurView \geq view)'$

Summary: Only the *Prepare* action can cause trouble, but its preconditions provide the conclusion.

Case 1. of 2

$\exists\, m \in ProposedMsg :$

$\quad \wedge\, m.view = view$

$\quad \wedge\, LL!Replica(cohort)!Prepare(m)$

Defn $m \triangleq$

$\quad \text{CHOOSE } m \in ProposedMsg :$

$\qquad \wedge\, m.view = view$

$\qquad \wedge\, LL!Replica(cohort)!Prepare(m)$

Step 1.1. of 2

$m.view = LL!Replica(cohort)!CurView$

Reasoning (1.1.): *Defn Prepare* action

Step 1.2. of 2

$(LL!Replica(cohort)!CurView') = LL!Replica(cohort)!CurView$

Reasoning (1.2.): *Defn Prepare* action leaves *CurView* unchanged

Reasoning (1.): last two steps, case conjunct 1

Case 2. of 2

$\forall\, m \in ProposedMsg :$

$(\neg$

$(\wedge\ m.view = view$

$\wedge\ LL!Replica(cohort)!Prepare(m)))$

Step 2.1. of 3

$(LL!SentMessages') \cap PreparedMsg =$

$LL!SentMessages \cap PreparedMsg$

Reasoning (2.1.): Only *Prepare* action sends *PreparedMsg*

Step 2.2. of 3

$Prepared(view,\ cohort,\ opn)$

Reasoning (2.2.): Definition *Prepared* relies only on variable *LL!SentMessages*

Step 2.3. of 3

$LL!Replica(cohort)!CurView \geq view$

Reasoning (2.3.): induction hypothesis

Reasoning (2.): *Ref:CurViewsMonotonic*

Reasoning: Case analysis.

Invariant *CurViewLaterThanAllProposeds*

Introduce $view \in ViewIds$

Introduce $cohort \in Cohorts$

Introduce $opn \in Opns$

Introduce $opv \in CsOps$

Assume

$ProposedAs(view,\ cohort,\ opn,\ opv) \Rightarrow LL!Replica(cohort)!CurView \geq view$

Assume $ProposedAs(view,\ cohort,\ opn,\ opv)'$

Prove $(LL!Replica(cohort)!CurView \geq view)'$

Summary: Only the Propose and *BecomePrimary* actions can cause trouble, but their preconditions provide the conclusion.

Case 1. of 3

$LL!Replica(cohort)!ProposeAction(view,\ opn,\ opv)$

Step 1.1. of 1

$LL!Replica(cohort)!CurView = view$

Reasoning (1.1.): *Defn* Propose action

Reasoning (1.): algebra

Case 2. of 3

$\wedge\ (\exists\, m \in PrimaryDesignatedMsg : LL!Replica(cohort)!BecomePrimary(m))$

$\wedge\ (\neg ProposedAs(view,\ cohort,\ opn,\ opv))$

Step 2.1. of 1

$(LL!Replica(cohort)!CurView') = view$

Reasoning (2.1.): If *ProposedAs* became true on this action, it's because *BecomePrimary* added a new message to *SentMessages*; *Defn BecomePrimary* says that all new messages have $m.view = CurView$.

Reasoning (2.): algebra

DefaultCase 3. of 3

Step 3.1. of 2

$\textsc{unchanged}\ \mathit{ProposedAs}(\mathit{view},\ \mathit{cohort},\ \mathit{opn},\ \mathit{opv})$

Reasoning (3.1.): Inspection of remaining actions: none add an appropriate *ProposedMsg* to *SentMessages*.

Step 3.2. of 2

$\mathit{LL}!\mathit{Replica}(\mathit{cohort})!\mathit{CurView} \geq \mathit{view}$

Reasoning (3.2.): induction hypothesis

Reasoning (3.): *Ref:CurViewsMonotonic*

Reasoning: Case analysis.

Invariant *CurViewLaterThanAllVcAckeds*

Introduce $\mathit{view} \in \mathit{ViewIds}$

Introduce $\mathit{cohort} \in \mathit{Cohorts}$

Assume $\mathit{VcAckedView}(\mathit{view},\ \mathit{cohort}) \Rightarrow \mathit{LL}!\mathit{Replica}(\mathit{cohort})!\mathit{CurView} \geq \mathit{view}$

Assume $\mathit{VcAckedView}(\mathit{view},\ \mathit{cohort})'$

Prove $(\mathit{LL}!\mathit{Replica}(\mathit{cohort})!\mathit{CurView} \geq \mathit{view})'$

Summary: Only the *VcAck* action can cause trouble, but its assignment of *CurView* provides the conclusion.

Case 1. of 2

$\exists\, \mathit{preparedOps} \in \mathit{PreparedOpsType} :$
$\quad \mathit{LL}!\mathit{Replica}(\mathit{cohort})!\mathit{VcAckAction}(\mathit{view},\ \mathit{preparedOps})$

Step 1.1. of 1

$(\mathit{LL}!\mathit{Replica}(\mathit{cohort})!\mathit{CurView}') = \mathit{view}$

Reasoning (1.1.): *Defn VcAck* action

Reasoning (1.): algebra

DefaultCase 2. of 2

Step 2.1. of 2

$\textsc{unchanged}\ \mathit{VcAckedView}(\mathit{view},\ \mathit{cohort})$

Reasoning (2.1.): Inspection of remaining actions: none add an appropriate *ProposedMsg* to *SentMessages*.

Step 2.2. of 2

$\mathit{LL}!\mathit{Replica}(\mathit{cohort})!\mathit{CurView} \geq \mathit{view}$

Reasoning (2.2.): induction hypothesis

Reasoning (2.): *Ref:CurViewsMonotonic*

Reasoning: Case analysis.

Invariant $\mathit{PrimaryDesignationSentByInitiator}$

Introduce $m \in \mathit{PrimaryDesignatedMsg} \cap \mathit{SentMessages}$

Assume $m.\mathit{sender} = m.\mathit{view}.\mathit{viewInitiator}$

Prove $(m.\mathit{sender} = m.\mathit{view}.\mathit{viewInitiator})'$

Case 1. of 2

$\exists\, \mathit{config} \in \mathit{DesignationConfigurations} :$

$\land\, \mathit{LL}!\mathit{Replica}(m.\mathit{sender})!\mathit{DesignatePrimary}(\mathit{config})$

$\land\, m \notin \mathit{SentMessages}$

Step 1.1. of 2

$m.\mathit{view} = \mathit{LL}!\mathit{Replica}(m.\mathit{sender})!\mathit{CurView}$

Reasoning (1.1.): *Defn DesignatePrimary*;*MakePrimaryDesignatedMsg*

Step 1.2. of 2

$\mathit{LL}!\mathit{Replica}(m.\mathit{sender})!\mathit{CurView}.\mathit{viewInitiator} = m.\mathit{sender}$

Reasoning (1.2.): *Defn DesignatePrimary*

Reasoning (1.): substitution

DefaultCase 2. of 2

Reasoning (2.): No other action could send m

Reasoning: Proof by case analysis

Invariant $\mathit{CurViewOfInitiatorLaterThanAllPrimaryDesignateds}$

Introduce $\mathit{view} \in \mathit{ViewIds}$

Assume

$\mathit{PrimaryDesignated}(\mathit{view}) \Rightarrow \mathit{LL}!\mathit{Replica}(\mathit{view}.\mathit{viewInitiator})!\mathit{CurView} \geq \mathit{view}$

Assume $\mathit{PrimaryDesignated}(\mathit{view})'$

Prove $(\mathit{LL}!\mathit{Replica}(\mathit{view}.\mathit{viewInitiator})!\mathit{CurView} \geq \mathit{view})'$

Summary: Only the *DesignatePrimary* action can cause trouble, but its preconditions provide the conclusion.

Case 1. of 2

$\exists\, \mathit{config} \in \mathit{DesignationConfigurations} :$

$\land\, \mathit{LL}!\mathit{Replica}(\mathit{config}.\mathit{designator})!\mathit{DesignatePrimary}(\mathit{config})$

$\land\, \mathit{config}.\mathit{view} = \mathit{view}$

Defn $\mathit{config} \triangleq$

$\textsc{choose}\ \mathit{config} \in \mathit{DesignationConfigurations} :$

$\land\, \mathit{LL}!\mathit{Replica}(\mathit{config}.\mathit{designator})!\mathit{DesignatePrimary}(\mathit{config})$

$\land\, \mathit{config}.\mathit{view} = \mathit{view}$

Step 1.1. of 1

$\mathit{config}.\mathit{designator} = \mathit{view}.\mathit{viewInitiator}$

Reasoning (1.1.): *Defn DesignatePrimary* action

Reasoning (1.): *Defn DesignatePrimary* action

DefaultCase 2. of 2

Reasoning (2.): No other actions send *PrimaryDesignatedMsg*; apply induction hypothesis ; apply *Ref* :*CurViewsMonotonic* .

Reasoning: Case analysis.

Invariant $\mathit{ProposedImpliesPrimary}$

Hypotheses of $\mathit{CurViewLaterThanAllProposeds}$

Introduce $\mathit{view} \in \mathit{ViewIds}$

Introduce $\mathit{cohort} \in \mathit{Cohorts}$

Introduce $\mathit{opn} \in \mathit{Opns}$

Introduce $\mathit{opv} \in \mathit{CsOps}$

Assume

$\land\ \mathit{LL}!\mathit{Replica}(\mathit{cohort})!\mathit{CurView} = \mathit{view}$

$\land\ \mathit{ProposedAs}(\mathit{view}, \mathit{cohort}, \mathit{opn}, \mathit{opv})$

$\Rightarrow$

$\lor\ \mathit{LL}!\mathit{Replica}(\mathit{cohort})!\mathit{IAmPrimary}$

$\lor\ \mathit{LL}!\mathit{Replica}(\mathit{cohort})!\mathit{StaleView}$

Assume

$(\land\ \mathit{LL}!\mathit{Replica}(\mathit{cohort})!\mathit{CurView} = \mathit{view}$

$\land\ \mathit{ProposedAs}(\mathit{view}, \mathit{cohort}, \mathit{opn}, \mathit{opv}))'$

Prove

$(\lor\ \mathit{LL}!\mathit{Replica}(\mathit{cohort})!\mathit{IAmPrimary}$

$\lor\ \mathit{LL}!\mathit{Replica}(\mathit{cohort})!\mathit{StaleView})'$

Case 1. of 5

$\exists\, m \in \mathit{PrimaryDesignatedMsg} : \mathit{LL}!\mathit{Replica}(\mathit{cohort})!\mathit{BecomePrimary}(m)$

Step 1.1. of 1

$\mathit{LL}!\mathit{Replica}(\mathit{cohort})!\mathit{IAmPrimary}'$

Reasoning (1.1.): Definition *BecomePrimary*

Reasoning (1.): algebra

Case 2. of 5

$\mathit{LL}!\mathit{Replica}(\mathit{cohort})!\mathit{ProposeAction}(\mathit{view}, \mathit{opn}, \mathit{opv})$

Reasoning (2.): Definition Propose

Case 3. of 5

$\exists\, m \in \mathit{VcInittedMsg} : \mathit{LL}!\mathit{Replica}(\mathit{cohort})!\mathit{VcAck}(m)$

Step 3.1. of 3

$\mathit{LL}!\mathit{Replica}(\mathit{cohort})!\mathit{CurView} < (\mathit{LL}!\mathit{Replica}(\mathit{cohort})!\mathit{CurView}')$

Reasoning (3.1.): Definition *VcAck*

Step 3.2. of 3

$(\mathit{LL}!\mathit{Replica}(\mathit{cohort})!\mathit{CurView}') = \mathit{view}$

Reasoning (3.2.): Antecedent

Step 3.3. of 3

$\mathit{view} \leq \mathit{LL}!\mathit{Replica}(\mathit{cohort})!\mathit{CurView}$

Reasoning (3.3.): *Ref* hypothesis:*CurViewLaterThanAllProposeds*

Reasoning (3.): Case eliminated by *contradiction*(*algebra*)

Case 4. of 5

$LL!Replica(cohort)!Crash$

Step 4.1. of 1

$LL!Replica(cohort)!StaleView'$

Reasoning (4.1.): Definition *Crash*

Reasoning (4.): algebra

DefaultCase 5. of 5

Step 5.1. of 1

$\land$ UNCHANGED $LL!Replica(cohort)!IAmPrimary$
$\land$ UNCHANGED $LL!Replica(cohort)!StaleView$

Reasoning (5.1.): inspection of remaining actions

Reasoning (5.): induction hypothesis

Reasoning: Case analysis.

Invariant *LastProposedTracksProposals*

Hypotheses of $ProposedImpliesPrimary$

Introduce $view \in ViewIds$

Introduce $cohort \in Cohorts$

Introduce $opn \in Opns$

Introduce $opv \in CsOps$

Assume

$\land LL!Replica(cohort)!CurView = view$
$\land LL!Replica(cohort)!IAmPrimary$
$\land ProposedAs(view, cohort, opn, opv)$
$\Rightarrow$
$opn \leq LL!Replica(cohort)!LastProposed$

Assume

$(\land LL!Replica(cohort)!CurView = view$
$\land LL!Replica(cohort)!IAmPrimary$
$\land ProposedAs(view, cohort, opn, opv))'$

Prove $(opn \leq LL!Replica(cohort)!LastProposed)'$

Case 1. of 5

$\exists\, m \in PrimaryDesignatedMsg :$
$\land LL!Replica(cohort)!BecomePrimary(m)$
$\land m.view = view$
$\land m.opn = opn$
$\land m.opv = opv$

Step 1.1. of 2

$\neg LL!Replica(cohort)!StaleView$

Reasoning (1.1.): Definition *BecomePrimary*

Step 1.2. of 2

$\neg ProposedAs(view, cohort, opn, opv)$

Reasoning (1.2.): *Ref* hypothesis:*ProposedImpliesPrimary*

Reasoning (1.): Case eliminated by contradiction

Case 2. of 5

$LL!Replica(cohort)!ProposeAction(view,\ opn,\ opv)$

Step 2.1. of 1

$(LL!Replica(cohort)!LastProposed') = opn$

Reasoning (2.1.): Definition Propose

Reasoning (2.): algebra

Case 3. of 5

$\exists\, m \in VcInittedMsg : LL!Replica(cohort)!VcAck(m)$

Reasoning (3.): Eliminate case by contradiction: Definition *VcAck* shows $\neg$ $LL!Replica(cohort)!IAmPrimary$

Case 4. of 5

$LL!Replica(cohort)!Crash$

Reasoning (4.): Eliminate case by contradiction: Definition *Crash* shows $\neg$ $LL!Replica(cohort)!IAmPrimary$

DefaultCase 5. of 5

Step 5.1. of 1

$\land$ UNCHANGED $LL!Replica(cohort)!CurView$
$\land$ UNCHANGED $LL!Replica(cohort)!IAmPrimary$
$\land$ UNCHANGED $LL!Replica(cohort)!LastProposed$
$\land$ UNCHANGED $ProposedAs(view,\ cohort,\ opn,\ opv)$

Reasoning (5.1.): inspection of remaining actions

Reasoning (5.): induction hypothesis

Reasoning: Proof by case analysis

Invariant *PrimaryDesignatedPrecludesDesignationNeeded*

Introduce $view \in ViewIds$

Assume

$\land\ PrimaryDesignated(view)$
$\land\ LL!Replica(view.viewInitiator)!CurView = view$
$\Rightarrow$
$(\neg LL!Replica(view.viewInitiator)!DesignationNeeded)$

Assume

$(\land\ PrimaryDesignated(view)$
$\ \ \land\ LL!Replica(view.viewInitiator)!CurView = view)'$

Prove $(\neg LL!Replica(view.viewInitiator)!DesignationNeeded)'$

Reasoning: Basic action analysis; probably some monotonicity; induction hypothesis .

Invariant *OneDesignationPerView*

Introduce $\mathit{cohort} \in \mathit{Cohorts}$

Assume

$\mathit{PrimaryDesignated}(\mathit{LL}!\mathit{Replica}(\mathit{cohort})!\mathit{CurView}) \Rightarrow$
$(\neg \mathit{LL}!\mathit{Replica}(\mathit{cohort})!\mathit{DesignationNeeded})$

Assume $\mathit{PrimaryDesignated}(\mathit{LL}!\mathit{Replica}(\mathit{cohort})!\mathit{CurView})'$

Prove $(\neg \mathit{LL}!\mathit{Replica}(\mathit{cohort})!\mathit{DesignationNeeded})'$

Case 1. of 3

$\exists\, \mathit{config} \in \mathit{DesignationConfigurations}:$
$\quad \wedge\ \mathit{LL}!\mathit{Replica}(\mathit{config.designator})!\mathit{DesignatePrimary}(\mathit{config})$
$\quad \wedge\ \mathit{config.designator} = \mathit{cohort}$
$\quad \wedge\ \mathit{config.view} = \mathit{LL}!\mathit{Replica}(\mathit{cohort})!\mathit{CurView}$

Reasoning (1.): *Defn DesignatePrimary* action

Case 2. of 3

$\exists\, m \in \mathit{VcInittedMsg} : \mathit{LL}!\mathit{Replica}(\mathit{cohort})!\mathit{VcAck}(m)$

Defn $m \triangleq \text{CHOOSE } m \in \mathit{VcInittedMsg} : \mathit{LL}!\mathit{Replica}(\mathit{cohort})!\mathit{VcAck}(m)$

Case 2.1. of 2

$\mathit{cohort} = m.\mathit{view}.\mathit{viewInitiator}$

Step 2.1.1. of 4

$\mathit{LL}!\mathit{Replica}(\mathit{cohort})!\mathit{CurView}'.\mathit{viewInitator} = \mathit{cohort}$

Step 2.1.1.1. of 1

$(\mathit{LL}!\mathit{Replica}(\mathit{cohort})!\mathit{CurView}') = m.\mathit{view}$

Reasoning (2.1.1.1.): *Defn VcAck*

Reasoning (2.1.1.): substitution with Case assumption

Step 2.1.2. of 4

$\mathit{PrimaryDesignated}(\mathit{LL}!\mathit{Replica}(\mathit{cohort})!\mathit{CurView}')$

Reasoning (2.1.2.): *VcAck* doesn't send a *PrimaryDesignatedMsg*

Step 2.1.3. of 4

$(\mathit{LL}!\mathit{Replica}(\mathit{cohort})!\mathit{CurView}') \leq \mathit{LL}!\mathit{Replica}(\mathit{cohort})!\mathit{CurView}$

Step 2.1.3.1. of 1

$(\mathit{LL}!\mathit{Replica}(\mathit{cohort})!\mathit{CurView}') \leq$
$\mathit{LL}!\mathit{Replica}(\mathit{LL}!\mathit{Replica}(\mathit{cohort})!\mathit{CurView}'.\mathit{viewInitiator})!\mathit{CurView}$

Reasoning (2.1.3.1.):*Ref*:*CurViewOfInitiatorLaterThanAllPrimaryDesignateds*

Reasoning (2.1.3.): substitution with *Ref*:Step 2.1.1.

Step 2.1.4. of 4

$(\mathit{LL}!\mathit{Replica}(\mathit{cohort})!\mathit{CurView}') = \mathit{LL}!\mathit{Replica}(\mathit{cohort})!\mathit{CurView}$

Reasoning (2.1.4.): Forced by *Ref*:*CurViewsMonotonic*

Reasoning (2.1.): Contradicts defn *VcAck* action,eliminating the case.

DefaultCase 2.2. of 2

Reasoning (2.2.): *Defn VcAck* sets $\mathit{DesignationNeeded}' = \text{FALSE}$, satisfying the goal.

Reasoning (2.): Proof by case analysis

DefaultCase 3. of 3

Reasoning (3.): No other action could have sent a message that would make *PrimaryDesignated* true; hence it was true before. Apply induction hypothesis . No action besides *VcAck* can set *DesignationNeeded* true, so $\mathit{DesignationNeeded}' = \text{FALSE}$.

Reasoning: Proof by case analysis

Invariant *UniquePrimaryDesignationMessage*

Hypotheses of $OneDesignationPerView$

Introduce $m1 \in PrimaryDesignatedMsg$

Introduce $m2 \in PrimaryDesignatedMsg$

Assume

$\land\ m1 \in SentMessages$

$\land\ m2 \in SentMessages$

$\land\ m1.view = m2.view$

$\Rightarrow$

$m1 = m2$

Assume

$(\land\ m1 \in SentMessages$

$\ \land\ m2 \in SentMessages$

$\ \land\ m1.view\ = m2.view)'$

Prove $(m1 = m2)'$

Step 1. of 1

Assume (1.A1.) $m1 \in SentMessages \Rightarrow m2 \in SentMessages$

Prove $m1 = m2$

Case 1.1. of 2

$m1 \notin SentMessages$

Defn $config \triangleq$

$\textsc{choose}\ config \in DesignationConfigurations :$

$\land\ LL!Replica(m1.sender)!DesignatePrimary(config)$

$\land\ config.view = m1.view$

$\land\ config.newPrimary = m1.newPrimary$

$\land\ MaxTruncationPoint(config.msgs) = m1.maxTruncationPoint$

$\land\ AggregatePreparedOps(config.msgs) = m1.preparedOps$

Step 1.1.1. of 4

$\land\ LL!Replica(m1.sender)!DesignatePrimary(config)$

$\land\ config.view = m1.view$

$\land\ config.newPrimary = m1.newPrimary$

$\land\ MaxTruncationPoint(config.msgs) = m1.maxTruncationPoint$

$\land\ AggregatePreparedOps(config.msgs) = m1.preparedOps$

Reasoning (1.1.1.): $m1$ was sent in this step; this configuration must have done it.

Step 1.1.2. of 4

$LL!Replica(m1.sender)!DesignationNeeded$

Reasoning (1.1.2.): *Defn DesignatePrimary* action

Step 1.1.3. of 4

$\neg PrimaryDesignated(LL!Replica(m1.sender)!CurView)$

Reasoning (1.1.3.): Contrapositive of *Ref* hypothesis:*OneDesignationPerView*

Step 1.1.4. of 4

$m2 \notin \mathit{SentMessages}$

Reasoning (1.1.4.): *Defn PrimaryDesignated*

Reasoning (1.1.): Message $m2$ was sent this step, and this action sent only one message ($m1$). So they must be the same message.

DefaultCase 1.2. of 2

Step 1.2.1. of 2

$m1 \in \mathit{SentMessages}$

Reasoning (1.2.1.): No other action could send $m1$

Step 1.2.2. of 2

$m2 \in \mathit{SentMessages}$

Reasoning (1.2.2.): Assumption *Ref*:Assumption 1.A1.

Reasoning (1.2.): induction hypothesis ;*Ref*:*PrimaryDesignatedMonotonic*

Reasoning (1.): Proof by case analysis

Reasoning: without loss of generality, we can apply the substep with $m1$ and $m2$ swapped.

Theorem *UniquePrimaryDesignated*

Hypotheses of $\mathit{UniquePrimaryDesignationMessage}$

Introduce $\mathit{view} \in \mathit{ViewIds}$

Introduce $\mathit{cohort}1 \in \mathit{Cohorts}$

Introduce $\mathit{cohort}2 \in \mathit{Cohorts}$

Assume $\mathit{PrimaryDesignatedAs}(\mathit{view}, \mathit{cohort}1)$

Assume $\mathit{PrimaryDesignatedAs}(\mathit{view}, \mathit{cohort}2)$

Prove $\mathit{cohort}1 = \mathit{cohort}2$

Summary: Easily falls out of *Ref* hypothesis:*UniquePrimaryDesignationMessage* .

Defn $m1 \triangleq$

$$\begin{array}{l} \text{CHOOSE } m \in \mathit{SentMessages} \cap \mathit{PrimaryDesignatedMsg} : \\ \quad \land\ m.\mathit{view} = \mathit{view} \\ \quad \land\ m.\mathit{newPrimary} = \mathit{cohort}1 \end{array}$$

Defn $m2 \triangleq$

$$\begin{array}{l} \text{CHOOSE } m \in \mathit{SentMessages} \cap \mathit{PrimaryDesignatedMsg} : \\ \quad \land\ m.\mathit{view} = \mathit{view} \\ \quad \land\ m.\mathit{newPrimary} = \mathit{cohort}2 \end{array}$$

Step 1. of 1

$m1 = m2$

Reasoning (1.): Assumptions guarantee *CHOOSEs* succeed; *Ref* hypothesis:*UniquePrimaryDesignationMessage*

Reasoning: $\mathit{cohort}1 = m1.\mathit{newPrimary} = m2.\mathit{newPrimary} = \mathit{cohort}2$

Invariant *PreparedOpsPreparedImpliesPrepared*

Introduce $v2 \in ViewIds$

Introduce $cohort \in Cohorts$

Introduce $opn \in Opns$

Assume

$opn \in \textsc{domain}\ LL!Replica(cohort)!PreparedOps \Rightarrow$

$Prepared(LL!Replica(cohort)!PreparedOps[opn].view,\ cohort,\ opn)$

Assume $(opn \in \textsc{domain}\ LL!Replica(cohort)!PreparedOps)'$

Prove $Prepared(LL!Replica(cohort)!PreparedOps[opn].view,\ cohort,\ opn)'$

Case 1. of 3

$\exists\, m \in ProposedMsg : LL!Replica(cohort)!Prepare(m)$

Defn $m \triangleq \textsc{choose}\ m \in ProposedMsg : LL!Replica(cohort)!Prepare(m)$

Case 1.1. of 2

$m.opn = opn$

Step 1.1.1. of 2

$Prepared(LL!Replica(cohort)!CurView,\ cohort,\ opn)'$

Reasoning (1.1.1.): *Defn Prepare* action arguments to *MakePreparedMsg*

Step 1.1.2. of 2

$(LL!Replica(cohort)!PreparedOps[opn].view') = LL!Replica(cohort)!CurView$

Reasoning (1.1.2.): *Defn Prepare* action construction of $PreparedOps'$

Reasoning (1.1.): substitution satisfies the proof goal

DefaultCase 1.2. of 2

Step 1.2.1. of 1

$opn \in \textsc{domain}\ LL!Replica(cohort)!PreparedOps$

Reasoning (1.2.1.): *Defn Prepare* defines DOMAIN $PreparedOps'$ with a union on old value

Reasoning (1.2.): Apply induction hypothesis

Reasoning (1.): Proof by case analysis

Case 2. of 3

$\exists\, m \in PrimaryDesignatedMsg :$

$\land\ m.view = v2$

$\land\ LL!Replica(cohort)!BecomePrimary(m)$

Defn $m \triangleq$

$\textsc{choose}\ m \in PrimaryDesignatedMsg :$

$\land\ m.view = v2$

$\land\ LL!Replica(cohort)!BecomePrimary(m)$

Step 2.1. of 1

$opn \in \textsc{domain}\ m.prevPrepares$

Reasoning (2.1.): *Defn BecomePrimary* sets $PreparedOps' = m.prevPrepares$

Reasoning (2.): *Defn BecomePrimary* action sends the required message (argument to *SendMessageSet*)

DefaultCase 3. of 3

Step 3.1. of 1

$opn \in \textsc{domain}\ LL!Replica(cohort)!PreparedOps$

Reasoning (3.1.): All other actions leave *PreparedOps* unchanged

Reasoning (3.): induction hypothesis ;*Ref* :*PreparedAsMonotonic*

Reasoning: Proof by case analysis

Invariant *VcAckPreparedImpliesPrepared*

Hypotheses of $PreparedOpsPreparedImpliesPrepared$

Introduce $v2 \in ViewIds$

Introduce $cohort \in Cohorts$

Introduce $opn \in Opns$

Introduce $preparedOpInfo \in PreparedOpInfo$

Assume

$VcAckPreparedOpAs(v2,\ cohort,\ opn,\ preparedOpInfo) \Rightarrow$
$Prepared(preparedOpInfo.view,\ cohort,\ opn)$

Assume $VcAckPreparedOpAs(v2,\ cohort,\ opn,\ preparedOpInfo)'$

Prove $Prepared(preparedOpInfo.view,\ cohort,\ opn)'$

Case 1. of 2

$\wedge\ (\exists\, m \in SentMessagesMatching(cohort,\ VcInittedMsg) :$
$\quad \wedge\ LL!Replica(cohort)!VcAck(m)$
$\quad \wedge\ m.view = v2)$
$\wedge\ (LL!Replica(cohort)!PreparedOps') = preparedOpInfo$

Step 1.1. of 4

$PreparedOpInfoFromPreparedOps(LL!Replica(cohort)!PreparedOps',\ opn) =$
$preparedOpInfo$

Reasoning (1.1.): *Defn VcAck* action

Step 1.2. of 4

$opn \in \textsc{domain}\ (LL!Replica(cohort)!PreparedOps')$

Step 1.2.1. of 1

$preparedOpInfo \neq PreparedOpZero$

Reasoning (1.2.1.): as defined when it was Introduced

Reasoning (1.2.): Definition *PreparedOpInfoFromPreparedOps*

Step 1.3. of 4

$Prepared(LL!Replica(cohort)!PreparedOps[opn].view,\ cohort,\ opn)'$

Reasoning (1.3.): *Ref* hypothesis:*PreparedOpsPreparedImpliesPrepared* supports *Ref* :*PreparedOpsPreparedImpliesPrepared*

Step 1.4. of 4

$(LL!Replica(cohort)!PreparedOps')[opn].view = preparedOpInfo.view$

Step 1.4.1. of 1

$(LL!Replica(cohort)!PreparedOps')[opn] = preparedOpInfo$

Reasoning (1.4.1.): Last conjunct of Case condition; *Defn VcAck*

Reasoning (1.4.): Substitution

Reasoning (1.): Substitution

DefaultCase 2. of 2

Step 2.1. of 2

$VcAckPreparedOpAs(v2, cohort, opn, preparedOpInfo)$

Reasoning (2.1.): No actions in this case send a message that could make the statement transition to true.

Step 2.2. of 2

$Prepared(preparedOpInfo.view, cohort, opn)$

Reasoning (2.2.): induction hypothesis

Reasoning (2.): *Ref:PreparedAsMonotonic*

Reasoning: Case analysis on actions

Invariant *IAmPrimaryImpliesPrimaryDesignated*

Introduce $view \in ViewIds$

Introduce $cohort \in Cohorts$

Assume

$\wedge LL!Replica(cohort)!CurView = view$

$\wedge LL!Replica(cohort)!IAmPrimary$

$\Rightarrow$

$PrimaryDesignatedAs(view, cohort)$

Assume

$(\wedge LL!Replica(cohort)!CurView = view$

$\wedge LL!Replica(cohort)!IAmPrimary)'$

Prove $PrimaryDesignatedAs(view, cohort)'$

Step 1. of 1

$PrimaryDesignatedAs(view, cohort)$

Case 1.1. of 2

$\exists m \in PrimaryDesignatedMsg :$

$\wedge m.view = view$

$\wedge LL!Replica(cohort)!BecomePrimary(m)$

Reasoning (1.1.): Message m is a witness to *PrimaryDesignatedAs*. *Defn BecomePrimary*; *Defn ReceiveMessage*; *Defn PrimaryDesignatedAs*

DefaultCase 1.2. of 2

Step 1.2.1. of 3

$LL!Replica(cohort)!IAmPrimary$

Reasoning (1.2.1.): No actions on cohort other than *BecomePrimary* make *IAmPrimary* transition to
TRUE

Step 1.2.2. of 3

Introduce $m \in VcInittedMsg$

Prove $\neg LL!Replica(cohort)!VcAck(m)$

Reasoning (1.2.2.): *VcAck* sets $IAmPrimary' = $ FALSE, which contradicts antecedent conjunct "*IAmPrimary*"

Step 1.2.3. of 3

$LL!Replica(cohort)!CurView = view$

Summary: No other actions on cohort change *CurView*

Step 1.2.3.1. of 1

UNCHANGED $LL!Replica(cohort)!CurView$

Reasoning (1.2.3.1.): No other actions on cohort change *CurView*

Reasoning (1.2.3.): Antecedent conjunct $CurView = view$

Reasoning (1.2.): induction hypothesis

Reasoning (1.): Case analysis.

Reasoning: *Ref*:*PrimaryDesignatedMonotonic*

Invariant *ProposedImpliesPrimaryDesignated*

Hypotheses of $IAmPrimaryImpliesPrimaryDesignated$

Introduce $view \in ViewIds$

Introduce $cohort \in Cohorts$

Introduce $opn \in Opns$

Introduce $opv \in CsOps$

Assume $ProposedAs(view, cohort, opn, opv) \Rightarrow PrimaryDesignatedAs(view, cohort)$

Assume $ProposedAs(view, cohort, opn, opv)'$

Prove $PrimaryDesignatedAs(view, cohort)'$

Step 1. of 1

$PrimaryDesignatedAs(view, cohort)$

Case 1.1. of 3

$LL!Replica(cohort)!ProposeAction(view, opn, opv)$

Step 1.1.1. of 2

$LL!Replica(cohort)!IAmPrimary$

Reasoning (1.1.1.): Definition of *ProposeAction*

Step 1.1.2. of 2

$LL!Replica(cohort)!CurView = view$

Reasoning (1.1.2.): Definition of *ProposeAction*

Reasoning (1.1.): *Ref* hypothesis:*IAmPrimaryImpliesPrimaryDesignated*

Defn $rec \triangleq$

CHOOSE $rec \in [cohort : Cohorts,\ m : PrimaryDesignatedMsg]$:

$LL!Replica(rec.cohort)!BecomePrimary(rec.m)$

Case 1.2. of 3

$LL!Replica(rec.cohort)!BecomePrimary(rec.m)$

Reasoning (1.2.): *rec.m* is the witness to $PrimaryDesignatedAs(view, cohort)$

DefaultCase 1.3. of 3

Step 1.3.1. of 1

$ProposedAs(view, cohort, opn, opv)$

Reasoning (1.3.1.): No other step emits a *ProposedMsg* for *opn*, which is needed for *ProposedAs* to transition from false to true.

Reasoning (1.3.): induction hypothesis

Reasoning (1.): Proof by case analysis

Reasoning: *Ref*:*PrimaryDesignatedMonotonic*

Invariant *ProposedsInSameViewDoNotConflict*

Hypotheses of $ProposedImpliesPrimaryDesignated$

Hypotheses of $LastProposedTracksProposals$

Hypotheses of $UniquePrimaryDesignationMessage$

Hypotheses of $ProposedImpliesActiveMember$

Hypotheses of $UniquePrimaryDesignated$

Introduce $view \in ViewIds$

Introduce $cohort1 \in Cohorts$

Introduce $cohort2 \in Cohorts$

Introduce $opn \in Opns$

Introduce $opv1 \in CsOps$

Introduce $opv2 \in CsOps$

Assume

$\land ProposedAs(view,\ cohort1,\ opn,\ opv1)$

$\land ProposedAs(view,\ cohort2,\ opn,\ opv2)$

$\Rightarrow$

$opv1 = opv2$

Assume

$(\land ProposedAs(view,\ cohort1,\ opn,\ opv1)$

$\land ProposedAs(view,\ cohort2,\ opn,\ opv2))'$

Prove $(opv1 = opv2)'$

Step 1. of 4

$cohort1 = cohort2$

Step 1.1. of 2

$PrimaryDesignatedAs(view,\ cohort1)'$

Reasoning (1.1.): Antecedent conjunct 1;*Ref*:*ProposedImpliesPrimaryDesignated*

Step 1.2. of 2

$PrimaryDesignatedAs(view,\ cohort2)'$

Reasoning (1.2.): Antecedent conjunct 2;*Ref*:*ProposedImpliesPrimaryDesignated*

Reasoning (1.): *Ref*:*UniquePrimaryDesignated*

Case 2. of 4

$\exists\, opv \in CsOps : LL!Replica(cohort1)!ProposeAction(view,\ opn,\ opv)$

Step 2.1. of 3

$LL!Replica(cohort1)!LastProposed = opn - 1$

Reasoning (2.1.): *Defn ProposeAction*

Step 2.2. of 3

$\land LL!Replica(cohort1)!CurView = view$

$\wedge\ LL!Replica(cohort1)!IAmPrimary$

Reasoning (2.2.): *Defn ProposeAction*

Step 2.3. of 3

$\neg Proposed(view,\ cohort1,\ opn)$

Reasoning (2.3.): *Ref* hypothesis:*LastProposedTracksProposals*

Reasoning (2.): No proposals for *opn* in previous state, and action only proposes a single *opv*, so the same proposal message must make both $ProposedAs'$ statements true; hence $opv1 = opv = opv2$.

Case 3. of 4

$\exists\ m \in PrimaryDesignatedMsg$:

$\wedge\ LL!Replica(cohort1)!BecomePrimary(m)$

$\wedge\ m.view = view$

$\wedge\ opn \in NotPrevPrepared(m)$

Summary: If *cohort*1 is just now becoming the primary, then it had proposed nothing (in this view) before this step. Therefore, whatever messages support the *ProposedAs*() assumptions must have been sent as a part of the *BecomePrimary* action.

Step 3.1. of 1

Introduce $opv \in CsOps$

Prove $\neg ProposedAs(view,\ cohort1,\ opn,\ opv)$

Step 3.1.1. of 2

$LL!Replica(cohort1)!CurView = view$

Reasoning (3.1.1.): *Defn BecomePrimary*

Step 3.1.2. of 2

$\neg LL!Replica(cohort1)!ActiveMember$

Reasoning (3.1.2.): *Defn BecomePrimary*

Reasoning (3.1.): Contrapositive of *Ref* hypothesis:*ProposedImpliesActiveMember*

Reasoning (3.): For each *opn*, either *BecomePrimary* sends no proposal for it, or it sends a *NoOp*, or it sends some *opv* from *m.prevPrepares*; but in any case, a single message. That message is the only one that can witness to the two assumptions, so they must have the equal values for *opv*.

DefaultCase 4. of 4

Reasoning (4.): No new proposals in this view for *opn* sent; apply induction hypothesis and Ref:*ProposedAsMonotonic*

Reasoning: Proof by case analysis

Theorem *PreparedsInSameViewDoNotConflict*

Hypotheses of *PreparedImpliesProposed*

Hypotheses of *ProposedsInSameViewDoNotConflict*

Introduce $v \in ViewIds$

Introduce $c \in Cohorts$

Introduce $opn \in Opns$

Introduce $opv1 \in CsOps$

Introduce $opv2 \in CsOps$

Assume $PreparedAs(v,\ c,\ opn,\ opv1)'$

Assume $PreparedAs(v,\ c,\ opn,\ opv2)'$

Prove $opv1 = opv2$

Step 1. of 2

$ProposedAs(v,\ c,\ opn,\ opv1)'$

Reasoning (1.): *Ref:PreparedImpliesProposed*

Step 2. of 2

$ProposedAs(v,\ c,\ opn,\ opv2)'$

Reasoning (2.): *Ref:PreparedImpliesProposed*

Reasoning: *Ref:ProposedsInSameViewDoNotConflict*

Invariant *PreparedOpsReflectViewRecentPrepare*

Hypotheses of *CurViewLaterThanAllPrepareds*

Hypotheses of *PreparedsInSameViewDoNotConflict*

Introduce $v1 \in ViewIds$

Introduce $c \in Cohorts$

Introduce $opn \in Opns$

Introduce $opv \in CsOps$

Assume

$$
\begin{array}{l}
\land\ PreparedAs(v1,\ c,\ opn,\ opv)\\
\land\ (\forall\ vi \in ViewIds :\\
\qquad \land\ v1 < vi\\
\qquad \land\ vi \leq LL!Replica(c)!CurView\\
\qquad \Rightarrow\\
\qquad (\neg Prepared(vi,\ c,\ opn)))\\
\Rightarrow\\
LL!Replica(c)!PreparedOps[opn] = [view \mapsto v1,\ opv \mapsto opv]
\end{array}
$$

Assume

$$
\begin{array}{l}
(\ \land\ PreparedAs(v1,\ c,\ opn,\ opv)\\
\ \ \land\ (\forall\ vi \in ViewIds :\\
\qquad \land\ v1 < vi\\
\qquad \land\ vi \leq LL!Replica(c)!CurView\\
\qquad \Rightarrow\\
\qquad (\neg Prepared(vi,\ c,\ opn))))'
\end{array}
$$

Prove $(LL!Replica(c)!PreparedOps[opn] = [view \mapsto v1,\ opv \mapsto opv])'$

Case 1. of 2

$$
\begin{array}{l}
\exists\ m \in ProposedMsg :\\
\qquad \land\ LL!Replica(c)!Prepare(m)\\
\qquad \land\ m.opn = opn
\end{array}
$$

Defn $m \triangleq$

$\textsc{choose}\ m \in ProposedMsg :$

$\wedge\ LL!Replica(c)!Prepare(m)$

$\wedge\ m.opn = opn$

Step 1.1. of 1

$LL!Replica(c)!PreparedOps[opn] = [view \mapsto v1,\ opv \mapsto opv]$

Case 1.1.1. of 3

$LL!Replica(c)!CurView < v1$

Reasoning (1.1.1.): *Ref* hypothesis:*CurViewLaterThanAllPrepareds* eliminates case by contradiction

Case 1.1.2. of 3

$v1 < LL!Replica(c)!CurView$

Reasoning (1.1.2.): Consider witness $vi = LL!Replica(c)!CurView$ where $Prepared(vi,\ c,\ opn)'$ (because *Prepare* sends that message): it shows the second antecedent conjunct to be false. Case eliminated by contradiction.

Case 1.1.3. of 3

$v1 = LL!Replica(c)!CurView$

Step 1.1.3.1. of 3

$PreparedAs(v1,\ c,\ opn,\ opv)'$

Reasoning (1.1.3.1.): *Ref*:*PreparedAsMonotonic*

Step 1.1.3.2. of 3

$PreparedAs(v1,\ c,\ opn,\ m.opv)'$

Reasoning (1.1.3.2.): *Defn ProposedMsg* sends a message that is witness to *PreparedAs*

Step 1.1.3.3. of 3

$m.opv = opv$

Reasoning (1.1.3.3.):*Ref*:*PreparedsInSameViewDoNotConflict*

Reasoning (1.1.3.): Conclusion follows from assignment to $LL!Replica(c)!PreparedOps[opn]$ in action defn

Reasoning (1.1.): Proof by case analysis

Reasoning (1.): Last step satisfies this obligation (it was down a level so it could use the Case pattern.)

DefaultCase 2. of 2

Step 2.1. of 3

$PreparedAs(v1,\ c,\ opn,\ opv)$

Reasoning (2.1.): No other action can send *PreparedMsg*, so UNCHANGED *PreparedAs*

Step 2.2. of 3

Introduce $vi \in ViewIds$

Assume $v1 < vi$

Assume (2.2.*A*1.)

$vi \leq (LL!Replica(c)!CurView') \Rightarrow (\neg(Prepared(vi,\ c,\ opn)'))$

Prove $vi \leq LL!Replica(c)!CurView \Rightarrow (\neg Prepared(vi,\ c,\ opn))$

Step 2.2.1. of 1

$vi \leq LL!Replica(c)!CurView \Rightarrow (\neg(Prepared(vi,\ c,\ opn)'))$

Reasoning (2.2.1.): *Ref*:Assumption 2.2.*A*1. ,*Ref*:*CurViewsMonotonic*

Reasoning (2.2.): *Ref*:*PreparedAsMonotonic*

Step 2.3. of 3

$LL!Replica(c)!PreparedOps[opn] = [view \mapsto v1,\ opv \mapsto opv]$

Reasoning (2.3.): induction hypothesis

Reasoning (2.): In this case (not a *Prepare* of *opn*), *PreparedOps*[*opn*] cannot change (Note: Truncate action could change *PreparedOps*, but current spec explicitly ignores log truncation.)

Reasoning: Proof by case analysis

Invariant *VcAckPreparedsReflectViewRecentPrepare*

Hypotheses of *PreparedOpsReflectViewRecentPrepare*

Hypotheses of *CurViewLaterThanAllVcAckeds*

Introduce $v1 \in ViewIds$

Introduce $v2 \in ViewIds$

Introduce $c \in Cohorts$

Introduce $opn \in Opns$

Introduce $opv \in CsOps$

Assume

$$
\begin{array}{l}
\land\ v1 < v2 \\
\land\ PreparedAs(v1,\ c,\ opn,\ opv) \\
\land\ (\forall\ vi \in ViewIds : \\
\qquad \land\ v1 < vi \\
\qquad \land\ vi\ < v2 \\
\qquad \Rightarrow \\
\qquad (\neg Prepared(vi,\ c,\ opn))) \\
\land\ VcAckedView(v2,\ c) \\
\Rightarrow \\
VcAckPreparedOpAs(v2,\ c,\ opn,\ [opv \mapsto opv,\ view \mapsto v1])
\end{array}
$$

Assume

$$
\begin{array}{l}
(\ \land\ v1 < v2 \\
\ \ \land\ PreparedAs(v1,\ c,\ opn,\ opv) \\
\ \ \land\ (\forall\ vi \in ViewIds : \\
\qquad \land\ v1 < vi \\
\qquad \land\ vi\ < v2 \\
\qquad \Rightarrow \\
\qquad (\neg Prepared(vi,\ c,\ opn))) \\
\ \ \land\ VcAckedView(v2,\ c))'
\end{array}
$$

Prove $VcAckPreparedOpAs(v2,\ c,\ opn,\ [opv \mapsto opv,\ view \mapsto v1])'$

Case 1. of 3

$\exists\ preparedOps \in PreparedOpsType : LL!Replica(c)!VcAckAction(v2,\ preparedOps)$

Step 1.1. of 4

$(LL!Replica(c)!CurView') = v2$

Reasoning (1.1.): *Defn VcAck* action

Step 1.2. of 4

$\land (\forall\, vi \in \mathit{ViewIds} :$

$\quad \land\, v1 < vi$

$\quad \land\, (vi < (\mathit{LL}!\mathit{Replica}(c)!\mathit{CurView}') \Rightarrow (\neg(\mathit{Prepared}(vi,\ c,\ opn)'))))$

Reasoning (1.2.): algebra applied to antecedent third conjunct

Step 1.3. of 4

$(\mathit{LL}!\mathit{Replica}(c)!\mathit{PreparedOps}')[opn] = [opv \mapsto opv,\ view \mapsto v1]$

Reasoning (1.3.):*Ref* :*PreparedOpsReflectViewRecentPrepare*

Step 1.4. of 4

$\mathit{VcAcked}(v2,\ c,\ (\mathit{LL}!\mathit{Replica}(c)!\mathit{PreparedOps}')[opn])'$

Reasoning (1.4.): *VcAck* action puts a message into *SentMessages* that serves as a witness to *VcAcked*().

Reasoning (1.): Definition of *VcAckPreparedOpAs*

Case 2. of 3

$\exists\, m \in \mathit{ProposedMsg} :$

$\quad \land\, m.view = v1$

$\quad \land\, m.opn = opn$

$\quad \land\, \mathit{LL}!\mathit{Replica}(c)!\mathit{Prepare}(m)$

Step 2.1. of 2

$\neg\, \mathit{VcAckedView}(v2,\ c)$

Step 2.1.1. of 2

$\mathit{LL}!\mathit{Replica}(c)!\mathit{CurView} = v1$

Reasoning (2.1.1.): *Defn Prepare*

Step 2.1.2. of 2

$\mathit{LL}!\mathit{Replica}(c)!\mathit{CurView} < v2$

Reasoning (2.1.2.): algebra

Reasoning (2.1.): Contrapositive of *Ref* hypothesis:*CurViewLaterThanAllVcAckeds*

Step 2.2. of 2

$\neg(\mathit{VcAckedView}(v2,\ c)')$

Reasoning (2.2.): This action doesn't send a *VcAckedMsg*

Reasoning (2.): case eliminated by contradiction

DefaultCase 3. of 3

Step 3.1. of 6

$\textsc{unchanged}\ \mathit{SentMessagesMatching}(c,\ \mathit{VcAckedMsg})$

Reasoning (3.1.): No other action sends a *VcAckedMsg*

Step 3.2. of 6

$\mathit{VcAckedView}(v2,\ c)$

Reasoning (3.2.): *VcAckedView* only varies in *SentMessagesMatching*(*c*, *VcAckedMsg*)

Step 3.3. of 6

$\textsc{unchanged}\ \mathit{SentMessagesMatching}(c,\ \mathit{PreparedMsg})$

Reasoning (3.3.): No other action sends a *PreparedMsg*

Step 3.4. of 6

$\mathit{PreparedAs}(v1,\ c,\ opn,\ opv)$

Reasoning (3.4.): *PreparedAs* only varies in *SentMessagesMatching*$(c,\ PreparedMsg)$

Step 3.5. of 6

$\land (\forall\, vi \in ViewIds :$

$\quad \land\, v1 < vi$

$\quad \land\, (vi < v2 \Rightarrow (\neg Prepared(vi,\ c,\ opn))))$

Reasoning (3.5.): Contrapositive of *Ref*:*PreparedAsMonotonic*

Step 3.6. of 6

$VcAckPreparedOpAs(v2,\ c,\ opn,\ [opv \mapsto opv,\ view \mapsto v1])$

Reasoning (3.6.): induction hypothesis

Reasoning (3.): *Ref*:*VcAckedMonotonic*

Reasoning: Proof by case analysis

Theorem *PlausibleElectionQuorumMonotonic*

Introduce $view \in ViewIds$

Introduce $quorum \in \textsc{subset}\ Cohorts$

Assume $PlausibleElectionQuorum(view,\ quorum)$

Prove $PlausibleElectionQuorum(view,\ quorum)'$

Step 1. of 2

$\forall\, cohort \in quorum : (\, VcAckedView(view,\ cohort)')$

Reasoning (1.): *Ref*:*VcAckedViewMonotonic*

Step 2. of 2

$DesignationReflectsVcAcks(view,\ quorum)'$

Reasoning (2.): *Ref*:*SentMessagesMonotonic* ; existential witnesses carry forward

Reasoning: Both conjuncts of *Defn* *PlausibleElectionQuorum*$'$ are satisfied

Invariant *MembershipMapDomain*

Introduce $cohort \in Cohorts$

Assume

$(\neg LL!Replica(cohort)!Crash) \Rightarrow$

$\textsc{domain}\ LL!Replica(cohort)!CsState.membershipMap =$

$(1\, ..\, (LL!Replica(cohort)!CsState.numExecuted + Alpha))$

Assume $(\neg LL!Replica(cohort)!Crash)'$

Prove

$(\textsc{domain}\ LL!Replica(cohort)!CsState.membershipMap =$

$(1\, ..\, (LL!Replica(cohort)!CsState.numExecuted + Alpha)))'$

Case 1. of 2

$\exists\, m \in CommittedMsg : LL!Replica(cohort)!Execute(m)$

Defn $m \triangleq \textsc{choose}\ m \in \mathit{CommittedMsg} : \mathit{LL}!\mathit{Replica}(\mathit{cohort})!\mathit{Execute}(m)$

Step 1.1. of 1

$(\mathit{LL}!\mathit{Replica}(\mathit{cohort})!\mathit{CsState}') = \mathit{CsTx}[\mathit{LL}!\mathit{Replica}(\mathit{cohort})!\mathit{CsState},\ m.\mathit{opv}]$

Reasoning (1.1.): *Defn Execute* action

Reasoning (1.): *Defn newMembershipMap* in *Defn CsTx*

DefaultCase 2. of 2

Reasoning (2.): Since we've ruled out *Crash* in the assumption, no other action updates *CsState*. Thus induction hypothesis carries forward into primed state.

Reasoning: Proof by case analysis

Invariant *MembershipMapChangesByExtension*

Hypotheses of *MembershipMapDomain*

Introduce $\mathit{cohort} \in \mathit{Cohorts}$

Assume

$(\neg \mathit{LL}!\mathit{Replica}(\mathit{cohort})!\mathit{Crash}) \Rightarrow$
$\mathit{FcnExtends}($
$\quad \mathit{LL}!\mathit{Replica}(\mathit{cohort})!\mathit{CsState}.\mathit{membershipMap}',$
$\quad \mathit{LL}!\mathit{Replica}(\mathit{cohort})!\mathit{CsState}.\mathit{membershipMap})$

Assume $(\neg \mathit{LL}!\mathit{Replica}(\mathit{cohort})!\mathit{Crash})'$

Prove

$\mathit{FcnExtends}($
$\quad \mathit{LL}!\mathit{Replica}(\mathit{cohort})!\mathit{CsState}.\mathit{membershipMap}',$
$\quad \mathit{LL}!\mathit{Replica}(\mathit{cohort})!\mathit{CsState}.\mathit{membershipMap})'$

Case 1. of 2

$\exists\, m \in \mathit{CommittedMsg} : \mathit{LL}!\mathit{Replica}(\mathit{cohort})!\mathit{Execute}(m)$

Defn $m \triangleq \textsc{choose}\ m \in \mathit{CommittedMsg} : \mathit{LL}!\mathit{Replica}(\mathit{cohort})!\mathit{Execute}(m)$

Step 1.1. of 3

$(\mathit{LL}!\mathit{Replica}(\mathit{cohort})!\mathit{CsState}') = \mathit{CsTx}[\mathit{LL}!\mathit{Replica}(\mathit{cohort})!\mathit{CsState},\ m.\mathit{opv}]$

Reasoning (1.1.): *Defn Execute* action

Step 1.2. of 3

$\textsc{domain}\ \mathit{LL}!\mathit{Replica}(\mathit{cohort})!\mathit{CsState}.\mathit{membershipMap} \subseteq$
$\textsc{domain}\ (\mathit{LL}!\mathit{Replica}(\mathit{cohort})!\mathit{CsState}.\mathit{membershipMap}')$

Step 1.2.1. of 2

$\textsc{domain}\ \mathit{LL}!\mathit{Replica}(\mathit{cohort})!\mathit{CsState}.\mathit{membershipMap} =$
$(1 \mathrel{..} (\mathit{LL}!\mathit{Replica}(\mathit{cohort})!\mathit{CsState}.\mathit{numExecuted} + \mathit{Alpha}))$

Reasoning (1.2.1.): *Ref* hypothesis:*MembershipMapDomain*

Step 1.2.2. of 2

$\textsc{domain}\ (\mathit{LL}!\mathit{Replica}(\mathit{cohort})!\mathit{CsState}.\mathit{membershipMap}') =$
$(1 \mathrel{..} ((\mathit{LL}!\mathit{Replica}(\mathit{cohort})!\mathit{CsState}.\mathit{numExecuted} + \mathit{Alpha}) + 1))$

Reasoning (1.2.2.): *Defn CsTx*

Reasoning (1.2.): *Defn ..*

Step 1.3. of 3

Introduce $x \in \textsc{domain}\ \mathit{LL}!\mathit{Replica}(\mathit{cohort})!\mathit{CsState}.\mathit{membershipMap}$

$(\mathit{LL}!\mathit{Replica}(\mathit{cohort})!\mathit{CsState}.\mathit{membershipMap}')[x] =$
$\mathit{LL}!\mathit{Replica}(\mathit{cohort})!\mathit{CsState}.\mathit{membershipMap}[x]$

Reasoning (1.3.): *Defn CsTx*

Reasoning (1.): We have satisfied *Defn FcnExtends*

DefaultCase 2. of 2

Reasoning (2.): Since we've ruled out *Crash* in the assumption, no other action updates *CsState*. Thus the reflexive *FcnExtends* is easily satisfied.

Reasoning: Proof by case analysis

Theorem *VolatileMembershipMapExtendsPersistentMembershipMap*

Hypotheses of $\mathit{MembershipMapChangesByExtension}$

Introduce $\mathit{cohort} \in \mathit{Cohorts}$

Assume

$$\begin{aligned}&\mathit{FcnExtends}(\\&\quad \mathit{LL}!\mathit{Replica}(\mathit{cohort})!\mathit{CsState}.\mathit{membershipMap},\\&\quad \mathit{LL}!\mathit{Replica}(\mathit{cohort})!\mathit{CsStateSnapshot}.\mathit{membershipMap})\end{aligned}$$

Prove

$$\begin{aligned}&\mathit{FcnExtends}(\\&\quad \mathit{LL}!\mathit{Replica}(\mathit{cohort})!\mathit{CsState}.\mathit{membershipMap},\\&\quad \mathit{LL}!\mathit{Replica}(\mathit{cohort})!\mathit{CsStateSnapshot}.\mathit{membershipMap})'\end{aligned}$$

Reasoning: Since we're not doing log truncation, this theorem is really boring: *CsStateSnapshot.membershipMap* never changes. When *CsState* does, *Ref* hypothesis:*MembershipMapChangesByExtension* is sufficient to show the theorem. If we had truncation, the *Persist* action is the only interesting case, and it's not very interesting: it makes both states equal, so *FcnExtends* follows because it is a reflexive relation.

Theorem *MembershipAsMonotonic*

Introduce $\mathit{opn} \in \mathit{Opns}$

Introduce $\mathit{membership} \in \mathit{Memberships}$

Assume $\mathit{MembershipAs}(\mathit{opn}, \mathit{membership}, \mathit{LL}!\mathit{SentMessages})$

Prove $\mathit{MembershipAs}(\mathit{opn}, \mathit{membership}, \mathit{LL}!\mathit{SentMessages}')$

Reasoning:

Invariant *MembershipChangesAreBroadcast*

Hypotheses of $\mathit{MembershipMapChangesByExtension}$

Introduce $\mathit{opn} \in \mathit{Opns}$

Introduce $cohort \in Cohorts$

Introduce $membership \in Memberships$

Assume

$\land\ opn \in \textsc{domain}\ LL!Replica(cohort)!CsState.membershipMap$

$\land\ membership = LL!Replica(cohort)!CsState.membershipMap[opn]$

$\Rightarrow$

$MembershipAs(opn,\ membership,\ LL!SentMessages)$

Assume

$(\land\ opn \in \textsc{domain}\ LL!Replica(cohort)!CsState.membershipMap$

$\land\ membership = LL!Replica(cohort)!CsState.membershipMap[opn])'$

Prove $MembershipAs(opn,\ membership,\ LL!SentMessages)'$

Defn $state \triangleq LL!Replica(cohort)!CsState$

Defn $snapshot \triangleq LL!Replica(cohort)!CsStateSnapshot$

Case 1. of 3

$\exists\, m \in CommittedMsg : LL!Replica(cohort)!Execute(m)$

Defn $m \triangleq \textsc{choose}\ m \in CommittedMsg : LL!Replica(cohort)!Execute(m)$

Case 1.1. of 2

$opn = m.opn + Alpha$

Step 1.1.1. of 2

$state'.membershipMap[(m.opn + Alpha)] = membership$

Step 1.1.1.1. of 1

$state'.membershipMap[opn] = membership$

Reasoning (1.1.1.1.): Antecedent

Reasoning (1.1.1.): substitution

Defn $sentMessage \triangleq MakeMembershipMsg(cohort,\ opn,\ membership)$

Step 1.1.2. of 2

$sentMessage \in (SentMessages')$

Reasoning (1.1.2.): *Defn Execute* sends a message

Reasoning (1.1.): *Defn MembershipAs*

DefaultCase 1.2. of 2

Step 1.2.1. of 3

$opn \in \textsc{domain}\ LL!Replica(cohort)!CsState.membershipMap$

Reasoning (1.2.1.): *Ref* hypothesis:*MembershipMapChangesByExtension* ; *Defn FcnExtends*

Step 1.2.2. of 3

$membership = state.membershipMap[opn]$

Reasoning (1.2.2.): $\textsc{if} - \textsc{else}$ in $CsTx$ leaves unchanged any $opn \neq m.opn + Alpha$

Step 1.2.3. of 3

$MembershipAs(opn,\ membership,\ LL!SentMessages)$

Reasoning (1.2.3.): induction hypothesis

Reasoning (1.2.): *Ref*:*MembershipAsMonotonic*

Reasoning (1.): Proof by case analysis

Case 2. of 3

$LL!Replica(cohort)!Crash$

Step 2.1. of 4

$\land\ opn \in \text{DOMAIN}\ snapshot'.membershipMap[opn]$

$\land\ cohort \in snapshot'.membershipMap[opn]$

Reasoning (2.1.): *Defn Crash* equates $CsState' = CsStateSnapshot'$

Step 2.2. of 4

$\land\ opn \in \text{DOMAIN}\ snapshot.membershipMap[opn]$

$\land\ cohort \in snapshot.membershipMap[opn]$

Reasoning (2.2.): *Defn Crash* leaves UNCHANGED *CsStateSnapshot*

Step 2.3. of 4

$\land\ opn \in \text{DOMAIN}\ state.membershipMap[opn]$

$\land\ cohort \in state.membershipMap[opn]$

Reasoning (2.3.): Ref:*VolatileMembershipMapExtendsPersistentMembershipMap* ; *Defn FcnExtends*

Step 2.4. of 4

$MembershipAs(opn,\ LL!Replica(cohort)!Membership,\ LL!SentMessages)$

Reasoning (2.4.): induction hypothesis

Reasoning (2.): *Ref:MembershipAsMonotonic*

DefaultCase 3. of 3

Reasoning (3.): No other actions update state (*CsState*) (this proof ignores the Transfer action); so we use the induction hypothesis and Ref:*MembershipAsMonotonic* .

Reasoning: Proof by case analysis

Theorem *MaxKnownOpnGrows*

$MaxKnownOpn \leq (MaxKnownOpn')$

Reasoning: *Ref:CommittedMonotonic* : Anything committed before will still be committed after any legal action.

Theorem *MembershipsAreUnique*

Hypotheses of $BroadcastMembershipsReflectKnownState$

Introduce $opn \in Opns$

Introduce $membership1 \in Memberships$

Introduce $membership2 \in Memberships$

Assume $MembershipAs(opn,\ membership1,\ LL!SentMessages)$

Assume $MembershipAs(opn,\ membership2,\ LL!SentMessages)$

Prove $membership1 = membership2$

Case 1. of 2

$Alpha < opn$

Step 1.1. of 2

$KnownState[(opn - Alpha)].membershipMap[opn] = membership1$

Reasoning (1.1.): *Ref* hypothesis:*BroadcastMembershipsReflectKnownState*

Step 1.2. of 2

$KnownState[(opn - Alpha)].membershipMap[opn] = membership2$

Reasoning (1.2.): *Ref* hypothesis:*BroadcastMembershipsReflectKnownState*

Reasoning (1.): Substitution

Case 2. of 2

$opn \leq Alpha$

Step 2.1. of 2

$MembershipAs(opn, membership1, LL!SentMessages) = MakeMembership(InitialHosts, 1)$

Reasoning (2.1.): *Defn MembershipAs*

Step 2.2. of 2

$MembershipAs(opn, membership2, LL!SentMessages) = MakeMembership(InitialHosts, 1)$

Reasoning (2.2.): *Defn MembershipAs*

Reasoning (2.): Substitution

Reasoning: Proof by case analysis

NB Unlike most, this theorem states properties about the primed state.

Theorem *MembershipAsDeterminesMembership*

Hypotheses of $MembershipsAreUnique$

Introduce $opn \in Opns$

Introduce $membership \in Memberships$

Assume $MembershipAs(opn, membership, LL!SentMessages')$

Prove $(Membership(opn)') = membership$

Defn $choices \triangleq \{m \in Memberships : MembershipAs(opn, m, LL!SentMessages')\}$

Step 1. of 2

$membership \in choices$

Reasoning (1.): *Defn* choices

Step 2. of 2

$Cardinality(choices) = 1$

Reasoning (2.): *Ref*:*MembershipsAreUnique* (to get primed statement)

Reasoning: CHOOSE in *Defn* $Membership(opn)'$ is fully constrained

Theorem *CsTxIncrementsEpochs*

Introduce $state \in CsStates$

Introduce $opv \in CsOps$

Prove

$\vee\ CsTx[state, opv].membershipMap[((state.numExecuted + Alpha) + 1)] =$
$state.membershipMap[(state.numExecuted + Alpha)]$

$\lor\ EpochOf(CsTx[state,\ opv].membershipMap[((state.numExecuted + Alpha) + 1)]) =$
$\quad EpochOf(state.membershipMap[(state.numExecuted + Alpha)]) + 1$

Reasoning: By construction of $CsTx$

Invariant $NumExecutedTicks$

Introduce $opn \in \text{DOMAIN } KnownState$

Assume $KnownState[opn].numExecuted = opn$

Prove $(KnownState[opn].numExecuted = opn)'$

Reasoning: Really boring induction induction hypothesis ; $CsTx$ shows the inductive step.

Invariant $LocalMembershipEpochOrdering$

Introduce $cohort \in Cohorts$

Assume

$\land\ EpochsOrdered(LL!Replica(cohort)!CsState.membershipMap)$
$\land\ EpochsOrdered(LL!Replica(cohort)!CsStateSnapshot.membershipMap)$

Prove

$(\land\ EpochsOrdered(LL!Replica(cohort)!CsState.membershipMap)$
$\ \land\ EpochsOrdered(LL!Replica(cohort)!CsStateSnapshot.membershipMap))'$

Case 1. of 3

$LL!Replica(cohort)!Crash$

Reasoning (1.): $CsState' = CsStateSnapshot' = CsStateSnapshot$; apply induction hypothesis .

Case 2. of 3

$\exists\, m \in CommittedMsg : LL!Replica(cohort)!Execute(m)$

Summary: Only $CsState$ changes, and it changes by extension by a single spot; we can apply $CsTx$ there to show that the invariant holds.

Defn $m \triangleq \text{CHOOSE } m \in CommittedMsg : LL!Replica(cohort)!Execute(m)$

Defn $map \triangleq LL!Replica(cohort)!CsState.membershipMap$

Step 2.1. of 2

Introduce $opn1 \in \text{DOMAIN } map$

Introduce $opn2 \in \text{DOMAIN } map$

$\land\ EpochOf(map[opn1]) \leq EpochOf(map[opn2])$
$\land\ (EpochOf(map[opn1]) = EpochOf(map[opn2]) \Rightarrow map[opn1] = map[opn2])$

Summary: If $opn2$ (and hence $opn1$) concern slots before the one being executed presently, then the induction hypothesis takes care of the proof. Otherwise, we use $CsTx$.

Case 2.1.1. of 2

$opn2 = m.opn$

Step 2.1.1.1. of 2

$\lor\ (map')[(opn2 - 1)] = (map')[opn2]$

$\lor \mathit{EpochOf}((\mathit{map}')[(\mathit{opn}2 - 1)]) < \mathit{EpochOf}((\mathit{map}')[\mathit{opn}2])$

Step 2.1.1.1.1. of 1

$(\mathit{LL}!\mathit{Replica}(\mathit{cohort})!\mathit{CsState}') = \mathit{CsTx}[\mathit{LL}!\mathit{Replica}(\mathit{cohort})!\mathit{CsState},\ m.\mathit{opv}]$

Reasoning (2.1.1.1.1.): *Defn Execute*

Reasoning (2.1.1.1.): Consider *Defn CsTx*, paying attention to the LET –IN variable *newMembership*

Step 2.1.1.2. of 2

$\lor (\mathit{map}')[(\mathit{opn}2 - 1)] = (\mathit{map}')[\mathit{opn}1]$
$\lor \mathit{EpochOf}((\mathit{map}')[\mathit{opn}1]) < \mathit{EpochOf}((\mathit{map}')[(\mathit{opn}2 - 1)])$

Reasoning (2.1.1.2.): Ref:*MembershipMapChangesByExtension* ; induction hypothesis

Reasoning (2.1.1.): algebra relates *opn*2 to *opn*1 via *opn*2 − 1

DefaultCase 2.1.2. of 2

Reasoning (2.1.2.): Apply Ref:*MembershipMapChangesByExtension* and induction hypothesis .

Reasoning (2.1.): Proof by case analysis.

Step 2.2. of 2

$\mathit{EpochsOrdered}(\mathit{LL}!\mathit{Replica}(\mathit{cohort})!\mathit{CsStateSnapshot}.\mathit{membershipMap})$

Reasoning (2.2.): *Defn Execute* implies UNCHANGED *CsStateSnapshot*; induction hypothesis

Reasoning (2.): First step proves *Defn EpochsOrdered* in first conjunct of proof goal; Second step proves second conjunct.

DefaultCase 3. of 3

Step 3.1. of 1

$\land$ UNCHANGED $\mathit{LL}!\mathit{Replica}(\mathit{cohort})!\mathit{CsState}$
$\land$ UNCHANGED $\mathit{LL}!\mathit{Replica}(\mathit{cohort})!\mathit{CsStateSnapshot}$

Reasoning (3.1.): No other actions change *CsState* and *CsStateSnapshot* (besides *Persist*, but this proof is ignoring persistence and log truncation, and anyway, *Persist* is easy like *Crash*.)

Reasoning (3.): apply induction hypothesis

Reasoning: Proof by case analysis.

Theorem *MembershipEpochOrdering*

Hypotheses of $\mathit{NumExecutedTicks}$
Introduce $\mathit{opn}1 \in \mathit{Opns}$
Introduce $\mathit{opn}2 \in \mathit{Opns}$
Assume $\mathit{Alpha} \leq \mathit{opn}1$
Assume $\mathit{opn}1 < \mathit{opn}2$
Assume $\mathit{opn}2 \leq \mathit{MaxKnownOpn} + \mathit{Alpha}$

Prove

$\land \mathit{EpochOf}(\mathit{KnownMembership}(\mathit{opn}1)) \leq \mathit{EpochOf}(\mathit{KnownMembership}(\mathit{opn}2))$
$\land (\mathit{EpochOf}(\mathit{KnownMembership}(\mathit{opn}1)) = \mathit{EpochOf}(\mathit{KnownMembership}(\mathit{opn}2)) \Rightarrow$

$KnownMembership(opn1) = KnownMembership(opn2))$

Step 1. of 2

Assume $\quad opn2 = opn1 + 1$

Prove

$\land\, EpochOf(KnownMembership(opn1)) \leq EpochOf(KnownMembership(opn2))$

$\land\, (EpochOf(KnownMembership(opn1)) = EpochOf(KnownMembership(opn2)) \Rightarrow$

$\quad KnownMembership(opn1) = KnownMembership(opn2))$

Reasoning (1.): Follows by algebra from *Ref:CsTxIncrementsEpochs*

Step 2. of 2

Assume (2.A1.)

$\land\, EpochOf(KnownMembership(opn1)) \leq EpochOf(KnownMembership(opn2))$

$\land\, (EpochOf(KnownMembership(opn1)) = EpochOf(KnownMembership(opn2)) \Rightarrow$

$\quad KnownMembership(opn1) = KnownMembership(opn2))$

Prove

$\land\, EpochOf(KnownMembership(opn1)) \leq EpochOf(KnownMembership(opn2 + 1))$

$\land\, (EpochOf(KnownMembership(opn1)) = EpochOf(KnownMembership(opn2 + 1)) \Rightarrow$

$\quad KnownMembership(opn1) = KnownMembership(opn2 + 1))$

Defn $\quad state \triangleq KnownState[(opn2 - Alpha)]$

Step 2.1. of 5

$KnownMembership(opn2) = state.membershipMap[(state.numExecuted + Alpha)]$

Reasoning (2.1.): *Defn KnownMembership*; *Defn* state

Defn $\quad opv \triangleq KnownOpv[(opn2 - Alpha + 1)]$

Step 2.2. of 5

$KnownState[(opn2 - Alpha + 1)] = CsTx[state,\ opv]$

Step 2.2.1. of 1

$KnownState[(opn2 - Alpha + 1)] =$

$CsTx[(KnownState[(opn2 \quad - Alpha + 1 - 1)]),\ (KnownOpv[(opn2 - Alpha + 1)])]$

Reasoning (2.2.1.): *Defn KnownState*

Reasoning (2.2.): algebra

Step 2.3. of 5

$KnownMembership(opn2 + 1) =$

$CsTx[state,\ opv].membershipMap[((state.numExecuted + Alpha) + 1)]$

Summary: Basically a boring bunch of algebra

Step 2.3.1. of 3

$KnownMembership(opn2 + 1) = KnownState[(opn2 + 1 - Alpha)].membershipMap[(opn2 + 1)]$

Reasoning (2.3.1.): *Defn KnownMembership*

Step 2.3.2. of 3

$KnownState[(opn2 + 1 - Alpha)].membershipMap[(opn2 + 1)] =$

$CsTx[state,\ opv].membershipMap[(opn2 + 1)]$

Reasoning (2.3.2.): *Defn KnownState*; *Defn* state;*Defn opv*

Step 2.3.3. of 3

$CsTx[state,\ opv].membershipMap[(opn2 + 1)] =$

$CsTx[state,\ opv].membershipMap[((state.numExecuted + Alpha) + 1)]$

Step 2.3.3.1. of 1

$opn2 + 1 = (state.numExecuted + Alpha) + 1$

Step 2.3.3.1.1. of 1

$state.numExecuted = opn2 - Alpha$

Reasoning (2.3.3.1.1.): *Defn* state; *Ref* hypothesis:*NumExecutedTicks*

Reasoning (2.3.3.1.): algebra

Reasoning (2.3.3.): algebra

Reasoning (2.3.): transitivity

Case 2.4. of 5

$CsTx[state,\ opv].membershipMap[((state.numExecuted + Alpha) + 1)] =$
$state.membershipMap[(state.numExecuted + Alpha)]$

Step 2.4.1. of 1

$KnownMembership(opn2 + 1) = KnownMembership(opn2)$

Step 2.4.1.1. of 4

$KnownMembership(opn2 + 1) =$
$KnownState[(opn2 - Alpha)].membershipMap$
$[$
$(KnownState[(opn2 - Alpha)].numExecuted + Alpha)$
$]$

Reasoning (2.4.1.1.): Case condition, with substitutions from *Ref*:Step 2.3. and *Defn* state

Step 2.4.1.2. of 4

$KnownState[(opn2 - Alpha)].membershipMap$
$[$
$(KnownState[(opn2 - Alpha)].numExecuted + Alpha)$
$]$
$=$
$KnownState[(opn2 - Alpha)].membershipMap[(opn2 - Alpha + Alpha)]$

Reasoning (2.4.1.2.): *Ref* hypothesis:*NumExecutedTicks*

Step 2.4.1.3. of 4

$KnownState[(opn2 - Alpha)].membershipMap[(opn2 - Alpha + Alpha)] =$
$KnownState[(opn2 - Alpha)].membershipMap[opn2]$

Reasoning (2.4.1.3.): algebra

Step 2.4.1.4. of 4

$KnownState[(opn2 - Alpha)].membershipMap[opn2] = KnownMembership(opn2)$

Reasoning (2.4.1.4.): *Defn KnownMembership*

Reasoning (2.4.1.): transitivity

Reasoning (2.4.): We can substitute into Ref:Assumption 2.A1. to produce the proof goal.

Case 2.5. of 5

$EpochOf(CsTx[state,\ opv].membershipMap[((state.numExecuted + Alpha) + 1)]) =$
$EpochOf(state.membershipMap[(state.numExecuted + Alpha)]) + 1$

Step 2.5.1. of 2

$EpochOf(KnownMembership(opn2 + 1)) = EpochOf(KnownMembership(opn2)) + 1$

Reasoning (2.5.1.): *Ref*:Step 2.1. ; *Ref*:Step 2.3.

Step 2.5.2. of 2

$EpochOf(KnownMembership(opn1)) < EpochOf(KnownMembership(opn2 + 1))$

Reasoning (2.5.2.): *Ref*:Step 2.5.1. ; inductive hypothesis

Reasoning (2.5.): The first conjunct of the goal is clearly satisfied by the previous step, and the antecedent of the second conjunct of the goal is denied by the previous step.

Reasoning (2.): Case analysis; complete by *Ref*:*CsTxIncrementsEpochs* .

Reasoning: By induction over $opn2$

Theorem *NonconflictingViewMemberships*

Hypotheses of $MembershipsAreUnique$

Hypotheses of $MembershipEpochOrdering$

Hypotheses of $BroadcastMembershipsReflectKnownState$

Introduce $view \in ViewIds$

Introduce $membership \in Memberships$

Introduce $opn \in Opns$

Assume $view.viewIniator.epoch = EpochOf(membership)$

Assume $MembershipAs(opn, membership, LL!SentMessages)$

Prove $ViewMembership(view) = membership$

Defn $satisfyingMemberships \triangleq$

$$\begin{array}{l}
\{potentialMembership \in Memberships : \\
\quad (\land (\exists\, opn2 \in Opns : MembershipAs(opn2, potentialMembership, LL!SentMessages)) \\
\quad\quad \land EpochOf(potentialMembership) = view.viewInitiator.epoch) \\
\}
\end{array}$$

Step 1. of 2

$membership \in satisfyingMemberships$

Reasoning (1.): Follows from assumptions and *Defn Membership*

Step 2. of 2

Introduce $m2 \in satisfyingMemberships$

Assume $m2 \neq membership$

Prove FALSE

Defn $opn2 \triangleq \text{CHOOSE } opn2 \in Opns : MembershipAs(opn2, m2, LL!SentMessages)$

Step 2.1. of 6

$MembershipAs(opn2, m2, LL!SentMessages)$

Reasoning (2.1.): *Defn satisfyingMemberships*

Step 2.2. of 6

$opn \neq opn2$

Reasoning (2.2.): *Ref*:*MembershipsAreUnique*

Step 2.3. of 6

$KnownState[(opn2 - Alpha)].membershipMap[opn2] = membership$

Reasoning (2.3.): *Ref* hypothesis:*BroadcastMembershipsReflectKnownState*

Step 2.4. of 6

$KnownMembership(opn2) = membership$

Reasoning (2.4.): *Defn KnownMembership*

Step 2.5. of 6

$EpochOf(m2) \neq EpochOf(membership)$

Reasoning (2.5.): Algebra on *Ref:MembershipEpochOrdering*

Step 2.6. of 6

$EpochOf(m2) \neq view.viewInitiator.epoch$

Reasoning (2.6.): That distinction is already claimed by membership

Reasoning (2.): We have arrived at a contradiction.

Reasoning: CHOOSE in *ViewMembership* is fully constrained

Theorem *NonconflictingViewMembershipsPrimed*

Hypotheses of $MembershipsAreUnique$

Hypotheses of $MembershipEpochOrdering$

Hypotheses of $BroadcastMembershipsReflectKnownState$

Introduce $view \in ViewIds$

Introduce $membership \in Memberships$

Introduce $opn \in Opns$

Assume $view.viewIniator.epoch = EpochOf(membership)$

Assume $MembershipAs(opn, membership, LL!SentMessages')$

Prove $(ViewMembership(view)') = membership$

Reasoning: Track each hypothesis back to the underlying invariants, use the invariants to push the statements into the primed state, apply Ref:*NonconflictingViewMemberships* to get the conclusion.

Invariant *PrimaryDesignatedImpliesElectingQuorum*

Hypotheses of $MembershipChangesAreBroadcast$

Hypotheses of $CsStateTypeInvariant$

Hypotheses of $NonconflictingViewMembershipsPrimed$

Hypotheses of $MembershipAsDeterminesMembership$

Introduce $view \in ViewIds$

Introduce $primary \in Cohorts$

Assume

$$PrimaryDesignatedAs(view, primary) \Rightarrow (\exists\, quorum \in QuoraOfMembership(ViewMembership(view)) : PlausibleElectionQuorum(view, quorum))$$

Assume $PrimaryDesignatedAs(view, primary)'$

Prove

$$(\exists\, quorum \in QuoraOfMembership(ViewMembership(view)) : PlausibleElectionQuorum(view, quorum))'$$

Summary: The interesting action is *DesignatePrimary*; all other actions fall out by monotonicity.

Case 1. of 2

$\exists\, \mathit{config} \in \mathit{DesignationConfigurations} :$

$\quad \wedge\, \mathit{config.view} = \mathit{view}$

$\quad \wedge\, \mathit{LL}!\mathit{Replica}(\mathit{config.designator})!\mathit{DesignatePrimary}(\mathit{config})$

Summary: If a primary was designated in this step, then we identify the *VcAcks* used to make that decision. The cohorts that sent those *VcAcks* (the electing quorum) must have formed a *PlausibleElectionQuorum*, so we prove the conjuncts of that definition.

Defn $\quad \mathit{config} \triangleq$

$\textsc{choose}\ \mathit{config} \in \mathit{DesignationConfigurations} :$

$\quad \wedge\, \mathit{config.view} = \mathit{view}$

$\quad \wedge\, \mathit{LL}!\mathit{Replica}(\mathit{config.designator})!\mathit{DesignatePrimary}(\mathit{config})$

Step 1.1. of 7

$\wedge\, \mathit{config.view} = \mathit{view}$

$\wedge\, \mathit{LL}!\mathit{Replica}(\mathit{config.designator})!\mathit{DesignatePrimary}(\mathit{config})$

Reasoning (1.1.): CHOOSE Axiom

Step 1.2. of 7

$\mathit{LL}!\mathit{Replica}(\mathit{config.designator})!\mathit{CurView} = \mathit{view}$

Summary: The configuration was chosen specifically to enforce this equality

Defn $\quad \mathit{witnessMsg} \triangleq \textsc{choose}\ m \in \mathit{config.msgs} : \textsc{true}$

Step 1.2.1. of 3

$\mathit{witnessMsg} \in \mathit{config.msgs}$

Reasoning (1.2.1.): *Defn DesignationConfigurations* explicitly disallows empty *.msgs* fields.

Step 1.2.2. of 3

$\mathit{witnessMsg.view} = \mathit{LL}!\mathit{Replica}(\mathit{config.designator})!\mathit{CurView}$

Reasoning (1.2.2.): universal quantifier in *Defn DesignatePrimaryAction*

Step 1.2.3. of 3

$\mathit{witnessMsg.view} = \mathit{view}$

Reasoning (1.2.3.): *Defn DesignationConfigurations*; *Ref*:Step 1.1.

Reasoning (1.2.): substitution

Step 1.3. of 7

$\mathit{config.designator} = \mathit{view.viewInitiator}$

Step 1.3.1. of 2

$\mathit{config.designator} = \mathit{LL}!\mathit{Replica}(\mathit{config.designator})!\mathit{ThisCohort}$

Reasoning (1.3.1.): *Defn DesignatePrimary*

Step 1.3.2. of 2

$\mathit{LL}!\mathit{Replica}(\mathit{config.designator})!\mathit{ThisCohort} =$
$\mathit{LL}!\mathit{Replica}(\mathit{config.designator})!\mathit{CurView.viewInitiator}$

Reasoning (1.3.2.): *Defn DesignatePrimaryAction*

Reasoning (1.3.): Substitution, including *Ref*:Step 1.2.

Step 1.4. of 7

$\forall\, \mathit{cohort} \in \mathit{config.quorum} : (\mathit{VcAckedView}(\mathit{view},\ \mathit{cohort})')$

Step 1.4.1. of 1

Introduce $\quad \mathit{cohort} \in \mathit{config.quorum}$

Prove $VcAckedView(view,\ cohort)'$

Summary: $config.msgs$ provides the collection of $VcAck$ messages.

Defn $vcAckMsg \triangleq \text{CHOOSE } vcAckMsg \in config.msgs : vcAckMsg.sender = cohort$

Step 1.4.1.1. of 3

$vcAckMsg.sender = cohort$

Reasoning (1.4.1.1.): *Defn DesignatePrimaryAction*; *Defn EachCohortSentAMessage*; CHOOSE axiom

Step 1.4.1.2. of 3

$VcAcked(view,\ cohort,\ vcAckMsg.preparedOps)$

Step 1.4.1.2.1. of 2

$vcAckMsg \in SentMessages$

Reasoning (1.4.1.2.1.): *Defn* $ReceiveMessageSet(config.msgs)$

Step 1.4.1.2.2. of 2

$vcAckMsg.view = view$

Reasoning (1.4.1.2.2.): *Defn DesignationConfigurations*

Reasoning (1.4.1.2.): *Defn vcAckMsg*; *Defn VcAcked*

Step 1.4.1.3. of 3

$VcAckedView(view,\ cohort)$

Reasoning (1.4.1.3.): $vcAckMsg.preparedOps$ is witness to the existential in *Defn VcAckedView*

Reasoning (1.4.1.): *Ref*: *VcAckedMonotonic*

Reasoning (1.4.): expand universal quantifier

Step 1.5. of 7

$DesignationReflectsVcAcks(view,\ config.quorum)'$

Summary: This step follows by the construction of the message sent in *DesignatePrimaryAction*.

Step 1.5.1. of 5

$\forall\, vcAckMsg \in config.msgs : vcAckMsg.sender \in config.quorum$

Reasoning (1.5.1.): *Defn EachCohortSentAMessage*

Step 1.5.2. of 5

$\forall\, vcAckMsg \in config.msgs : vcAckMsg.view = view$

Reasoning (1.5.2.): *Defn DesignatePrimaryAction*;
Defn DesignationConfigurations

Defn $designationMsg \triangleq$

$MakePrimaryDesignatedMsg($
$view,$
$view.viewInitiator,$
$primary,$
$MaxTruncationPoint(config.msgs),$
$AggregatePreparedOps(config.msgs))$

Step 1.5.3. of 5

$designationMsg.view = view$

Reasoning (1.5.3.): *Defn designationMsg*; *Defn MakePrimaryDesignatedMsg*

Step 1.5.4. of 5

$\mathit{designationMsg.prevPrepares} = \mathit{AggregatePreparedOps}(\mathit{config.msgs})$

Reasoning (1.5.4.): *Defn designationMsg*; *Defn MakePrimaryDesignatedMsg*

Step 1.5.5. of 5

$\mathit{DesignationReflectsVcAcks}(\mathit{view}, \mathit{config.quorum})'$

Reasoning (1.5.5.): With existential witnesses *designationMsg* \is *designationMsg* and *vcAckMsgSet* \is *config.msgs*, we have satisfied each conjunct of *DesignationReflectsVcAcks*(*view*, *config.quorum*).

Reasoning (1.5.):*Ref*:*DesignationReflectsVcAcksMonotonic*

Step 1.6. of 7

$\mathit{PlausibleElectionQuorum}(\mathit{view}, \mathit{config.quorum})'$

Reasoning (1.6.): Prior two steps satisfy *Defn PlausibleElectionQuorum′*

Step 1.7. of 7

$\mathit{config.quorum} \in \mathit{QuoraOfMembership}(\mathit{ViewMembership}(\mathit{view})')$

Defn $\mathit{membershipOpn} \triangleq$

$\textsc{choose}\ \mathit{membershipOpn} \in \mathit{Opns} :$

$\mathit{view.viewInitiator} \in$

$\mathit{LL}!\mathit{Replica}(\mathit{view.viewInitiator})!\mathit{CsState}'.\mathit{membershipMap}[\mathit{membershipOpn}]$

Defn $\mathit{membership} \triangleq$

$\mathit{LL}!\mathit{Replica}(\mathit{view.viewInitiator})!\mathit{CsState}'.\mathit{membershipMap}[\mathit{membershipOpn}]$

Step 1.7.1. of 1

$(\mathit{ViewMembership}(\mathit{view})') = \mathit{membership}$

Summary: Sketch: *ActiveMember* ⇒ *LL*!*Membership* is defined. ⇒ it's been recorded in the message history ⇒ *LL*!*Membership* is in the set of globally-known memberships (recorded in message history) (an invariant; not sure how many steps) ⇒ it's the only one (a different invariant, coinductive with nonconflicting-commits) ⇒ it's the one chosen by *Defn ViewMembership*

Step 1.7.1.1. of 3

$\mathit{view.viewInitiator} \in \mathit{membership}$

Reasoning (1.7.1.1.): *Defn ActiveMember′*; CHOOSE axiom

Step 1.7.1.2. of 3

$\mathit{MembershipAs}(\mathit{membershipOpn}, \mathit{membership}, \mathit{LL}!\mathit{SentMessages}')$

Reasoning (1.7.1.2.): *Ref* hypothesis:*MembershipChangesAreBroadcast* ($\mathit{opn} = \mathit{membershipOpn}$, $\mathit{cohort} = \mathit{view.viewInitiator}$, $\mathit{membership} = \mathit{membership}$)

Step 1.7.1.3. of 3

$\mathit{view.viewIniator.epoch} = \mathit{EpochOf}(\mathit{membership})$

Step 1.7.1.3.1. of 2

$(\mathit{Membership}(\mathit{membershipOpn})') = \mathit{membership}$

Reasoning (1.7.1.3.1.):*Ref*:*MembershipAsDeterminesMembership*

Step 1.7.1.3.2. of 2

$\mathit{membership} \in \mathit{Memberships}$

Reasoning (1.7.1.3.2.): *Ref*:*CsStateTypeInvariant*

Reasoning (1.7.1.3.): *Defn EpochOf*; *Defn Memberships*

Reasoning (1.7.1.): We have satisfied the antecedent of Ref:*NonconflictingViewMembershipsPrimed* ($\mathit{view} = \mathit{view}$, $\mathit{membership} = \mathit{membership}$, $\mathit{opn} = \mathit{membershipOpn}$)

Reasoning (1.7.): *Defn DesignatePrimaryAction* provides $\mathit{config.quorum} = \mathit{LL}!\mathit{Replica}(\mathit{view.viewInitiator})!\mathit{Membership}$; expand *Defn Quora*

Reasoning (1.): We have exhibited a witness *config.quorum* to existential variable quorum

DefaultCase 2. of 2

Summary: When no "interesting" action has occurred, the relevant predicates are monotonic.

Step 2.1. of 3

$\mathit{PrimaryDesignatedAs}(\mathit{view}, \mathit{primary})$

Reasoning (2.1.): No other actions send *PrimaryDesignated* messages for this view, so *PrimaryDesignatedAs*() cannot have changed.

Step 2.2. of 3

$\exists\, \mathit{quorum} \in \mathit{QuoraOfMembership}(\mathit{ViewMembership}(\mathit{view})) :$
$\quad \mathit{PlausibleElectionQuorum}(\mathit{view}, \mathit{quorum})$

Reasoning (2.2.): induction hypothesis

Defn $\mathit{quorum} \triangleq$
$\quad \text{CHOOSE } \mathit{quorum} \in \mathit{QuoraOfMembership}(\mathit{ViewMembership}(\mathit{view})) :$
$\quad \mathit{PlausibleElectionQuorum}(\mathit{view}, \mathit{quorum})$

Step 2.3. of 3

$\mathit{PlausibleElectionQuorum}(\mathit{view}, \mathit{quorum})'$

Reasoning (2.3.): CHOOSE axiom;*Ref*:*PlausibleElectionQuorumMonotonic*

Reasoning (2.): We have exhibited a witness variable quorum

Reasoning: Case analysis

Invariant *IAmPrimaryImpliesElectingQuorum*

Hypotheses of *PrimaryDesignatedImpliesElectingQuorum*

Introduce $\mathit{view} \in \mathit{ViewIds}$

Introduce $\mathit{primary} \in \mathit{Cohorts}$

Assume

$\land\ \mathit{LL}!\mathit{Replica}(\mathit{primary})!\mathit{CurView} = \mathit{view}$
$\land\ \mathit{LL}!\mathit{Replica}(\mathit{primary})!\mathit{IAmPrimary}$
$\Rightarrow$
$(\exists\, \mathit{quorum} \in \mathit{QuoraOfMembership}(\mathit{ViewMembership}(\mathit{view})) :$
$\quad \mathit{PlausibleElectionQuorum}(\mathit{view}, \mathit{quorum}))$

Assume

$(\land\ \mathit{LL}!\mathit{Replica}(\mathit{primary})!\mathit{CurView} = \mathit{view}$
$\ \land\ \mathit{LL}!\mathit{Replica}(\mathit{primary})!\mathit{IAmPrimary})'$

Prove

$(\exists\, \mathit{quorum} \in \mathit{QuoraOfMembership}(\mathit{ViewMembership}(\mathit{view})) :$
$\quad \mathit{PlausibleElectionQuorum}(\mathit{view}, \mathit{quorum}))'$

Case 1. of 2

$\exists\, m \in \mathit{PrimaryDesignatedMsg} : \mathit{LL}!\mathit{Replica}(\mathit{primary})!\mathit{BecomePrimary}(m)$

Step 1.1. of 1

$\mathit{PrimaryDesignatedAs}(\mathit{view}, \mathit{primary})$

Reasoning (1.1.): *Defn BecomePrimary* shows that *m* is a witness to *Defn PrimaryDesignatedAs*.

Reasoning (1.): *Ref* hypothesis:*PrimaryDesignatedImpliesElectingQuorum*

DefaultCase 2. of 2

Step 2.1. of 3

$\land\ \mathit{LL}!\mathit{Replica}(\mathit{primary})!\mathit{CurView} = \mathit{view}$
$\land\ \mathit{LL}!\mathit{Replica}(\mathit{primary})!\mathit{IAmPrimary}$

Reasoning (2.1.): No actions on cohort other than *BecomePrimary* make *IAmPrimary* transition to TRUE. (*VcAck* changes *CurView*, but it also assigns $\mathit{IAmPrimary}' = \text{FALSE}$, which we have assumed isn't the case.)

Defn $\mathit{quorum} \triangleq$
$\text{CHOOSE } \mathit{quorum} \in \mathit{QuoraOfMembership}(\mathit{ViewMembership}(\mathit{view})) :$
$\mathit{PlausibleElectionQuorum}(\mathit{view}, \mathit{quorum})$

Step 2.2. of 3

$\mathit{PlausibleElectionQuorum}(\mathit{view}, \mathit{quorum})$

Reasoning (2.2.): induction hypothesis

Step 2.3. of 3

$\mathit{PlausibleElectionQuorum}(\mathit{view}, \mathit{quorum})'$

Reasoning (2.3.):*Ref*:*PlausibleElectionQuorumMonotonic*

Reasoning (2.): We have exhibited a witness variable quorum

Reasoning: Case analysis

Invariant *ProposedImpliesElectingQuorum*

which also incorporates the hypotheses of *ProposedImpliesElectingQuorum*

Hypotheses of $\mathit{IAmPrimaryImpliesElectingQuorum}$
Hypotheses of $\mathit{PrimaryDesignatedImpliesElectingQuorum}$
Introduce $\mathit{view} \in \mathit{ViewIds}$
Introduce $\mathit{opn} \in \mathit{Opns}$

Assume

$\mathit{ProposedByAny}(\mathit{view}, \mathit{opn}) \Rightarrow$
$(\exists\, \mathit{quorum} \in \mathit{QuoraOfMembership}(\mathit{ViewMembership}(\mathit{view})) :$
$\mathit{PlausibleElectionQuorum}(\mathit{view}, \mathit{quorum}))$

Assume $\mathit{ProposedByAny}(\mathit{view}, \mathit{opn})'$

Prove

$(\exists\, \mathit{quorum} \in \mathit{QuoraOfMembership}(\mathit{ViewMembership}(\mathit{view})) :$
$\mathit{PlausibleElectionQuorum}(\mathit{view}, \mathit{quorum}))'$

Defn $\mathit{primary} \triangleq \text{CHOOSE } \mathit{primary} \in \mathit{Cohorts} : (\mathit{Proposed}(\mathit{view}, \mathit{primary}, \mathit{opn})')$

Step 1. of 4

$\mathit{Proposed}(\mathit{view}, \mathit{primary}, \mathit{opn})'$

Reasoning (1.): *Defn ProposedByAny*; CHOOSE axiom

Case 2. of 4

$\exists\, opv \in CsOps : LL!Replica(primary)!ProposeAction(view,\ opn,\ opv)$

Step 2.1. of 1

$\land\ LL!Replica(primary)!CurView = view$
$\land\ LL!Replica(primary)!IAmPrimary$

Reasoning (2.1.): *Defn ProposeAction*

Reasoning (2.): *Ref* hypothesis:*IAmPrimaryImpliesElectingQuorum*

Case 3. of 4

$\exists\, m \in PrimaryDesignatedMsg : LL!Replica(primary)!BecomePrimary(m)$

Step 3.1. of 1

$PrimaryDesignatedAs(view,\ primary)$

Reasoning (3.1.): *Defn BecomePrimary* shows that m is a witness to *Defn PrimaryDesignatedAs*.

Reasoning (3.): *Ref* hypothesis:*PrimaryDesignatedImpliesElectingQuorum*

DefaultCase 4. of 4

Step 4.1. of 3

$Proposed(view,\ primary,\ opn)$

Reasoning (4.1.): No other actions change proposals

Defn $quorum \triangleq$
CHOOSE $quorum \in QuoraOfMembership(\,ViewMembership(view)) :$
$PlausibleElectionQuorum(view,\ quorum)$

Step 4.2. of 3

$PlausibleElectionQuorum(view,\ quorum)$

Reasoning (4.2.): *Ref* hypothesis:*ProposedImpliesElectingQuorum*

Step 4.3. of 3

$PlausibleElectionQuorum(view,\ quorum)'$

Reasoning (4.3.):*Ref*:*PlausibleElectionQuorumMonotonic*

Reasoning (4.): We have exhibited a witness variable quorum

Reasoning: Case analysis

Invariant *CsStateTypeInvariant*

Introduce $cohort \in Cohorts$

Assume

$\land\ LL!Replica(cohort)!CsState \in CsStates$
$\land\ LL!Replica(cohort)!CsStateSnapshot \in CsStates$

Prove

$(\land\ LL!Replica(cohort)!CsState \in CsStates$
$\land\ LL!Replica(cohort)!CsStateSnapshot \in CsStates)'$

Reasoning:

Theorem *OpnInMembershipMapImpliesMembershipDefined*

Hypotheses of $MembershipChangesAreBroadcast$

Hypotheses of $CsStateTypeInvariant$

Introduce $opn \in Opns$

Introduce $cohort \in Cohorts$

Assume $opn \in \textsc{domain}\ LL!Replica(cohort)!CsState.membershipMap$

Prove $MembershipDefined(opn)$

Defn $membership \triangleq LL!Replica(cohort)!CsState.membershipMap[opn]$

- Step 1. of 2

  $MembershipAs(opn, membership, LL!SentMessages)$

  - Reasoning (1.): *Ref* hypothesis:*MembershipChangesAreBroadcast*

- Step 2. of 2

  $membership \in Memberships$

  - Reasoning (2.): *Ref* hypothesis:*CsStateTypeInvariant*

- Reasoning: membership is witness to *Defn MembershipDefined*

Invariant *ProposedImpliesMembershipDefined*

Hypotheses of $OpnInMembershipMapImpliesMembershipDefined$

Introduce $view \in ViewIds$

Introduce $cohort \in Cohorts$

Introduce $opn \in Opns$

Assume $Proposed(view, cohort, opn) \Rightarrow MembershipDefined(opn)$

Assume $Proposed(view, cohort, opn)'$

Prove $MembershipDefined(opn)'$

- Step 1. of 1

  $MembershipDefined(opn)$

  - Case 1.1. of 2

    $\exists\, opv \in CsOps : LL!Replica(cohort)!ProposeAction(view, opn, opv)$

    - Step 1.1.1. of 1

      $opn \in \textsc{domain}\ LL!Replica(cohort)!CsState.membershipMap$

      - Reasoning (1.1.1.): *Def ActiveMember*

    - Reasoning (1.1.):*Ref*:*OpnInMembershipMapImpliesMembershipDefined*

  - *DefaultCase* 1.2. of 2

    - Step 1.2.1. of 1

      $Proposed(view, cohort, opn)$

      - Reasoning (1.2.1.): No other action changes *Proposed*

    - Reasoning (1.2.): induction hypothesis

  - Reasoning (1.): Case analysis

- Reasoning: *Ref*:*MembershipDefinedMonotonic*

Invariant *LastProposedReflectsPrevPreps*

Hypotheses of $UniquePrimaryDesignationMessage$

Hypotheses of $PrimaryDesignatedPrecludesDesignationNeeded$

Hypotheses of $IAmPrimaryImpliesPrimaryDesignated$

Introduce $view \in ViewIds$

Introduce $primary \in Cohorts$

Introduce $opn \in Opns$

Introduce $opv \in CsOps$

Assume

$$\begin{array}{l}
\land\ PrimaryDesignatedPrevPrep(view,\ opn,\ opv) \\
\land\ LL!Replica(primary)!CurView = view \\
\land\ LL!Replica(primary)!IAmPrimary \\
\Rightarrow \\
opn \leq LL!Replica(primary)!LastProposed
\end{array}$$

Assume

$$\begin{array}{l}
(\ \land\ PrimaryDesignatedPrevPrep(view,\ opn,\ opv) \\
\quad \land\ LL!Replica(primary)!CurView = view \\
\quad \land\ LL!Replica(primary)!IAmPrimary)'
\end{array}$$

Prove $(opn \leq LL!Replica(primary)!LastProposed)'$

Case 1. of 5

$\exists\, m \in PrimaryDesignatedMsg : LL!Replica(primary)!BecomePrimary(m)$

Summary: Action assigns *LastProposed* suitably.

Defn $m1 \triangleq$

$$\begin{array}{l}
\text{CHOOSE}\ m1 \in SentMessages \cap PrimaryDesignatedMsg : \\
\quad \land\ m1.view = view \\
\quad \land\ opn \in \text{DOMAIN}\ m1.prevPrepares \\
\quad \land\ m1.prevPrepares[opn] = opv
\end{array}$$

Defn $m2 \triangleq$

$\text{CHOOSE}\ m2 \in PrimaryDesignatedMsg : LL!Replica(primary)!BecomePrimary(m2)$

Step 1.1. of 2

$m1 = m2$

Reasoning (1.1.): Antecedent makes first CHOOSE succeed; Case condition makes second CHOOSE succeed; *Ref* hypothesis: *UniquePrimaryDesignationMessage*

Step 1.2. of 2

$opn \leq MaxPreparedOpn(m1)$

Reasoning (1.2.): *Defn MaxPreparedOpn*

Reasoning (1.): substitution

Case 2. of 5

$$\begin{array}{l}
\exists\, opn2 \in Opns,\ opv2 \in CsOps : \\
\quad LL!Replica(primary)!ProposeAction(view,\ opn2,\ opv2)
\end{array}$$

Summary: induction hypothesis holds in unprimed state, and action increments *LastProposed*, so things only get better.

Step 2.1. of 3

$\wedge\ PrimaryDesignatedPrevPrep(view,\ opn,\ opv)$
$\wedge\ LL!Replica(primary)!CurView = view$
$\wedge\ LL!Replica(primary)!IAmPrimary$

Reasoning (2.1.): Propose doesn't send a *PrimaryDesignatedMsg*, and leaves *CurView* and *IAmPrimary* UNCHANGED

Step 2.2. of 3

$opn \leq LL!Replica(primary)!LastProposed$

Reasoning (2.2.): induction hypothesis

Step 2.3. of 3

$LL!Replica(primary)!LastProposed < (LL!Replica(primary)!LastProposed')$

Reasoning (2.3.): *Defn* Propose

Reasoning (2.): transitivity

Case 3. of 5

$\exists\ config \in DesignationConfigurations :$
$\wedge\ config.view = view$
$\wedge\ LL!Replica(config.designator)!DesignatePrimary(config)$

Summary: If the cohort is already operating as a primary, we won't see $a(nother)DesignatePrimary$ action on this view.

Step 3.1. of 3

$\wedge\ LL!Replica(view.viewInitiator)!DesignationNeeded$
$\wedge\ LL!Replica(view.viewInitiator)!CurView = view$

Reasoning (3.1.): *Defn DesignatePrimary*;substitution

Step 3.2. of 3

$\neg PrimaryDesignated(view)$

Reasoning (3.2.): *Ref* hypothesis:*PrimaryDesignatedPrecludesDesignationNeeded* ; algebra

Step 3.3. of 3

$PrimaryDesignatedAs(view,\ primary)$

Reasoning (3.3.): *Ref* hypothesis:*IAmPrimaryImpliesPrimaryDesignated*

Reasoning (3.): Case eliminated by contradiction

Case 4. of 5

$LL!Replica(primary)!Crash$

Summary: This action cannot have happened if $IAmPrimary'$ is TRUE.

Step 4.1. of 1

$\neg(LL!Replica(primary)!IAmPrimary')$

Reasoning (4.1.): *Defn Crash* action

Reasoning (4.): Case eliminated by contradiction

DefaultCase 5. of 5

Step 5.1. of 5

$PrimaryDesignatedPrevPrep(view,\ opn,\ opv)$

Reasoning (5.1.): No action other than *DesignatePrimary* sends a *PrimaryDesignatedMsg*

Step 5.2. of 5

$LL!Replica(primary)!CurView = view$

Reasoning (5.2.): Only $VcAck$ action updates $CurView$, and it requires $\neg IAmPrimary'$, so it cannot have happened.

Step 5.3. of 5

$LL!Replica(primary)!IAmPrimary$

Reasoning (5.3.): No action other than $BecomePrimary$ changes $IAmPrimary$ to TRUE

Step 5.4. of 5

$opn \leq LL!Replica(primary)!LastProposed$

Reasoning (5.4.): induction hypothesis

Step 5.5. of 5

$opn \leq (LL!Replica(primary)!LastProposed')$

Reasoning (5.5.): No action other than Propose, Crash, and $BecomePrimary$ changes $LastProposed$.

Reasoning (5.): Done.

Reasoning: case analysis

Invariant $ProposalsRespectPrevPrepares$

Hypotheses of $ProposedImpliesPrimaryDesignated$

Hypotheses of $PrimaryDesignatedPrecludesDesignationNeeded$

Hypotheses of $IAmPrimaryImpliesPrimaryDesignated$

Hypotheses of $UniquePrimaryDesignated$

Hypotheses of $LastProposedReflectsPrevPreps$

Introduce $view \in ViewIds$

Introduce $opn \in Opns$

Introduce $opv1 \in CsOps$

Introduce $opv2 \in CsOps$

Assume

$\wedge\ PrimaryDesignatedPrevPrep(view,\ opn,\ opv1)$

$\wedge\ ProposedByAnyAs(view,\ opn,\ opv2)$

$\Rightarrow$

$opv1 = opv2$

Assume

$(\wedge\ PrimaryDesignatedPrevPrep(view,\ opn,\ opv1)$

$\wedge\ ProposedByAnyAs(view,\ opn,\ opv2))'$

Prove $(opv1 = opv2)'$

Summary: Three actions are interesting: We show designation cannot occur (again) if a proposal has already been made. A Propose action cannot occur, because $LastProposed$ will prevent it. A $BecomePrimary$ action will respect $PrevPrepares$.

Defn $primary \triangleq$ CHOOSE $primary \in Cohorts : PrimaryDesignatedAs(view,\ primary)$

Step 1. of 5

$PrimaryDesignatedAs(view,\ primary)$

Reasoning (1.): *Ref* hypothesis:*ProposedImpliesPrimaryDesignated* and some expansion of quantifiers

Case 2. of 5

$\exists\, \mathit{config} \in \mathit{DesignationConfigurations} :$
$\quad \mathit{LL}!\mathit{Replica}(\mathit{config}.\mathit{designator})!\mathit{DesignatePrimary}(\mathit{config})$

Summary: Since there has already been a proposal in the view, this action cannot be enabled.

Defn $\mathit{config} \triangleq$
$\quad \textsc{choose}\ \mathit{config} \in \mathit{DesignationConfigurations} :$
$\quad\quad \mathit{LL}!\mathit{Replica}(\mathit{config}.\mathit{designator})!\mathit{DesignatePrimary}(\mathit{config})$

Step 2.1. of 1

$\neg \mathit{LL}!\mathit{Replica}(\mathit{config}.\mathit{designator})!\mathit{DesignationNeeded}$

Reasoning (2.1.): A bunch of substitutions on *Defn DesignatePrimary*; then apply *Ref* hypothesis:*PrimaryDesignatedPrecludesDesignationNeeded*

Reasoning (2.): Case eliminated by contradiction with *Defn DesignatePrimary*

Case 3. of 5

$\exists\, \mathit{cohort} \in \mathit{Cohorts} : \mathit{LL}!\mathit{Replica}(\mathit{cohort})!\mathit{ProposeAction}(\mathit{view},\ \mathit{opn},\ \mathit{opv}2)$

Summary: A Propose action cannot occur, because *LastProposed* will prevent it.

Defn $\mathit{cohort} \triangleq$
$\quad \textsc{choose}\ \mathit{cohort} \in \mathit{Cohorts} : \mathit{LL}!\mathit{Replica}(\mathit{cohort})!\mathit{ProposeAction}(\mathit{view},\ \mathit{opn},\ \mathit{opv}2)$

Step 3.1. of 2

$\mathit{LL}!\mathit{Replica}(\mathit{primary})!\mathit{ProposeAction}(\mathit{view},\ \mathit{opn},\ \mathit{opv}2)$

Step 3.1.1. of 3

$\mathit{LL}!\mathit{Replica}(\mathit{cohort})!\mathit{IAmPrimary}$

Reasoning (3.1.1.): *Defn* Propose

Step 3.1.2. of 3

$\mathit{PrimaryDesignatedAs}(\mathit{view},\ \mathit{cohort})$

Reasoning (3.1.2.): *Ref* hypothesis:*IAmPrimaryImpliesPrimaryDesignated*

Step 3.1.3. of 3

$\mathit{cohort} = \mathit{primary}$

Reasoning (3.1.3.): *Ref*:*UniquePrimaryDesignated*

Reasoning (3.1.): Substitution into case condition

Step 3.2. of 2

$\mathit{opn} \leq \mathit{LL}!\mathit{Replica}(\mathit{cohort})!\mathit{LastProposed}$

Reasoning (3.2.):*Ref*:*LastProposedReflectsPrevPreps*

Reasoning (3.): Case eliminated by contradiction with *Defn ProposeAction* ($\mathit{opn} = \mathit{LastProposed} + 1$)

Case 4. of 5

$\exists\, m \in \mathit{PrimaryDesignatedMsg} :$
$\quad \land\ \mathit{LL}!\mathit{Replica}(m.\mathit{newPrimary})!\mathit{BecomePrimary}(m)$
$\quad \land\ m.\mathit{view} = \mathit{view}$
$\quad \land\ \mathit{opn} \in \textsc{domain}\ m.\mathit{prevPrepares}$

Defn $m \triangleq$
$\quad \textsc{choose}\ m \in \mathit{PrimaryDesignatedMsg} :$
$\quad\quad \land\ \mathit{LL}!\mathit{Replica}(m.\mathit{newPrimary})!\mathit{BecomePrimary}(m)$
$\quad\quad \land\ m.\mathit{view} = \mathit{view}$

$\land\ opn \in \textsc{domain}\ m.prevPrepares$

Step 4.1. of 1

$ProposedAs(view,\ m.newPrimary,\ opn,\ m.prevPrepares[opn])'$

Reasoning (4.1.): *Defn BecomePrimary* sends proposal message

Reasoning (4.):*Ref:ProposedsInSameViewDoNotConflict*

DefaultCase 5. of 5

Step 5.1. of 2

$PrimaryDesignatedPrevPrep(view,\ opn,\ opv1)$

Reasoning (5.1.): *PrimaryDesignatedPrevPrep* cannot change without a *DesignatePrimary* action

Step 5.2. of 2

$ProposedByAnyAs(view,\ opn,\ opv2)$

Reasoning (5.2.): *ProposedByAnyAs*(*view*, *opn*, *opv*2) cannot change without a suitable Propose or *BecomePrimary* action

Reasoning (5.): induction hypothesis ; conclusion is a constant expression

Reasoning: case analysis

Invariant *ViewInitiatorElectsPrimaryInSameEpoch*

Hypotheses of $CsStateTypeInvariant$

Introduce $view \in ViewIds$

Introduce $primary \in Cohorts$

Assume

$PrimaryDesignatedAs(view,\ primary) \Rightarrow view.viewInitiator.epoch = primary.epoch$

Assume $PrimaryDesignatedAs(view,\ primary)'$

Prove $(view.viewInitiator.epoch = primary.epoch)'$

Case 1. of 2

$\exists\ config \in DesignationConfigurations :$

$\quad \land\ config.view = view$

$\quad \land\ config.newPrimary = primary$

$\quad \land\ LL!Replica(config.designator)!DesignatePrimary(config)$

Defn $config \triangleq$

$\textsc{choose}\ config \in DesignationConfigurations :$

$\quad \land\ config.view = view$

$\quad \land\ config.newPrimary = primary$

$\quad \land\ LL!Replica(config.designator)!DesignatePrimary(config)$

Step 1.1. of 3

$view.viewInitiator \in LL!Replica(config.designator)!Membership$

Step 1.1.1. of 2

$config.designator \in LL!Replica(config.designator)!Membership$

Reasoning (1.1.1.): *Defn DesignatePrimary*; *Defn ActiveMember*

Step 1.1.2. of 2

$\mathit{config.designator} = \mathit{view.viewInitiator}$

Reasoning (1.1.2.): *Defn DesignatePrimary*

Reasoning (1.1.): substitution

Step 1.2. of 3

$\mathit{primary} \in \mathit{LL}!\mathit{Replica}(\mathit{config.designator})!\mathit{Membership}$

Step 1.2.1. of 5

$\mathit{config.quorum} \in \mathit{LL}!\mathit{Replica}(\mathit{config.designator})!\mathit{Quora}$

Reasoning (1.2.1.): *Defn DesignatePrimary*

Step 1.2.2. of 5

$\mathit{config.quorum} \in \mathit{QuoraOfMembership}(\mathit{LL}!\mathit{Replica}(\mathit{config.designator})!\mathit{Membership})$

Reasoning (1.2.2.): *Defn Quora*

Step 1.2.3. of 5

$\mathit{config.quorum} \subseteq \mathit{LL}!\mathit{Replica}(\mathit{config.designator})!\mathit{Membership}$

Reasoning (1.2.3.): *Defn QuoraOfMembership*

Step 1.2.4. of 5

$\mathit{config.newPrimary} \in \mathit{config.quorum}$

Reasoning (1.2.4.): *Defn DesignationConfigurations*

Step 1.2.5. of 5

$\mathit{config.newPrimary} \in \mathit{LL}!\mathit{Replica}(\mathit{config.designator})!\mathit{Membership}$

Reasoning (1.2.5.): Substitution

Reasoning (1.2.): Substitution

Step 1.3. of 3

$\mathit{LL}!\mathit{Replica}(\mathit{config.designator})!\mathit{Membership} \in \mathit{Memberships}$

Step 1.3.1. of 1

$\mathit{Range}(\mathit{LL}!\mathit{Replica}(\mathit{config.designator})!\mathit{CsState.membershipMap}) = \mathit{Memberships}$

Reasoning (1.3.1.): *Ref* hypothesis:*CsStateTypeInvariant*

Reasoning (1.3.): *Defn Membership*

Reasoning (1.): *Defn Memberships* requires all members to share a common epoch.

DefaultCase 2. of 2

Step 2.1. of 1

$\mathit{PrimaryDesignatedAs}(\mathit{view}, \mathit{primary})$

Reasoning (2.1.): No other action could have sent a witness *PrimaryDesignatedMsg*, so it must already have been in *SentMessages*

Reasoning (2.): induction hypothesis

Reasoning: Proof by case analysis

Invariant *PrimaryAndViewInitiatorInSameEpoch*

Hypotheses of $\mathit{ViewInitiatorElectsPrimaryInSameEpoch}$

Introduce $\mathit{primary} \in \mathit{Cohorts}$

Assume

$\mathit{LL}!\mathit{Replica}(\mathit{primary})!\mathit{IAmPrimary} \Rightarrow$

$\mathit{LL}!\mathit{Replica}(\mathit{primary})!\mathit{CurView}.\mathit{viewInitiator}.\mathit{epoch} = \mathit{primary}.\mathit{epoch}$

Assume $\mathit{LL}!\mathit{Replica}(\mathit{primary})!\mathit{IAmPrimary}'$

Prove $(\mathit{LL}!\mathit{Replica}(\mathit{primary})!\mathit{CurView}.\mathit{viewInitiator}.\mathit{epoch} = \mathit{primary}.\mathit{epoch})'$

Case 1. of 2

$\exists\, m \in \mathit{PrimaryDesignatedMsg} : \mathit{LL}!\mathit{Replica}(\mathit{primary})!\mathit{BecomePrimary}(m)$

Defn $m \triangleq$

CHOOSE $m \in \mathit{PrimaryDesignatedMsg} : \mathit{LL}!\mathit{Replica}(\mathit{primary})!\mathit{BecomePrimary}(m)$

Step 1.1. of 1

$\mathit{PrimaryDesignatedAs}(\mathit{LL}!\mathit{Replica}(\mathit{primary})!\mathit{CurView},\ \mathit{primary})$

Reasoning (1.1.): *Defn BecomePrimary* constraints *CurView*

Reasoning (1.): *Ref* hypothesis: *ViewInitiatorElectsPrimaryInSameEpoch*

DefaultCase 2. of 2

Step 2.1. of 4

$\mathit{LL}!\mathit{Replica}(\mathit{primary})!\mathit{IAmPrimary}$

Reasoning (2.1.): No actions besides *BecomePrimary* change *IAmPrimary* to TRUE, so it must have stayed TRUE.

Step 2.2. of 4

$\mathit{LL}!\mathit{Replica}(\mathit{primary})!\mathit{CurView}.\mathit{viewInitiator}.\mathit{epoch} = \mathit{primary}.\mathit{epoch}$

Reasoning (2.2.): induction hypothesis

Step 2.3. of 4

$\neg(\exists\, m \in \mathit{VcInittedMsg} : \mathit{LL}!\mathit{Replica}(\mathit{primary})!\mathit{VcAck}(m))$

Reasoning (2.3.): *VcAck* action requires $\mathit{LL}!\mathit{Replica}(\mathit{primary})!\mathit{IAmPrimary}' = \text{FALSE}$

Step 2.4. of 4

UNCHANGED $\mathit{LL}!\mathit{Replica}(\mathit{primary})!\mathit{CurView}$

Reasoning (2.4.): inspection of remaining actions

Reasoning (2.): substitution

Reasoning: Proof by case analysis

Invariant *ProposedConstrainsViewMembership*

Hypotheses of *MembershipChangesAreBroadcast*

Hypotheses of *PrimaryAndViewInitiatorInSameEpoch*

Hypotheses of *MembershipAsDeterminesMembership*

Hypotheses of *ProposedImpliesMembershipAs*

Hypotheses of *NonconflictingViewMembershipsPrimed*

Introduce $\mathit{view} \in \mathit{ViewIds}$

Introduce $\mathit{opn} \in \mathit{Opns}$

Assume $\mathit{ProposedByAny}(\mathit{view},\ \mathit{opn}) \Rightarrow \mathit{ViewMembership}(\mathit{view}) = \mathit{Membership}(\mathit{opn})$

Assume $\mathit{ProposedByAny}(\mathit{view},\ \mathit{opn})'$

Prove $(\mathit{ViewMembership}(\mathit{view}) = \mathit{Membership}(\mathit{opn}))'$

Defn $\mathit{rec} \triangleq$

CHOOSE $rec \in [cohort : Cohorts,\ opv : CsOps]$:

$LL!Replica(rec.cohort)!ProposeAction(view,\ opn,\ rec.opv)$

Case 1. of 2

$LL!Replica(rec.cohort)!ProposeAction(view,\ opn,\ rec.opv)$

Step 1.1. of 8

$\land\ opn \in \text{DOMAIN}\ LL!Replica(rec.cohort)!CsState.membershipMap$
$\land\ rec.cohort \in LL!Replica(rec.cohort)!CsState.membershipMap[opn]$

Reasoning (1.1.): *Defn* Propose

Step 1.2. of 8

$\exists\ membership \in Memberships :$
$\quad \land\ opn \in \text{DOMAIN}\ LL!Replica(rec.cohort)!CsState.membershipMap$
$\quad \land\ membership = LL!Replica(rec.cohort)!CsState.membershipMap[opn]$
$\quad \land\ rec.cohort \in membership$

Reasoning (1.2.): logical rewrite

Step 1.3. of 8

$\exists\ membership \in Memberships :$
$\quad \land\ MembershipAs(opn,\ membership,\ LL!SentMessages)$
$\quad \land\ rec.cohort \in membership$

Reasoning (1.3.): Replace first two conjuncts using *Ref* hypothesis:*MembershipChangesAreBroadcast*

Defn $membership \triangleq$
CHOOSE $membership \in Memberships$:
$\quad \land\ MembershipAs(opn,\ membership,\ LL!SentMessages)$
$\quad \land\ rec.cohort \in membership$

Step 1.4. of 8

$EpochOf(membership) = view.viewInitiator.epoch$

Step 1.4.1. of 1

$EpochOf(membership) = rec.cohort.epoch$

Reasoning (1.4.1.): *Defn Memberships*; *Defn EpochOf*

Reasoning (1.4.): *Defn* Propose action gives *IAmPrimary*; *Ref* hypothesis:*PrimaryAndViewInitiatorInSameEpoch*

Step 1.5. of 8

$ViewMembership(view) = membership$

Reasoning (1.5.): *Defn ViewMembership*

Step 1.6. of 8

$MembershipAs(opn,\ membership,\ LL!SentMessages')$

Reasoning (1.6.): *Ref*:*MembershipAsMonotonic*

Step 1.7. of 8

$(Membership(opn)') = membership$

Reasoning (1.7.):*Ref*:*MembershipAsDeterminesMembership*

Step 1.8. of 8

$(ViewMembership(view)') = membership$

Reasoning (1.8.):*Ref*:*NonconflictingViewMembershipsPrimed*

Reasoning (1.): transitivity

DefaultCase 2. of 2

Step 2.1. of 5

$ProposedByAny(view, opn)$

Step 2.1.1. of 1

UNCHANGED $ProposedByAny(view, opn)$

Reasoning (2.1.1.): Case condition

Reasoning (2.1.): antecedent

Step 2.2. of 5

$ViewMembership(view) = Membership(opn)$

Reasoning (2.2.): induction hypothesis

Step 2.3. of 5

$MembershipAs(opn, Membership(opn), LL!SentMessages')$

Step 2.3.1. of 1

$MembershipAs(opn, Membership(opn), LL!SentMessages)$

Reasoning (2.3.1.):*Ref:ProposedImpliesMembershipAs*

Reasoning (2.3.): *Ref:MembershipAsMonotonic*

Step 2.4. of 5

$(Membership(opn)') = Membership(opn)$

Reasoning (2.4.):*Ref:MembershipAsDeterminesMembership*

Step 2.5. of 5

$(ViewMembership(view)') = Membership(opn)$

Reasoning (2.5.):*Ref:NonconflictingViewMembershipsPrimed*

Reasoning (2.): transitivity

Reasoning: case analysis

Theorem *QuorumPreparationPreventsConflictingProposal*

Hypotheses of $VcAckPreparedsReflectViewRecentPrepare$

Hypotheses of $ProposedsInSameViewDoNotConflict$

Hypotheses of $VcAckPreparedImpliesPrepared$

Hypotheses of $ProposalsRespectPrevPrepares$

Hypotheses of $ProposedImpliesElectingQuorum$

Hypotheses of $ProposedConstrainsViewMembership$

Hypotheses of $PreparedImpliesProposed$

Introduce $v1 \in ViewIds$

Introduce $v2 \in ViewIds$

Introduce $opn \in Opns$

Introduce $opv1 \in CsOps$

Introduce $opv2 \in CsOps$

Assume ($A1.$) $QuorumPreparedAs(v1, opn, opv1)$

Assume $v1 < v2$

Assume ($A2.$) $opv2 \neq opv1$

Prove $\neg ProposedByAnyAs(v2, opn, opv2)$

Summary: The proof of this theorem is the core of the *Paxos* proof. We proceed by contradiction: We are given one witness view $v1$ in which a quorum prepares opn as $\mathit{opv}1$, and a later view $v2$ in which a primary manages to propose opn as $\mathit{opv}2$. Once we have those witnesses, we know that there is some earliest view with such a conflicting proposal (which view this proof calls vm). The quorum that elected the primary of vm must have allowed the conflicting proposal, and the quorum that prepared $\mathit{opv}1$ in $v1$ should not have. We identify a spoiler cohort that belongs to both quorums (that's the point of a quorum), show that he must have quorum prepared in $v1$ before electing in vm, and show that he must have maintained and relayed to the primary of vm *preparedOps* info that would have precluded the conflicting proposal.

Step 1. of 1

Assume (1.A1.) $\mathit{ProposedByAnyAs}(v2,\ \mathit{opn},\ \mathit{opv}2)$

Prove FALSE

Defn $\mathit{InconsistentProposalView}(\mathit{vi}) \triangleq$

$\exists\, \mathit{opv}3 \in \mathit{CsOps} :$

$\land\ v1 < \mathit{vi}$

$\land\ \mathit{ProposedByAnyAs}(\mathit{vi},\ \mathit{opn},\ \mathit{opv}3)$

$\land\ \mathit{opv}3 \neq \mathit{opv}1$

Defn $\mathit{vm} \triangleq \mathit{Minimum}(\{v \in \mathit{ViewIds} : \mathit{InconsistentProposalView}(v)\})$

Step 1.1. of 10

$\mathit{InconsistentProposalView}(\mathit{vm})$

Reasoning (1.1.): True when $\mathit{vm} = v2$; maybe for some earlier view

Defn $\mathit{PreparingQuorum} \triangleq \{c \in \mathit{Cohorts} : \mathit{PreparedAs}(v1,\ c,\ \mathit{opn},\ \mathit{opv}1)\}$

Step 1.2. of 10

$\mathit{PreparingQuorum} \in \mathit{Quora}(\mathit{opn})$

Reasoning (1.2.): *Ref*:Assumption A1. ; def *QuorumPreparedAs*

Defn $\mathit{ElectionQuorum} \triangleq$

$\text{CHOOSE}\ \mathit{quorum} \in \mathit{Quora}(\mathit{opn}) : \mathit{PlausibleElectionQuorum}(\mathit{vm},\ \mathit{quorum})$

Step 1.3. of 10

$\mathit{PlausibleElectionQuorum}(\mathit{vm},\ \mathit{ElectionQuorum})$

Step 1.3.1. of 3

$\mathit{ProposedByAny}(\mathit{vm},\ \mathit{opn})$

Reasoning (1.3.1.): *Ref*:Step 1.1. ; *Defn InconsistentProposalView*

Step 1.3.2. of 3

$\exists\, \mathit{quorum} \in \mathit{QuoraOfMembership}(\mathit{ViewMembership}(\mathit{vm})) :$

$\mathit{PlausibleElectionQuorum}(\mathit{vm},\ \mathit{quorum})$

Reasoning (1.3.2.): *Ref* hypothesis:*ProposedImpliesElectingQuorum*

Step 1.3.3. of 3

$\mathit{QuoraOfMembership}(\mathit{ViewMembership}(\mathit{vm})) = \mathit{Quora}(\mathit{opn})$

Step 1.3.3.1. of 1

$\mathit{ViewMembership}(\mathit{vm}) = \mathit{Membership}(\mathit{opn})$

Reasoning (1.3.3.1.): *Ref* hypothesis:*ProposedConstrainsViewMembership*

Reasoning (1.3.3.): *Defn Quora*

Reasoning (1.3.): CHOOSE axiom

Defn $\mathit{spoiler} \triangleq \text{CHOOSE}\ c \in \mathit{PreparingQuorum} \cap \mathit{ElectionQuorum} : \text{TRUE}$

Step 1.4. of 10

$\mathit{spoiler} \in \mathit{PreparingQuorum} \cap \mathit{ElectionQuorum}$

Summary: The definition of *QuoraOfMembership* and the pigeon-hole principle ensure that the two quora overlap.

Step 1.4.1. of 1

Assume $PreparingQuorum \cap ElectionQuorum = \{\}$

Prove FALSE

Defn $bothQuora \triangleq PreparingQuorum \cup ElectionQuorum$

Step 1.4.1.1. of 2

$bothQuora \subseteq Membership(opn)$

Reasoning (1.4.1.1.): *Defn PreparingQuorum*; *Defn ElectionQuorum*; *Defn Quora*(*opn*); *Defn QuoraOfMembership*; *Defn* SUBSET ; property of ∪

Step 1.4.1.2. of 2

$Cardinality(bothQuora) > Cardinality(Membership(opn))$

Reasoning (1.4.1.2.): Since *bothQuora* is composed of a union of disjoint sets, its size is the sum of the sizes of the operands of the union. Defn *QuoraOfMembership* provides a minimum on the size of each operand.

Reasoning (1.4.1.): Strangely, the size of the set is bigger than the size of its superset.

Reasoning (1.4.): Proof by contradiction

Step 1.5. of 10

$VcAckedView(vm, spoiler)$

Reasoning (1.5.): defn *ElectionQuorum*,*PlausibleElectionQuorum*

Step 1.6. of 10

$PreparedAs(v1, spoiler, opn, opv1)$

Reasoning (1.6.): defn *PreparingQuorum*;*QuorumPrepared*

Step 1.7. of 10

$VcAckPreparedOpAs(vm, spoiler, opn, [view \mapsto v1, opv \mapsto opv1])$

Defn $ViewPreparesOpn(va) \triangleq$
$\exists\, opv4 \in Opns : va < vm \wedge PreparedAs(va, spoiler, opn, opv4)$

Defn $lastViewPreparingOpnBeforeConflict \triangleq$
$Maximum(\{v \in ViewIds : ViewPreparesOpn(v)\})$

Step 1.7.1. of 6

$ViewPreparesOpn(lastViewPreparingOpnBeforeConflict)$

Step 1.7.1.1. of 1

$\exists\, v \in ViewIds : ViewPreparesOpn(v)$

Reasoning (1.7.1.1.): assumption 1 provides a witness $ViewPreparesOpn(v1)$

Reasoning (1.7.1.): CHOOSE axiom

Step 1.7.2. of 6

$v1 \leq lastViewPreparingOpnBeforeConflict$

Reasoning (1.7.2.): $v1$ satisfies *ViewPreparesOpn*, and hence provides a lower bound for *Maximum*()

Defn $lastOpvPreparedBeforeConflict \triangleq$
CHOOSE $opv4 : PreparedAs(lastViewPreparingOpnBeforeConflict, spoiler, opn, opv4)$

Step 1.7.3. of 6

$PreparedAs($
$lastViewPreparingOpnBeforeConflict, spoiler, opn, lastOpvPreparedBeforeConflict)$

Reasoning (1.7.3.): CHOOSE axiom

Step 1.7.4. of 6

$VcAckPreparedOpAs($
$vm,$
$spoiler,$
$opn,$
$[$
$opv \mapsto lastOpvPreparedBeforeConflict,$
$view \mapsto lastViewPreparingOpnBeforeConflict$
$])$

Reasoning (1.7.4.): *Ref* hypothesis:*VcAckPreparedsReflectViewRecentPrepare* ($v1 = lastViewPreparingOpnBeforeConflict$, $v2 = vm$, $opv = lastOpvPreparedBeforeConflict$)

Step 1.7.5. of 6

$v1 \leq lastViewPreparingOpnBeforeConflict$

Reasoning (1.7.5.): assumption provides a witness to a max value of *lastViewPreparingOpnBeforeConflict*

Step 1.7.6. of 6

$lastOpvPreparedBeforeConflict = opv1$

Case 1.7.6.1. of 2

$v1 < lastViewPreparingOpnBeforeConflict$

Reasoning (1.7.6.1.): *Defn vm* requires *lastViewPreparingOpnBeforeConflict* to prepare $opv1$. (By contradiction: if it prepares something else, then *Ref* hypothesis:*PreparedImpliesProposed* requires it to be proposed there, which contradicts the definition of *vm* as the minimum view in which an *opv* other than $opv1$ was proposed.)

Case 1.7.6.2. of 2

$v1 = lastViewPreparingOpnBeforeConflict$

Step 1.7.6.2.1. of 2

$ProposedByAnyAs(v1, opn, lastOpvPreparedBeforeConflict)$

Reasoning (1.7.6.2.1.): *Ref* hypothesis:*PreparedImpliesProposed*

Step 1.7.6.2.2. of 2

$ProposedByAnyAs(v1, opn, opv1)$

Step 1.7.6.2.2.1. of 1

$\exists\, c \in Cohorts : PreparedAs(v1, c, opn, opv1)$

Reasoning (1.7.6.2.2.1.): *Ref*:Assumption *A*1. ;defn *QuorumPreparedAs*

Reasoning (1.7.6.2.2.): *Ref* hypothesis:*PreparedImpliesProposed*

Reasoning (1.7.6.2.): *Ref* hypothesis:*ProposedsInSameViewDoNotConflict*

Reasoning (1.7.6.): Proof by cases

Reasoning (1.7.): *Ref*:Step 1.7.4. ; *Ref*:Step 1.7.6.

Step 1.8. of 10

Introduce $c \in Cohorts$

Assume $VcAckedView(vm, c)$

Prove

$\lor\ ChooseVcAckPreparedOpInfo(vm, c, opn).opv = opv1$
$\lor\ ChooseVcAckPreparedOpInfo(vm, c, opn).view < v1$

Defn $\mathit{latestPreparedView} \triangleq \mathit{ChooseVcAckPreparedOpInfo}(\mathit{vm},\ c,\ \mathit{opn}).\mathit{view}$

Case 1.8.1. of 2

$\mathit{latestPreparedView} < v1$

Reasoning (1.8.1.): satisfies second disjunct of prove goal

Case 1.8.2. of 2

$v1 \leq \mathit{latestPreparedView}$

Step 1.8.2.1. of 5

$\mathit{Prepared}(\mathit{latestPreparedView},\ c,\ \mathit{opn})$

Reasoning (1.8.2.1.): *Ref* hypothesis: *VcAckPreparedImpliesPrepared*

Step 1.8.2.2. of 5

$\forall\ \mathit{vi} \in \mathit{ViewIds} : v1 < \mathit{vi} \land \mathit{vi} < v2 \Rightarrow (\neg \mathit{Prepared}(\mathit{vi},\ c,\ \mathit{opn}))$

Reasoning (1.8.2.2.): Contrapositive of *Ref* hypothesis: *VcAckPreparedsReflectViewRecentPrepare*

Step 1.8.2.3. of 5

Introduce $\mathit{opv}3 \in \mathit{ViewIds}$

Assume $\mathit{opv}3 \neq \mathit{opv}1$

Prove $\neg \mathit{PreparedAs}(\mathit{latestPreparedView},\ c,\ \mathit{opn},\ \mathit{opv}3)$

Step 1.8.2.3.1. of 1

Assume $\mathit{PreparedAs}(\mathit{latestPreparedView},\ c,\ \mathit{opn},\ \mathit{opv}3)$

Prove FALSE

Step 1.8.2.3.1.1. of 2

$\mathit{ProposedByAnyAs}(\mathit{latestPreparedView},\ \mathit{opn},\ \mathit{opv}3)$

Reasoning (1.8.2.3.1.1.): *Ref* hypothesis: *PreparedImpliesProposed*

Step 1.8.2.3.1.2. of 2

$\mathit{InconsistentProposalView}(\mathit{latestPreparedView})$

Reasoning (1.8.2.3.1.2.): *Defn InconsistentProposalView*

Reasoning (1.8.2.3.1.): *latestPreparedView* is a witness to the non-minimality of *vm*

Reasoning (1.8.2.3.): By contradiction

Step 1.8.2.4. of 5

$\mathit{PreparedAs}(\mathit{latestPreparedView},\ c,\ \mathit{opn},\ \mathit{opv}1)$

Reasoning (1.8.2.4.): *Defn Prepared* gives witness *opv*; only $\mathit{opv}1$ satisfies previous step

Step 1.8.2.5. of 5

$\mathit{ChooseVcAckPreparedOpInfo}(\mathit{vm},\ c,\ \mathit{opn}).\mathit{opv} = \mathit{opv}1$

Reasoning (1.8.2.5.): *Ref* hypothesis: *VcAckPreparedsReflectViewRecentPrepare*

Reasoning (1.8.2.): satisfies first disjunct of prove goal

Reasoning (1.8.): Proof by case analysis

Step 1.9. of 10

$\mathit{PrimaryDesignatedPrevPrep}(\mathit{vm},\ \mathit{opn},\ \mathit{opv}1)$

Reasoning (1.9.): No conflicting *VcAckPrevPreps* have views later than $v1$, so any conflict is dominated by $\mathit{VcAckPreparedOp}(\mathit{vm},\ \mathit{spoiler},\ \mathit{opn})$

Step 1.10. of 10

$\mathit{opv}1 = \mathit{opv}2$

Reasoning (1.10.): *Ref*:Assumption 1.$A1$. and *Ref*:Step 1.9. satisfy *Ref* hypothesis:*ProposalsRespectPrevPrepares*

Reasoning (1.): We have arrived at a contradiction with *Ref*:Assumption $A2$.

Reasoning: Proof by contradiction.

Theorem *QuorumPreparedImpliesProposed*

Hypotheses of $PreparedImpliesProposed$
Introduce $v \in ViewIds$
Introduce $opn \in Opns$
Introduce $opv \in CsOps$
Assume $QuorumPreparedAs(v, opn, opv)$
Prove $ProposedByAnyAs(v, opn, opv)$

Step 1. of 1
$\exists\, c \in Cohorts : PreparedAs(v, c, opn, opv)$
Reasoning (1.): *Defn QuorumPreparedAs*

Reasoning: *Ref* hypothesis:*PreparedImpliesProposed*

Theorem *NoConflictingQuorumPreparationInOrderedViews*

Hypotheses of $QuorumPreparedImpliesProposed$
Hypotheses of $QuorumPreparationPreventsConflictingProposal$
Hypotheses of $ProposedImpliesElectingQuorum$
Hypotheses of $ProposedConstrainsViewMembership$
Introduce $v1 \in ViewIds$
Introduce $v2 \in ViewIds$
Introduce $opn \in Opns$
Introduce $opv1 \in CsOps$
Introduce $opv2 \in CsOps$
Assume $v1 < v2$
Assume $QuorumPreparedAs(v1, opn, opv1)$
Assume $QuorumPreparedAs(v2, opn, opv2)$
Prove $opv1 = opv2$

Step 1. of 1
$ProposedByAnyAs(v2, opn, opv2)$
Reasoning (1.): *Ref*:*QuorumPreparedImpliesProposed*

Reasoning:*Ref*:*QuorumPreparationPreventsConflictingProposal* and some algebra

Theorem *NoConflictingQuorumPreparation*

Hypotheses of $\mathit{QuorumPreparedImpliesProposed}$

Hypotheses of $\mathit{ProposedsInSameViewDoNotConflict}$

Hypotheses of $\mathit{NoConflictingQuorumPreparationInOrderedViews}$

Introduce $v1 \in \mathit{ViewIds}$

Introduce $v2 \in \mathit{ViewIds}$

Introduce $\mathit{opn} \in \mathit{Opns}$

Introduce $\mathit{opv}1 \in \mathit{CsOps}$

Introduce $\mathit{opv}2 \in \mathit{CsOps}$

Assume $\mathit{QuorumPreparedAs}(v1, \mathit{opn}, \mathit{opv}1)$

Assume $\mathit{QuorumPreparedAs}(v2, \mathit{opn}, \mathit{opv}2)$

Prove $\mathit{opv}1 = \mathit{opv}2$

Case 1. of 3

$v1 < v2$

Reasoning (1.):*Ref*:$\mathit{NoConflictingQuorumPreparationInOrderedViews}(v1 = v1, v2 = v2)$

Case 2. of 3

$v1 = v2$

Step 2.1. of 2

$\mathit{ProposedByAnyAs}(v1, \mathit{opn}, \mathit{opv}1)$

Reasoning (2.1.):*Ref*:*QuorumPreparedImpliesProposed*

Step 2.2. of 2

$\mathit{ProposedByAnyAs}(v2, \mathit{opn}, \mathit{opv}2)$

Reasoning (2.2.):*Ref*:*QuorumPreparedImpliesProposed*

Reasoning (2.): *Ref* hypothesis:*ProposedsInSameViewDoNotConflict*

Case 3. of 3

$v1 > v2$

Reasoning (3.):*Ref*:$\mathit{NoConflictingQuorumPreparationInOrderedViews}(v1 = v2, v2 = v1)$

Reasoning: Proof by case analysis

Invariant *CommittedImpliesQuorumPrepared*

Hypotheses of $\mathit{LocalMembershipEpochOrdering}$

Introduce $\mathit{opn} \in \mathit{Opns}$

Introduce $\mathit{opv} \in \mathit{CsOps}$

Assume

$\mathit{CommittedByAnyAs}(\mathit{opn}, \mathit{opv}) \Rightarrow (\exists\, v \in \mathit{ViewIds} : \mathit{QuorumPreparedAs}(v, \mathit{opn}, \mathit{opv}))$

Assume $\mathit{CommittedByAnyAs}(\mathit{opn}, \mathit{opv})'$

Prove $(\exists\, v \in \mathit{ViewIds} : \mathit{QuorumPreparedAs}(v, \mathit{opn}, \mathit{opv}))'$

Defn $\mathit{rec} \triangleq$

$$\begin{array}{l} \textsc{choose}\ \mathit{rec} \in [\mathit{cohort} : \mathit{Cohorts}, \mathit{preparedMsgProto} : \mathit{PreparedMsg}] : \\ \quad \land\ \mathit{rec.preparedMsgProto.opn} = \mathit{opn} \\ \quad \land\ \mathit{LL}!\mathit{Replica}(\mathit{rec.cohort})!\mathit{Commit}(\mathit{rec.preparedMsgProto}) \end{array}$$

Case 1. of 2

$\land \mathit{rec.preparedMsgProto.opn} = \mathit{opn}$

$\land \mathit{LL}!\mathit{Replica}(\mathit{rec.cohort})!\mathit{Commit}(\mathit{rec.preparedMsgProto})$

Defn $\mathit{rec2} \triangleq$

CHOOSE

$\mathit{rec2} \in [\mathit{mSet} : \textsc{subset}\ \mathit{ConsensusMessage},\ \mathit{quorum} : \mathit{LL}!\mathit{Replica}(\mathit{rec.cohort})!\mathit{Quora}]$

:

$\land \mathit{LL}!\mathit{Replica}(\mathit{rec.cohort})!\mathit{ReceiveMessageSet}(\mathit{rec2.mSet})$

$\land \mathit{LL}!\mathit{Replica}(\mathit{rec.cohort})!\mathit{MessagesMatchPrototype}(\mathit{rec2.mSet},\ \mathit{rec.preparedMsgProto})$

$\land \mathit{LL}!\mathit{Replica}(\mathit{rec.cohort})!\mathit{EachCohortSentAMessage}(\mathit{rec2.quorum},\ \mathit{rec2.mSet})$

Step 1.1. of 3

$\land \mathit{LL}!\mathit{Replica}(\mathit{rec.cohort})!\mathit{ReceiveMessageSet}(\mathit{rec2.mSet})$

$\land \mathit{LL}!\mathit{Replica}(\mathit{rec.cohort})!\mathit{MessagesMatchPrototype}(\mathit{rec2.mSet},\ \mathit{rec.preparedMsgProto})$

$\land \mathit{LL}!\mathit{Replica}(\mathit{rec.cohort})!\mathit{EachCohortSentAMessage}(\mathit{rec2.quorum},\ \mathit{rec2.mSet})$

Reasoning (1.1.): *Defn Commit*; *Defn ReceiveFromQuorum*

Step 1.2. of 3

$\mathit{Quora}(\mathit{opn}) = \mathit{LL}!\mathit{Replica}(\mathit{rec.cohort})!\mathit{Quora}$

Step 1.2.1. of 1

$\mathit{Membership}(\mathit{opn}) = \mathit{LL}!\mathit{Replica}(\mathit{rec.cohort})!\mathit{Membership}$

Step 1.2.1.1. of 4

$\mathit{opn} \in \textsc{domain}\ \mathit{LL}!\mathit{Replica}(\mathit{rec.cohort})!\mathit{CsState.membershipMap}$

Reasoning (1.2.1.1.): *Defn Commit* implies *ActiveMember*

Step 1.2.1.2. of 4

$\mathit{MembershipAs}($
$\quad \mathit{opn},\ \mathit{LL}!\mathit{Replica}(\mathit{rec.cohort})!\mathit{CsState.membershipMap}[\mathit{opn}],\ \mathit{LL}!\mathit{SentMessages})$

Reasoning (1.2.1.2.): *Ref* hypothesis:*MembershipChangesAreBroadcast*

Step 1.2.1.3. of 4

$\mathit{Membership}(\mathit{opn}) = \mathit{LL}!\mathit{Replica}(\mathit{rec.cohort})!\mathit{CsState.membershipMap}[\mathit{opn}]$

Reasoning (1.2.1.3.): UNPRIMED version of Ref:*MembershipAsDeterminesMembership*

Step 1.2.1.4. of 4

$\mathit{LL}!\mathit{Replica}(\mathit{rec.cohort})!\mathit{CsState.membershipMap}[\mathit{opn}] =$
$\mathit{LL}!\mathit{Replica}(\mathit{rec.cohort})!\mathit{Membership}$

Step 1.2.1.4.1. of 2

$\mathit{rec.cohort} \in \mathit{LL}!\mathit{Replica}(\mathit{rec.cohort})!\mathit{CsState.membershipMap}[\mathit{opn}]$

Reasoning (1.2.1.4.1.): *ActiveMember*

Step 1.2.1.4.2. of 2

$\forall\, \mathit{opn2} \in \textsc{domain}\ \mathit{LL}!\mathit{Replica}(\mathit{rec.cohort})!\mathit{CsState.membershipMap} :$
$\quad \mathit{rec.cohort} \in \mathit{LL}!\mathit{Replica}(\mathit{rec.cohort})!\mathit{CsState.membershipMap} \Rightarrow$
$\quad \mathit{LL}!\mathit{Replica}(\mathit{rec.cohort})!\mathit{CsState.membershipMap}[\mathit{opn2}] =$
$\quad \mathit{LL}!\mathit{Replica}(\mathit{rec.cohort})!\mathit{CsState.membershipMap}[\mathit{opn}]$

Reasoning (1.2.1.4.2.): Any two memberships both containing *rec.cohort* must have the same epoch; with *Ref* hypothesis:*LocalMembershipEpochOrdering* , they must be the same membership.

Reasoning (1.2.1.4.): CHOOSE in *Defn* $LL!Replica(rec.cohort)!Membership$ is fully-constrained

Reasoning (1.2.1.): substitution

Reasoning (1.2.): *Defn* $Quora$; *Defn* $LL!Replica(rec.cohort)!Quora$

Step 1.3. of 3

Introduce $member \in rec2.quorum$

Prove $PreparedAs(rec.preparedMsgProto.view,\ member,\ opn,\ opv)$

Defn $memberMessage \triangleq \text{CHOOSE}\ m \in rec2.mSet : m.sender = member$

Step 1.3.1. of 3

$memberMessage.sender = member$

Reasoning (1.3.1.): *Defn* $EachCohortSentAMessage$;CHOOSE axiom

Step 1.3.2. of 3

$\land\ memberMessage.view = rec.preparedMsgProto.view$
$\land\ memberMessage.opn = opn$
$\land\ memberMessage.opv = opv$

Reasoning (1.3.2.): *Defn* $MessagesMatchPrototype$

Step 1.3.3. of 3

$memberMessage \in SentMessages$

Reasoning (1.3.3.): *Defn* $ReceiveMessageSet$

Reasoning (1.3.): *Defn* $PreparedAs$

Reasoning (1.): $rec2.quorum$ is witness to $QuorumPreparedAs.quorum$, and $rec.preparedMsgProto.view$ is witness to v in proof obligation.

DefaultCase 2. of 2

Step 2.1. of 2

$CommittedByAnyAs(opn,\ opv)$

Step 2.1.1. of 1

UNCHANGED $CommittedByAnyAs(opn,\ opv)$

Reasoning (2.1.1.): $CommittedByAnyAs$ only changes when a $CommittedMsg$ is sent; Inspection of actions shows that only happens in Case I.

Reasoning (2.1.): Assumption; *Defn* UNCHANGED

Step 2.2. of 2

$\exists\, v \in ViewIds : QuorumPreparedAs(v,\ opn,\ opv)$

Reasoning (2.2.): induction hypothesis

Reasoning (2.): *Ref*:$QuorumPreparedAsMonotonic$

Reasoning: Proof by case analysis

Theorem $NoConflictingCommits$

Hypotheses of $CommittedImpliesQuorumPrepared$

Hypotheses of $NoConflictingQuorumPreparation$

Introduce $opn \in Opns$

Introduce $opv1 \in CsOps$

Introduce $opv2 \in CsOps$

Assume $CommittedByAnyAs(opn, opv1)$

Assume $CommittedByAnyAs(opn, opv2)$

Prove $opv1 = opv2$

Summary: We push the problem back from when the operations were Committed to when they were *QuorumPrepared*, and invoke *NoConflictingQuorumPreparation*.

Defn $QuorumPreparedViews(opv) \triangleq \{v \in ViewIds : QuorumPreparedAs(v, opn, opv)\}$

Defn $v1 \triangleq \text{CHOOSE } v1 : v1 \in QuorumPreparedViews(opv1)$

Defn $v2 \triangleq \text{CHOOSE } v2 : v2 \in QuorumPreparedViews(opv2)$

Step 1. of 2

$v1 \in QuorumPreparedViews(opv1)$

Reasoning (1.): *Ref* hypothesis:*CommittedImpliesQuorumPrepared*

Step 2. of 2

$v2 \in QuorumPreparedViews(opv2)$

Reasoning (2.): *Ref* hypothesis:*CommittedImpliesQuorumPrepared*

Reasoning: Apply *Ref*:*NoConflictingQuorumPreparation*

Theorem *KnownOpvFcnExtended*

Hypotheses of *NoConflictingCommits*

$FcnExtends(KnownOpv', KnownOpv)$

Step 1. of 2

$\text{DOMAIN } KnownOpv \subseteq \text{DOMAIN } (KnownOpv')$

Reasoning (1.): *Ref*:*MaxKnownOpnGrows*

Step 2. of 2

Introduce $opn \in \text{DOMAIN } KnownOpv$

$KnownOpv[opn] = (KnownOpv')[opn]$

Summary: If an operation was known in the previous step, we'll commit no conflicting operations in this step, so we can be sure that *KnownOpv′* makes the same operation assignment.

Step 2.1. of 3

$CommittedByAnyAs(opn, KnownOpv[opn])$

Reasoning (2.1.): *Defn KnownOpv*; *Defn MaxKnownOpn*

Step 2.2. of 3

$CommittedByAnyAs(opn, KnownOpv[opn])'$

Reasoning (2.2.): *Ref*:*CommittedMonotonic*

Step 2.3. of 3

$\forall\, opv \in CsOps : (CommittedByAnyAs(opn, opv)') \Rightarrow opv = KnownOpv[opn]$

Reasoning (2.3.): *Ref*:*NoConflictingCommits*

Reasoning (2.): *Defn KnownOpv*; CHOOSE constrained to singleton set

Reasoning: *Defn FcnExtends*

Theorem *KnownStateFcnExtended*

Hypotheses of $KnownOpvFcnExtended$

Prove $FcnExtends(KnownState', KnownState)$

Step 1. of 2

$\textsc{domain}\ KnownState \subseteq \textsc{domain}\ (KnownState')$

Reasoning (1.): *Ref:MaxKnownOpnGrows*

Step 2. of 2

Introduce $opn \in \textsc{domain}\ KnownState$

Prove $KnownState[opn] = (KnownState')[opn]$

Step 2.1. of 2

Prove $KnownState[0] = (KnownState')[0]$

Reasoning (2.1.): $CsInit = CsInit$

Step 2.2. of 2

Assume $0 < opn$

Assume $KnownState[(opn - 1)] = (KnownState')[(opn - 1)]$

Prove $KnownState[opn] = (KnownState')[opn]$

Step 2.2.1. of 1

$KnownOpv[opn] = (KnownOpv')[opn]$

Step 2.2.1.1. of 1

$opn \in \textsc{domain}\ KnownOpv$

Reasoning (2.2.1.1.): *Defn KnownOpv*

Reasoning (2.2.1.): *Ref:KnownOpvFcnExtended*

Reasoning (2.2.): Substitution of equal terms in ELSE clause of *Defn KnownState*.

Reasoning (2.): Proof by induction on *opn*

Reasoning: *Defn FcnExtends*

Invariant *LocalStateConsonantWithKnownState*

Hypotheses of $NoConflictingCommits$

Introduce $cohort \in Cohorts$

Defn $state \triangleq LL!Replica(cohort)!CsState$

Defn $snapshot \triangleq LL!Replica(cohort)!CsStateSnapshot$

Assume

$\land\ Consonant(state)$

$\land\ Consonant(snapshot)$

Prove

$(\ \land\ Consonant(state)$

$\ \ \land\ Consonant(snapshot))'$

Summary: We first establish lemmas showing that if either state or snapshot doesn't change, the corresponding variable holds its consonance. With those lemmas, we can charge through a case analysis of the three actions that touch state or snapshot.

Step 1. of 6

Assume UNCHANGED *state*

Prove $Consonant(state)'$

Step 1.1. of 1

$\land (state') \in Range(KnownState)$

$\land (state') = KnownState[state'.numExecuted]$

Reasoning (1.1.): UNCHANGED assumption; induction hypothesis

Reasoning (1.): *Ref*:*KnownStateFcnExtended* ; *Defn Consonant*

Step 2. of 6

Assume UNCHANGED *snapshot*

Prove $Consonant(snapshot)'$

Step 2.1. of 1

$\land (snapshot') \in Range(KnownState)$

$\land (snapshot') = KnownState[snapshot'.numExecuted]$

Reasoning (2.1.): UNCHANGED assumption; induction hypothesis

Reasoning (2.): *Ref*:*KnownStateFcnExtended* ; *Defn Consonant*

Case 3. of 6

$LL!Replica(cohort)!Persist$

Summary: The state is UNCHANGED by *Persist*; the snapshot part relies on the consonance of state in the prior state.

Step 3.1. of 2

$Consonant(state)'$

Step 3.1.1. of 1

UNCHANGED *state*

Reasoning (3.1.1.): *Defn Crash* action

Reasoning (3.1.): *Ref*:Step 1.

Step 3.2. of 2

$Consonant(snapshot)'$

Step 3.2.1. of 2

$(snapshot') = state$

Reasoning (3.2.1.): *Defn Persist* action

Step 3.2.2. of 2

$\land (snapshot') \in Range(KnownState)$

$\land (snapshot') = KnownState[snapshot'.numExecuted]$

Reasoning (3.2.2.): UNCHANGED assumption;induction hypothesis

Reasoning (3.2.): *Ref*:*KnownStateFcnExtended* ; *Defn Consonant*

Reasoning (3.): We've shown both conjuncts of the proof goal

Case 4. of 6

$LL!Replica(cohort)!Crash$

Summary: This case is the mirror of the previous. The snapshot is UNCHANGED by a *Crash*; the state part relies on the consonance of snapshot in the prior state.

Step 4.1. of 2

$Consonant(snapshot)'$

Step 4.1.1. of 1

UNCHANGED *snapshot*

Reasoning (4.1.1.): *Defn Crash* action

Reasoning (4.1.): *Ref*:Step 2.

Step 4.2. of 2

$\mathit{Consonant}(\mathit{state})'$

Step 4.2.1. of 2

$(\mathit{state}') = \mathit{snapshot}$

Reasoning (4.2.1.): *Defn Crash* action

Step 4.2.2. of 2

$\land\ (\mathit{state}') \in \mathit{Range}(\mathit{KnownState})$

$\land\ (\mathit{state}') = \mathit{KnownState}[\mathit{state}'.\mathit{numExecuted}]$

Reasoning (4.2.2.): induction hypothesis

Reasoning (4.2.): *Ref*:*KnownStateFcnExtended* ; *Defn Consonant*

Reasoning (4.): We've shown both conjuncts of the proof goal

Case 5. of 6

$\exists\, m \in \mathit{CommittedMsg} : \mathit{LL}!\mathit{Replica}(\mathit{cohort})!\mathit{Execute}(m)$

Defn $\quad m \triangleq \textsc{choose}\ m \in \mathit{CommittedMsg} : \mathit{LL}!\mathit{Replica}(\mathit{cohort})!\mathit{Execute}(m)$

Step 5.1. of 2

$\mathit{Consonant}(\mathit{snapshot})'$

Step 5.1.1. of 1

$\textsc{unchanged}\ \mathit{snapshot}$

Reasoning (5.1.1.): *Defn Crash* action

Reasoning (5.1.): *Ref*:Step 2.

Step 5.2. of 2

$\mathit{Consonant}(\mathit{state})'$

Step 5.2.1. of 2

$\mathit{state}'.\mathit{numExecuted} \in \textsc{domain}\ \mathit{KnownState}$

Step 5.2.1.1. of 2

$\mathit{state}.\mathit{numExecuted} + 1 \leq \mathit{MaxKnownOpn}$

Step 5.2.1.1.1. of 2

$\forall\, \mathit{opn} \in 1 \,..\, \mathit{state}.\mathit{numExecuted} : \mathit{CommittedByAny}(\mathit{opn})$

Step 5.2.1.1.1.1. of 1

$\mathit{state}.\mathit{numExecuted} \leq \mathit{MaxKnownOpn}$

Step 5.2.1.1.1.1.1. of 1

$\mathit{state}.\mathit{numExecuted} \in \textsc{domain}\ \mathit{KnownState}$

Reasoning (5.2.1.1.1.1.1.): induction hypothesis ;*Defn Consonant*

Reasoning (5.2.1.1.1.1.): *Defn KnownState*

Reasoning (5.2.1.1.1.): *Defn MaxKnownOpn*

Step 5.2.1.1.2. of 2

$\mathit{CommittedByAny}(\mathit{state}.\mathit{numExecuted} + 1)$

Step 5.2.1.1.2.1. of 1

$\mathit{CommittedByAnyAs}(m.\mathit{opn},\ m.\mathit{opv})$

Reasoning (5.2.1.1.2.1.): According to *Defn Commit*, Message m is a witness

Reasoning (5.2.1.1.2.): *Defn Execute* action

Reasoning (5.2.1.1.): *Defn MaxKnownOpn*

Step 5.2.1.2. of 2

$state.numExecuted + 1 \in \textsc{domain}\ KnownState$

Reasoning (5.2.1.2.): *Defn KnownState*

Reasoning (5.2.1.): *Defn CsTx*

Step 5.2.2. of 2

$(state') = KnownState[state'.numExecuted]$

Step 5.2.2.1. of 4

$KnownState[(state'.numExecuted - 1)] = state$

Step 5.2.2.1.1. of 2

$state = KnownState[state.numExecuted]$

Reasoning (5.2.2.1.1.): induction hypothesis ;*Defn Consonant*

Step 5.2.2.1.2. of 2

$state.numExecuted = state'.numExecuted - 1$

Reasoning (5.2.2.1.2.): *Defn Execute*; *Defn CsTx*

Reasoning (5.2.2.1.): Substitution

Step 5.2.2.2. of 4

$KnownOpv[state'.numExecuted] = m.opv$

Step 5.2.2.2.1. of 2

$CommittedByAnyAs(m.opn,\ m.opv)$

Reasoning (5.2.2.2.1.): m is a witness

Step 5.2.2.2.2. of 2

$\forall\ opv \in CsOps : CommittedByAnyAs(m.opn,\ opv) \Rightarrow opv = m.opv$

Reasoning (5.2.2.2.2.): *Ref*:*NoConflictingCommits*

Reasoning (5.2.2.2.): CHOOSE in *Defn KnownOpv* is fully constrained.

Step 5.2.2.3. of 4

$KnownState[state'.numExecuted] = CsTx[state,\ m.opv]$

Reasoning (5.2.2.3.): *Defn KnownState*; *Ref*:Step 5.2.2.1. ; *Ref*:Step 5.2.2.2.

Step 5.2.2.4. of 4

$(state') = CsTx[state,\ m.opv]$

Reasoning (5.2.2.4.): *Defn Execute* action

Reasoning (5.2.2.): Substitution

Reasoning (5.2.): *Defn Consonant*

Reasoning (5.): We've shown both conjuncts of the proof goal

DefaultCase 6. of 6

Step 6.1. of 1

$\land\ \textsc{unchanged}\ state$

$\land\ \textsc{unchanged}\ snapshot$

Reasoning (6.1.): All other actions leave UNCHANGED *CsState* and *CsStateSnapshot*.

Reasoning (6.): *Ref*:Step 1. ; *Ref*:Step 2.

Reasoning: Proof by case analysis

Invariant *BroadcastMembershipsReflectKnownState*

Hypotheses of $\mathit{LocalStateConsonantWithKnownState}$

Introduce $\mathit{opn} \in \mathit{Opns}$

Introduce $\mathit{membership} \in \mathit{Memberships}$

Assume

$\land\ \mathit{Alpha} < \mathit{opn}$

$\land\ \mathit{MembershipAs}(\mathit{opn},\ \mathit{membership},\ \mathit{LL}!\mathit{SentMessages})$

$\Rightarrow$

$\land\ \mathit{opn} - \mathit{Alpha} \in \textsc{domain}\ \mathit{KnownState}$

$\land\ \mathit{opn} \in \textsc{domain}\ \mathit{KnownState}[(\mathit{opn} - \mathit{Alpha})].\mathit{membershipMap}$

$\land\ \mathit{KnownState}[(\mathit{opn} - \mathit{Alpha})].\mathit{membershipMap}[\mathit{opn}] = \mathit{membership}$

Assume

$(\land\ \mathit{Alpha} < \mathit{opn}$

$\land\ \mathit{MembershipAs}(\mathit{opn},\ \mathit{membership},\ \mathit{LL}!\mathit{SentMessages}))'$

Prove

$(\land\ \mathit{opn} - \mathit{Alpha} \in \textsc{domain}\ \mathit{KnownState}$

$\land\ \mathit{opn} \in \textsc{domain}\ \mathit{KnownState}[(\mathit{opn} - \mathit{Alpha})].\mathit{membershipMap}$

$\land\ \mathit{KnownState}[(\mathit{opn} - \mathit{Alpha})].\mathit{membershipMap}[\mathit{opn}] = \mathit{membership})'$

Case 1. of 2

$\exists\ \mathit{membershipMsg} \in \mathit{MembershipMsg}:$

$\land\ \mathit{membershipMsg}.\mathit{opn} = \mathit{opn}$

$\land\ \mathit{membershipMsg}.\mathit{membership} = \mathit{membership}$

$\land\ \mathit{membershipMsg} \notin \mathit{SentMessages}$

$\land\ \mathit{membershipMsg} \in (\mathit{SentMessages}')$

Defn $\mathit{rec} \triangleq$

$\textsc{choose}\ \mathit{rec} \in [\mathit{cohort} : \mathit{Cohorts},\ \mathit{commitMsg} : \mathit{CommittedMsg}] :$

$\land\ \mathit{LL}!\mathit{Replica}(\mathit{rec}.\mathit{cohort})!\mathit{Execute}(\mathit{rec}.\mathit{commitMsg})$

$\land\ \mathit{rec}.\mathit{commitMsg}.\mathit{opn} = \mathit{opn} - \mathit{Alpha}$

Step 1.1. of 6

$\land\ \mathit{LL}!\mathit{Replica}(\mathit{rec}.\mathit{cohort})!\mathit{Execute}(\mathit{rec}.\mathit{commitMsg})$

$\land\ \mathit{rec}.\mathit{commitMsg}.\mathit{opn} = \mathit{opn} - \mathit{Alpha}$

Reasoning (1.1.): Only such an *Execute* action sends a *MembershipMessage* matching the Case condition.

Step 1.2. of 6

$\land\ \mathit{rec}.\mathit{commitMsg}.\mathit{opn} \in \textsc{domain}\ \mathit{KnownState}$

$\land\ (\mathit{LL}!\mathit{Replica}(\mathit{rec}.\mathit{cohort})!\mathit{CsState}') = \mathit{KnownState}[\mathit{rec}.\mathit{commitMsg}.\mathit{opn}]$

Step 1.2.1. of 3

$\mathit{rec}.\mathit{commitMsg}.\mathit{opn} = \mathit{LL}!\mathit{Replica}(\mathit{rec}.\mathit{cohort})!\mathit{CsState}'.\mathit{numExecuted}$

Reasoning (1.2.1.): *Defn Execute*; *Defn CsTx*

Step 1.2.2. of 3

$\land\ \mathit{LL}!\mathit{Replica}(\mathit{rec}.\mathit{cohort})!\mathit{CsState}'.\mathit{numExecuted} \in \textsc{domain}\ (\mathit{KnownState}')$

$\land\ (\mathit{LL}!\mathit{Replica}(\mathit{rec}.\mathit{cohort})!\mathit{CsState}') =$

$(\mathit{KnownState}')[\mathit{LL}!\mathit{Replica}(\mathit{rec}.\mathit{cohort})!\mathit{CsState}'.\mathit{numExecuted}]$

Reasoning (1.2.2.):*Ref:LocalStateConsonantWithKnownState* ; *Defn Consonant*

Step 1.2.3. of 3

UNCHANGED $KnownState$

Reasoning (1.2.3.): *Execute* action sends no *CommittedMsgs*; *KnownState* only varies over $SentMessages \cap CommittedMsgs$.

Reasoning (1.2.): substitution

Step 1.3. of 6

$opn \in \textsc{domain}\ LL!Replica(rec.cohort)!CsState'.membershipMap$

Reasoning (1.3.): *Defn Execute*; *Defn CsTx*

Step 1.4. of 6

$LL!Replica(rec.cohort)!CsState'.membershipMap[opn] = membership$

Reasoning (1.4.): This action was responsible for sending *membershipMsg* (*Defn SendMessage*), and *Defn* Execute constrains what message we send to match the membership in $CsState'$.

Step 1.5. of 6

$opn \in \textsc{domain}\ KnownState[(opn - Alpha)].membershipMap$

Reasoning (1.5.): Substitute into *Ref*:Step 1.3. second conjunct of *Ref*:Step 1.2. ; substitute in *opn*-Alpha from second conjunct of *Ref*:Step 1.1. .

Step 1.6. of 6

$KnownState[(opn - Alpha)].membershipMap[opn] = membership$

Step 1.6.1. of 2

$KnownState[rec.commitMsg.opn].membershipMap[opn] =$
$LL!Replica(rec.cohort)!CsState'.membershipMap[opn]$

Reasoning (1.6.1.): *Ref*:Step 1.2.

Step 1.6.2. of 2

$KnownState[rec.commitMsg.opn].membershipMap[opn] = membership$

Reasoning (1.6.2.): *Ref*:Step 1.4.

Reasoning (1.6.): substitute in *opn*-Alpha from second conjunct of *Ref*:Step 1.1. .

Reasoning (1.): We've satisfied each required conjunct.

DefaultCase 2. of 2

Step 2.1. of 1

UNCHANGED $MembershipAs(opn, membership, LL!SentMessages)$

Reasoning (2.1.): Case condition

Reasoning (2.): induction hypothesis ;*Ref*:*KnownStateFcnExtended*

Reasoning: Proof by case analysis