Verifying Fault-Tolerant Distributed Algorithms In The Heard-Of Model∗

Henri Debrat1 and Stephan Merz2
1 Université de Lorraine & LORIA
2 Inria Nancy Grand-Est & LORIA
Villers-lès-Nancy, France
August 16, 2018

Distributed computing is inherently based on replication, promising increased tolerance to failures of individual computing nodes or communication channels. Realizing this promise, however, involves quite subtle algorithmic mechanisms, and requires precise statements about the kinds and numbers of faults that an algorithm tolerates (such as process crashes, communication faults or corrupted values). The landmark theorem due to Fischer, Lynch, and Paterson shows that it is impossible to achieve Consensus among $N$ asynchronously communicating nodes in the presence of even a single permanent failure. Existing solutions must rely on assumptions of “partial synchrony”.

Indeed, there have been numerous misunderstandings on what exactly a given algorithm is supposed to realize in what kinds of environments. Moreover, the abundance of subtly different computational models complicates comparisons between different algorithms. Charron-Bost and Schiper introduced the Heard-Of model for representing algorithms and failure assumptions in a uniform framework, simplifying comparisons between algorithms. In this contribution, we represent the Heard-Of model in Isabelle/HOL. We define two semantics of runs of algorithms with different unit of atomicity and relate these through a reduction theorem that allows us to verify algorithms in the coarse-grained semantics (where proofs are easier) and infer their correctness for the fine-grained one (which corresponds to actual executions). We instantiate the framework by verifying six Consensus algorithms that differ in the underlying algorithmic mechanisms and the kinds of faults they tolerate.

∗Bernadette Charron-Bost introduced us to the Heard-Of model and accompanied this work by suggesting algorithms to study, providing or simplifying hand proofs, and giving most valuable feedback on our formalizations. Mouna Chaouch-Saad contributed an initial draft formalization of the reduction theorem.
Contents

1 Introduction 4

2 Heard-Of Algorithms 5
  2.1 The Consensus Problem .......................... 5
  2.2 A Generic Representation of Heard-Of Algorithms .... 7

3 Reduction Theorem 13
  3.1 Fine-Grained Semantics .......................... 14
  3.2 Properties of the Fine-Grained Semantics .............. 17
  3.3 From Fine-Grained to Coarse-Grained Runs .......... 24
  3.4 Locally Similar Runs and Local Properties .......... 26
  3.5 Consensus as a Local Property ........................ 28

4 Utility Lemmas About Majorities 30

5 Verification of the One-Third Rule Consensus Algorithm 31
  5.1 Model of the Algorithm .......................... 31
  5.2 Communication Predicate for One-Third Rule ......... 33
  5.3 The One-Third Rule Heard-Of Machine ............... 34
  5.4 Proof of Integrity ............................... 34
  5.5 Proof of Agreement ............................... 37
  5.6 Proof of Termination ............................... 41
  5.7 One-Third Rule Solves Consensus ...................... 43

6 Verification of the UniformVoting Consensus Algorithm 44
  6.1 Model of the Algorithm .......................... 44
  6.2 Communication Predicate for UniformVoting ........... 47
  6.3 The UniformVoting Heard-Of Machine ................ 47
  6.4 Preliminary Lemmas ............................... 47
  6.5 Proof of Irrevocability, Agreement and Integrity ...... 50
  6.6 Proof of Termination ............................... 57
  6.7 UniformVoting Solves Consensus ...................... 60

7 Verification of the LastVoting Consensus Algorithm 61
  7.1 Model of the Algorithm .......................... 61
  7.2 Communication Predicate for LastVoting ............... 65
  7.3 The LastVoting Heard-Of Machine ...................... 65
1 Introduction

We are interested in the verification of fault-tolerant distributed algorithms. The archetypical problem in this area is the Consensus problem that requires a set of distributed nodes to achieve agreement on a common value in the presence of faults. Such algorithms are notoriously hard to design and to get right. This is particularly true in the presence of asynchronous communication: the landmark theorem by Fischer, Lynch, and Paterson [9] shows that there is no algorithm solving the Consensus problem for asynchronous systems in the presence of even a single, permanent fault. Existing solutions therefore rely on assumptions of “partial synchrony” [8].

Different computational models, and different concepts for specifying the kinds and numbers of faults such algorithms must tolerate, have been introduced in the literature on distributed computing. This abundance of subtly different notions makes it very difficult to compare different algorithms, and has sometimes even led to misunderstandings and misinterpretations of what an algorithm claims to achieve. The general lack of rigorous, let alone formal, correctness proofs for this class of algorithms makes it even harder to understand the field.

In this contribution, we formalize in Isabelle/HOL the Heard-Of (HO) model, originally introduced by Charron-Bost and Schiper [7]. This model can represent algorithms that operate in communication-closed rounds, which is true of virtually all known fault-tolerant distributed algorithms. Assumptions on failures tolerated by an algorithm are expressed by communication predicates that impose bounds on the set of messages that are not received during executions. Charron-Bost and Schiper show how the known failure hypotheses from the literature can be represented in this format. The Heard-Of model therefore makes an interesting target for formalizing different algorithms, and for proving their correctness, in a uniform way. In particular, different assumptions can be compared, and the suitability of an algorithm for a particular situation can be evaluated.

The HO model has subsequently been extended [3] to encompass algorithms designed to tolerate value (also known as malicious or Byzantine) faults. In the present work, we propose a generic framework in Isabelle/HOL that encompasses the different variants of HO algorithms, including resilience to benign or value faults, as well as coordinated and non-coordinated algorithms.

A fundamental design decision when modeling distributed algorithm is to determine the unit of atomicity. We formally relate in Isabelle two definitions of runs: we first define “coarse-grained” executions, in which entire rounds are executed atomically, and then define “fine-grained” executions that correspond to conventional interleaving representations of asynchronous networks. We formally prove that every fine-grained execution corresponds
to a certain coarse-grained execution, such that every process observes the same sequence of local states in the two executions, up to stuttering. As a corollary, a large class of correctness properties, including Consensus, can be transferred from coarse-grained to fine-grained executions.

We then apply our framework for verifying six different distributed Consensus algorithms w.r.t. their respective communication predicates. The first three algorithms, One-Third Rule, UniformVoting, and LastVoting, tolerate benign failures. The three remaining algorithms, $\mathcal{U}_{T,E,\alpha}$, $\mathcal{A}_{T,E,\alpha}$, and EIG-Byz$_f$, are designed to tolerate value failures, and solve a weaker variant of the Consensus problem.


\begin{verbatim}
theory HOModel
imports Main
begin

declare if-split-asm [split] — perform default perform case splitting on conditionals

2 Heard-Of Algorithms

2.1 The Consensus Problem

We are interested in the verification of fault-tolerant distributed algorithms. The Consensus problem is paradigmatic in this area. Stated informally, it assumes that all processes participating in the algorithm initially propose some value, and that they may at some point decide some value. It is required that every process eventually decides, and that all processes must decide the same value.

More formally, we represent runs of algorithms as $\omega$-sequences of configurations (vectors of process states). Hence, a run is modeled as a function of type $\text{n}at \Rightarrow \text{proc} \Rightarrow \text{pst}$ where type variables $\text{proc}$ and $\text{pst}$ represent types of processes and process states, respectively. The Consensus property is expressed with respect to a collection $vals$ of initially proposed values (one per process) and an observer function $\text{dec}::\text{pst} \Rightarrow \text{val option}$ that retrieves the decision (if any) from a process state. The Consensus problem is stated as the conjunction of the following properties:

**Integrity.** Processes can only decide initially proposed values.

**Agreement.** Whenever processes $p$ and $q$ decide, their decision values must be the same. (In particular, process $p$ may never change the value it
decides, which is referred to as Irrevocability.)

**Termination.** Every process decides eventually.

The above properties are sometimes only required of non-faulty processes, since nothing can be required of a faulty process. The Heard-Of model does not attribute faults to processes, and therefore the above formulation is appropriate in this framework.

**type-synonym**

\[(\texttt{proc, pst}) \text{ run} = \text{nat} \Rightarrow \texttt{proc} \Rightarrow \texttt{pst}\]

**definition**

\[\text{consensus} :: (\texttt{proc} \Rightarrow \texttt{val}) \Rightarrow (\texttt{pst} \Rightarrow \texttt{val} \text{ option}) \Rightarrow (\texttt{proc, pst}) \text{ run} \Rightarrow \text{bool}\]

**where**

\[\text{consensus} \text{ vals} \text{ dec rho} \equiv \]

\[(\forall n p v. \text{ dec (rho n p)} = \text{Some v} \rightarrow v \in \text{range vals})\]

\[\land (\forall m n p q v w. \text{ dec (rho m p)} = \text{Some v} \land \text{ dec (rho n q)} = \text{Some w} \rightarrow v = w)\]

\[\land (\forall p. \exists n. \text{ dec (rho n p)} \neq \text{None})\]

A variant of the Consensus problem replaces the Integrity requirement by

**Validity.** If all processes initially propose the same value \(v\) then every process may only decide \(v\).

**definition**

\[\text{weak-consensus} \text{ where}\]

\[\text{weak-consensus} \text{ vals} \text{ dec rho} \equiv\]

\[(\forall v. (\forall p. \text{vals p} = v) \rightarrow (\forall n p w. \text{ dec (rho n p)} = \text{Some w} \rightarrow w = v))\]

\[\land (\forall m n p q v w. \text{ dec (rho m p)} = \text{Some v} \land \text{ dec (rho n q)} = \text{Some w} \rightarrow v = w)\]

\[\land (\forall p. \exists n. \text{ dec (rho n p)} \neq \text{None})\]

Clearly, \text{consensus} implies \text{weak-consensus}.

**lemma** \text{consensus-then-weak-consensus}:

**assumes** \text{consensus} \text{ vals} \text{ dec rho}

**shows** \text{weak-consensus} \text{ vals} \text{ dec rho}

**using** \text{assms by} (auto simp: consensus-def weak-consensus-def image-def)

Over Boolean values ("binary Consensus"), \text{weak-consensus} implies \text{consensus}, hence the two problems are equivalent. In fact, this theorem holds more generally whenever at most two different values are proposed initially (i.e., \text{card} (\text{range vals}) \leq 2).

**lemma** \text{binary-weak-consensus-then-consensus}:

**assumes** \text{bc: weak-consensus (vals::'proc \Rightarrow bool) dec rho}

**shows** \text{consensus} \text{ vals} \text{ dec rho}

**proof**

\{- Show the Integrity property, the other conjuncts are the same. \}

\[\text{fix} n p v \]
assume \( \text{dec} : \text{dec} (\rho n p) = \text{Some} v \)

have \( v \in \text{range vals} \)

proof (cases \( \exists w. \forall p. \text{vals} p = w \))
  case True
  then obtain \( w \) where \( w : \forall p. \text{vals} p = w \)

next
  case False
  — In this case both possible values occur in \( \text{vals} \), and the result is trivial.
  thus \( \text{thesis} \) by (auto simp: weak-consensus-def)

qed

2.2 A Generic Representation of Heard-Of Algorithms

Charron-Bost and Schiper [7] introduce the Heard-Of (HO) model for representing fault-tolerant distributed algorithms. In this model, algorithms execute in communication-closed rounds: at any round \( r \), processes only receive messages that were sent for that round. For every process \( p \) and round \( r \), the “heard-of set” \( \text{HO}(p, r) \) denotes the set of processes from which \( p \) receives a message in round \( r \). Since every process is assumed to send a message to all processes in each round, the complement of \( \text{HO}(p, r) \) represents the set of faults that may affect \( p \) in round \( r \) (messages that were not received, e.g. because the sender crashed, because of a network problem etc.).

The HO model expresses hypotheses on the faults tolerated by an algorithm through “communication predicates” that constrain the sets \( \text{HO}(p, r) \) that may occur during an execution. Charron-Bost and Schiper show that standard fault models can be represented in this form.

The original HO model is sufficient for representing algorithms tolerating benign failures such as process crashes or message loss. A later extension for algorithms tolerating Byzantine (or value) failures [3] adds a second collection of sets \( \text{SHO}(p, r) \subseteq \text{HO}(p, r) \) that contain those processes \( q \) from which process \( p \) receives the message that \( q \) was indeed supposed to send for round \( r \) according to the algorithm. In other words, messages from processes in \( \text{HO}(p, r) \setminus \text{SHO}(p, r) \) were corrupted, be it due to errors during message transmission or because of the sender was faulty or lied deliberately. For both benign and Byzantine errors, the HO model registers the fault but
does not try to identify the faulty component (i.e., designate the sending or receiving process, or the communication channel as the “culprit”).

Executions of HO algorithms are defined with respect to collections $HO(p, r)$ and $SHO(p, r)$. However, the code of a process does not have access to these sets. In particular, process $p$ has no way of determining if a message it received from another process $q$ corresponds to what $q$ should have sent or if it has been corrupted.

Certain algorithms rely on the assignment of “coordinator” processes for each round. Just as the collections $HO(p, r)$, the definitions assume an external coordinator assignment such that $coord(p, r)$ denotes the coordinator of process $p$ and round $r$. Again, the correctness of algorithms may depend on hypotheses about coordinator assignments – e.g., it may be assumed that processes agree sufficiently often on who the current coordinator is.

The following definitions provide a generic representation of HO and SHO algorithms in Isabelle/HOL. A (coordinated) HO algorithm is described by the following parameters:

- a finite type $'proc$ of processes,
- a type $'pst$ of local process states,
- a type $'msg$ of messages sent in the course of the algorithm,
- a predicate $CinitState$ such that $CinitState p st crd$ is true precisely of the initial states $st$ of process $p$, assuming that $crd$ is the initial coordinator of $p$,
- a function $sendMsg$ where $sendMsg r p q st$ yields the message that process $p$ sends to process $q$ at round $r$, given its local state $st$, and
- a predicate $CnextState$ where $CnextState r p st msgs crd st'$ characterizes the successor states $st'$ of process $p$ at round $r$, given current state $st$, the vector $msgs :: 'proc => 'msg option$ of messages that $p$ received at round $r$ ($msgs q = None$ indicates that no message has been received from process $q$), and process $crd$ as the coordinator for the following round.

Note that every process can store the coordinator for the current round in its local state, and it is therefore not necessary to make the coordinator a parameter of the message sending function $sendMsg$.

We represent an algorithm by a record as follows.

```haskell
record ('proc, 'pst, 'msg) CHOAlgorithm =
  CinitState :: 'proc => 'pst => 'proc => bool
  sendMsg :: nat => 'proc => 'proc => 'pst => 'msg
  CnextState :: nat => 'proc => 'pst => ('proc => 'msg option) => 'proc => 'pst => bool
```

8
For non-coordinated HO algorithms, the coordinator argument of functions \textit{CinitState} and \textit{CnextState} is irrelevant, and we define utility functions that omit that argument.

\textbf{definition} \textit{isNCAlgorithm} where
\begin{align*}
\text{isNCAlgorithm alg} \equiv \\
(\forall p \, \text{st} \, \text{crd} \, \text{crd}'. \, \text{CinitState alg p st crd} = \text{CinitState alg p st crd}') \\
\land (\forall r \, p \, \text{msgs} \, \text{crd} \, \text{crd}' \, \text{st}'. \, \text{CnextState alg r p st msgs crd st'} = \text{CnextState alg r p st msgs crd st'})
\end{align*}

\textbf{definition} \textit{initState} where
\begin{align*}
\text{initState alg p st} \equiv \text{CinitState alg p st undefined}
\end{align*}

\textbf{definition} \textit{nextState} where
\begin{align*}
\text{nextState alg r p st msgs st'} \equiv \text{CnextState alg r p st msgs undefined st'}
\end{align*}

A \textit{heard-of assignment} associates a set of processes with each process. The following type is used to represent the collections \textit{HO}(p, r) and \textit{SHO}(p, r) for fixed round \textit{r}. Similarly, a \textit{coordinator assignment} associates a process (its coordinator) to each process.

\textbf{type-synonym}
\begin{align*}
'\text{proc HO} = '\text{proc} \Rightarrow '\text{proc set}
\end{align*}

\textbf{type-synonym}
\begin{align*}
'\text{proc coord} = '\text{proc} \Rightarrow '\text{proc}
\end{align*}

An execution of an HO algorithm is defined with respect to HO and SHO assignments that indicate, for every round \textit{r} and every process \textit{p}, from which sender processes \textit{p} receives messages (resp., uncorrupted messages) at round \textit{r}.

The following definitions formalize this idea. We define “coarse-grained” executions whose unit of atomicity is the round of execution. At each round, the entire collection of processes performs a transition according to the \textit{CnextState} function of the algorithm. Consequently, a system state is simply described by a configuration, i.e. a function assigning a process state to every process.

The predicate \textit{CSHOfinConfig} describes the possible initial configurations for algorithm \textit{A} (remember that a configuration is a function that assigns local states to every process).

\textbf{definition} \textit{CSHOfinConfig} where
\begin{align*}
\text{CSHOfinConfig Alg cfg (coord::'}\text{proc coord}') \equiv \forall p. \text{CinitState Alg p (cfg p) (coord p)}
\end{align*}
Given the current configuration $cfg$ and the HO and SHO sets $HOp$ and $SHOp$ for process $p$ at round $r$, the function $SHOmsgVectors$ computes the set of possible vectors of messages that process $p$ may receive. For processes $q \notin HOp$, $p$ receives no message (represented as value $None$). For processes $q \in SHOp$, $p$ receives the message that $q$ computed according to the $sendMsg$ function of the algorithm. For the remaining processes $q \in HOp - SHOp$, $p$ may receive some arbitrary value.

**definition** $SHOmsgVectors$

$SHOmsgVectors A r p cfg HOp SHOp \equiv$

\[ \{ \mu. (\forall q. q \in HOp \iff \mu \cdot q \neq None) \land (\forall q. q \in SHOp \cap HOp \rightarrow \mu \cdot q = Some (sendMsg A r q p (cfg q))) \}\]

Predicate $CSHOnextConfig$ uses the preceding function and the algorithm’s $CnextState$ function to characterize the possible successor configurations in a coarse-grained step, and predicate $CSHORun$ defines (coarse-grained) executions $rho$ of an HO algorithm.

**definition** $CSHOnextConfig$

$CSHOnextConfig A r cfg HO SHO coord cfg' \equiv$

\[ \forall p. \exists \mu \in SHOmsgVectors A r p cfg (HO p) (SHO p).\]

$CnextState A r p (cfg p) \mu (coord p) (cfg' p)$

**definition** $CSHORun$

$CSHORun A rho HOs SHOs coords \equiv$

$CHOinitConfig A rho 0 (coords 0)$

\[ \land (\forall r. CSHOnextConfig A r (rho r) (HOs r) (SHOs r) (coords (Suc r)) (rho (Suc r))) \]

For non-coordinated algorithms. the $coord$ arguments of the above functions are irrelevant. We define similar functions that omit that argument, and relate them to the above utility functions for these algorithms.

**definition** $HOinitConfig$

$HOinitConfig A cfg \equiv CHOinitConfig A cfg (\lambda q. undefined)$

**lemma** $HOinitConfig-eq$

$HOinitConfig A cfg = (\forall p. initState A p (cfg p))$

**by** (auto simp: $HOinitConfig-def$ $CHOinitConfig-def$ $initState-def$)

**definition** $SHOnextConfig$

$SHOnextConfig A r cfg HO SHO cfg' \equiv$

$CSHOnextConfig A r cfg HO SHO (\lambda q. undefined) cfg'$

**lemma** $SHOnextConfig-eq$

$SHOnextConfig A r cfg HO SHO cfg' =$

\[ (\forall p. \exists \mu \in SHOmsgVectors A r p cfg (HO p) (SHO p).\]

$CnextState A r p (cfg p) \mu (cfg' p))\]

**by** (auto simp: $SHOnextConfig-def$ $CSHOnextConfig-def$ $SHOmsgVectors-def$ $CnextState-def$)
definition \( \text{SHORun} \) where
\[
\text{SHORun} \ A \ \rho \ \text{HOs} \ \text{SHOs} \equiv \\
\text{CSHORun} \ A \ \rho \ \text{HOs} \ \text{SHOs} \ (\lambda \ r \ q. \ \text{undefined})
\]

lemma \( \text{SHORun-eq} \):
\[
\text{SHORun} \ A \ \rho \ \text{HOs} \ \text{SHOs} = \\
(\text{HOinitConfig} \ A \ (\rho \ 0)) \\
\land \ (\forall \ r. \ \text{SHOnextConfig} \ A \ r \ (\rho \ r) \ (\text{HOs} \ r) \ (\text{SHOs} \ r) \ (\rho \ (\text{Suc} \ r)))
\]
by (auto simp: \text{SHORun-def} \text{CSHORun-def} \text{HOinitConfig-def} \text{SHOnextConfig-def})

Algorithms designed to tolerate benign failures are not subject to message corruption, and therefore the SHO sets are irrelevant (more formally, each SHO set equals the corresponding HO set). We define corresponding special cases of the definitions of successor configurations and of runs, and prove that these are equivalent to simpler definitions that will be more useful in proofs. In particular, the vector of messages received by a process in a benign execution is uniquely determined from the current configuration and the HO sets.

definition \( \text{HOrcvdMsgs} \) where
\[
\text{HOrcvdMsgs} \ A \ r \ p \ \text{HO} \ \text{cfg} \equiv \\
\lambda q. \ \text{if} \ q \in \text{HO} \ \text{then} \ \text{Some} \ (\text{sendMsg} \ A \ r \ q \ p \ \text{cfg} \ q) \ \text{else} \ \text{None}
\]

lemma \( \text{SHOmsgVectors-HO} \):
\[
\text{SHOmsgVectors} \ A \ r \ p \ \text{cfg} \ \text{HO} \ \text{HO} = \{\text{HOrcvdMsgs} \ A \ r \ p \ \text{HO} \ \text{cfg}\}
\]
unfolding \( \text{SHOmsgVectors-def} \) \( \text{HOrcvdMsgs-def} \) by auto

With coordinators
definition \( \text{CHOnextConfig} \) where
\[
\text{CHOnextConfig} \ A \ r \ \text{cfg} \ \text{HO} \ \text{coord} \ \text{cfg}' \equiv \\
\text{CSHOnextConfig} \ A \ r \ \text{cfg} \ \text{HO} \ \text{coord} \ \text{cfg}'
\]

lemma \( \text{CHOnextConfig-eq} \):
\[
\text{CHOnextConfig} \ A \ r \ \text{cfg} \ \text{HO} \ \text{coord} \ \text{cfg}' = \\
(\forall \ p. \ \text{CnextState} \ A \ r \ p \ (\text{cfg} \ p) \ (\text{HOrcvdMsgs} \ A \ r \ p \ (\text{HO} \ p) \ \text{cfg}) \\
(\text{coord} \ p) \ (\text{cfg}' \ p))
\]
by (auto simp: \text{CHOnextConfig-def} \text{CSHOnextConfig-def} \text{SHOmsgVectors-HO})

definition \( \text{CHORun} \) where
\[
\text{CHORun} \ A \ \rho \ \text{HOs} \ \text{coords} \equiv \text{CSHORun} \ A \ \rho \ \text{HOs} \ \text{HOs} \ \text{coords}
\]

lemma \( \text{CHORun-eq} \):
\[
\text{CHORun} \ A \ \rho \ \text{HOs} \ \text{coords} = \\
(\text{CHOinitConfig} \ A \ (\rho \ 0) \ (\text{coords} \ 0)) \\
\land \ (\forall \ r. \ \text{CHOnextConfig} \ A \ r \ (\rho \ r) \ (\text{HOs} \ r) \ (\text{coords} \ (\text{Suc} \ r)) \ (\rho \ (\text{Suc} \ r)))
\]
by (auto simp: \text{CHORun-def} \text{CSHORun-def} \text{CHOinitConfig-def} \text{CHOnextConfig-def})

Without coordinators
definition \( \text{HOnextConfig} \) where
lemma HOnextConfig-eq:
\[ HOnextConfig A r \ cfg \ HO \ cfg' = (\forall p. \ nextState A r p (\cfg p) (HOnextConfig A r p (\HO p) \ cfg) (\cfg' p)) \]
by (auto simp: HOnextConfig-def SHOnextConfig-eq SHOmsgVectors-HO)

definition HORun where
HORun A rho HOs \equiv SHORun A rho HOs HOs

lemma HORun-eq:
\[ HORun A rho \ HOs = \]
\[ (HOinitConfig A \ (rho 0) \]
\[ \land (\forall r. HOnextConfig A r \ (rho r) (\HO s r) (\rho (Suc r)))) \]
by (auto simp: HORun-def SHORun-eq HOnextConfig-def)

The following derived proof rules are immediate consequences of the definition of CHORun; they simplify automatic reasoning.

lemma CHORun-0:
assumes CHORun A rho HOs coords
and \(\forall cfg. \ CHOinitConfig A \ cfg \ (coords 0) \implies P \ cfg\)
shows P \(\rho 0\)
using assms unfolding CHORun-eq by blast

lemma CHORun-Suc:
assumes CHORun A rho HOs coords
and \(\forall r. HOnextConfig A r \ (rho r) (\HO s r) (coords (Suc r)) (\rho (Suc r))\)
\[ \implies P \ r \]
shows P \(n\)
using assms unfolding CHORun-eq by blast

lemma CHORun-induct:
assumes run: CHORun A rho HOs coords
and init: CHOinitConfig A \(\rho 0\) (coords 0) \implies P \ 0
and step: \(\forall r. [ P \ r; HOnextConfig A r \ (rho r) (\HO s r) (coords (Suc r)) \]
\[ (\rho (Suc r)) ] \implies P \ (Suc r) \]
shows P \(n\)
using run unfolding CHORun-eq by (induct n, auto elim: init step)

Because algorithms will not operate for arbitrary HO, SHO, and coordinator assignments, these are constrained by a communication predicate. For convenience, we split this predicate into a per Round part that is expected to hold at every round and a global part that must hold of the sequence of \((S)HO\) assignments and may thus express liveness assumptions.

In the parlance of [7], a \emph{HO machine} is an HO algorithm augmented with a communication predicate. We therefore define \((C)(S)HO\) machines as the corresponding extensions of the record defining an HO algorithm.

record \('\proc', 'pst', 'msg'\) HOMachine = \('\proc', 'pst', 'msg'\) \CHOAlgorithm +
3 Reduction Theorem

We have defined the semantics of HO algorithms such that rounds are executed atomically, by all processes. This definition is surprising for a model of asynchronous distributed algorithms since it models a synchronous execution of rounds. However, it simplifies representing and reasoning about the algorithms. For example, the communication network does not have to be modeled explicitly, since the possible sets of messages received by processes can be computed from the global configuration and the collections of HO and SHO sets.

We will now define a more conventional “fine-grained” semantics where communication is modeled explicitly and rounds of processes can be arbitrarily interleaved (subject to the constraints of the communication predicates). We will then establish a reduction theorem that shows that for every fine-grained run there exists an equivalent round-based (“coarse-grained”) run in the sense that the two runs exhibit the same sequences of local states of all processes, modulo stuttering. We prove the reduction theorem for the most general class of coordinated SHO algorithms. It is easy to see that the theorem equally holds for the special cases of uncoordinated or HO algorithms, and since we have in fact defined these classes of algorithms from the more general ones, we can directly apply the general theorem.

As a corollary, interesting properties remain valid in the fine-grained semantics if they hold in the coarse-grained semantics. It is therefore enough to verify such properties in the coarse-grained semantics, which is much eas-
ier to reason about. The essential restriction is that properties may not depend on states of different processes occurring simultaneously. (For example, the coarse-grained semantics ensures by definition that all processes execute the same round at any instant, which is obviously not true of the fine-grained semantics.) We claim that all “reasonable” properties of fault-tolerant distributed algorithms are preserved by our reduction. For example, the Consensus (and Weak Consensus) problems fall into this class. The proofs follow Chaouch-Saad et al. [4], where the reduction theorem was proved for uncoordinated HO algorithms.

### 3.1 Fine-Grained Semantics

In the fine-grained semantics, a run of an HO algorithm is represented as an $\omega$-sequence of system configurations. Each configuration is represented as a record carrying the following information:

- for every process $p$, the current round that process $p$ is executing,
- the local state of every process,
- for every process $p$, the set of processes to which $p$ has already sent a message for the current round,
- for all processes $p$ and $q$, the message (if any) that $p$ has received from $q$ for the round that $p$ is currently executing, and
- the set of messages in transit, represented as triples of the form $(p, r, q, m)$ meaning that process $p$ sent message $m$ to process $q$ for round $r$, but $q$ has not yet received that message.

As explained earlier, the coordinators of processes are not recorded in the configuration, but algorithms may record them as part of the process states.

```plaintext
record ('pst', 'proc', 'msg') config =
  round :: 'proc ⇒ nat
  state :: 'proc ⇒ 'pst
  sent :: 'proc ⇒ 'proc set
  rcvd :: 'proc ⇒ 'proc ⇒ 'msg option
  network :: ('proc * nat * 'proc * 'msg) set

type-synonym ('pst , 'proc , 'msg) fgrun = nat ⇒ ('pst , 'proc , 'msg) config

definition fg-init-config where
  fg-init-config A (config::('pst, 'proc, 'msg) config) (coord::'proc coord) ≡
```

14
\[
\text{round config} = (\lambda p. 0)
\wedge (\forall p. \text{CinitState A p (state config p) (coord p)})
\wedge \text{sent config} = (\lambda p. \{\})
\wedge \text{rcvd config} = (\lambda p q. \text{None})
\wedge \text{network config} = \{\}
\]

In the fine-grained semantics, we have three types of transitions due to

- some process sending a message,
- some process receiving a message, and
- some process executing a local transition.

The following definition models process \( p \) sending a message to process \( q \). The transition is enabled if \( p \) has not yet sent any message to \( q \) for the current round. The message to be sent is computed according to the algorithm’s \( \text{sendMsg} \) function. The effect of the transition is to add \( q \) to the \text{sent} component of the configuration and the message quadruple to the \text{network} component.

\textbf{definition} \( \text{fg-send-msg} \) where
\[
f_{g\text{-send-msg}} A \ p \ q \ \text{config config}' \equiv
q \notin (\text{sent config } p) \\
\wedge \text{config'} = \text{config } (\|)
\wedge \text{sent} := (\text{sent config }) (p := (\text{sent config } p) \cup \{q\}),
\wedge \text{network} := \text{network config } \cup
\{(p, \text{round config } p, q, \text{sendMsg } A (\text{round config } p) p q (\text{state config } p))\}) \}
\]

The following definition models the reception of a message by process \( p \) from process \( q \). The action is enabled if \( q \) is in the heard-of set \( \text{HO} \) of process \( p \) for the current round, and if the network contains some message from \( q \) to \( p \) for the round that \( p \) is currently executing. W.l.o.g., we model message corruption at reception: if \( q \) is not in \( p \)'s \( \text{SHO} \) set (parameter \( \text{SHO} \)), then an arbitrary value \( m' \) is received instead of \( m \).

\textbf{definition} \( \text{fg-rcv-msg} \) where
\[
f_{g\text{-rcv-msg}} p \ q \ \text{HO} \ \text{SHO} \ \text{config config}' \equiv
\exists m m'. (q, (\text{round config } p), p, m) \in \text{network config}
\wedge q \in \text{HO}
\wedge \text{config'} = \text{config } (\|)
\wedge \text{rcvd} := (\text{rcvd config })(p := (\text{rcvd config } p) (q :=
\text{if } q \in \text{SHO then Some } m \text{ else Some } m'))
\wedge \text{network} := \text{network config } \cup \{(q, (\text{round config } p), p, m))\} \}
\]

Finally, we consider local state transition of process \( p \). A local transition is enabled only after \( p \) has sent all messages for its current round and has received all messages that it is supposed to receive according to its current
HO set (parameter $HO$). The local state is updated according to the algorithm’s $C_{nextState}$ relation, which may depend on the coordinator $crd$ of the following round. The round of process $p$ is incremented, and the $sent$ and $rcvd$ components for process $p$ are reset to initial values for the new round.

**definition fg-local where**

\[ fg\text{-}local \ A \ p \ HO \ crd \ config \ config' \equiv \]
\[ \begin{align*}
    & \text{sent \ config \ p} = UNIV \\
    \land & \text{dom (rcvd config p)} = HO \\
    \land & (\exists s. \ C_{nextState} \ A (\text{round config p}) \ p (\text{state config p}) (\text{rcvd config p}) \ crd \ s \\
    \land & \text{config'} = \text{config} [] \\
    \land & \text{round := (round config)(p := Suc (round config p))}, \\
    \land & \text{state := (state config)(p := s)}, \\
    \land & \text{sent := (sent config)(p := {})}, \\
    \land & \text{rcvd := (rcvd config)(p := \lambda q. \text{None})})
\end{align*} \]

The next-state relation for process $p$ is just the disjunction of the above three types of transitions.

**definition fg-next-config where**

\[ fg\text{-}next\text{-}config \ A \ p \ HO \ SHO \ crd \ config \ config' \equiv \]
\[ \begin{align*}
    & (\exists q. \ fg\text{-}send\text{-}msg \ A \ p \ q \ config \ config') \\
    \lor & (\exists q. \ fg\text{-}rcv\text{-}msg \ p \ q \ HO \ SHO \ config \ config') \\
    \lor & fg\text{-}local \ A \ p \ HO \ crd \ config \ config'
\end{align*} \]

Fine-grained runs are infinite sequences of configurations that start in an initial configuration and where each step corresponds to some process sending a message, receiving a message or performing a local step. We also require that every process eventually executes every round – note that this condition is implicit in the definition of coarse-grained runs.

**definition fg-run where**

\[ fg\text{-}run \ A \ \rho \ HO\_{s} \ SHO\_{s} \ coords \equiv \]
\[ \begin{align*}
    & fg\text{-}init\text{-}config \ A (\rho \ 0) (coords \ 0) \\
    \land & (\forall i. \ \exists p. \ fg\text{-}next-config \ A \ p \ (HO\_{s} (\text{round (\rho \ i) \ p}) \ p) \\
    \land & (SHO\_{s} (\text{round (\rho \ i) \ p}) \ p) \\
    \land & (coords (\text{round (\rho \ (Suc \ i)) \ p}) \ p) \\
    \land & (\rho \ i) (\rho \ (Suc \ i))) \\
    \land & (\forall p \ r. \ \exists n. \ \text{round (\rho \ n) \ p} = r
\end{align*} \]

The following function computes at which “time point” (index in the fine-grained computation) process $p$ starts executing round $r$. This function plays an important role in the correspondence between the two semantics, and in the subsequent proofs.

**definition fg-start-round where**

\[ fg\text{-}start\text{-}round \ \rho \ p \ r \equiv \text{LEAST} \ (n::\text{nat}). \ \text{round (\rho \ n) \ p} = r \]
3.2 Properties of the Fine-Grained Semantics

In preparation for the proof of the reduction theorem, we establish a number of consequences of the above definitions.

Process states change only when round numbers change during a fine-grained run.

**lemma fg-state-change:**

**assumes** rho: fg-run A rho HOs SHOs coords
  and rd: round (rho (Suc n)) p = round (rho n) p
**shows** state (rho (Suc n)) p = state (rho n) p

**proof** –
  from rho have \( \exists p'. \ fg-next-config A p' (HOs (round (rho n) p')) p' \)
  (SHOs (round (rho n) p') p')
  (coords (round (rho (Suc n)) p') p')
  (rho n) (rho (Suc n))
  by (auto simp: fg-run-def)
  with rd show \(?thesis\)
  by (auto simp: fg-next-config-def fg-send-msg-def fg-rcv-msg-def fg-local-def)

**qed**

Round numbers never decrease.

**lemma fg-round-numbers-increase:**

**assumes** rho: fg-run A rho HOs SHOs coords and n: \( n \leq m \)
**shows** round (rho n) p \( \leq \) round (rho m) p

**proof** –
  from n obtain k where k: \( m = n+k \) by (auto simp: le_iff_add)
  { fix i have round (rho n) p \( \leq \) round (rho (n+i)) p (is \(?P i\))
    proof (induct i)
      show \(?P 0\) by simp
    next
      fix j assume ih: \(?P j\)
      from rho have \( \exists p'. \ fg-next-config A p' (HOs (round (rho (n+j)) p')) p' \)
      (SHOs (round (rho (n+j)) p') p')
      (coords (round (rho (Suc (n+j))) p') p')
      (rho (n+j)) (rho (Suc (n+j)))
      by (auto simp: fg-run-def)
      hence round (rho (n+j)) p \( \leq \) round (rho (n + Suc j)) p
      by (auto simp: fg-next-config-def fg-send-msg-def fg-rcv-msg-def fg-local-def)
      with ih show \(?P (Suc j)\) by auto
    qed
  }
  with k show \(?thesis\) by simp

**qed**

Combining the two preceding lemmas, it follows that the local states of
process $p$ at two configurations are the same if these configurations have the
same round number.

**Lemma fg-same-round-same-state:**
assumes $\rho$: fg-run $A$ $\rho$ HOs SHOs coords
and $rd$: round ($\rho m$) $p$ = round ($\rho n$) $p$
shows state ($\rho m$) $p$ = state ($\rho n$) $p$

**Proof** –

$$
\begin{array}{c}
\text{fix } k
\text{ have round } (\rho (k+i)) p = \text{round } (\rho k) p \\
\implies \text{state } (\rho (k+i)) p = \text{state } (\rho k) p
\end{array}
$$

(is $?R i \implies ?S i$)

**Proof** (induct $i$

show $?S 0$ by simp

next

fix $j$
assume $ih$: $?R j \implies ?S j$
and $r$: round ($\rho (k + \text{Suc } j)$) $p$ = round ($\rho k$) $p$
from $\rho$ have 1: round ($\rho k$) $p$ $\leq$ round ($\rho (k+j)$) $p$
by (auto elim: fg-round-numbers-increase)
from $\rho$ have 2: round ($\rho (k+j)$) $p$ $\leq$ round ($\rho (k + \text{Suc } j)$) $p$
by (auto elim: fg-round-numbers-increase)
from 1 2 $r$ have 3: round ($\rho (k+j)$) $p$ = round ($\rho k$) $p$ by auto
with $r$ have round ($\rho (\text{Suc } (k+j))$) $p$ = round ($\rho (k+j)$) $p$ by simp
with $\rho$ have state ($\rho (\text{Suc } (k+j))$) $p$ = state ($\rho (k+j)$) $p$
by (auto elim: fg-state-change)
with 3 $ih$ show $?S (\text{Suc } j)$ by simp

qed

Note aux = this

show $?thesis$

**Proof** (cases $n \leq m$

case $\text{True}$
then obtain $k$ where $m = n+k$ by (auto simp: le-iff-add)
with $rd$ show $?thesis$ by (auto simp: aux)

next

case $\text{False}$
hence $m \leq n$ by simp
then obtain $k$ where $n = m+k$ by (auto simp: le-iff-add)
with $rd$ show $?thesis$ by (auto simp: aux)

qed

Since every process executes every round, function $fg-startRound$ is well-defined. We also list a few facts about $fg-startRound$ that will be used to show that it is a “stuttering sampling function”, a notion introduced in the theories about stuttering equivalence.

**Lemma fg-start-round:**
assumes \( \text{fg-run} \ A \ \rho \ \text{HOs SHOs coords} \)
shows \( \text{round} \ (\rho \ (\text{fg-start-round } \rho \ p \ r)) \ p = r \)
using assms by (auto simp: \text{fg-run-def} \ \text{fg-start-round-def} \ intro: LeastI-ex)

lemma \( \text{fg-start-round-smallest} \):
assumes \( \text{round} \ (\rho \ k) \ p = r \)
shows \( \text{fg-start-round } \rho \ p \ r \ \leq (\text{k::nat}) \)
using assms unfolding \( \text{fg-start-round-def} \) by (rule Least-le)

lemma \( \text{fg-start-round-later} \):
assumes \( \rho: \text{fg-run} \ A \ \rho \ \text{HOs SHOs coords} \)
and \( r: \text{round} \ (\rho \ n) \ p = r \ \text{and} \ r': r < r' \)
shows \( n < \text{fg-start-round } \rho \ p \ r' \) (is \( < \ ?\text{start} \))
proof (rule ccontr)
assume \( \neg \ ?\text{thesis} \)
hence \( \text{start} \ : ?\text{start} \leq n \) by simp  
from \( \rho \) this have \( \text{round} \ (\rho \ ?\text{start}) \ p \leq \text{round} \ (\rho \ n) \ p \)
by (rule \text{fg-round-numbers-increase})
with \( r \) have \( r' \leq r \) by (simp add: \( \text{fg-start-round} \)[OF \( \rho \)])
with \( r' \) show \( \text{False} \) by simp
qed

lemma \( \text{fg-start-round-0} \):
assumes \( \rho: \text{fg-run} \ A \ \rho \ \text{HOs SHOs coords} \)
shows \( \text{fg-start-round } \rho \ p \ 0 = 0 \)
proof  
from \( \rho \) this have \( \text{round} \ (\rho \ 0) \ p = 0 \) by (auto simp: \( \text{fg-run-def} \ \text{fg-init-config-def} \))
by (rule \( \text{fg-start-round-smallest} \))
thus \( \?\text{thesis} \) by simp
qed

lemma \( \text{fg-start-round-strict-mono} \):
assumes \( \rho: \text{fg-run} \ A \ \rho \ \text{HOs SHOs coords} \)
shows \( \text{strict-mono} \ (\text{fg-start-round } \rho \ p) \)
proof  
fix \( r \ r' \)
assume \( r: \ (\text{r::nat}) < r' \)
from \( \rho \) have \( \text{round} \ (\rho \ (\text{fg-start-round } \rho \ p \ r)) \ p = r \) by (rule \( \text{fg-start-round} \))
from \( \rho \) this \( r \) show \( \text{fg-start-round } \rho \ p \ r < \text{fg-start-round } \rho \ p \ r' \)
by (rule \( \text{fg-start-round-later} \))
qed

Process \( p \) is at round \( r \) at all configurations between the start of round \( r \) and the start of round \( r + 1 \). By lemma \( \text{fg-same-round-same-state} \), this implies that the local state of process \( p \) is the same at all these configurations.

lemma \( \text{fg-round-between-start-rounds} \):
assumes \( \rho: \text{fg-run} \ A \ \rho \ \text{HOs SHOs coords} \)
and \( 1: \text{fg-start-round } \rho \ p \ r \ \leq n \)
and \( 2: n < \text{fg-start-round } \rho \ p \ (\text{Suc } r) \)
shows  \( \text{round } (\rho n) = r \) (is \( ?rd = r \))

proof (rule antisym)
  from 1 have \( \text{round } (\rho (\text{fg-start-round } \rho p r)) \leq ?rd \)
    by (rule \( \text{fg-round-numbers-increase}[\OF \rho] \))
  thus \( r \leq ?rd \) by (simp add: \( \text{fg-start-round}[\OF \rho] \))

next
  show \( ?rd \leq r \)
  proof (rule ccontr)
    assume \( \neg \thesis \)
    hence \( \text{Suc } r \leq ?rd \) by simp
    hence \( \text{fg-start-round } \rho p (\text{Suc } r) \leq \text{fg-start-round } \rho p ?rd \)
      by (rule \( \text{rho}[\THEN \text{fg-start-round-strict-mono}, \THEN \text{strict-mono-strict-mono}, \THEN \text{monoD}] \))
    also have \( ... \leq n \) by (auto intro: \( \text{fg-start-round-smallest} \))
    also note 2
    finally show False by simp
  qed
  qed

For any process \( p \) and round \( r \) there is some instant \( n \) where \( p \) executes a local transition from round \( r \). In fact, \( n+1 \) marks the start of round \( r+1 \).

lemma \( \text{fg-local-transition-from-round} \):
assumes \( \rho: \text{fg-run } A \rho \text{ HOs SHOs coords} \)
obtains \( n \) where \( \text{round } (\rho n) = r \)
  and \( \text{fg-start-round } \rho p (\text{Suc } r) = \text{Suc } n \)
  and \( \text{fg-local } A p (\text{HOs } r p) (\text{coords } (\text{Suc } r) p) (\rho n) (\rho (\text{Suc } n)) \)
proof
  have \( \text{fg-start-round } \rho p (\text{Suc } r) \neq 0 \) (is \( ?\text{start} \neq 0 \))
    proof
      assume \( \neg \text{start} = 0 \)
      from \( \rho \) have \( \text{round } (\text{fg ?start}) p = \text{Suc } r \) by (rule \( \text{fg-start-round} \))
      with \( \neg \text{start} \rho \) show False by (auto simp: \text{fg-run-def \text{fg-init-config-def}})
    qed
  then obtain \( n \) where \( n: ?\text{start} = \text{Suc } n \) by (auto simp: \text{gr0-conv-Suc})
  with \( \text{fg-start-round}[\OF \rho, \OF p \text{ Suc } r] \)
  have 0: \( \text{round } (\rho (\text{Suc } n)) p = \text{Suc } r \) by simp
  have 1: \( \text{round } (\rho n) = r \)
    proof (rule \( \text{fg-round-between-start-rounds}[\OF \rho] \))
      have \( \text{fg-start-round } \rho p r < \text{fg-start-round } \rho p (\text{Suc } r) \)
        by (rule \( \text{fg-start-round-strict-mono}[\OF \rho, \THEN \text{strict-monoD}] \))
      with \( n \) show \( \text{fg-start-round } \rho p r \leq n \) by simp
    qed
  next
    from \( n \) show \( n < ?\text{start} \) by simp
  qed
from \( \rho \) obtain \( p' \) where
  \( \text{fg-next-config } A p' (\text{HOs } \text{round } (\rho n) p') p' \)
  (\( \text{SHOs } (\text{round } (\rho n) p') p' \))
  (\( \text{coords } (\text{round } (\rho (\text{Suc } n)) p') p' \))
  (\( \rho n \) (\( \rho (\text{Suc } n) \)))
lemma fg-invariant1:
  assumes rho: fg-run A rho HOs SHOs coords
  and m: (p, r, q, m) ∈ network (rho n) (is ?msg n)
  shows m = sendMsg A r p q (state (rho (fg-start-round rho p r)) p)
using m proof (induct n)
  — the base case is trivial because the network is empty
  assume ?msg 0 with rho show ?thesis
    by (auto simp: fg-run-def fg-init-config-def)
next
  fix n
  assume m': ?msg (Suc n) and ih: ?msg n ⇒ ?thesis
  from rho obtain p' where
    fg-next-config A p' (HOs (round (rho n) p') p')
     (SHOs (round (rho n) p') p')
     (coords (round (rho (Suc n)) p') p')
     (rho n) (rho (Suc n))
    (is fg-next-config - - ?HO ?SHO ?crd ?cfg ?cfg')
    by (force simp: fg-run-def)
  thus ?thesis
proof (auto simp: fg-next-config-def)

Only fg-send-msg transitions for process p are interesting, since all other transitions
cannot add a message for p, hence we can apply the induction hypothesis.

  fix q'
  show ?thesis
proof (cases ?msg n)
  case True
  with ih show ?thesis .
next
case False
  with send m' have 1: p' = p round ?cfg p = r
  and 2: m = sendMsg A r p q (state ?cfg p)
  by (auto simp: fg-send-msg-def)
from rho 1 have state ?cfg p = state (rho (fg-start-round rho p r)) p
  by (auto simp: fg-start-round fg-same-round-same-state)
  with 1 2 show ?thesis by simp
qed
next
fix q'
  with m' have ?msg n by (auto simp: fg-rcv-msg-def)
  with ih show ?thesis .
next
  with m' have ?msg n by (auto simp: fg-local-def)
  with ih show ?thesis .
qed
qed

The second invariant states that if process q received message m from process p, then (a) p is in q's HO set for that round m, and (b) if p is moreover in q's SHO set, then m is the message that p computed at the start of that round.

lemma fg-invariant2a:
  assumes rho: fg-run A rho HOs SHOs coords
  and m: rcvd (rho n) q p = Some m (is ?rcvd n)
  shows p ∈ HOs (round (rho n) q) q
  (is p ∈ HOs (?rd n) q is ?P n)
using m proof (induct n)
  — The base case is trivial because q has not received any message initially
  assume ?rcvd 0 with rho show ?P 0
  by (auto simp: fg-run-def fg-init-config-def)
next
fix n
  assume rcvd: ?rcvd (Suc n) and ih: ?rcvd n −→ ?P n
  — For the inductive step we distinguish the possible transitions
from rho obtain p' where
  fg-next-config A p' (HOs (round (rho n) p) p')
  (SHOs (round (rho n) p) p')
  (coords (round (rho (Suc n)) p') p')
  (rho n) (rho (Suc n))
  (is fg-next-config - - ?HO ?SHO ?crd ?cfg ?cfg')
by (force simp: fg-run-def)
thus ?P (Suc n)
proof (auto simp: fg-next-config-def)

Except for fg-rev-msg steps of process $q$, the proof is immediately reduced to the
induction hypothesis.

fix $q'$
assume rcvmsg: fg-rev-msg $p'$ $q'$ $r$ $s$ $h$ $s$ $c$

hence rd: $?rd$ ($Suc$ $n$) = $?rd$ $n$ by (auto simp: fg-rev-msg-def)
show $?P$ ($Suc$ $n$)
proof (cases $?rcvd$ $n$)
case True
with ih rd show $?thesis$ by simp
next
case False
with rcvd rcvmsg rd show $?thesis$ by (auto simp: fg-rev-msg-def)
qed

next
fix $q'$
assume fg-send-msg $A$ $p'$ $q'$ $c$

with rcvd have $?rcvd$ $n$ and $?rd$ ($Suc$ $n$) = $?rd$ $n$
by (auto simp: fg-send-msg-def)
with ih show $?P$ ($Suc$ $n$) by simp
qed

next
assume fg-local $A$ $p'$ $r$ $c$

with rcvd have $?rcvd$ $n$ and $?rd$ ($Suc$ $n$) = $?rd$ $n$
in fact, $p' = q$ is impossible because the $rcvd$ field of $p'$ is cleared
by (auto simp: fg-local-def)
with ih show $?P$ ($Suc$ $n$) by simp
qed

lemma fg-invariant2b:
assumes rho: fg-run $A$ $rho$ $HOs$ $SHOs$ $coords$
and m: rcvd ($rho$ $n$) $q$ $p$ = Some $m$ ($is$ $?rcvd$ $n$)
and sho: $p$ $\in$ $SHOs$ ($round$ ($rho$ $n$) $q$) $q$ ($is$ $p$ $\in$ $SHOs$ ($?rd$ $n$) $q$)
shows m = sendMsg $A$ ($?rd$ $n$) $p$ $q$
  (state ($rho$ ($fg$-start-round $rho$ $p$ ($?rd$ $n$))) $p$)
  ($is$ $?P$ $n$)
using m sho proof (induct $n$)
— The base case is trivial because $q$ has not received any message initially
assume $?rcvd$ 0 with rho show $?P$ 0
by (auto simp: fg-run-def fg-init-config-def)

next
fix $n$
assume rcvd: $?rcvd$ ($Suc$ $n$) and $p$: $p$ $\in$ $SHOs$ ($?rd$ ($Suc$ $n$)) $q$
and ih: $?rcvd$ $n$ $\Longrightarrow$ $p$ $\in$ $SHOs$ ($?rd$ $n$) $q$ $\Longrightarrow$ $?P$ $n$
— For the inductive step we again distinguish the possible transitions
from rho obtain $p'$ where
fg-next-config $A$ $p'$ ($HOs$ ($round$ ($rho$ $n$) $p'$) $p'$)
  ($SHOs$ ($round$ ($rho$ $n$) $p'$) $p'$)
(coords (round (rho (Suc n)) p') p')
(rho n) (rho (Suc n))
(is fg-next-config - - ?HO ?SHO ?crd ?cfg ?cfg')
by (force simp: fg-run-def)
thus ?P (Suc n)
proof (auto simp: fg-next-config-def)

Except for fg-rcv-msg steps of process q, the proof is immediately reduced to the
induction hypothesis.

fix q'
hence rd: ?rd (Suc n) = ?rd n by (auto simp: fg-rcv-msg-def)
show ?P (Suc n)
proof (cases ?rcvd n)
case True
  with ih p rd show ?thesis by simp
next
case False
from rcvmsg obtain m' m'' where
  (q', round ?cfg p', p', m') ∈ network ?cfg
  rcvd ?cfg' = (rcvd ?cfg)(p' := (rcvd ?cfg p')(q' :=
    if q' ∈ ?SHO then Some m' else Some m''))
  by (auto simp: fg-rcv-msg-def split del: if-split-asm)
with False rcvd p rd have (p, ?rd n, q, m) ∈ network ?cfg by auto
with rho rd show ?thesis by (auto simp: fg-invariant1)
qed
next
fix q'
assume fg-send-msg A p' q' ?cfg ?cfg'
with rcvd have ?rcvd n and ?rd (Suc n) = ?rd n
by (auto simp: fg-send-msg-def)
with p ih show ?P (Suc n) by simp
next
with rcvd have ?rcvd n and ?rd (Suc n) = ?rd n
  — in fact, p' = q is impossible because the rcvd field of p' is cleared
  by (auto simp: fg-local-def)
with p ih show ?P (Suc n) by simp
qed

3.3 From Fine-Grained to Coarse-Grained Runs

The reduction theorem asserts that for any fine-grained run rho there is a
coarse-grained run such that every process sees the same sequence of local
states in the two runs, modulo stuttering. In other words, no process can
locally distinguish the two runs.

Given fine-grained run rho, the corresponding coarse-grained run sigma is
defined as the sequence of state vectors at the beginning of every round. Notice in particular that the local states \( \sigma_r p \) and \( \sigma_r q \) of two different processes \( p \) and \( q \) appear at different instants in the original run \( \rho \). Nevertheless, we prove that \( \sigma \) is a coarse-grained run of the algorithm for the same HO, SHO, and coordinator assignments. By definition (and the fact that local states remain equal between \( fg-start-round \) instants), the sequences of process states in \( \rho \) and \( \sigma \) are easily seen to be stuttering equivalent, and this will be formally stated below.

**definition coarse-run**

\[
\text{coarse-run } \rho \ r \ p \equiv \text{state } (\rho (fg-start-round \rho p r)) p
\]

**theorem** reduction:

- **assumes** \( \rho; fg-run A \rho HOs SHOs coords \)
- **shows** CSHORun A (coarse-run \( \rho \)) HOs SHOs coords
  \[
  (\text{is CSHORun - ?cr - - -})
  \]
- **proof** (auto simp: CSHORun-def)
- **from** \( \rho \) **show** CHOinitConfig A (?cr 0) (coords 0)
  \[
  \text{by (auto simp: fg-run-def fg-init-config-def CHOinitConfig-def coarse-run-def fg-start-round-0[OF rho])}
  \]

**next**

- **fix** \( r \)
- **show** CSHOnextConfig A r (?cr r) (HOs r) (SHOs r) (coords (Suc r))
  \[
  (?cr (Suc r))
  \]
- **proof** (auto simp add: CSHOnextConfig-def)
- **fix** \( p \)
- **from** \( \rho[\text{THEN \text{fg-local-transition-from-round}] \text{ obtain} n \)
- **where** \( n: \text{round } (\rho n) p = r \)
  \[
  \text{and start: } \text{fg-start-round } \rho p (\Suc r) = \Suc n \text{ (is ?start = -)}
  \]
  \[
  \text{and loc: } \text{fg-local } A p (\HOs r p) (\coords (\Suc r) p) (\rho n) (\rho (\Suc n))
  \]
  \[
  (\text{is fg-local - - ?HO ?crd ?cfg ?cfg'})
  \]
  \[
  \text{by blast}
  \]
- **have** \( \text{cfg: } ?\text{cr } r \ p = \text{state } \text{?cfg } p \)
  \[
  \text{unfolding coarse-run-def proof (rule fg-same-round-same-state[OF rho])}
  \]
- **from** \( n \) **show** \( \text{round } (\rho (fg-start-round \rho p r)) p = \text{round } ?\text{cfg } p \)
  \[
  \text{by (simp add: fg-start-round[OF rho])}
  \]
- **qed**
- **from** \( \text{start} \) **have** \( \text{cfg': } ?\text{cr (Suc r) p = state } ?\text{cfg'} p \)
  \[
  \text{by (simp add: coarse-run-def)}
  \]
- **have** \( \text{rcvd: } \text{rcvd } \text{?cfg } p \in \text{SHOmsgVectors A r p (?cr r) ?HO (SHOs r p)} \)
- **proof** (auto simp: SHOmsgVectors-def)
- **fix** \( q \)
- **assume** \( q \in ?\text{HO} \)
- **with** \( n \) **show** \( \exists m. \text{rcvd } ?\text{cfg } p \ q = \text{Some } m \)
  \[
  \text{by (auto simp: fg-local-def)}
  \]
- **next**
- **fix** \( q \ m \)
- **assume** \( \text{rcvd } ?\text{cfg } p \ q = \text{Some } m \)
- **with** \( \rho \) **show** \( q \in ?\text{HO} \) **by** (auto simp: fg-invariant2a)
- **next**

25
3.4 Locally Similar Runs and Local Properties

We say that two sequences of configurations (vectors of process states) are **locally similar** if for every process the sequences of its process states are stuttering equivalent. Observe that different stuttering reduction may be applied for every process, hence the original sequences of configurations need not be stuttering equivalent and can indeed differ wildly in the combinations of local states that occur.

A property of a sequence of configurations is called **local** if it is insensitive to local similarity.

**definition** locally-similar where

\[ \text{locally-similar} \ (\sigma :: \text{nat} \Rightarrow \text{'proc} \Rightarrow \text{'pst}) \ 
\forall p :: \text{'proc}. (\lambda n. \sigma n p) \approx (\lambda n. \tau n p) \]

**definition** local-property where

\[ \text{local-property} \ P \ \equiv \ 
\forall \sigma \tau. \text{locally-similar} \sigma \tau \rightarrow P \sigma \rightarrow P \tau \]

Local similarity is an equivalence relation.

**lemma** locally-similar-refl: locally-similar \( \sigma \sigma \)

by (simp add: locally-similar-def stutter-equiv-refl)

**lemma** locally-similar-sym: locally-similar \( \sigma \tau \rightarrow \rightarrow \text{locally-similar} \tau \sigma \)

by (simp add: locally-similar-def stutter-equiv-sym)

**lemma** locally-similar-trans [trans]:

\[ \text{locally-similar} \ g \sigma \rightarrow \text{locally-similar} \sigma \tau \rightarrow \text{locally-similar} \ g \tau \]

by (force simp add: locally-similar-def elim: stutter-equiv-trans)

**lemma** local-property-eq:

\[ \text{local-property} \ P = (\forall \sigma \tau. \text{locally-similar} \sigma \tau \rightarrow P \sigma = P \tau) \]

by (auto simp: local-property-def dest: locally-similar-sym)

Consider any fine-grained run \( \text{rho} \). The projection of \( \text{rho} \) to vectors of
process states is locally similar to the coarse-grained run computed from $\rho$.

**lemma** coarse-run-locally-similar:

**assumes** $\rho$: fg-run $A$ $\rho$ HOs SHOs coords

**shows** locally-similar ($state \circ \rho$) (coarse-run $\rho$)

**proof** (auto simp: locally-similar-def)

fix $p$

show ($\lambda n. state \ (\rho n) \ p \approx (\lambda n. coarse-run \ \rho \ n \ p)$) (is $\ ?fgr \approx \ ?cgr$)

**proof** (rule stutter-equivI)

show stutter-sampler ($fg-start-round \ \rho \ p \ ?fgr$)

**proof** (auto simp: stutter-sampler-def)

next

show strict-mono ($fg-start-round \ \rho \ p$)

by (rule $fg-start-round-strict-mono$)

next

fix $r \ n$

assume $fg-start-round \ \rho \ p \ r < n$ and $n < fg-start-round \ \rho \ p \ (Suc \ r)$

with $\rho$ have round ($\rho n$) $p$ = round ($\rho \ (fg-start-round \ \rho \ r)$) $p$

by (simp add: $fg-start-round-fg-round-between-start-rounds$)

with $\rho$ show state ($\rho n$) $p$ = state ($\rho \ (fg-start-round \ \rho \ r)$) $p$

by (rule $fg-same-round-same-state$)

qed

next

show stutter-sampler id $\ ?cgr$

by (rule id-stutter-sampler)

next

show $\ ?fgr \circ fg-start-round \ \rho \ p \ = \ ?cgr \circ id$

by (auto simp: coarse-run-def)

qed

Therefore, in order to verify a local property $P$ for a fine-grained run over given HO, SHO, and coord collections, it is enough to show that $P$ holds for all coarse-grained runs for these same collections. Indeed, one may restrict attention to coarse-grained runs whose initial states agree with that of the given fine-grained run.

**theorem** local-property-reduction:

**assumes** $\rho$: fg-run $A$ $\rho$ HOs SHOs coords

and $P$: local-property $P$

and coarse-correct:

$\forall \ crho. \ [\ CSHORun \ A \ crho \ HOs \ SHOs \ coords; \ crho \ 0 = \ state \ (\rho \ 0)] \Rightarrow P \ crho$

**shows** $P \ (state \circ \rho)$

**proof**

have coarse-run $\rho \ 0 = state \ (\rho \ 0)$

by (rule ext, simp add: coarse-run-def $fg-start-round-0$[OF $\rho$])

27
from rho \[ \text{THEN reduction} \] this
have \( P \) (coarse-run rho) by (rule coarse-correct)
with coarse-run-locally-similar[OF rho] \( P \)
show \( ?\text{thesis} \) by (auto simp: local-property-eq)
qed

3.5 Consensus as a Local Property

Consensus and Weak Consensus are local properties and can therefore be verified just over coarse-grained runs, according to theorem \( \text{local-property-reduction} \).

lemma integrity-is-local:
assumes \( \text{sim: locally-similar } \sigma \tau \)
and \( \text{val: } \bigwedge n. \text{dec} (\sigma n p) = \text{Some } v \implies v \in \text{range vals} \)
and \( \text{dec: } \text{dec} (\tau n p) = \text{Some } v \)
shows \( v \in \text{range vals} \)
proof –
from \( \text{sim have } (\lambda r. \sigma r p) \approx (\lambda r. \tau r p) \) by (simp add: locally-similar-def)
then obtain \( m \) where \( \sigma m p = \tau n p \) by (rule stutter-equiv-element-left)
from \( \text{sym[OF this] dec show } ?\text{thesis} \) by (auto elim: val)
qed

lemma validity-is-local:
assumes \( \text{sim: locally-similar } \sigma \tau \)
and \( \text{val: } \bigwedge n. \text{dec} (\sigma n p) = \text{Some } w = v \)
and \( \text{dec: } \text{dec} (\tau n p) = \text{Some } w \)
shows \( w = v \)
proof –
from \( \text{sim have } (\lambda r. \sigma r p) \approx (\lambda r. \tau r p) \) by (simp add: locally-similar-def)
then obtain \( m \) where \( \sigma m p = \tau n p \) by (rule stutter-equiv-element-left)
from \( \text{sym[OF this] dec show } ?\text{thesis} \) by (auto elim: val)
qed

lemma agreement-is-local:
assumes \( \text{sim: locally-similar } \sigma \tau \)
and \( \text{agr: } \bigwedge m n. \left[ \text{dec} (\sigma m p) = \text{Some } v; \text{dec} (\sigma n q) = \text{Some } w \right] \implies v=w \)
and \( v: \text{dec} (\tau m p) = \text{Some } v \) and \( w: \text{dec} (\tau n q) = \text{Some } w \)
shows \( v = w \)
proof –
from \( \text{sim have } (\lambda r. \sigma r p) \approx (\lambda r. \tau r p) \) by (simp add: locally-similar-def)
then obtain \( m' \) where \( \sigma m' p = \tau m p \) by (rule stutter-equiv-element-left)
from \( \text{sim have } (\lambda r. \sigma r q) \approx (\lambda r. \tau r q) \) by (simp add: locally-similar-def)
then obtain \( n' \) where \( \sigma n' q = \tau n q \) by (rule stutter-equiv-element-left)
from \( \text{sym[OF m'] sym[OF n'] v w show } v = w \) by (auto elim: agr)
qed

lemma termination-is-local:
assumes \( \text{sim: locally-similar } \sigma \tau \)
and \( \text{trm: } \text{dec} (\sigma m p) = \text{Some } v \)
shows \( \exists n. \text{dec} (\tau n p) = \text{Some } v \)
proof
  from sim have \((\lambda r. \sigma r p) \approx (\lambda r. \tau r p)\) by (simp add: locally-similar-def)
  then obtain \(n\) where \(\sigma m p = \tau n p\) by (rule stutter-equiv-element-right)
  with trm show ?thesis by auto
qed

theorem consensus-is-local: local-property (consensus vals dec)
proof (auto simp: local-property-def consensus-def)
  fix \(\sigma \tau n p v\)
  assume locally-similar \(\sigma \tau\)
  and \(\forall n p v. \text{dec}(\sigma n p) = \text{Some } v \rightarrow v \in \text{range vals}\)
  and \(\text{dec}(\tau n p) = \text{Some } v\)
  thus \(v \in \text{range vals}\) by (blast intro: integrity-is-local)
next
  fix \(\sigma \tau m n p q v w\)
  assume locally-similar \(\sigma \tau\)
  and \(\forall m n p q v w. \text{dec}(\sigma m p) = \text{Some } v \rightarrow v = w\)
  and \(\text{dec}(\tau m p) = \text{Some } v\) and \(\text{dec}(\tau n q) = \text{Some } w\)
  thus \(v = w\) by (blast intro: agreement-is-local)
next
  fix \(\sigma \tau p\)
  assume locally-similar \(\sigma \tau\)
  and \(\forall p. \exists m v. \text{dec}(\sigma m p) = \text{Some } v\)
  thus \(\exists n w. \text{dec}(\tau n p) = \text{Some } w\) by (blast dest: termination-is-local)
qed

theorem weak-consensus-is-local: local-property (weak-consensus vals dec)
proof (auto simp: local-property-def weak-consensus-def)
  fix \(\sigma \tau n p v w\)
  assume locally-similar \(\sigma \tau\)
  and \(\forall n p v w. \text{dec}(\sigma n p) = \text{Some } v \rightarrow v = w\)
  and \(\text{dec}(\tau n p) = \text{Some } v\) and \(\text{dec}(\tau n q) = \text{Some } w\)
  thus \(v = w\) by (blast intro: validity-is-local)
next
  fix \(\sigma \tau m n p q v w\)
  assume locally-similar \(\sigma \tau\)
  and \(\forall m n p q v w. \text{dec}(\sigma m p) = \text{Some } v \rightarrow v = w\)
  and \(\text{dec}(\tau m p) = \text{Some } v\) and \(\text{dec}(\tau n q) = \text{Some } w\)
  thus \(v = w\) by (blast intro: agreement-is-local)
next
  fix \(\sigma \tau p\)
  assume locally-similar \(\sigma \tau\)
  and \(\forall p. \exists m v. \text{dec}(\sigma m p) = \text{Some } v\)
  thus \(\exists n w. \text{dec}(\tau n p) = \text{Some } w\) by (blast dest: termination-is-local)
qed

end
theory Majorities
4 Utility Lemmas About Majorities

Consensus algorithms usually ensure that a majority of processes proposes the same value before taking a decision, and we provide a few utility lemmas for reasoning about majorities.

Any two subsets $S$ and $T$ of a finite set $E$ such that the sum of their cardinalities is larger than the size of $E$ have a non-empty intersection.

**Lemma abs-majorities-intersect:**

**Assumes**
- $\text{card } E < \text{card } S + \text{card } T$
- $S \subseteq E$ and $T \subseteq E$ and $e$: finite $E$

**Shows**
- $S \cap T \neq \{\}$

**Proof**

- **Clarify**
- **Assume**
  - $S \cap T = \{\}$
- **Have**
  - finite $S$ and finite $T$ by (auto simp: finite-subset)
- **With**
  - $\text{card } S \cap T = \{\}$
- **Show**
  - finite $S \cup T$ by (simp add: card-Un-Int)
- **Moreover**
  - $\text{card } S \cup T \leq \text{card } E$ by (simp add: card-mono)
- **Ultimately**
  - $\text{Show } False$ by simp

**Lemma abs-majoritiesE:**

**Assumes**
- $\text{card } E < \text{card } S + \text{card } T$
- $S \subseteq E$ and $T \subseteq E$ and $e$: finite $E$

**Obtains**
- $p$ where $p \in S$ and $p \in T$

**Proof**

- **From**
  - $\text{card } S \cap T \neq \{\}$
- **By** (rule abs-majorities-intersect)
- **Then**
  - $\text{Obtain } p$ where $p \in S \cap T$ by blast
- **With**
  - that $\text{Show } ?thesis$ by auto

**Lemma abs-majoritiesE’:**

**Assumes**
- $\text{card } S > (\text{card } E) \div 2$ and $\text{card } T > (\text{card } E) \div 2$
- $S \subseteq E$ and $T \subseteq E$ and $e$: finite $E$

**Obtains**
- $p$ where $p \in S$ and $p \in T$

**Proof**

- **Rule** abs-majoritiesE[OF $s t e$]
- **From**
  - $\text{Smaj Tmaj}$
  - $\text{Show } \text{card } E < \text{card } S + \text{card } T$ by auto

**Qed**

We restate the above theorems for the case where the base type is finite (taking $E$ as the universal set).

**Lemma** majority-intersect:
assumes crd: \( \text{card} (\text{UNIV}::('a::finite) \text{ set}) < \text{card} (S::'a \text{ set}) + \text{card} T \)
shows \( S \cap T \neq \{\} \)
by (rule abs-majorities-intersect[of crd]) auto

lemma majoritiesE:
assumes crd: \( \text{card} (\text{UNIV}::('a::finite) \text{ set}) < \text{card} (S::'a \text{ set}) + \text{card} (T::'a \text{ set}) \)
obtains \( p \) where \( p \in S \) and \( p \in T \)
using crd majorities-intersect by blast

lemma majoritiesE':
assumes \( S: \text{card} (S::('a::finite) \text{ set}) > (\text{card} (\text{UNIV}::'a \text{ set})) \div 2 \)
and \( T: \text{card} (T::'a \text{ set}) > (\text{card} (\text{UNIV}::'a \text{ set})) \div 2 \)
obtains \( p \) where \( p \in S \) and \( p \in T \)
by (rule abs-majoritiesE[of S T]) auto

5 Verification of the One-Third Rule Consensus Algorithm

We now apply the framework introduced so far to the verification of concrete algorithms, starting with algorithm One-Third Rule, which is one of the simplest algorithms presented in [7]. Nevertheless, the algorithm has some interesting characteristics: it ensures safety (i.e., the Integrity and Agreement) properties in the presence of arbitrary benign faults, and if everything works perfectly, it terminates in just two rounds. One-Third Rule is an uncoordinated algorithm tolerating benign faults, hence SHO or coordinator sets do not play a role in its definition.

5.1 Model of the Algorithm

We begin by introducing an anonymous type of processes of finite cardinality that will instantiate the type variable 'proc of the generic HO model.

typedecl Proc — the set of processes
axiomatization where Proc-finite: OFCLASS(Proc, finite-class)
instance Proc :: finite by (rule Proc-finite)

abbreviation
\( N \equiv \text{card} (\text{UNIV}::\text{Proc set}) \)

The state of each process consists of two fields: \( x \) holds the current value proposed by the process and \( \text{decide} \) the value (if any, hence the option type) it has decided.

31
record ′val pstate =
  x :: ′val
  decide :: ′val option

The initial value of field x is unconstrained, but no decision has been taken initially.

definition OTR-initState where
  OTR-initState p st ≡ decide st = None

Given a vector msgs of values (possibly null) received from each process, HOV msgs v denotes the set of processes from which value v was received.

definition HOV :: (Proc ⇒ ′val option) ⇒ ′val ⇒ Proc set where
  HOV msgs v ≡ { q . msgs q = Some v }

MFR msgs v ("most frequently received") holds for vector msgs if no value has been received more frequently than v.

Some such value always exists, since there is only a finite set of processes and thus a finite set of possible cardinalities of the sets HOV msgs v.

definition MFR :: (Proc ⇒ ′val option) ⇒ ′val ⇒ bool where
  MFR msgs v ≡ ∀ w. card (HOV msgs w) ≤ card (HOV msgs v)

lemma MFR-exists: ∃ v. MFR msgs v
proof −
  let ?cards = { card (HOV msgs v) | v . True }
  let ?mfr = Max ?cards
  have ∀ v. card (HOV msgs v) ≤ N by (auto intro: card-mono)
  hence ?cards ⊆ { 0 .. N } by auto
  hence fin: finite ?cards by (metis atLeast0AtMost finite-atMost finite-subset)
  hence ?mfr ∈ ?cards by (rule Max-in) auto
  then obtain v where v: ?mfr = card (HOV msgs v) by auto
  have MFR msgs v
    proof (auto simp: MFR-def)
      fix w
      from fin have card (HOV msgs w) ≤ ?mfr by (rule Max-ge) auto
      thus card (HOV msgs w) ≤ card (HOV msgs v) by (unfold v)
    qed
  thus ?thesis ..
  qed

Also, if a process has heard from at least one other process, the most frequently received values are among the received messages.

lemma MFR-in-msgs:
  assumes HO:HOs m p ≠ {} and v: MFR (HOrcvdMsgs OTR-M m p (HOs m p) (rho m)) v
  (is MFR ?msgs v)
  shows ∃ q ∈ HOs m p. v = the (?msgs q)
proof −
from \( \text{HO} \) obtain \( q \) where \( q : q \in \text{HOs} m p \)
by \( \text{auto} \)
with \( v \) have \( \text{HOV} \ ?\text{msgs} \ (\text{the} \ (?\text{msgs} \ q)) \neq \{} \)
by \( \text{(auto simp: \text{HO-def} \ \text{HOrcvdMsgs-def})} \)
\text{hence} \( \text{HOp} : 0 < \text{card} \ (\text{HOV} \ ?\text{msgs} \ (\text{the} \ (?\text{msgs} \ q))) \)
by \( \text{auto} \)
also from \( v \) have \( \ldots \leq \text{card} \ (\text{HOV} \ ?\text{msgs} \ v) \)
by \( \text{(simp add: \text{MFR-def})} \)
finally have \( \text{HOV} \ ?\text{msgs} \ v \neq \{} \)
by \( \text{auto} \)
\text{thus} \ ?\text{thesis} 
by \( \text{(auto simp: \text{HO-def} \ \text{HOrcvdMsgs-def})} \)
\text{qed}

TwoThirds \?\text{msgs} \ v \) holds if value \( v \) has been received from more than \( 2/3 \) of all processes.

\text{definition} TwoThirds \text{where}
\( \text{TwoThirds} \ ?\text{msgs} \ v \equiv (2 \ast N) \div3 < \text{card} \ (\text{HOV} \ ?\text{msgs} \ v) \)

The next-state relation of algorithm \text{One-Third Rule} for every process is defined as follows: if the process has received values from more than \( 2/3 \) of all processes, the \( x \) field is set to the smallest among the most frequently received values, and the process decides value \( v \) if it received \( v \) from more than \( 2/3 \) of all processes. If \( p \) hasn’t heard from more than \( 2/3 \) of all processes, the state remains unchanged. (Note that \text{Some} is the constructor of the option datatype, whereas \( \epsilon \) is Hilbert’s choice operator.) We require the type of values to be linearly ordered so that the minimum is guaranteed to be well-defined.

\text{definition} OTR-\text{nextState} \text{where}
\( \text{OTR-\text{nextState}} \ r \ p \ (\text{st::(val::\text{linorder}) \ pstate}) \ ?\text{msgs} \ ?\text{st}' \equiv 
\begin{cases}
\text{if} \ (2 \ast N) \div3 < \text{card} \ \{q. \ ?\text{msgs} q \neq \text{None}\}, \\
\text{then} \ ?\text{st}' = (|x = \text{Min} \ \{v. \ \text{MFR} \ ?\text{msgs} v\}, \\
\text{decide} = (\text{if} \ (\exists v. \ \text{TwoThirds} \ ?\text{msgs} v) \\
\text{then} \text{Some} \ (\epsilon \ v. \ \text{TwoThirds} \ ?\text{msgs} v) \\
\text{else} \text{decide} \ ?\text{st}) \}) \\
\text{else} \ ?\text{st}' = \ ?\text{st}
\end{cases} 
\)

The message sending function is very simple: at every round, every process sends its current proposal (field \( x \) of its local state) to all processes.

\text{definition} OTR-\text{sendMsg} \text{where}
\( \text{OTR-\text{sendMsg}} \ r \ p \ q \ ?\text{st} \equiv x \ ?\text{st} \)

5.2 Communication Predicate for \text{One-Third Rule}

We now define the communication predicate for the \text{One-Third Rule} algorithm to be correct. It requires that, infinitely often, there is a round where all processes receive messages from the same set \( \Pi \) of processes where \( \Pi \)
contains more than two thirds of all processes. The “per-round” part of the communication predicate is trivial.

**Definition** OTR-commPerRd where

\[\text{OTR-commPerRd } H\text{Ors } \equiv \text{True}\]

**Definition** OTR-commGlobal where

\[\text{OTR-commGlobal } H\text{Os } \equiv \forall r. \exists r_0 \Pi. r_0 \geq r \land (\forall p. H\text{Os } r_0 p = \Pi) \land \text{card } \Pi > (2N) \text{ div } 3\]

### 5.3 The One-Third Rule Heard-Of Machine

We now define the HO machine for the One-Third Rule algorithm by assembling the algorithm definition and its communication-predicate. Because this is an uncoordinated algorithm, the \(\text{crd}\) arguments of the initial- and next-state predicates are unused.

**Definition** OTR-HOMachine where

\[
\text{OTR-HOMachine } = \\
\langle \text{CinitState } = (\lambda p \text{ st } \text{ crd}. \text{OTR-initState } p \text{ st}), \text{sendMsg } = \text{OTR-sendMsg}, \text{CnextState } = (\lambda r p \text{ st } \text{ msg} s \text{ crd st}'. \text{OTR-nextState } r p \text{ st } \text{ msg} s \text{ st'}), \text{HOcommPerRd } = \text{OTR-commPerRd}, \text{HOcommGlobal } = \text{OTR-commGlobal } \rangle
\]

**Abbreviation** OTR-M \(\equiv\) OTR-HOMachine":(Proc, 'val::linorder pstate, 'val) HOMachine

end

theory OneThirdRuleProof
imports OneThirdRuleDefs ../Reduction ../Majorities
begin

We prove that One-Third Rule solves the Consensus problem under the communication predicate defined above. The proof is split into proofs of the Integrity, Agreement, and Termination properties.

### 5.4 Proof of Integrity

Showing integrity of the algorithm is a simple, if slightly tedious exercise in invariant reasoning. The following inductive invariant asserts that the values of the \(x\) and \(\text{decide}\) fields of the process states are limited to the \(x\) values present in the initial states since the algorithm does not introduce any new values.

**Definition** \(V\text{Inv}\) where

\[
\text{VInv } \rho n \equiv \\
\text{let } x\text{init } = (\text{range } (x \circ (\rho 0))) \\
\text{in } \text{range } (x \circ (\rho n)) \subseteq x\text{init} \\
\land \text{range } (\text{decide } \circ (\rho n)) \subseteq \{\text{None}\} \cup (\text{Some } x\text{init})
\]
lemma vinv-invariant:
  assumes run: HORun OTR-M rho HOs
  shows VInv rho n
proof (induct n)
  from run show VInv rho 0
    by (simp add: HORun-eq HOinitConfig-eq OTR-HOMachine-def initState-def
      OTR-initState-def VInv-def image-def)
next
  fix m
  assume ih: VInv rho m
  let ?xinit = range (x ◦ (rho m))
  have range (x ◦ (rho (Suc m))) ⊆ ?xinit
    proof (clarsimp)
      fix p
      from run
      have nxt: OTR-nextState m p (rho m p)
        (HOrcvdMsgs OTR-M m p (HOs m p) (rho m))
        (rho (Suc m) p)
        (is OTR-nextState - - ?st ?msgs ?st')
      by (simp add: HORun-eq HOnextConfig-eq OTR-HOMachine-def nextState-def)
    show x ?st' ∈ ?xinit
      proof (cases (2*N) div 3 < card (HOs m p))
        case True
        hence HO: HOs m p ≠ {} by auto
        let ?MFRs = {v. MFR ?msgs v}
        have Min ?MFRs ∈ ?MFRs
          proof (rule Min-in)
            from HO
            have ?MFRs ⊆ (the ?msgs)’(HOs m p)
            by (auto simp: image-def intro: MFR-in-msgs)
          thus finite ?MFRs by (auto elim: finite-subset)
        next
          from MFR-exists show ?MFRs ≠ {} by auto
        qed
        with HO have ∃ q ∈ HOs m p. Min ?MFRs = the (?msgs q)
          by (intro MFR-in-msgs auto)
        hence ∃ q ∈ HOs m p. Min ?MFRs = x (rho m q)
          by (auto simp: HOrcvdMsgs-def OTR-HOMachine-def OTR-sendMsg-def)
        moreover
          from True nxt have x ?st' = Min ?MFRs
            by (simp add: OTR-nextState-def HOrcvdMsgs-def)
        ultimately
          show ?thesis using ih
            by (auto simp: VInv-def image-def)
      next
        case False
        with nxt ih show ?thesis
          by (auto simp: OTR-nextState-def VInv-def HOrcvdMsgs-def Let-def)
      qed
    qed
  qed

35
moreover have \( \forall p. \ Decide ((\rho (Suc \ m)) p) \in \{ \text{None} \} \cup (\text{Some ' xinit}) \)

proof
  fix \( p \)
  from run have \( \text{nxt}: \text{OTR-nextState} m p (\rho m p) \)
  \((\text{HOrcvdMsgs} \ \text{OTR-M} m p (\text{HOs} m p) (\rho m)) \)
  \((\rho (Suc \ m) p) \)
  \((\text{is OTR-nextState - - ?st ?msgs ?st'}) \)
  by (simp add: \text{HORun-eq} \text{HOnextConfig-eq} \text{OTR-HOMachine-def} \text{nextState-def})
  show \( \text{decide ?st' \in \{ \text{None} \} \cup (\text{Some ' xinit})} \)
  proof (cases (2*\( N \)) div 3 < card \{ \text{q. ?msgs q \neq None} \})
    assume \( \text{HO}: (2*\( N \)) \text{ div } 3 < \text{card} \{ \text{q. ?msgs q \neq None} \} \)
    show \( \text{?thesis} \)
    proof (cases \( \exists v. \text{TwoThirds} \ ?msgs v \))
      case True
      let \( ?\text{dec} = \epsilon v. \text{TwoThirds} \ ?msgs v \)
      from True have \( \text{TwoThirds} \ ?msgs \ ?\text{dec} \) by (rule twoThirds)
      hence \( \text{HOV} \ ?msgs \ ?\text{dec} \neq \{ \} \) by (auto simp add: \text{TwoThirds-def})
      then obtain \( q \) where \( x (\rho m q) = \ ?\text{dec} \)
      by (auto simp: \text{HOV-def} \text{HOrcvdMsgs-def} \text{OTR-HOMachine-def} \text{OTR-sendMsg-def})
      from sym [OF this] \( \text{nxt} \ \text{ih} \) show \( \text{?thesis} \)
      by (auto simp: \text{OTR-nextState-def} \text{VInv-def} \text{image-def})
    next
      case False
      with \( \text{HO} \ \text{nxt} \ \text{ih} \) show \( \text{?thesis} \)
      by (auto simp: \text{OTR-nextState-def} \text{VInv-def} \text{HOrcvdMsgs-def} \text{image-def})
    qed
    qed
  next
  case False
  with \( \text{nxt} \ \text{ih} \) show \( \text{?thesis} \)
  by (auto simp: \text{OTR-nextState-def} \text{VInv-def} \text{image-def})
  qed
  qed
  hence \( \text{range (decide o (\rho (Suc \ m)))} \subseteq \{ \text{None} \} \cup (\text{Some ' xinit}) \) by auto
  ultimately show \( \text{VInv} \ \rho (Suc \ m) \) by (auto simp: \text{VInv-def} \text{image-def})
  qed

Integrity is an immediate consequence.

theorem \text{OTR-integrity}: assumes run: \text{HORun} \ \text{OTR-M} \ \rho \ \text{HOs} \ and \ \text{dec}: \text{decide} (\rho n p) = \text{Some v}
shows \( \exists q. \ v = x (\rho 0 q) \)
proof –
  let \( ?\text{xinit} = \text{range (x o (\rho 0))} \)
  from run have \( \text{VInv} \ \rho \ \text{n} \ by (rule vine-invariant} \)
  hence \( \text{range (decide o (\rho n))} \subseteq \{ \text{None} \} \cup (\text{Some ' ?xinit}) \)
  by (auto simp: \text{VInv-def} \text{Let-def})
hence \( \text{decide} \ (\rho_n, p) \in \{\text{None}\} \cup \{\text{Some } \xi \text{init}\} \)
by (auto simp: image-def)
with \(\text{dec} \ \text{show} \ \theta\text{thesis by auto} \)
qed

5.5 Proof of Agreement

The following lemma \(A1\) asserts that if process \(p\) decides in a round on a value \(v\) then more than \(2/3\) of all processes have \(v\) as their \(x\) value in their local state.

We show a few simple lemmas in preparation.

**lemma** nextState-change:
assumes \(\text{HORun OTR-M } \rho \ \text{HOs}\)
and \(\neg ((2 \ast N) \div 3 < \text{card } \{q. (\text{HOrcvdMsgs OTR-M } n \ p \ (\text{HOs n p}) (\rho n)) \ q \neq \text{None}\})\)
shows \(\rho (\text{Suc } n) \ p = \rho n \ p\)
using \(\text{assms by (auto simp: HORun-eq HOnextConfig-eq OTR-HOMachine-def nextState-def OTR-nextState-def)}\)

**lemma** nextState-decide:
assumes \(\text{run: HORun OTR-M } \rho \ \text{HOs}\)
and \(\text{chg: decide } (\rho (\text{Suc } n) \ p) \neq \text{decide } (\rho n \ p)\)
shows \(\text{TwoThirds } \text{HOrcvdMsgs OTR-M } n \ p \ (\text{HOs n p}) (\rho n)\)
using \(\text{assms by (auto simp: OTR-nextState-def elim: someI)}\)

**lemma A1**:
assumes \(\text{run: HORun OTR-M } \rho \ \text{HOs}\)
and \(\text{dec: decide } (\rho (\text{Suc } n) \ p) = \text{Some } v\)
and \(\text{chg: decide } (\rho (\text{Suc } n) \ p) \neq \text{decide } (\rho n \ p) \ (\text{is decide } \eta \neq \text{decide } \xi)\)
shows \((2 \ast N) \div 3 < \text{card } \{q. x (\rho n \ q) = v\}\)
proof –
from \(\text{run} \ \text{chg}\)
have \(\text{TwoThirds } \text{HOrcvdMsgs OTR-M } n \ p \ (\text{HOs n p}) (\rho n)\)
using \(\text{assms by (rule nextState-decide)}\)
with \(\text{dec} \ \text{have} \ \text{TwoThirds } \eta \text{msgs v by simp} \)
hence \((2 \ast N) \div 3 < \text{card } \{q. \eta \text{msgs } q = \text{Some } v\}\)
by (simp add: TwoThirds-def HOV-def)
moreover
have \{ q . ?msgs q = Some v \} \subseteq \{ q . x (\rho n q) = v \}
by (auto simp: OTR-HOMachine-def OTR-sendMsg-def HOrcvdMsgs-def)
hence card \{ q . ?msgs q = Some v \} \leq card \{ q . x (\rho n q) = v \}
by (simp add: card-mono)
ultimately
show ?thesis by simp
qed

The following lemma A2 contains the crucial correctness argument: if more
than 2/3 of all processes send \( v \) and process \( p \) hears from more than 2/3 of
all processes then the \( x \) field of \( p \) will be updated to \( v \).

lemma A2:
assumes run: \( \text{HORun OTR-M rho HO} \)
and \( \text{HO: (2N) div 3 < card \{ q . HOrcvdMsgs OTR-M n p (HOs n p) (\rho n) q \neq None \} } \)
and \( \text{maj: (2N) div 3 < card \{ q . x (\rho n q) = v \} } \)
shows \( x (\rho (Suc n) p) = v \)
proof –
from run
have nxt: \( \text{OTR-nextState n p (\rho n p) } \)
(\( \text{HOrcvdMsgs OTR-M n p (HOs n p) (\rho n) } \)
(\( \text{rho (Suc n) p) } \)
(is \( \text{OTR-nextState - - ?st ?msgs ?st'} \)
by (simp add: HORun-eq HOnextConfig-eq OTR-HOMachine-def nextState-def)
let \( ?HOVothers = \bigcup \{ \text{HOV} ?msgs w | w . w \neq v \} \)
— processes from which \( p \) received values different from \( v \)
have w: card ?HOVothers \leq N div 3
proof –
have card ?HOVothers \leq card (UNIV - \{ q . x (\rho n q) = v \} )
by (auto simp: HOM-def HOrcvdMsgs-def OTR-HOMachine-def OTR-sendMsg-def
intro: card-mono)
also have \ldots = N - card \{ q . x (\rho n q) = v \}
by (auto simp: card-Diff-subset)
also from maj have \ldots \leq N div 3 by auto
finally show ?thesis ,
qed

have hov: \( \text{HOV} ?msgs v = \{ q . ?msgs q \neq None \} - \text{HOVothers} \)
by (auto simp: HOV-def) blast

have othHO: \( \text{HOVothers} \subseteq \{ q . ?msgs q \neq None \} \)
by (auto simp: HOV-def)

Show that \( v \) has been received from more than \( N/3 \) processes.
from \( \text{HO} \) have \( N \text{ div 3 < card \{ q . ?msgs q \neq None \} } - (N \text{ div 3}) \)
by auto
also from \( w \) \text{HO have} \ldots \leq card \{ q . ?msgs q \neq None \} - \text{card \text{HOVothers} }

38
by auto
also from hov othHO have \ldots = card (HOV ?msgs v)
  by (auto simp: card-Diff-subset)
finally have HOV: N div 3 < card (HOV ?msgs v).

All other values are received from at most N/3 processes.

have \( \forall w. w \neq v \rightarrow card (HOV ?msgs w) \leq card \ HOV\text{others} \)
  by (force intro: card-mono)
with w have cardw: \( \forall w. w \neq v \rightarrow card (HOV ?msgs w) \leq N \) div 3 by auto

In particular, \( v \) is the single most frequently received value.

with HOV have MFR ?msgs v by (auto simp: MFR-def)

moreover
have \( \forall w. w \neq v \rightarrow -(MFR ?msgs w) \)
proof (auto simp: MFR-def not-le)
fix w
  assume w \neq v
  with cardw HOV have card (HOV ?msgs w) < card (HOV ?msgs v) by auto
thus \( \exists v. card (HOV ?msgs w) < card (HOV ?msgs v) \).
qed

ultimately
have mfrv: \{ w. MFR ?msgs w \} = \{ v \} by auto

have card \{ q . ?msgs q = Some v \} \leq card \{ q . ?msgs q \neq None \}
  by (auto intro: card-mono)
with HO mfrv nxt show ?thesis by (auto simp: OTR-nextState-def)
qed

Therefore, once more than two thirds of the processes hold \( v \) in their \( x \) field,
this will remain true forever.

lemma \( A3: \)
  assumes \( \text{run:HORun OTR-M rho HOs} \)
  and \( n: (2 \ast N) \) div 3 < card \{ q . (rho n q) = v \} (is \( \text{?twothird n} \))
  shows \( ?\text{twothird} (n+k) \)
proof (induct k)
  from n show \( ?\text{twothird} (n+0) \) by simp
next
  fix m
  assume m: ?twothird \( (n+m) \)
  have \( \forall q. x (rho (n+m) q) = v \rightarrow x (rho (n + Suc m) q) = v \)
  proof (rule+)
    fix q
      assume q: x \( ((rho (n+m)) q) = v \)
    let \( ?msgs = \text{HORcvdMsgs OTR-M (n+m) q \{ HOs (n+m) q \} (rho (n+m))} \)
    show x \( (rho (n + Suc m) q) = v \)
    proof (cases \( (2\ast N) \) div 3 < card \{ q . ?msgs q \neq None \})
      case True
      qed

39
from m have \((2 \times N) \div 3 < \text{card}\ \{q . x (\rho (n+m)) q = v\}\) by simp

next

case False

with run q show \(?\text{thesis}\) by (auto dest: nextState-change)

qed

hence \(\text{card}\ \{q . x (\rho (n+m)) q = v\}\leq \text{card}\ \{q . x (\rho (n + \text{Suc} m)) q = v\}\)

by (auto intro: card-mono)

with m show \(?\text{twothird}\ (n + \text{Suc} m)\) by simp

qed

It now follows that once a process has decided on some value \(v\), more than two thirds of all processes continue to hold \(v\) in their \(x\) field.

**Lemma A4:**

assumes run: HORun OTR-M rho HOs

and \(\forall k. (2 \times N) \div 3 < \text{card}\ \{q . x (\rho (n+k)) q = v\}\)

using \(\text{dec}\)

— The base case is trivial since no process has decided

assume \(?\text{dec} 0\) with run show \(?\text{twothird}\ (0 + k)\)

by (simp add: HORun-eq HOinitConfig-eq OTR-HOMachine-def

initState-def OTR-initState-def)

next

— For the inductive step, we assume that process \(p\) has decided on \(v\).

fix m

assume ih: \(?\text{dec m} \implies \forall k. \?\text{twothird}\ (m+k)\) and m: \(?\text{dec} (\text{Suc} m)\)

show \(?\text{twothird}\ ((\text{Suc} m) + k)\)

proof

fix k

have \(?\text{twothird}\ (m + \text{Suc} k)\)

There are two cases to consider: if \(p\) had already decided on \(v\) before, the assertion follows from the induction hypothesis. Otherwise, the assertion follows from lemmas \(A1\) and \(A3\).

proof (cases \(?\text{dec m}\)

  case True with ih show \(?\text{thesis}\) by blast

next

  case False

  with run m have \(?\text{twothird} m\) by (auto elim: A1)

  with run show \(?\text{thesis}\) by (blast dest: A3)

  qed

  thus \(?\text{twothird}\ ((\text{Suc} m) + k)\) by simp

  qed

  qed

The Agreement property follows easily from lemma A4: if processes \(p\) and \(q\) decide values \(v\) and \(w\), respectively, then more than two thirds of the
processes must propose \( v \) and more than two thirds must propose \( w \). Because these two majorities must have an intersection, we must have \( v = w \).

We first prove an “asymmetric” version of the agreement property before deriving the general agreement theorem.

**lemma A5:**

**assumes** run: HORun OTR-M rho HOs

**and** \( p: \) decide (rho n p) = Some v

**and** \( p': \) decide (rho (n+k) p') = Some w

**shows** \( v = w \)

**proof** –

- **from** run p
  - **have** \((2 \ast N) \div 3 < \text{card } \{ q. x (\rho (n+k) q) = v \} \) (is - < card ?V)
    - by (blast dest: A4)
  - **moreover**
    - **from** run p'
      - **have** \((2 \ast N) \div 3 < \text{card } \{ q. x (\rho ((n+k)+0) q) = w \} \) (is - < card ?W)
        - by (blast dest: A4)
    - **ultimately**
      - **have** \( N < \text{card } ?V + \text{card } ?W \) by auto
      - **then obtain** proc where proc \( \in ?V \cap ?W \) by (auto dest: majorities-intersect)
      - **thus** ?thesis by auto

**qed**

**theorem OTR-agreement:**

**assumes** run: HORun OTR-M rho HOs

**and** \( p: \) decide (rho n p) = Some v

**and** \( p': \) decide (rho m p') = Some w

**shows** \( v = w \)

**proof** (cases \( n \leq m \))

- **case** True
  - **then obtain** \( k \) where \( m = n+k \) by (auto simp add: le-iff-add)
  - **with** run p p' show ?thesis by (auto elim: A5)

- **next**
  - **case** False
    - **hence** \( m \leq n \) by auto
    - **then obtain** \( k \) where \( n = m+k \) by (auto simp add: le-iff-add)
    - **with** run p p' **have** \( w = v \) by (auto elim: A5)
    - **thus** ?thesis ..

**qed**

### 5.6 Proof of Termination

We now show that every process must eventually decide.

The idea of the proof is to observe that the communication predicate guarantees the existence of two uniform rounds where every process hears from the same two-thirds majority of processes. The first such round serves to ensure that all \( x \) fields hold the same value, the second round copies that
value into all decision fields.

Lemma A2 is instrumental in this proof.

**Theorem OTR-termination:**

- **Assumes** \( \text{run: HORun OTR-M \varrho HOs} \)
  - **and** \( \text{commG: HOcommGlobal OTR-M HOs} \)
- **Shows** \( \exists r, v. \text{decide (\varrho r p) = Some v} \)

**Proof**

- From \( \text{commG} \) obtain \( r0 \) II where
  - \( pi: \forall q. \text{HOs r0 q = II and pic: card II > (2N) div 3} \)
  - By \( \text{(auto simp: OTR-HOMachine-def OTR-commGlobal-def)} \)
- Let \( \forall \text{msgs q r = HORcvdMsgs OTR-M r q (HOs r q)} \)

- From \( \text{run pi} \) have \( \forall p q. \text{msgs q r0 = msgs p r0} \)
  - By \( \text{(auto simp: HORun-eq OTR-HOMachine-def HORcvdMsgs-def OTR-sendMsg-def)} \)
- Then obtain \( \mu \) where \( \forall q. \text{msgs q r0 = \mu by auto} \)

Furthermore

- From \( pi \) \( \text{pic have} \forall p q. \text{(2N) div 3 < card \{ q. \text{msgs q r0 = None} \}} \)
  - By \( \text{(auto simp: HORun-eq HOnextConfig-eq HORcvdMsgs-def)} \)
- With \( \text{run have} \forall q. x (\varrho (\text{Suc r0}) q) = \text{Min (v . MFR (msgs q r0)) v} \)
  - By \( \text{(auto simp: HORun-eq HOnextConfig-eq OTR-HOMachine-def HOnextState-def OTR-nextState-def)} \)

Ultimately

- Have \( \forall q. x (\varrho (\text{Suc r0}) q) = \text{Min (Min MFR \mu v)} \) by auto
- Then obtain \( v \) where \( v \forall q. x (\varrho (\text{Suc r0}) q) = v \) by auto

Have \( P : \forall k. \forall q. x (\varrho (\text{Suc r0 + k}) q) = v \)

**Proof**

- Fix \( k \)
- Show \( \forall q. x (\varrho (\text{Suc r0 + k}) q) = v \)
- **Proof (induct \( k \))**
  - From \( v \) show \( \forall q. x (\varrho (\text{Suc r0 + 0}) q) = v \) by simp

- Next
  - Fix \( k \)
  - Assume \( \text{ih: \forall q. x (\varrho (\text{Suc r0 + k}) q) = v} \)
  - Show \( \forall q. x (\varrho (\text{Suc r0 + Suc k}) q) = v \)
  - **Proof (cases (2N) div 3 < card \{ p . \text{msgs q (Suc r0 + k) p = None} \})**
    - Case True
      - Have \( \text{N > 0} \) by \( \text{(rule finite-UNIV-card-ge-0 simp)} \)
      - With \( \text{ih} \)
    - Have \( (2N) \text{ div 3 < card \{ p . x (\varrho (\text{Suc r0 + k}) p) = v \}} \) by auto
      - With \( \text{True run show \?thesis by (auto elim: A2)} \)
  - Next
    - Case False
      - With \( \text{run \ ih show \?thesis by (auto dest: nextState-change)} \)
      qed
      qed

42
qed

5.7 One-Third Rule Solves Consensus

Summing up, all (coarse-grained) runs of One-Third Rule for HO collections that satisfy the communication predicate satisfy the Consensus property.

theorem OTR-consensus:
  assumes run: HORun OTR-M rho HOs and commG: HOcommGlobal OTR-M HOs
  shows consensus (x ◦ (rho 0)) decide rho
    using OTR-integrity[OF run] OTR-agreement[OF run] OTR-termination[OF run commG]
    by (auto simp: consensus-def image-def)

By the reduction theorem, the correctness of the algorithm also follows for fine-grained runs of the algorithm. It would be much more tedious to establish this theorem directly.

theorem OTR-consensus-fg:
  assumes run: fg-run OTR-M rho HOs HOs (λr q. undefined) and commG: HOcommGlobal OTR-M HOs
  shows consensus (λp. x (state (rho 0) p)) decide (state ◦ rho)
    (is consensus ?inits - -)
  proof (rule local-property-reduction[OF run consensus-is-local])
    fix crun
assume crun: CSHORun OTR-M crun HOs HOs (λr q. undefined)
and init: crun 0 = state (rho 0)
from crun have HORun OTR-M crun HOs by (unfold HORun-def SHORun-def)
from this commG have consensus (x ◦ (crun 0)) decide crun by (rule OTR-consensus)
with init show consensus ?inits decide crun by (simp add: o-def)
qed

end
theory UvDefs
imports ..../HOModel
begin

6 Verification of the UniformVoting Consensus Algorithm

Algorithm UniformVoting is presented in [7]. It can be considered as a

deterministic version of Ben-Or’s well-known probabilistic Consensus algo-

rithm [2]. We formalize in Isabelle the correctness proof given in [7], using
the framework of theory HOModel.

6.1 Model of the Algorithm

We begin by introducing an anonymous type of processes of finite cardinality
that will instantiate the type variable ‘proc of the generic HO model.

typedecl Proc — the set of processes
axiomatization where Proc-finite: OFCLASS(Proc, finite-class)
instance Proc :: finite by (rule Proc-finite)

abbreviation
N ≡ card (UNIV::Proc set) — number of processes

The algorithm proceeds in phases of 2 rounds each (we call steps the in-

dividual rounds that constitute a phase). The following utility functions

compute the phase and step of a round, given the round number.

abbreviation nSteps ≡ 2

definition phase where phase (r::nat) ≡ r div nSteps

definition step where step (r::nat) ≡ r mod nSteps

The following record models the local state of a process.

record ‘val pstate =
x :: ‘val — current value held by process
vote :: ‘val option — value the process voted for, if any
decide :: ‘val option — value the process has decided on, if any
Possible messages sent during the execution of the algorithm, and characteristic predicates to distinguish types of messages.

```
datatype 'val msg =
    Val 'val
  | ValVote 'val 'val option
  | Null — dummy message in case nothing needs to be sent
```

```
definition isValVote where isValVote m ≡ ∃ z v. m = ValVote z v
```

```
definition isVal where isVal m ≡ ∃ v. m = Val v
```

Selector functions to retrieve components of messages. These functions have a meaningful result only when the message is of appropriate kind.

```
fun getvote where
    getvote (ValVote z v) = v
```

```
fun getval where
    getval (ValVote z v) = z
  | getval (Val z) = z
```

The $x$ field of the initial state is unconstrained, all other fields are initialized appropriately.

```
definition UV-initState where
    UV-initState p st ≡ (vote st = None) ∧ (decide st = None)
```

We separately define the transition predicates and the send functions for each step and later combine them to define the overall next-state relation.

```
definition msgRcvd where — processes from which some message was received
    msgRcvd (msgs :: Proc ↭ 'val msg) = { q . msgs q ≠ None}
```

```
definition smallestValRcvd where
    smallestValRcvd (msgs :: Proc ↭ ('val::linorder) msg) ≡
    Min { v. ∃ q. msgs q = Some (Val v) }
```

In step 0, each process sends its current $x$ value. It updates its $x$ field to the smallest value it has received. If the process has received the same value $v$ from all processes from which it has heard, it updates its $vote$ field to $v$.

```
definition send0 where
    send0 r p q st ≡ Val (x st)
```

```
definition next0 where
    next0 r p st (msgs :: Proc ↭ ('val::linorder) msg) st' ≡
    (∃ v. (∀ q ∈ msgRcvd msgs. msgs q = Some (Val v))
      ∧ st' = st ( q || vote := Some v, x := smallestValRcvd msgs () )
    ∨ ¬ (∃ v. ∀ q ∈ msgRcvd msgs. msgs q = Some (Val v))
    ∧ st' = st ( q || x := smallestValRcvd msgs )
```

45
In step 1, each process sends its current \( x \) and \( \text{vote} \) values.

**definition** \texttt{send1 where}

\( \text{send1 } r \ p \ q \ st \equiv \text{ValVote} \ (x \ st) \ (\text{vote} \ st) \)

**definition** \texttt{valVoteRcvd where}

— processes from which values and votes were received

\( \text{valVoteRcvd} \ (\text{msgs :: Proc} \to \ ('\text{val} \ \text{msg}) \equiv \{q \cdot \exists z \ v. \ \text{msgs} \ q = \text{Some} \ (\text{ValVote} \ z \ v)\} \)

**definition** \texttt{smallestValNoVoteRcvd where}

\( \text{smallestValNoVoteRcvd} \ (\text{msgs :: Proc} \to \ ('\text{val} \ \text{msg}) \equiv \text{Min} \ \{v. \exists q. \ \text{msgs} \ q = \text{Some} \ (\text{ValVote} \ v \ \text{None})\} \)

**definition** \texttt{someVoteRcvd where}

— set of processes from which some vote was received

\( \text{someVoteRcvd} \ (\text{msgs :: Proc} \to \ ('\text{val} \ \text{msg}) \equiv \{q. \ q \in \text{msgRcvd} \ \text{msgs} \ \wedge \ \text{isValVote} \ (\text{the} \ (\text{msgs} \ q)) \ \wedge \ \text{getvote} \ (\text{the} \ (\text{msgs} \ q)) \neq \text{None} \} \)

**definition** \texttt{identicalVoteRcvd where}

\( \text{identicalVoteRcvd} \ (\text{msgs :: Proc} \to \ ('\text{val} \ \text{msg}) \equiv \forall q. \ q \in \text{msgRcvd} \ \text{msgs}, \ \text{isValVote} \ (\text{the} \ (\text{msgs} \ q)) \ \wedge \ \text{getvote} \ (\text{the} \ (\text{msgs} \ q)) = \text{Some} \ v \)

**definition** \texttt{x-update where}

\( \text{x-update} \ st \ \text{msgs} \ st' \equiv (\exists q \in \text{someVoteRcvd} \ \text{msgs} \ . \ x \ st' = \text{the} \ (\text{getvote} \ (\text{the} \ (\text{msgs} \ q)))) \lor \text{someVoteRcvd} \ \text{msgs} = \{\} \ \wedge \ x \ st' = \text{smallestValNoVoteRcvd} \ \text{msgs} \)

**definition** \texttt{dec-update where}

\( \text{dec-update} \ st \ \text{msgs} \ st' \equiv (\exists v. \ \text{identicalVoteRcvd} \ \text{msgs} \ v \ \wedge \ \text{decide} \ st' = \text{Some} \ v) \lor \neg(\exists v. \ \text{identicalVoteRcvd} \ \text{msgs} \ v) \ \wedge \ \text{decide} \ st' = \text{decide} \ st \)

**definition** \texttt{next1 where}

\( \text{next1 } r \ p \ st \ \text{msgs} \ st' \equiv \text{x-update} \ st \ \text{msgs} \ st' \land \text{dec-update} \ st \ \text{msgs} \ st' \land \ \text{vote} \ st' = \text{None} \)

The overall send function and next-state relation are simply obtained as the composition of the individual relations defined above.

**definition** \texttt{UV-sendMsg where}

\( \text{UV-sendMsg} \ (r :: \text{nat}) \equiv \text{if step } r = 0 \ \text{then} \ 	ext{send0} \ r \ \text{else} \ 	ext{send1} \ r \)

**definition** \texttt{UV-nextState where}

\( \text{UV-nextState} \ r \equiv \text{if step } r = 0 \ \text{then} \ 	ext{next0} \ r \ \text{else} \ 	ext{next1} \ r \)
6.2 Communication Predicate for *UniformVoting*

We now define the communication predicate for the *UniformVoting* algorithm to be correct.

The round-by-round predicate requires that for any two processes there is always one process heard by both of them. In other words, no “split rounds” occur during the execution of the algorithm [7]. Note that in particular, heard-of sets are never empty.

**definition** UV-commPerRd where

\[ \text{UV-commPerRd} \ HOrs \equiv \forall p \ q. \ \exists \ pq \in \ HOrs \ p \cap \ HOrs \ q \]

The global predicate requires the existence of a (space-)uniform round during which the heard-of sets of all processes are equal. (Observe that [7] requires infinitely many uniform rounds, but the correctness proof uses just one such round.)

**definition** UV-commGlobal where

\[ \text{UV-commGlobal} \ HOs \equiv \exists r. \ \forall p \ q. \ HOs r p = HOs r q \]

6.3 The *UniformVoting* Heard-Of Machine

We now define the HO machine for *Uniform Voting* by assembling the algorithm definition and its communication predicate. Notice that the coordinator arguments for the initialization and transition functions are unused since *UniformVoting* is not a coordinated algorithm.

**definition** UV-HOMachine where

\[ \text{UV-HOMachine} = (\lambda \ p \ st \ crd. \ UV-initState \ p \ st), \text{sendMsg} = UV-sendMsg, \]

\[ CnextState = (\lambda r p st msgs crd st'. \ UV-nextState \ r p st msgs st'), \]

\[ \text{HOcommPerRd} = UV-commPerRd, \]

\[ \text{HOcommGlobal} = UV-commGlobal \]

**abbreviation**

\[ \text{UV-M} \equiv (\text{UV-HOMachine}::(\text{Proc}, \ 'val':\text{linorder pstate}, \ 'val msg) \ HOMachine) \]

end
theory UvProof
imports UvDefs ../Reduction
begin

6.4 Preliminary Lemmas

At any round, given two processes \( p \) and \( q \), there is always some process which is heard by both of them, and from which \( p \) and \( q \) have received the same message.
lemma some-common-msg:
  assumes HOcommPerRd UV-M (HOs r)
  shows \( \exists pq. \ pq \in \text{msgRcvd} \HOrcvdMsgs UV-M r p (\text{HOs } r) (\rho r)) \)
  \( \land \ pq \in \text{msgRcvd} \HOrcvdMsgs UV-M r q (\text{HOs } r) (\rho r) \)
  \( \land (\HOrcvdMsgs UV-M r p (\text{HOs } r p) (\rho r)) \ pq \)
  \( = (\HOrcvdMsgs UV-M r q (\text{HOs } r q) (\rho r)) \ pq \)
  using assms
  by (auto simp: UV-HOMachine-def UV-commPerRd-def HOrcvdMsgs-def
       UV-sendMsg-def send0-def send1-def msgRcvd-def)

When executing step 0, the minimum received value is always well defined.

lemma minval-step0:
  assumes com: HOcommPerRd UV-M (HOs r) and s0: step r = 0
  shows smallestValRcvd (HOrcvdMsgs UV-M r q (HOs r) (\rho r)) \in \{ v. \exists p. (\HOrcvdMsgs UV-M r q (\text{HOs } r q) (\rho r)) \ p = Some (Val v)\}
    (is smallestValRcvd ?msgs \in {?vals})
  unfolding smallestValRcvd-def proof (rule Min-in)
  have ?vals \subseteq \text{getval}' ((\text{the } ?msgs)' (\text{HOs } r q))
    by (auto simp: HOrcvdMsgs-def image-def)
  thus finite ?vals by (auto simp: finite-subset)
next
from some-common-msg[of HOs, OF com]
obtain p where p \in msgRcvd ?msgs by blast
with s0 show ?vals \not= {} by (auto simp: msgRcvd-def HOrcvdMsgs-def UV-HOMachine-def
UV-sendMsg-def send0-def)
qed

When executing step 1 and no vote has been received, the minimum among values received in messages carrying no vote is well defined.

lemma minval-step1:
  assumes com: HOcommPerRd UV-M (HOs r) and s1: step r \not= 0
  and nov: someVoteRcvd (HOrcvdMsgs UV-M r q (HOs r) (\rho r)) = {}
  shows smallestValNoVoteRcvd (HOrcvdMsgs UV-M r q (HOs r) (\rho r)) \in \{ v. \exists p. (\HOrcvdMsgs UV-M r q (\text{HOs } r q) (\rho r)) \ p = Some (ValVote v None)\}
    (is smallestValNoVoteRcvd ?msgs \in {?vals})
  unfolding smallestValNoVoteRcvd-def proof (rule Min-in)
  have ?vals \subseteq \text{getval}' ((\text{the } ?msgs)' (\text{HOs } r q))
    by (auto simp: HOrcvdMsgs-def image-def)
  thus finite ?vals by (auto simp: finite-subset)
next
from some-common-msg[of HOs, OF com]
obtain p where p \in msgRcvd ?msgs by blast
with s1 nov show ?vals \not= {} by (auto simp: msgRcvd-def HOrcvdMsgs-def someVoteRcvd-def isValVote-def
UV-HOMachine-def UV-sendMsg-def send1-def)
qed
The `vote` field is reset every time a new phase begins.

**Lemma** reset-vote:
- **Assumes** `run: HORun UV-M rho HOs and s0: step r' = 0`
- **Shows** `vote(rho r' p) = None`
- **Proof** (cases `r'`)
  - **Assume** `r' = 0`
  - **With** `run` show `?thesis`
    - (auto simp: UV-HOMachine-def HORun-eq HOinitConfig-eq
      initState-def UV-initState-def)

**Next**
- **Fix** `r`
- **Assume** `sucr: r' = Suc r`
- **From** `run`
  - **Have** `nxt: nextState UV-M r p (rho r p)`
    - `(HOrcvdMsgs UV-M r p (HOs r p) (rho r))`
    - `(rho (Suc r) p)`
    - (auto simp: UV-HOMachine-def HORun-eq HOnextConfig-eq nextState-def)
  - **From** `s0 suc` **Have** `step r = 1` by (auto simp: step-def mod-Suc)
  - **With** `nxt suc` show `?thesis`
    - (auto simp: UV-HOMachine-def nextState-def UV-nextState-def next1-def)
- **QED**

Processes only vote for the value they hold in their `x` field.

**Lemma** x-vote-eq:
- **Assumes** `run: HORun UV-M rho HOs`
  - **And** `com: \( \forall r. \) HOr CommPerRd UV-M (HOs r)`
  - **And** `vote: vote(rho r p) = Some v`
- **Shows** `v = x (rho r p)`
- **Proof** (cases `r`)
  - **Case** `0`
    - **With** `run vote` **Have** `nxt0 r' p (rho r p)`
    - (auto simp: UV-HOMachine-def HORun-eq HOinitConfig-eq
      initState-def UV-initState-def)
  - **Fix** `r'`
    - **Assume** `r: r = Suc r'`
    - **Let** `msgs = HOrcvdMsgs UV-M r' p (HOs r' p) (rho r')`
    - **From** `run` **Have** `nextState UV-M r' p (rho r' p) ?msgs (rho (Suc r') p)`
      - (auto simp: HORun-eq HOnextConfig-eq nextState-def)
    - **With** `vote r` **Have** `nxt0: nxt0 r' p (rho r' p) ?msgs (rho r p) and s0: step r' = 0`
      - (auto simp: nextState-def UV-HOMachine-def UV-nextState-def next1-def)
    - **From** `run s0` **Have** `vote(rho r' p) = None` by (rule reset-vote)
    - **With** `vote nxt0`
      - **Have** `idv: \( \forall q \in msgRcvd ?msgs. \) ?msgs q = Some (Val v)`
      - **And** `x: x (rho r p) = smallestValRcvd ?msgs`
        - (auto simp: nxt0-def)
    - **Moreover**
      - **From** `com` **Obtain** `q where q \in msgRcvd ?msgs`
by \((\text{force dest: some-common-msg})\)

with \(\text{idv have} \{ x . \exists q q. \ ?msgs q q = \text{Some} \ (\text{Val} \ x) \} = \{v\}\)

by \((\text{auto simp: msgRcvd-def})\)

hence \(\text{smallestValRcvd} \ ?msgs = v\)

by \((\text{auto simp: smallestValRcvd-def})\)

ultimately

show \(?\text{thesis by simp}\)

qed

6.5 Proof of Irrevocability, Agreement and Integrity

A decision can only be taken in the second round of a phase.

**lemma decide-step:**

assumes \(\text{run: HORun} \ UV-M \ \rho HOs\)

and \(\text{decide: decide} \ (\rho \ (\text{Suc} \ r) \ p) \neq \text{decide} \ (\rho \ r \ p)\)

shows \(\text{step} \ r = 1\)

**proof –**

let \(?msgs = \text{HOrcvdMsgs} \ UV-M \ r \ p \ (\text{HOs} \ r \ p) \ (\rho \ r)\)

from \(\text{run have} \ \text{nextState} \ UV-M \ r \ p \ (\rho \ r \ p) \ ?msgs \ (\rho \ (\text{Suc} \ r) \ p)\)

by \((\text{auto simp: HORun-eq HOnextConfig-eq nextState-def})\)

with \(\text{decide show} \ ?\text{thesis}\)

by \((\text{auto simp: nextState-def UV-HOMachine-def UV-nextState-def next0-def step-def})\)

qed

No process ever decides \(\text{None}\).

**lemma decide-nonnull:**

assumes \(\text{run: HORun} \ UV-M \ \rho HOs\)

and \(\text{decide: decide} \ (\rho \ (\text{Suc} \ r) \ p) \neq \text{decide} \ (\rho \ r \ p)\)

shows \(\text{decide} \ (\rho \ (\text{Suc} \ r) \ p) \neq \text{None}\)

**proof –**

let \(?msgs = \text{HOrcvdMsgs} \ UV-M \ r \ p \ (\text{HOs} \ r \ p) \ (\rho \ r)\)

from \(\text{assms have} \ s1: \text{step} \ r = 1 \ \text{by} \ ((\text{rule decide-step})\)

with \(\text{run have} \ \text{next1} \ r \ p \ (\rho \ r \ p) \ ?msgs \ (\rho \ (\text{Suc} \ r) \ p)\)

by \((\text{auto simp: UV-HOMachine-def HORun-eq HOnextConfig-eq nextState-def UV-nextState-def})\)

with \(\text{decide show} \ ?\text{thesis}\)

by \((\text{auto simp: next1-def dec-update-def})\)

qed

If some process \(p\) votes for \(v\) at some round \(r\), then any message that \(p\) received in \(r\) was holding \(v\) as a value.

**lemma msgs-unanimity:**

assumes \(\text{run: HORun} \ UV-M \ \rho HOs\)

and \(\text{vote: vote} \ (\rho \ (\text{Suc} \ r) \ p) = \text{Some} \ v\)

and \(q : q \in \text{msgRcvd} \ (\text{HOrcvdMsgs} \ UV-M \ r \ p \ (\text{HOs} \ r \ p) \ (\rho \ r))\)

\(\text{(is -} \in \text{msgRcvd \ ?msgs)}\)

shows \(\text{getval} \ (\text{the} \ (\ ?msgs \ q)) = v\)
proof

have \( s_0 \): step \( r = 0 \)
proof (rule ccontr)
  assume \( \text{step } r \neq 0 \)
  hence \( \text{step } (\text{Suc } r) = 0 \) by (simp add: step-def mod-Suc)
with \text{run} vote show False by (auto simp: reset-vote)
qed

with \text{run} have \text{novote}: vote (\text{rho } r \text{ p}) = \text{None} by (auto simp: reset-vote)
from \text{run} have \text{nextState UV-M } r \text{ p } (\text{rho } r \text{ p}) \ ?msgs (\text{rho } (\text{Suc } r) \text{ p})
  by (auto simp: HORun-eq HOnextConfig-eq nextState-def)
with \text{s0} have \text{nxt}: next0 \text{ r p } (\text{rho } r \text{ p}) \ ?msgs (\text{rho } (\text{Suc } r) \text{ p})
  by (auto simp: UV-HOMachine-def nextState-def UV-nextState-def)
with \text{novote} vote q show \text{thesis} by (auto simp: next0-def)
qed

Any two processes can only vote for the same value.

lemma \text{vote-agreement}:
assumes \text{run}: HORun UV-M \text{ rho } \text{HOs}
  and \text{com}: \( \forall r. \text{HOcommPerRd } \text{UV-M } (\text{HOs } r) \)
and \( p \): vote (\text{rho } r \text{ p}) = \text{Some } v
and \( q \): vote (\text{rho } r \text{ q}) = \text{Some } w
shows \( v = w \)
proof (cases \text{r})
  case \text{0}
with \text{run} \text{p} show \text{thesis} — no votes in initial state
  by (auto simp: UV-HOMachine-def HORun-eq HOinitConfig-eq initState-def UV-initState-def)
next
fix \text{r}'
assume \text{r}: \text{r} = \text{Suc } r'
let \(?msgs \text{ p} = \text{HORcvdMsgs } \text{UV-M } r' \text{ p } (\text{HOs } r')\) (\text{rho } r')
from \text{com} obtain \text{pq}
  where \(?msgs \text{ p} \text{ pq} = \text{?msgs } q \text{ pq} \)
    and \text{smq}: \text{pq} \in \text{msgRcvd } (\text{?msgs } p) \text{ and } \text{smq}: \text{pq} \in \text{msgRcvd } (\text{?msgs } q)
  by (force dest: some-common-msg)
moreover
from \text{run} \text{p} \text{ smp} \text{ r} have getval (the (\text{?msgs } \text{ p} \text{ pq})) = v
  by (simp add: msgs-unanimity)
moreover
from \text{run} \text{q} \text{ smq} \text{ r} have getval (the (\text{?msgs } \text{ q} \text{ pq})) = w
  by (simp add: msgs-unanimity)
ultimately
show \text{thesis} by simp
qed

If a process decides value \( v \) then all processes must have \( v \) in their \text{x} fields.

lemma \text{decide-equals-x}:
assumes \text{run}: HORun UV-M \text{ rho } \text{HOs}
  and \text{com}: \( \forall r. \text{HOcommPerRd } \text{UV-M } (\text{HOs } r) \)
and decide: \(\text{decide}(\rho (\text{Suc } r) p) \neq \text{decide}(\rho r p)\)

and decval: \(\text{decide}(\rho (\text{Suc } r) p) = \text{Some } v\)

shows \(\rho (\text{Suc } r) q = v\)

proof –

let \(?msgs p' = \text{HOrcvdMsgs } UV-M r p' (\text{HOs } r p') (\rho r)\)

from run decide have \(s_1: \text{step } r = 1\) by (rule decide-step)

from run have nextState \(UV-M r p (\rho r p) (?msgs p) (\rho (\text{Suc } r) p)\) by (auto simp: HORun-eq HOnextConfig-eq nextState-def)

with \(s_1\) have \(\text{nxtp: } \text{next1 } r p (\rho r p) (?msgs p) (\rho (\text{Suc } r) p)\) by (auto simp: UV-HOMachine-def nextState-def UV-nextState-def)

from run have nextState \(UV-M r q (\rho r q) (?msgs q) (\rho (\text{Suc } r) q)\) by (auto simp: HORun-eq HOnextConfig-eq nextState-def)

with \(s_1\) have \(\text{nxtq: } \text{next1 } r q (\rho r q) (?msgs q) (\rho (\text{Suc } r) q)\) by (auto simp: UV-HOMachine-def nextState-def UV-nextState-def)

obtain \(pq\) where \(pq: pq \in \text{msgRcvd } ?msgs\) \(pq \in \text{msgRcvd } ?msgs\) \(pq = (?msgs q) pq\)

by (force dest: some-common-msg)

with decide decidval have \(\text{vote: } \text{isValVote } (\rho (\text{Suc } r) q)\) \(\text{getvote } (\text{the } (?msgs p pq)) = \text{Some } v\)

by (auto simp: next1-def dec-update-def identicalVoteRcvd-def)

with \(\text{nxtq pq}\) obtain \(q'\) where \(q': q' \in \text{someVoteRcvd } ?msgs\)

\(x (\rho (\text{Suc } r) q) = \text{the } (\text{getvote } (\text{the } (?msgs q q')))\)

by (auto simp: next1-def x-update-def someVoteRcvd-def)

with \(s_1 pq\) show \(?thesis\) by (auto simp: UV-HOMachine-def UV-sendMsg-def send1-def someVoteRcvd-def msgRcvd-def vote-agreement[OF run com])

qed

If at some point all processes hold value \(v\) in their \(x\) fields, then this will still be the case at the next step.

lemma same-x-stable:

assumes run: \(\text{HORun } UV-M \rho \text{ HOs}\)

and comm: \(\forall r. \text{HOcommPerRd } UV-M (\text{HOs } r)\)

and \(x: \forall p. x (\rho r p) = v\)

shows \(x (\rho (\text{Suc } r) q) = v\)

proof –

let \(?msgs = \text{HOrcvdMsgs } UV-M r q (\text{HOs } r q) (\rho r)\)

from comm obtain \(p\) where \(p: p \in \text{msgRcvd } ?msgs\)

by (force dest: some-common-msg)

from run have nextState \(UV-M r q (\rho r q) (?msgs (\rho (\text{Suc } r) q)\)

by (auto simp: HORun-eq HOnextConfig-eq nextState-def)

hence \(\text{nxt0 } r q (\rho r q) (?msgs (\rho (\text{Suc } r) q) \land \text{step } r = 0\)

\(\lor \text{nxt1 } r q (\rho r q) (?msgs (\rho (\text{Suc } r) q) \land \text{step } r \neq 0\)

(is \(?nxt0 \lor \text{nxt1}\) )

by (auto simp: UV-HOMachine-def nextState-def UV-nextState-def)

52
thus \(?thesis\)

proof

assume \(nxt0\): \(?nxt0\)

hence \(x\ (\rho\ (\text{Suc } r)\ q) = \text{smallestValRcvd } \?msgs\)

by (auto simp: \(\text{next0-def}\))

moreover

from \(nxt0\) \(x\) have \(\forall p \in \text{msgRcvd } \?msgs. \ ?msgs p = \text{Some } (\text{Val } v)\)

by (auto simp: \(\text{UV-HOMachine-def } \text{HOrcvdMsgs-def } \text{UV-sendMsg-def } \text{msgRcvd-def send0-def}\))

from this \(p\) have \(\{x . \exists p. \ ?msgs p = \text{Some } (\text{Val } x)\} = \{v\}\)

by (auto simp: \(\text{msgRcvd-def } \text{msgRcvd-def}\))

hence \(\text{smallestValRcvd } \?msgs = v\)

by (auto simp: \(\text{smallestValRcvd-def}\))

ultimately

show \(?thesis\) by simp

next

assume \(nxt1\): \(?nxt1\)

show \(?thesis\)

proof (cases \(\text{someVoteRcvd } \?msgs = \{\}\))

case True

with \(nxt1\) \(x\) True

have \(\forall p \in \text{msgRcvd } \?msgs. \ ?msgs p = \text{Some } (\text{ValVote } v \text{ None})\)

by (auto simp: \(\text{UV-HOMachine-def } \text{HOrcvdMsgs-def } \text{UV-sendMsg-def } \text{msgRcvd-def send1-def someVoteRcvd-def isValVote-def}\))

from this \(p\) have \(\{x . \exists p. \ ?msgs p = \text{Some } (\text{ValVote } x \text{ None})\} = \{v\}\)

by (auto simp: \(\text{msgRcvd-def}\))

hence \(\text{smallestValNoVoteRcvd } \?msgs = v\)

by (auto simp: \(\text{smallestValNoVoteRcvd-def}\))

ultimately show \(?thesis\) by simp

next

case False

with \(nxt1\) obtain \(p'\ v'\ where\)

\(p'\in \text{msgRcvd } \?msgs\ \text{isValVote } \text{the } (\?msgs p')\)

getvote (the (\?msgs p')) = \text{Some } v' (\rho\ (\text{Suc } r)\ q) = v'

by (auto simp: \(\text{someVoteRcvd-def next1-def x-update-def}\))

with \(nxt1\) \(x\ (\rho\ (\text{Suc } r)\ q) = x\ (\rho\ r\ p')\)

by (auto simp: \(\text{UV-HOMachine-def } \text{HOrcvdMsgs-def } \text{UV-sendMsg-def}\))

\(\text{msgRcvd-def send1-def isValVote-def x-vote-eq[OF run comm]}\)

with \(x\) show \(?thesis\) by auto

qed

Combining the last two lemmas, it follows that as soon as some process decides value \(v\), all processes hold \(v\) in their \(x\) fields.
lemma safety-argument:
  assumes run: HORun UV-M rho HOs
  and com: ∀ r. HOcommPerRd UV-M (HOs r)
  and decide: decide (rho (Suc r) p) ≠ decide (rho r p) 
  and decval: decide (rho (Suc r) p) = Some v
  shows x (rho (Suc r+k) q) = v
proof (induct k arbitrary: q)
  fix q
  from decide-equals-x[OF assms] show x (rho (Suc r + 0) q) = v by simp
next
  fix k q
  assume \( \forall q. x (\rho (Suc r+k) q) = v \)
  with run com show x (rho (Suc r + Suc k) q) = v
  by (auto dest: same-x-stable)
qed

Any process that holds a non-null decision value has made a decision some-time in the past.

lemma decided-then-past-decision:
  assumes run: HORun UV-M rho HOs
  and dec: decide (rho n p) = Some v
  shows \( \exists m < n. \) decide (rho (Suc m) p) ≠ decide (rho m p) 
  \( \land \) decide (rho (Suc m) p) = Some v
proof –
  let \( ?\text{dec} \) k = decide (rho k p)
  have \( \forall m < n. \) ?dec (Suc m) \( \neq ?\text{dec} \) (Suc m) \( \neq \) Some v
  \( \rightarrow ?\text{dec} \) n \( \neq \) Some v
  (is \( ?P n \) is \( ?A n \rightarrow - \))
proof (induct n)
  from run show \( ?P \) 0
  by (auto simp: HORun-eq UV-HOMachine-def HOinitConfig-eq
  initState-def UV-initState-def)
next
  fix n
  assume ih: \( ?P \) n thus \( ?P \) (Suc n) by force
qed
with dec show \( ?\text{thesis} \) by auto
qed

We can now prove the safety properties of the algorithm, and start with proving Integrity.

lemma x-values-initial:
  assumes run: HORun UV-M rho HOs
  and com: \( \forall r. \) HOcommPerRd UV-M (HOs r)
  shows \( \exists q. x (\rho r p) = x (\rho 0 q) \)
proof (induct r arbitrary: p)
  fix p
  show \( \exists q. x (\rho 0 p) = x (\rho 0 q) \) by auto
next
fix r p
assume ih: ∃ q. x (ρ r p) = x (ρ 0 q)
let run have nextState UV-M r p (ρ r p) ?msgs (ρ (Suc r) p)
  by (auto simp: HORun-eq HOnextConfig-eq nextState-def)
hence nextState r p (ρ r p) ?msgs (ρ (Suc r) p) ∧ step r = 0
  ∨ nextState r p (ρ r p) ?msgs (ρ (Suc r) p) ∧ step r ≠ 0
  (is ?nxt0 ∨ ?nxt1)
  by (auto simp: UV-HOMachine-def nextState-def UV-nextState-def)
thus ∃ q. x (ρ (Suc r) p) = x (ρ 0 q)
proof
  assume ?nxt0: ?nxt0
  hence x (ρ (Suc r) p) = smallestValRcvd ?msgs
    by (auto simp: next0-def)
  also with com ?nxt0 have ... ∈ {v . ∃ q. ?msgs q = Some (Val v)}
    by (intro minval-step0) auto
  also with nextState True have ... = { x (ρ r q) | q . q ∈ msgRcvd ?msgs }
    by (auto simp: UV-HOMachine-def HOrcvdMsgs-def UV-sendMsg-def
      msgRcvd-def send0-def)
  finally obtain q where x (ρ (Suc r) p) = x (ρ r q)
    by auto
  with ih show ?thesis by auto
next
  assume ?nxt1: ?nxt1
  show ?thesis
  proof (cases someVoteRcvd ?msgs = {})
    case True
    with nextState True have x (ρ (Suc r) p) = smallestValNoVoteRcvd ?msgs
      by (auto simp: next1-def x-update-def)
    also with com ?nxt1 True have ... ∈ {v . ∃ q. ?msgs q = Some (ValVote v None)}
      by (intro minval-step1) auto
    also with nextState True have ... = { x (ρ r q) | q . q ∈ msgRcvd ?msgs }
      by (auto simp: UV-HOMachine-def HOrcvdMsgs-def UV-sendMsg-def
        someVoteRcvd-def isValVote-def msgRcvd-def send1-def)
    finally obtain q where x (ρ (Suc r) p) = x (ρ r q) by auto
    with ih show ?thesis by auto
next
  case False
  with nextState False obtain q where
    q ∈ someVoteRcvd ?msgs
    x (ρ (Suc r) p) = the (getvote (the (?msgs q)))
    by (auto simp: next1-def x-update-def)
  with nextState False have vote (ρ r q) = Some (x (ρ (Suc r) p))
    by (auto simp: UV-HOMachine-def HOrcvdMsgs-def UV-sendMsg-def
      someVoteRcvd-def isValVote-def msgRcvd-def send1-def)
  with run com have x (ρ (Suc r) p) = x (ρ r q)
    by (rule x-vote-eq)
  with ih show ?thesis by auto
theorem uv-integrity:
assumes \( \text{run: HORun UV-M \rho HOs} \)
and \( \text{com: \forall r. HOcommPerRd UV-M (HOs r)} \)
and \( \text{dec: decide (\rho r p) = Some v} \)
shows \( \exists q. v = x (\rho 0 q) \)
proof –
from \( \text{run \ dec \ obtain \ k \ where} \)
\( \text{decide (\rho (Suc k) p) \neq decide (\rho k p)} \)
\( \text{decide (\rho (Suc k) p) = Some v} \)
by (auto dest: decided-then-post-decision)
with \( \text{run \ com \ have} \ x (\rho (Suc k) p) = v \)
by (rule decide-equals-x)
with \( \text{run \ com \ show \ ?thesis} \)
by (auto dest: x-values-initial)
qed

We now turn to Agreement.

lemma two-decisions-agree:
assumes \( \text{run: HORun UV-M \rho HOs} \)
and \( \text{com: \forall r. HOcommPerRd UV-M (HOs r)} \)
and \( \text{decidep: decide (\rho (Suc r) p) \neq decide (\rho r p)} \)
and \( \text{decvalp: decide (\rho (Suc r) p) = Some v} \)
and \( \text{decideq: decide (\rho (Suc (r+k)) q) \neq decide (\rho (r+k) q)} \)
and \( \text{decvalq: decide (\rho (Suc (r+k)) q) = Some w} \)
shows \( v = w \)
proof –
from \( \text{run \ com \ decidep \ decvalp \ have} \ x (\rho (Suc r+k) q) = v \)
by (rule safety-argument)
moreover
from \( \text{run \ com \ decideq \ decvalq \ have} \ x (\rho (Suc (r+k)) q) = w \)
by (rule decide-equals-x)
ultimately
show \( \text{?thesis by simp} \)
qed

theorem uv-agreement:
assumes \( \text{run: HORun UV-M \rho HOs} \)
and \( \text{com: \forall r. HOcommPerRd UV-M (HOs r)} \)
and \( \text{p: decide (\rho m p) = Some v} \)
and \( \text{q: decide (\rho n q) = Some w} \)
shows \( v = w \)
proof –
from \( \text{run \ p \ obtain \ k \ where} \)
\( k: \text{decide (\rho (Suc k) p) \neq decide (\rho k p)} \)
\( \text{decide (\rho (Suc k) p) = Some v} \)
by (auto dest: decided-then-past-decision)

from run q obtain l where
l: decide (rho (Suc l) q) ≠ decide (rho l q)
decide (rho (Suc l) q) = Some w
by (auto dest: decided-then-past-decision)
show ?thesis
proof (cases k ≤ l)
case True
then obtain m where m: l = k+m by (auto simp: le-iff-add)
from run com k l m show ?thesis by (blast dest: two-decisions-agree)
next
case False
hence l ≤ k by simp
then obtain m where m: k = l+m by (auto simp: le-iff-add)
from run com k l m show ?thesis by (blast dest: two-decisions-agree)
qed
qed

Irrevocability is a consequence of Agreement and the fact that no process
can decide None.

theorem uv-irrevocability:
assumes run: HORun UV-M rho HOs
and com: ∀ r. HOrunCommPerRd UV-M (HOs r)
and p: decide (rho m p) = Some v
shows decide (rho (m+n) p) = Some v
proof (induct n)
from p show decide (rho (m+0) p) = Some v by simp
next
fix n
assume ih: decide (rho (m+n) p) = Some v
show decide (rho (m + Suc n) p) = Some v
proof (rule classical)
assume ¬ ?thesis
with run ih obtain w where w: decide (rho (m + Suc n) p) = Some w
by (auto dest!: decide-nonnull)
with p have w = v by (auto simp: uv-agreement[OF run com])
with w show ?thesis by simp
qed
qed

6.6 Proof of Termination

Two processes having the same Heard-Of set at some round will hold the
same value in their x variable at the next round.

lemma hoeq-xeq:
assumes run: HORun UV-M rho HOs
and com: ∀ r. HOrunCommPerRd UV-M (HOs r)
and hoeq: HOs r p = HOs r q
show $x (\rho (\text{Suc } r) p) = x (\rho (\text{Suc } r) q)$

proof –

let $?msgs p = \text{HOrcvdMsgs UV-M r p (HOs r p) (rho r)}$

from hoeq have msgeq: $?msgs p = $?msgs q

by (auto simp: UV-HOMachine-def HOrcvdMsgs-def UV-sendMsg-def
    send0-def send1-def)

show $?thesis

proof (cases step r = 0)

case True

with run

have $\forall p. \text{next0 r p (rho r p) ( msgs p) (rho (Suc r) p) (is p. ?nxt0 p)}$

by (force simp: UV-HOMachine-def HORun-eq HOnextConfig-eq
    nextState-def UV-nextState-def)

hence $?nxt0 p ?nxt0 q by auto

with msgeq show $?thesis by (auto simp: next0-def)

next

assume stp: step $r \neq 0$

with run

have $\forall p. \text{next1 r p (rho r p) (msgs p) (rho (Suc r) p) (is p. ?nxt1 p)}$

by (force simp: UV-HOMachine-def HORun-eq HOnextConfig-eq
    nextState-def UV-nextState-def)

hence $x\text{-update (rho r p) (msgs p) (rho (Suc r) p)}$

$x\text{-update (rho r q) (msgs q) (rho (Suc r) q)}$

by (auto simp: next1-def)

with msgeq have

$x'': x\text{-update (rho r p) (msgs p) (rho (Suc r) p)}$

$x\text{-update (rho r q) (msgs p) (rho (Suc r) q)}$

by auto

show $?thesis

proof (cases someVoteRcvd (?msgs p) = {})

  case True

    with $x'\text{ show } ?thesis$

    by (auto simp: x-update-def)

next

  case False

  with $x'\text{ stp obtain qp qq where}$

  vote (rho r qp) = Some ($x (\rho (\text{Suc } r) p))$ and

  vote (rho r qq) = Some ($x (\rho (\text{Suc } r) q))$

  by (force simp: UV-HOMachine-def HOrcvdMsgs-def UV-sendMsg-def
    x-update-def someVoteRcvd-def isValVote-def
    msgRcvd-def send1-def)

  with run com show $?thesis by (rule vote-agreement)$

qed

qed

We now prove that UniformVoting terminates.

theorem uv-termination:
assumes run: HORun UV-M rho HOs
    and commR: ∀ r. HOcommPerRd UV-M (HOs r)
    and commG: HOcommGlobal UV-M HOs
shows ∃ r v. decide (rho r p) = Some v
proof

First obtain a round where all x values agree.

from commG obtain r0 where r0: ∀ q. HOs r0 q = HOs r0 p
    by (force simp: UV-HOMachine-def UV-commGlobal-def)
let ?v = x (rho (Suc r0) p)
from run commR r0 have xs: ∀ q. x (rho r0 q) = ?v
    by (auto dest: hoeq-xeq)

Now obtain a round where all votes agree.

define r′ where r′ = (if step (Suc r0) = 0 then Suc r0 else Suc (Suc r0))
have stp′: step r′ = 0
    by (simp add: r′-def step-def mod-Suc)
have x′: ∀ q. x (rho r′ q) = ?v
    proof (auto simp: r′-def)
    fix q
    from xs show x (rho (Suc r0) q) = ?v ...
next
    fix q
    from run commR xs show x (rho (Suc r0) q) = ?v
    by (rule same-x-stable)
qed
have vote′: ∀ q. vote (rho (Suc r′) q) = Some ?v
    proof
    fix q
    let ?msgs = HOrcvdMsgs UV-M r′ p (HOs r′ p) (rho r′)
    from run stp′ have next0 r′ q (rho r′ q) ?msgs (rho (Suc r′) q)
        by (force simp: UV-HOMachine-def HORun-eq HOnextConfig-eq
            nextState-def UV-nextState-def)
    moreover
    from stp′ x′ have ∀ q′ ∈ msgRcvd ?msgs. ?msgs q′ = Some (Val ?v)
        by (auto simp: UV-HOMachine-def HOrcvdMsgs-def UV-sendMsg-def
            send0-def msgRcvd-def)
    moreover
    from commR have msgRcvd ?msgs ≠ {}
        by (force dest: some-common-msg)
    ultimately
    show vote (rho (Suc r′) q) = Some ?v
        by (auto simp: next0-def)
    qed

At the subsequent round, process p will decide.

let ?r'' = Suc r′
let ?msgs'' = HOrcvdMsgs UV-M ?r'' p (HOs ?r'' p) (rho ?r'')
from stp′ have stp'': step ?r'' = 1
by (simp add: step-def mod-Suc)
with run have next1 ?r'' p (rho ?r'' p) ?msgs' (rho (Suc ?r'') p)
  by (auto simp: UV-HOMachine-def HORun-eq HOnextConfig-eq
       nextState-def UV-nextState-def)
moreover
from stp'' vote' have identicalVoteRecvd ?msgs' ?v
  by (auto simp: UV-HOMachine-def HOrcvdMsgs-def UV-sendMsg-def
       send1-def identicalVoteRecvd-def isValVote-def)
moreover
from commR have msgRecvd ?msgs' \neq {}
  by (force dest: some-common-msg)
ultimately
have decide (rho (Suc ?r'') p) = Some ?v
  by (force simp: next1-def dec-update-def identicalVoteRecvd-def
       msgRecvd-def isValVote-def)
thus ?thesis by blast
qed

6.7 UniformVoting Solves Consensus

Summing up, all (coarse-grained) runs of UniformVoting for HO collections that satisfy the communication predicate satisfy the Consensus property.

theorem uv-consensus:
  assumes run: HORun UV-M rho HOs
  and commR: \forall r. HOcommPerRd UV-M (HOs r)
  and commG: HOcommGlobal UV-M HOs
  shows consensus (x \circ (rho 0)) decide rho
  using assms unfolding consensus-def image-def
  by (auto elim: uv-integrity uv-agreement uv-termination)

By the reduction theorem, the correctness of the algorithm carries over to the fine-grained model of runs.

theorem uv-consensus-fg:
  assumes run: fg-run UV-M rho HOs HOs (\lambda r g. undefined)
  and commR: \forall r. HOcommPerRd UV-M (HOs r)
  and commG: HOcommGlobal UV-M HOs
  shows consensus (\lambda p. x (state (rho 0) p)) decide (state o rho)
  (is consensus ?inits - -)
  proof (rule local-property-reduction[OF run consensus-is-local])
  fix crun
  assume crun: CSHORun UV-M crun HOs HOs (\lambda r g. undefined)
  and init: crun 0 = state (rho 0)
  from crun have HORun UV-M crun HOs
    by (unfold HORun-def SHORun-def)
  from this commR commG have consensus (x \circ (crun 0)) decide crun
    by (rule uv-consensus)
7 Verification of the *LastVoting* Consensus Algorithm

The *LastVoting* algorithm can be considered as a representation of Lamport’s Paxos consensus algorithm [11] in the Heard-Of model. It is a coordinated algorithm designed to tolerate benign failures. Following [7], we formalize its proof of correctness in Isabelle, using the framework of theory *HOModel*.

7.1 Model of the Algorithm

We begin by introducing an anonymous type of processes of finite cardinality that will instantiate the type variable `proc` of the generic CHO model.

```isar
typedef Proc — the set of processes

axiomatization where Proc-finite: OFCLASS(Proc, finite-class)

instance Proc :: finite by (rule Proc-finite)

abbreviation
  \[ N \equiv \text{card} (\text{UNIV::Proc set}) \] — number of processes
```

The algorithm proceeds in *phases* of 4 rounds each (we call *steps* the individual rounds that constitute a phase). The following utility functions compute the phase and step of a round, given the round number.

```isar
definition phase where phase (\text{nat} r) \equiv r \div 4

definition step where step (\text{nat} r) \equiv r \mod 4

lemma phase-zero [simp]: phase 0 = 0
  by (simp add: phase-def)

lemma step-zero [simp]: step 0 = 0
  by (simp add: step-def)

lemma phase-step: (phase r * 4) + step r = r
  by (auto simp add: phase-def step-def)
```

61
The following record models the local state of a process.

```
record 'val pstate =
x :: 'val          — current value held by process
vote :: 'val option — value the process voted for, if any
commit :: bool     — did the process commit to the vote?
ready :: bool      — for coordinators: did the round finish successfully?
timestamp :: nat   — time stamp of current value
decide :: 'val option — value the process has decided on, if any
coord Φ :: Proc    — coordinator for current phase
```

Possible messages sent during the execution of the algorithm.

```
datatype 'val msg =
  ValStamp 'val nat  — dummy message in case nothing needs to be sent
  Vote 'val
  Ack
  Null
```

Characteristic predicates on messages.

```
definition isValStamp where isValStamp m ≡ ∃v ts. m = ValStamp v ts
definition isVote where isVote m ≡ ∃v. m = Vote v
definition isAck where isAck m ≡ m = Ack
```

Selector functions to retrieve components of messages. These functions have a meaningful result only when the message is of an appropriate kind.

```
fun val where
  val (ValStamp v ts) = v
  val (Vote v) = v

fun stamp where
  stamp (ValStamp v ts) = ts
```

The x field of the initial state is unconstrained, all other fields are initialized appropriately.

```
definition LV-initState where
  LV-initState p st crd ≡
  vote st = None
  ∧ ¬(commit st)
  ∧ ¬(ready st)
  ∧ timestamp st = 0
  ∧ decide st = None
  ∧ coord Φ st = crd
```

We separately define the transition predicates and the send functions for each step and later combine them to define the overall next-state relation.

— processes from which values and timestamps were received
**definition** valStampsRcvd **where**

\[
\text{valStampsRcvd} \ (\text{msgs} :: \text{Proc} \rightarrow '\text{val msg}) \equiv \\
\{ q . \exists v \ ts. \text{msgs} q = \text{Some} \ (\text{ValStamp} v \ ts)\}
\]

**definition** highestStampRcvd **where**

\[
\text{highestStampRcvd} \ \text{msgs} \equiv \\
\text{Max} \ \{ts . \exists q v. (\text{msgs} :: \text{Proc} \rightarrow '\text{val msg}) q = \text{Some} \ (\text{ValStamp} v \ ts)\}
\]

In step 0, each process sends its current \(x\) and \(\text{timestamp}\) values to its coordinator.

A process that considers itself to be a coordinator updates its \(\text{vote}\) field if it has received messages from a majority of processes. It then sets its \(\text{commit}\) field to true.

**definition** send0 **where**

\[
\text{send0} \ r p q \ st \equiv \\
\text{if} \ q = \text{coord} \Phi \ st \ \text{then} \ \text{ValStamp} \ (x \ st) \ (\text{timestamp} \ st) \ \text{else} \ \text{Null}
\]

**definition** next0 **where**

\[
\text{next0} \ r p st \ msgs \ crd \ st' \equiv \\
\text{if} \ p = \text{coord} \Phi \ st \ \land \ \text{card} \ (\text{valStampsRcvd} \ \text{msgs}) > N \ \text{div} \ 2 \ \text{then} \ (\exists p v. \ \text{msgs} \ p = \text{Some} \ (\text{ValStamp} v \ (\text{highestStampRcvd} \ \text{msgs}))) \\
\quad \land \ st' = st \ [(\ text{vote} := \text{Some} \ v, \ \text{commit} := \text{True} \ )] \\
\text{else} \ st' = st
\]

In step 1, coordinators that have committed send their vote to all processes. Processes update their \(x\) and \(\text{timestamp}\) fields if they have received a vote from their coordinator.

**definition** send1 **where**

\[
\text{send1} \ r p q \ st \equiv \\
\text{if} \ p = \text{coord} \Phi \ st \ \land \ \text{commit} \ st \ \text{then} \ \text{Vote} \ (\text{the} \ (\text{vote} \ st)) \ \text{else} \ \text{Null}
\]

**definition** next1 **where**

\[
\text{next1} \ r p st \ msgs \ crd \ st' \equiv \\
\text{if} \ \text{msgs} \ (\text{coord} \Phi \ st) \neq \ \text{None} \ \land \ \text{isVote} \ (\text{the} \ (\text{msgs} \ (\text{coord} \Phi \ st))) \\
\text{then} \ st' = st \ [(x := \text{val} \ (\text{the} \ (\text{msgs} \ (\text{coord} \Phi \ st))), \ \text{timestamp} := \text{Suc} \ (\text{phase} \ r) \ ] \\
\text{else} \ st' = st
\]

In step 2, processes that have current timestamps send an acknowledgement to their coordinator.

A coordinator sets its \(\text{ready}\) field to true if it receives a majority of acknowledgements.

**definition** send2 **where**

\[
\text{send2} \ r p q \ st \equiv \\
\text{if} \ \text{timestamp} \ st = \text{Suc} \ (\text{phase} \ r) \ \land \ q = \text{coord} \Phi \ st \ \text{then} \ \text{Ack} \ \text{else} \ \text{Null}
\]

— processes from which an acknowledgement was received

**definition** acksRcvd **where**
\textbf{acksRcvd} (msgs :: \texttt{Proc} \to \texttt{val msg}) \equiv \\
\{ \ q . \ msgs \ q \neq \text{None} \land \text{isAck} (\text{the} (\text{msgs} \ q)) \ \}\n
definition \textit{next2} where
\textit{next2} \ r \ p \ st \ msgs \ crd \ st' \equiv \\
\text{if } p = \text{coord}\Phi \ st \land \text{card} (\text{acksRcvd} \ msgs) > N \div 2 \\
\text{then } st' = st (| \text{ready} := \text{True} |) \\
\text{else } st' = st

In step 3, coordinators that are ready send their vote to all processes.
Processes that received a vote from their coordinator decide on that value.
Coordinators reset their \textit{ready} and \textit{commt} fields to false. All processes reset
the coordinators as indicated by the parameter of the operator.

definition \textit{send3} where
\textit{send3} \ r \ p \ q \ st \equiv \\
\text{if } p = \text{coord}\Phi \ st \land \text{ready} \text{ then } \text{Vote} (\text{the} (\text{vote} \ st)) \text{ else } \text{Null}

definition \textit{next3} where
\textit{next3} \ r \ p \ st \ msgs \ crd \ st' \equiv \\
(\text{if } \text{msgs} (\text{coord}\Phi \ st) \neq \text{None} \land \text{isVote} (\text{the} (\text{msgs} (\text{coord}\Phi \ st))) \\
\text{then decide} \ st' = \text{Some} (\text{val} (\text{the} (\text{msgs} (\text{coord}\Phi \ st)))) \\
\text{else decide} \ st' = \text{decide} \ st) \\
\land (\text{if } p = \text{coord}\Phi \ st \\
\text{then } \lnot (\text{ready} \ st') \land \lnot (\text{commt} \ st') \\
\text{else ready} \ st' = \text{ready} \ st \land \text{commt} \ st' = \text{commt} \ st) \\
\land \ x \ st' = x \ st \\
\land \text{vote} \ st' = \text{vote} \ st \\
\land \text{timestamp} \ st' = \text{timestamp} \ st \\
\land \text{coord}\Phi \ st' = \text{crd}

The overall send function and next-state relation are simply obtained as the
composition of the individual relations defined above.

definition \textit{LV-sendMsg} :: \texttt{nat} \Rightarrow \texttt{Proc} \Rightarrow \texttt{Proc} \Rightarrow \texttt{\textit{val pstate} \Rightarrow \texttt{\textit{val msg}}} where
\textit{LV-sendMsg} (r::\texttt{nat}) \equiv \\
\text{if } \text{step} \ r = 0 \text{ then } \text{send0} \ r \\
\text{else if } \text{step} \ r = 1 \text{ then } \text{send1} \ r \\
\text{else if } \text{step} \ r = 2 \text{ then } \text{send2} \ r \\
\text{else } \text{send3} \ r

definition \textit{LV-nextState} :: \texttt{nat} \Rightarrow \texttt{Proc} \Rightarrow \texttt{\textit{val pstate} \Rightarrow (Proc \Rightarrow \texttt{\textit{val msg}})} \Rightarrow \texttt{Proc} \Rightarrow \texttt{\textit{val pstate} \Rightarrow \texttt{bool}} where
\textit{LV-nextState} \ r \equiv \\
\text{if } \text{step} \ r = 0 \text{ then } \text{next0} \ r \\
\text{else if } \text{step} \ r = 1 \text{ then } \text{next1} \ r \\
\text{else if } \text{step} \ r = 2 \text{ then } \text{next2} \ r \\
\text{else } \text{next3} \ r
7.2 Communication Predicate for \textit{LastVoting}

We now define the communication predicate that will be assumed for the correctness proof of the \textit{LastVoting} algorithm. The “per-round” part is trivial: integrity and agreement are always ensured.

For the “global” part, Charron-Bost and Schiper propose a predicate that requires the existence of infinitely many phases $ph$ such that:

- all processes agree on the same coordinator $c$,
- $c$ hears from a strict majority of processes in steps 0 and 2 of phase $ph$, and
- every process hears from $c$ in steps 1 and 3 (this is slightly weaker than the predicate that appears in [7], but obviously sufficient).

Instead of requiring infinitely many such phases, we only assume the existence of one such phase (Charron-Bost and Schiper note that this is enough.)

\textbf{definition}\n
$LV\text{-}comm\text{Per}Rd\ where$

$LV\text{-}comm\text{Per}Rd\ r\ (HO::Proc\ HO)\ (coord::Proc\ coord) \equiv True$

\textbf{definition}\n
$LV\text{-}comm\text{Global\ where}$

$LV\text{-}comm\text{Global}\ HOs\ coords \equiv$

\begin{align*}
\exists ph::\text{nat}. \exists c::\text{Proc}. & \\
& (\forall p.\ \text{coords}\ (4*ph)\ p = c) \\
& \land \text{card}\ (HOs\ (4*ph)\ c) > N \ div \ 2 \\
& \land \text{card}\ (HOs\ (4*ph+2)\ c) > N \ div \ 2 \\
& \land (\forall p.\ c \in HOs\ (4*ph+1)\ p \cap HOs\ (4*ph+3)\ p)
\end{align*}

7.3 The \textit{LastVoting} Heard-Of Machine

We now define the coordinated HO machine for the \textit{LastVoting} algorithm by assembling the algorithm definition and its communication-predicate.

\textbf{definition} \textit{LV\text{-}CHO}Machine \textbf{where}

$LV\text{-}CHO\text{Machine} \equiv$

\begin{align*}
\langle & \text{InitState} = LV\text{-}init\text{State}, \\
& \text{send\text{Msg}} = LV\text{-}send\text{Msg}, \\
& \text{NextState} = LV\text{-}next\text{State}, \\
& CHO\text{comm}\text{Per}Rd = LV\text{-}comm\text{Per}Rd, \\
& CHO\text{comm}\text{Global} = LV\text{-}comm\text{Global} \rangle
\end{align*}

\textbf{abbreviation}\n
$LV\text{-}M \equiv (LV\text{-}CHO\text{Machine}::(Proc, 'val\ ps\ state, 'val\ msg)\ CHOMachine)$

\textbf{end}
theory LastVotingProof

imports LastVotingDefs ../Majorities ../Reduction

begin

7.4 Preliminary Lemmas

We begin by proving some simple lemmas about the utility functions used in the model of LastVoting. We also specialize the induction rules of the generic CHO model for this particular algorithm.

lemma timeStampsRcvdFinite:
  \begin{align*}
  \text{finite } \{ ts . \exists q v. (msgs::Proc \rightarrow 'val msg) q = Some (ValStamp v ts) \}
  \\
  \text{(is finite ?ts)}
  \\
  \text{proof} - \\
  \text{have ?ts = stamp ' the ' msgs ' (valStampsRcvd msgs)}
  \\
  \text{by (force simp add: valStampsRcvd-def image-def)}
  \\
  \text{thus ?thesis by auto}
  \end{align*}

qed

lemma highestStampRcvd-exists:
  \begin{align*}
  \text{assumes nempty: valStampsRcvd msgs \neq \{}}
  \\
  \text{obtains p v where msgs p = Some (ValStamp v (highestStampRcvd msgs))}
  \end{align*}

proof
- 
  \text{let \( \text{?ts = \{ ts . \exists q v. msgs q = Some (ValStamp v ts) \} } \)}
  \\
  \text{from nempty have \( ?ts \neq \{ \) by (auto simp add: valStampsRcvd-def)}
  \\
  \text{with timeStampsRcvdFinite}
  \\
  \text{have highestStampRcvd msgs \in ?ts}
  \\
  \text{unfolding highestStampRcvd-def by (rule Max-in)}
  \\
  \text{then obtain p v where msgs p = Some (ValStamp v (highestStampRcvd msgs))}
  \\
  \text{by (auto simp add: highestStampRcvd-def)}
  \\
  \text{with that show thesis .}

qed

lemma highestStampRcvd-max:
  \begin{align*}
  \text{assumes \( \text{msgs p = Some (ValStamp v ts) } \)}
  \\
  \text{shows \( ts \leq \text{highestStampRcvd msgs } \)}
  \\
  \text{using assms unfolding highestStampRcvd-def}
  \\
  \text{by (blast intro: Max-ge timeStampsRcvdFinite)}
  \end{align*}

lemma phase-Suc:
  \begin{align*}
  \text{phase (Suc r) = (if step r = 3 then Suc (phase r)}
  \\
  \text{else phase r)}
  \\
  \text{unfolding step-def phase-def by presburger}
  \end{align*}

Many proofs are by induction on runs of the LastVoting algorithm, and we derive a specific induction rule to support these proofs.

lemma LV-induct:
  \begin{align*}
  \text{assumes \( \text{run: CHORun LV-M rho HOs coords } \)}
  \\
  \text{and init: \( \forall p. CinitState LV-M p (rho 0 p) (coords 0 p) \Rightarrow P 0 \)}
  \end{align*}
and step0: \( \forall r. \) 
\[
\begin{align*}
\text{step } r &= 0; P r; \text{phase } (\text{Suc } r) = \text{phase } r; \text{step } (\text{Suc } r) = 1; \\
\forall p. \text{next0 } r \ p \ (\rho \ r \ p) \\
&\quad \text{(HOrcvdMsgs } L V-M \ r \ p \ (\text{HOs } r \ p) \ (\rho \ r) \} \\
&\quad \text{(coords } (\text{Suc } r) \ p) \\
&\quad \text{(rho } (\text{Suc } r) \ p) \\
\implies P (\text{Suc } r)
\end{align*}
\]

and step1: \( \forall r. \) 
\[
\begin{align*}
\text{step } r &= 1; P r; \text{phase } (\text{Suc } r) = \text{phase } r; \text{step } (\text{Suc } r) = 2; \\
\forall p. \text{next1 } r \ p \ (\rho \ r \ p) \\
&\quad \text{(HOrcvdMsgs } L V-M \ r \ p \ (\text{HOs } r \ p) \ (\rho \ r) \} \\
&\quad \text{(coords } (\text{Suc } r) \ p) \\
&\quad \text{(rho } (\text{Suc } r) \ p) \\
\implies P (\text{Suc } r)
\end{align*}
\]

and step2: \( \forall r. \) 
\[
\begin{align*}
\text{step } r &= 2; P r; \text{phase } (\text{Suc } r) = \text{phase } r; \text{step } (\text{Suc } r) = 3; \\
\forall p. \text{next2 } r \ p \ (\rho \ r \ p) \\
&\quad \text{(HOrcvdMsgs } L V-M \ r \ p \ (\text{HOs } r \ p) \ (\rho \ r) \} \\
&\quad \text{(coords } (\text{Suc } r) \ p) \\
&\quad \text{(rho } (\text{Suc } r) \ p) \\
\implies P (\text{Suc } r)
\end{align*}
\]

and step3: \( \forall r. \) 
\[
\begin{align*}
\text{step } r &= 3; P r; \text{phase } (\text{Suc } r) = \text{Suc } (\text{phase } r); \text{step } (\text{Suc } r) = 0; \\
\forall p. \text{next3 } r \ p \ (\rho \ r \ p) \\
&\quad \text{(HOrcvdMsgs } L V-M \ r \ p \ (\text{HOs } r \ p) \ (\rho \ r) \} \\
&\quad \text{(coords } (\text{Suc } r) \ p) \\
&\quad \text{(rho } (\text{Suc } r) \ p) \\
\implies P (\text{Suc } r)
\end{align*}
\]

shows \( P n \) 

proof (rule CHORun-induct(OF \ run))
assume CHOinitConfig LV-M (\rho 0) (coords 0)
thus \( P 0 \) by (auto simp add: CHOinitConfig-def init)

next
fix \( r \)
assume ih: \( P \ r \)
and nxt: CHOnextConfig LV-M r (\rho \ r) (HOs \ r)

have step \( r \in \{0,1,2,3\} \) by (auto simp add: step-def)
thus \( P \ (\text{Suc } r) \)

proof auto
assume stp: step \( r = 0 \)
hence step (Suc \( r \)) = 1
by (auto simp add: step-def mod-Suc)
with ih nxt stp show ?thesis
by (intro step0)
(apply simp: LV-CHOMachine-def CHOnextConfig-eq
LV-nextState-def LV-sendMsg-def phase-Suc)

next
assume stp: step \( r = \text{Suc } 0 \)
hence \( \text{step} (\text{Suc } r) = 2 \)
by (auto simp add: step-def mod-Suc)
with \( \text{ih} \) \( \text{nxt} \) \( \text{stp} \) show \( \text{thesis} \)
by (intro step1)
(auto simp: LV-CHOMachine-def CHOnextConfig-eq
LV-nextState-def LV-sendMsg-def phase-Suc)

next
assume \( \text{stp} : \text{step} r = 2 \)
hence \( \text{step} (\text{Suc } r) = 3 \)
by (auto simp add: step-def mod-Suc)
with \( \text{ih} \) \( \text{nxt} \) \( \text{stp} \) show \( \text{thesis} \)
by (intro step2)
(auto simp: LV-CHOMachine-def CHOnextConfig-eq
LV-nextState-def LV-sendMsg-def phase-Suc)

next
assume \( \text{stp} : \text{step} r = 3 \)
hence \( \text{step} (\text{Suc } r) = 0 \)
by (auto simp add: step-def mod-Suc)
with \( \text{ih} \) \( \text{nxt} \) \( \text{stp} \) show \( \text{thesis} \)
by (intro step3)
(auto simp: LV-CHOMachine-def CHOnextConfig-eq
LV-nextState-def LV-sendMsg-def phase-Suc)

qed
qed

The following rule similarly establishes a property of two successive configurations of a run by case distinction on the step that was executed.

**lemma** \( \text{LV-Suc} \):
**assumes** \( \text{run} : \text{CHORun} \text{ LV-M } \text{rho} \text{ HO} \text{os} \text{ coords} \)
**and** \( \text{step}0 \) : \[ \text{step} r = 0; \text{step} (\text{Suc } r) = 1; \text{phase} (\text{Suc } r) = \text{phase } r; \]
\[ \forall p. \text{next}0 r p (\text{rho } p) \]
\[ \text{HO} \text{rcvd} \text{Msgs } \text{LV-M } r p (\text{HO} \text{os } r p) (\text{rho } r) \]
\[ \text{coords } (\text{Suc } r) p (\text{rho } (\text{Suc } r) p) \] \[ \Rightarrow P r \]
**and** \( \text{step}1 \) : \[ \text{step} r = 1; \text{step} (\text{Suc } r) = 2; \text{phase} (\text{Suc } r) = \text{phase } r; \]
\[ \forall p. \text{next}1 r p (\text{rho } p) \]
\[ \text{HO} \text{rcvd} \text{Msgs } \text{LV-M } r p (\text{HO} \text{os } r p) (\text{rho } r) \]
\[ \text{coords } (\text{Suc } r) p (\text{rho } (\text{Suc } r) p) \] \[ \Rightarrow P r \]
**and** \( \text{step}2 \) : \[ \text{step} r = 2; \text{step} (\text{Suc } r) = 3; \text{phase} (\text{Suc } r) = \text{phase } r; \]
\[ \forall p. \text{next}2 r p (\text{rho } p) \]
\[ \text{HO} \text{rcvd} \text{Msgs } \text{LV-M } r p (\text{HO} \text{os } r p) (\text{rho } r) \]
\[ \text{coords } (\text{Suc } r) p (\text{rho } (\text{Suc } r) p) \] \[ \Rightarrow P r \]
**and** \( \text{step}3 \) : \[ \text{step} r = 3; \text{step} (\text{Suc } r) = 0; \text{phase} (\text{Suc } r) = \text{Suc } (\text{phase } r); \]
\[ \forall p. \text{next}3 r p (\text{rho } p) \]
\[ \text{HO} \text{rcvd} \text{Msgs } \text{LV-M } r p (\text{HO} \text{os } r p) (\text{rho } r) \]
\[ \text{coords } (\text{Suc } r) p (\text{rho } (\text{Suc } r) p) \] \[ \Rightarrow P r \]
shows $P r$

proof –

from run

have $\text{nxt: CHOnextConfig LV-M r (rho r) (HOs r)}$
  $(\text{coords (Suc r)) (rho (Suc r))}$
  by (auto simp add: CHORun-eq)

have $\text{step r \in \{0,1,2,3\}}$ by (auto simp add: step-def)

thus $P r$

proof (auto)

  assume $\text{stp: step r = 0}$
  hence $\text{step (Suc r) = 1}$
  by (auto simp add: step-def mod-Suc)

  with $\text{nxt stp show \ ?thesis}$
  by (intro step0)
    (auto simp: LV-CHOMachine-def CHOnextConfig-eq
       LV-nextState-def LV-sendMsg-def phase-Suc)

next

  assume $\text{stp: step r = Suc 0}$
  hence $\text{step (Suc r) = 2}$
  by (auto simp add: step-def mod-Suc)

  with $\text{nxt stp show \ ?thesis}$
  by (intro step1)
    (auto simp: LV-CHOMachine-def CHOnextConfig-eq
       LV-nextState-def LV-sendMsg-def phase-Suc)

next

  assume $\text{stp: step r = 2}$
  hence $\text{step (Suc r) = 3}$
  by (auto simp add: step-def mod-Suc)

  with $\text{nxt stp show \ ?thesis}$
  by (intro step2)
    (auto simp: LV-CHOMachine-def CHOnextConfig-eq
       LV-nextState-def LV-sendMsg-def phase-Suc)

next

  assume $\text{stp: step r = 3}$
  hence $\text{step (Suc r) = 0}$
  by (auto simp add: step-def mod-Suc)

  with $\text{nxt stp show \ ?thesis}$
  by (intro step3)
    (auto simp: LV-CHOMachine-def CHOnextConfig-eq
       LV-nextState-def LV-sendMsg-def phase-Suc)

qed

Sometimes the assertion to prove talks about a specific process and follows from the next-state relation of that particular process. We prove corresponding variants of the induction and case-distinction rules. When these variants are applicable, they help automating the Isabelle proof.

lemma $\text{LV-induct'}$:
  assumes $\text{run: CHORun LV-M rho HOs coords}$

69
and \textit{init:} \textit{CinitState LV-M \rho (\rho 0 \rho) (coords 0 \rho) \implies P \rho 0 }

and \textit{step0:} \quad \wedge r. \quad \forall step r = 0; \ P \rho r; \ phase (Suc r) = phase r; \ step (Suc r) = 1; 
\quad \text{next0} \ r \ p \ (\rho r p) 
\quad (HOrcvdMsgs LV-M \ r \ p \ (HOs r p) (rho r)) 
\quad (coords (Suc r) p) (rho (Suc r) p) ] 
\implies P \rho (Suc r) 

and \textit{step1:} \quad \wedge r. \quad \forall step r = 1; \ P \rho r; \ phase (Suc r) = phase r; \ step (Suc r) = 2; 
\quad \text{next1} \ r \ p \ (\rho r p) 
\quad (HOrcvdMsgs LV-M \ r \ p \ (HOs r p) (rho r)) 
\quad (coords (Suc r) p) (rho (Suc r) p) ] 
\implies P \rho (Suc r) 

and \textit{step2:} \quad \wedge r. \quad \forall step r = 2; \ P \rho r; \ phase (Suc r) = phase r; \ step (Suc r) = 3; 
\quad \text{next2} \ r \ p \ (\rho r p) 
\quad (HOrcvdMsgs LV-M \ r \ p \ (HOs r p) (rho r)) 
\quad (coords (Suc r) p) (rho (Suc r) p) ] 
\implies P \rho (Suc r) 

and \textit{step3:} \quad \wedge r. \quad \forall step r = 3; \ P \rho r; \ phase (Suc r) = Suc (phase r); \ step (Suc r) = 0; 
\quad \text{next3} \ r \ p \ (\rho r p) 
\quad (HOrcvdMsgs LV-M \ r \ p \ (HOs r p) (rho r)) 
\quad (coords (Suc r) p) (rho (Suc r) p) ] 
\implies P \rho (Suc r) 

\textit{shows \ P \rho n.}

\textit{by \ (rule LV-induct[OF run])}

(\textit{auto intro: init step0 step1 step2 step3})

\textbf{lemma \ LV-Suc':}
\textbf{assumes \ run: \ CHORun LV-M \rho HO\scoords}
\textit{and step0:} \quad \exists step r = 0; \ step (Suc r) = phase r; 
\quad next0 \ r \ p \ (\rho r p) 
\quad (HOrcvdMsgs LV-M \ r \ p \ (HOs r p) (rho r)) 
\quad (coords (Suc r) p) (rho (Suc r) p) ] 
\implies P \rho r 

\textit{and step1:} \quad \exists step r = 1; \ step (Suc r) = phase r; 
\quad next1 \ r \ p \ (\rho r p) 
\quad (HOrcvdMsgs LV-M \ r \ p \ (HOs r p) (rho r)) 
\quad (coords (Suc r) p) (rho (Suc r) p) ] 
\implies P \rho r 

\textit{and step2:} \quad \exists step r = 2; \ step (Suc r) = phase r; 
\quad next2 \ r \ p \ (\rho r p) 
\quad (HOrcvdMsgs LV-M \ r \ p \ (HOs r p) (rho r)) 
\quad (coords (Suc r) p) (rho (Suc r) p) ] 
\implies P \rho r 

\textit{and step3:} \quad \exists step r = 3; \ step (Suc r) = Suc (phase r); 
\quad next3 \ r \ p \ (\rho r p) 
\quad (HOrcvdMsgs LV-M \ r \ p \ (HOs r p) (rho r)) 
\quad (coords (Suc r) p) (rho (Suc r) p) ] 
\implies P \rho r 

\textit{shows \ P \rho r}
7.5 Boundedness and Monotonicity of Timestamps

The timestamp of any process is bounded by the current phase.

**Lemma LV-timestamp-bounded:**

**Assumes** \( \text{run} : \text{CHORun} \ LV-M \ \rho \ \text{HOs} \ \text{coords} \)

**Shows** \( \text{timestamp} \ (\rho \ n \ p) \leq (\text{if} \ \text{step} \ n < 2 \ \text{then} \ \text{phase} \ n \ \text{else} \ \text{Suc} \ (\text{phase} \ n)) \)

(by \( \text{rule LV-induct'} \ [\text{OF run}, \ \text{where} \ P = \text{?P}] \))

(auto simp: \text{LV-CHOMachine-def LV-initState-def next0-def next1-def next2-def next3-def})

Moreover, timestamps can only grow over time.

**Lemma LV-timestamp-increasing:**

**Assumes** \( \text{run} : \text{CHORun} \ LV-M \ \rho \ \text{HOs} \ \text{coords} \)

**Shows** \( \text{timestamp} \ (\rho \ n \ p) \leq \text{timestamp} \ (\rho \ (\text{Suc} \ n) \ p) \)

(is \( \text{is} \ ?\text{ts} \leq - \))

**Proof** (by \( \text{rule LV-Suc' [OF run, where P=?P]} \))

(auto simp: \text{LV-timestamp-bounded [OF run, where P=?P]})

The case of \text{next1} is the only interesting one because the timestamp may change: here we use the previously established fact that the timestamp is bounded by the phase number.

**Assume** \text{stp}: \text{step} \ n = 1

and \text{nxt}: \text{next1} \ n \ p \ (\rho \ n \ p)

\( \text{HOrcvdMsgs} \ LV-M \ n \ p \ (\text{HOs} \ n \ p) \ (\rho \ n) \)

\( \text{coords} \ (\text{Suc} \ n) \ p \ (\rho \ (\text{Suc} \ n) \ p) \)

from \text{stp} have \( ?\text{ts} \leq \text{phase} \ n \)

using \text{LV-timestamp-bounded [OF run, where n=n, where p=p]} by auto

with \text{nxt} show \( ?\text{thesis} \) by (auto simp add: \text{next1-def})

qed (auto simp add: \text{next0-def next2-def next3-def})

**Lemma LV-timestamp-monotonic:**

**Assumes** \( \text{run} : \text{CHORun} \ LV-M \ \rho \ \text{HOs} \ \text{coords} \) and \text{le}: \( m \leq n \)

**Shows** \( \text{timestamp} \ (\rho \ m \ p) \leq \text{timestamp} \ (\rho \ n \ p) \)

(is \( ?\text{ts} \leq - \))

**Proof**

from \text{le} obtain \( k \) where \( k: \ n = m+k \)

by (auto simp add: \text{le-iff-add})

have \( ?\text{ts} \leq ?\text{ts} \ (m+k) \ (\text{is} \ ?P \ k) \)

proof (induct \( k \))

case \( 0 \) show \( ?P \ 0 \) by simp

next

fix \( k \)

assume \( \text{ih}: \ ?P \ k \)

from \text{run} have \( ?\text{ts} \ (m+k) \leq ?\text{ts} \ (m + \text{Suc} \ k) \)

by (auto simp add: \text{LV-timestamp-increasing})
The following definition collects the set of processes whose timestamp is beyond a given bound at a system state.

definition procsBeyondTS where
procsBeyondTS ts cfg ≡ { p . ts ≤ timestamp (cfg p) }

Since timestamps grow monotonically, so does the set of processes that are beyond a certain bound.

lemma procsBeyondTS-monotonic:
assumes run: CHORun LV-M rho HOs coords
and p: p ∈ procsBeyondTS ts (rho m)
and le: m ≤ n
shows p ∈ procsBeyondTS ts (rho n)
proof –
from p have ts ≤ timestamp (rho m)
  (is - ≤ ?ts m)
  by (simp add: procsBeyondTS-def)
moreover
from run le have ?ts m ≤ ?ts n
  by (rule LV-timestamp-monotonic)
ultimately show ?thesis
  by (simp add: procsBeyondTS-def)
qed

7.6 Obvious Facts About the Algorithm

The following lemmas state some very obvious facts that follow “immediately” from the definition of the algorithm. We could prove them in one fell swoop by defining a big invariant, but it appears more readable to prove them separately.

Coordinators change only at step 3.

lemma notStep3EqualCoord:
assumes run: CHORun LV-M rho HOs coords and stp:step r ≠ 3
shows coordΦ (rho (Suc r) p) = coordΦ (rho r p) (is ?P p r)
by (rule LV-Suc′ [OF run, where P=:?P])
  (auto simp: stp next0-def next1-def next2-def)

lemma coordinators:
assumes run: CHORun LV-M rho HOs coords
shows coordΦ (rho r p) = coords (4*(phase r)) p
proof –
let ?r0 = (4*(phase r))
let ?r1 = (4*(phase r))
have coordΦ (rho ?r1 p) = coords ?r1 p
proof (cases phase r > 0)
  case False
hence \( \text{phase } r = 0 \) by \text{auto}

with run show \(?\text{thesis}\)

by (auto simp: LV-CHO\text{Machine-def} CHORun-eq CHO\text{initConfig-def} LV-initState-def)

next

\text{case } \text{True}

hence \( \text{step } (\text{Suc } ?r0) = 0 \) by (auto simp: step-def)

hence \( \text{step } ?r0 = 3 \) by (auto simp: mod-Suc step-def)

moreover

from run

have LV-nextState \(?r0 \ p (\rho \ ?r0 \ p) (H\text{OrcvdMsgs} \ LV-M \ ?r0 \ p (H\text{Os} \ ?r0 \ p) (\rho \ ?r0)) \ (\text{coords} (\text{Suc } ?r0) \ p (\rho (\text{Suc } ?r0) \ p)

by (auto simp: LV-CHO\text{Machine-def} CHORun-eq CHO\text{nextConfig-def})

ultimately

have \text{next3 } ?r0 \ p (\rho \ ?r0 \ p) (H\text{OrcvdMsgs} \ LV-M \ ?r0 \ p (H\text{Os} \ ?r0 \ p) (\rho \ ?r0)) \ (\text{coords} (\text{Suc } ?r0) \ p (\rho (\text{Suc } ?r0) \ p)

by (auto simp: LV\text{-nextState-def})

hence \text{coord} \Phi (\rho (\text{Suc } ?r0) \ p) = \text{coords} (\text{Suc } ?r0) \ p

by (auto simp: next3-def)

with \text{True} show \(?\text{thesis}\) by \text{auto}

qed

moreover

have \( r \in \{?r1, \text{Suc } ?r1, \text{Suc } (\text{Suc } ?r1)\} \)

by (auto simp: notStep3EqualCoord step-def phase-def mod-Suc)

ultimately

show \(?\text{thesis}\) by \text{auto}

qed

Votes only change at step 0.

\textbf{lemma} notStep0EqualVote \[\text{rule-format}]:

\textbf{assumes} run: CHORun LV-M \(\rho\) HOs coords

\textbf{shows} \( \text{step } r \neq 0 \rightarrow \text{vote} (\rho (\text{Suc } r) \ p) = \text{vote} (\rho \ r \ p) \ (\text{is } ?P \ p \ r) \)

by (rule LV-Suc\[^{\text{OF run, where } P=?P}]\)

(auto simp: next0-def next1-def next2-def next3-def)

Commit status only changes at steps 0 and 3.

\textbf{lemma} notStep03EqualCommit \[\text{rule-format}]:

\textbf{assumes} run: CHORun LV-M \(\rho\) HOs coords

\textbf{shows} \( \text{step } r \neq 0 \land \text{step } r \neq 3 \rightarrow \text{commit} (\rho (\text{Suc } r) \ p) = \text{commit} (\rho \ r \ p) \ (\text{is } ?P \ p \ r) \)

by (rule LV-Suc\[^{\text{OF run, where } P=?P}]\)
Timestamps only change at step 1.

**lemma notStep1EqualTimestamp [rule-format]:**

assumes `run`: CHORun LV-M rho HOs coords

shows `step r ≠ 1 → timestamp (rho (Suc r) p) = timestamp (rho r p)`

by (rule LV-Suc[OF run, where `P = ?P`])

(auto simp: next0-def next1-def next2-def next3-def)

The x field only changes at step 1.

**lemma notStep1EqualX [rule-format]:**

assumes `run`: CHORun LV-M rho HOs coords

shows `step r ≠ 1 → x (rho (Suc r) p) = x (rho r p)`

by (rule LV-Suc[OF run, where `P = ?P`])

(auto simp: next0-def next1-def next2-def next3-def)

A process `p` has its `commit` flag set only if the following conditions hold:

- the step number is at least 1,
- `p` considers itself to be the coordinator,
- `p` has a non-null `vote`,
- a majority of processes consider `p` as their coordinator.

**lemma commitE:**

assumes `run`: CHORun LV-M rho HOs coords and `cmt`: commit (rho r p)

and `conds`: `[ 1 ≤ step r; coordΦ (rho r p) = p; vote (rho r p) ≠ None; card {q . coordΦ (rho r q) = p} > N div 2 ]` 

shows `A`

proof (rule LV-induct[OF run, where `P = ?P`])

—the only interesting step is step 0

fix `n`

assume `next`: `next0 n p (rho n p) (HOrcedMsgs LV-M n p (HOs n p) (rho n))`

and `ph`: `phase (Suc n) = phase n`

and `stp`: `step n = 0 and stp`: `step (Suc n) = 1`

and `ih`: `?P p n`

show `?P p (Suc n)`
proof
assume cm': commt (rho (Suc n) p)
from stp ih have cm: ¬ commt (rho n p) by simp
with nxt cm'
have coord\Phi (rho n p) = p
  ∧ vote (rho (Suc n) p) ≠ None
  ∧ card (valStampsRcvd (HOrcvdMsgs LV-M n p (HOs n p) (rho n)))
  > N div 2
by (auto simp add: next0-def)
moreover
from stp
have valStampsRcvd (HOrcvdMsgs LV-M n p (HOs n p) (rho n))
  ⊆ \{ q . coord\Phi (rho n q) = p\}
by (auto simp: valStampsRcvd-def LV-CHOMachine-def
     HOrcvdMsgs-def LV-sendMsg-def send0-def)
hence card (valStampsRcvd (HOrcvdMsgs LV-M n p (HOs n p) (rho n)))
  ≤ card \{ q . coord\Phi (rho n q) = p\}
by (auto intro: card-mono)
moreover
note stp stp' run
ultimately
show ?R (Suc n) by (auto simp: notStep3EqualCoord)
qed
— the remaining cases are all solved by expanding the definitions
qed (auto simp: LV-CHOMachine-def LV-initState-def next1-def next2-def
     next3-def notStep3EqualCoord[OF run])
with cmt show ?thesis by (intro conds, auto)
qed

A process has a current timestamp only if:

- it is at step 2 or beyond,
- its coordinator has committed,
- its x value is the vote of its coordinator.

lemma currentTimestampE:
assumes run: CHORun LV-M rho HOs coords
and ts: timestamp (rho r p) = Suc (phase r)
and conds: \[ 2 \leq step r;
    \] commt (rho r (coord\Phi (rho r p)));
    \[ x (rho r p) = the (vote (rho r (coord\Phi (rho r p))))\]
\[ \] ⇒ A
shows A
proof
let ?ts n = timestamp (rho n p)
let ?crd n = coord\Phi (rho n p)
have ?ts r = Suc (phase r) ⇒
  \[ 2 \leq step r \]
∧ commt (rho r (?crd r))
∧ x (rho r p) = the (vote (rho r (?crd r)))
(is ?Q p r is - → ?R r)
proof (rule LV-induct[OF run, where P=\?Q])
   — The assertion is trivially true initially because the timestamp is 0.
   assume CinitState LV-M p (rho 0 p) (coords 0 p) thus ?Q p 0
   by (auto simp: LV-CHOMachine-def LV-initState-def)
next

The assertion is trivially preserved by step 0 because the timestamp in the post-state cannot be current (cf. lemma LV-timestamp-bounded).

   fix n
   assume stp': step (Suc n) = 1
   with run LV-timestamp-bounded[where n=Suc n]
   have ?ts (Suc n) ≤ phase (Suc n) by auto
   thus ?Q p (Suc n) by simp
next

Step 1 establishes the assertion by definition of the transition relation.

   fix n
   assume stp: step n = 1 and stp':step (Suc n) = 2
   and ph: phase (Suc n) = phase n
   and nxt: next1 n p (rho n p) (HOrcvdMsgs LV-M n p (HOs n p) (rho n))
           (coords (Suc n) p) (rho (Suc n) p)
   show ?Q p (Suc n)
   proof
      assume ts: ?ts (Suc n) = Suc (phase (Suc n))
      from run stp LV-timestamp-bounded[where n=n]
      have ?ts n ≤ phase n by auto
      moreover
      from run stp
      have vote (rho (Suc n) (?crd (Suc n))) = vote (rho n (?crd n))
           by (auto simp: notStep3EqualCoord notStep0EqualVote)
      moreover
      from run stp
      have commt (rho (Suc n) (?crd (Suc n))) = commt (rho n (?crd n))
           by (auto simp: notStep3EqualCoord notStep1EqualCommit)
      moreover
      note ts nxt stp stp' ph
      ultimately
      show ?R (Suc n)
      by (auto simp: LV-CHOMachine-def HOrcvdMsgs-def LV-sendMsg-def
           next1-def send1-def isVote-def)
    qed
next

For step 2, the assertion follows from the induction hypothesis, observing that none of the relevant state components change.

   fix n
assume \(stp: \text{step } n = 2\) and \(stp': \text{step } (\text{Suc } n) = 3\) and \(ph: \text{phase } (\text{Suc } n) = \text{phase } n\) and \(ih: ?Q p n\) and \(nxt: \text{next}2 n p (\rho n) (\text{HOrcvdMsgs } LV-M n p (\text{HOs } n p) (\rho n)) (\text{coords } (\text{Suc } n) p) (\rho (\text{Suc } n) p)\)

show \(?Q p (\text{Suc } n)\)

proof

assume \(ts: ?ts (\text{Suc } n) = \text{Suc } (\text{phase } (\text{Suc } n))\)

from \(\text{run } stp\) have \(vt: \text{vote } (\rho (\text{Suc } n) ((\text{?crd } (\text{Suc } n)))) = \text{vote } (\rho n ((\text{?crd } n)))\)

by (auto simp add: notStep3EqualCoord notStep0EqualVote)

from \(\text{run } stp\) have \(cmt: \text{commt } (\rho (\text{Suc } n) ((\text{?crd } (\text{Suc } n)))) = \text{commt } (\rho n ((\text{?crd } n)))\)

by (auto simp add: notStep3EqualCoord notStep0EqualCommit)

with \(vt\ ts\ ph\ stp\ stp'\ ih\ nxt\)

show \(?R (\text{Suc } n)\)

by (auto simp add: next2-def)

qed

next

The assertion is trivially preserved by step 3 because the timestamp in the post-state cannot be current (cf. lemma \(LV\text{-timestamp-bounded}\)).

fix \(n\)

assume \(stp': \text{step } (\text{Suc } n) = 0\)

with \(\text{run } LV\text{-timestamp-bounded}[\text{where } n=\text{Suc } n]\)

have \(?ts (\text{Suc } n) \leq \text{phase } (\text{Suc } n)\) by auto

thus \(?Q p (\text{Suc } n)\) by simp

qed

with \(ts\) show \(?\text{thesis}\) by (intro conds) auto

qed

If a process \(p\) has its \textit{ready} bit set then:

- it is at step 3,
- it considers itself to be the coordinator of that phase and
- a majority of processes considers \(p\) to be the coordinator and has a current timestamp.

**lemma** \textit{readyE}:

assumes \(\text{run: CHORun } LV-M \rho \text{ HOs coords and rdy: ready } (\rho r p)\) and \(\text{conds: } [\text{step } r = 3; \text{coord}\Phi (\rho r p) = p; \text{card } \{ q . \text{coord}\Phi (\rho r q) = p \land \text{timestamp } (\rho r q) = \text{Suc } (\text{phase } r) \} > N \text{ div } 2]\)

shows \(P\)

proof

- let \(?qs n = \{ q . \text{coord}\Phi (\rho n q) = p \land \text{timestamp } (\rho n q) = \text{Suc } (\text{phase } n) \}\)
have \( \text{ready} \ (\rho r p) \implies \)
\[
\begin{align*}
&\text{\( \text{step} \ r = 3 \) \\
&\land \text{\( \text{coord} \Phi \ (\rho r p) = p \) \\
&\land \text{\( \text{card} \ (?qs \ r) > N \div 2 \)}}
\end{align*}
\]
\(\text{(is \(?Q \ p \ r \ \text{is} - \implies \ ?R \ p \ r\))}\)

\text{proof (rule \( \text{LV-induct[\( \text{OF \ run, \ where} \ P=\?Q\)]}\))}

— the interesting case is step 2

\text{fix} \ n \\
assume \( \text{stp} \): \( \text{step} \ n = 2 \) \text{ and} \( \text{stp}' \): \( \text{step} \ (\text{Suc} \ n) = 3 \)

and \( \text{ih} \): \( \forall Q \ p \ n \text{ and} \text{ph} \): \( \text{phase} \ (\text{Suc} \ n) = \text{phase} \ n \)

and \( \text{nxt} \): \( \text{next} \text{2} \ n \ p \ (\rho n \ p) \ (\text{HOrcvdMsgs} \ \text{LV-M} \ n \ p \ (\text{HOs} \ n \ p) \ (\rho n)) \)

(\text{coords} \ (\text{Suc} \ n) \ p) \ (\rho \ (\text{Suc} \ n) \ p) \\
\text{show} \ \forall Q \ p \ (\text{Suc} \ n) \\
\text{proof (clarify)} \\
\text{fix} \ q \\
assume \( q \in \ ?acks\)

with \( \text{stp} \)
have \( n \): \( \text{coord} \Phi \ (\rho n \ q) = p \land \text{timestamp} \ (\rho n \ q) = \text{Suc} \ (\text{phase} \ n) \)

by (auto simp: \( \text{LV-CHOMachine-def} \ \text{HOrcvdMsgs-def} \ \text{LV-sendMsg-def} \ \text{acksRcvd-def} \ \text{send2-def} \ \text{isAck-def})

with \( \text{run} \ \text{stp} \ \text{ph} \)
show \( \text{coord} \Phi \ (\rho \ (\text{Suc} \ n) \ q) = p \land \text{timestamp} \ (\rho \ (\text{Suc} \ n) \ q) = \text{Suc} \ (\text{phase} \ (\text{Suc} \ n)) \)

by (simp add: \( \text{notStep3EqualCoord} \ \text{notStep1EqualTimestamp} \))
\text{qed}

hence \( \text{?acks} \subseteq \ ?qs \ (\text{Suc} \ n) \)

\text{proof (clarify)} \\
\text{fix} \ q \\
assume \( q \in \ ?acks \)

with \( \text{stp} \)
have \( n \): \( \text{coord} \Phi \ (\rho n \ q) = p \land \text{timestamp} \ (\rho n \ q) = \text{Suc} \ (\text{phase} \ n) \)

by (auto simp: \( \text{LV-CHOMachine-def} \ \text{HOrcvdMsgs-def} \ \text{LV-sendMsg-def} \ \text{acksRcvd-def} \ \text{send2-def} \ \text{isAck-def})

with \( \text{run} \ \text{stp} \ \text{ph} \)
show \( \text{coord} \Phi \ (\rho \ (\text{Suc} \ n) \ q) = p \land \text{timestamp} \ (\rho \ (\text{Suc} \ n) \ q) = \text{Suc} \ (\text{phase} \ (\text{Suc} \ n)) \)

by (simp add: \( \text{notStep3EqualCoord} \ \text{notStep1EqualTimestamp} \))
\text{qed}

— the remaining steps are all solved trivially
\text{qed (auto simp: \( \text{LV-CHOMachine-def} \ \text{LV-initState-def} \ \text{next0-def} \ \text{next1-def} \ \text{next3-def})}\)

\text{with} \( \text{rdy} \ \text{show} \ \?\text{thesis} \ \text{by (blast intro: cons)} \)
\text{qed}

A process decides only if the following conditions hold:

\bullet \text{ it is at step 3,}
• its coordinator votes for the value the process decides on,
• the coordinator has its ready and commit bits set.

lemma decisionE:
assumes run: CHORun LV-M rho HOs coords
and dec: decide (rho (Suc r) p) ≠ decide (rho r p)
and conds: []
  step r = 3;
  decide (rho (Suc r) p) = Some (the (vote (rho r (coord \Phi (rho r p)))))
  ready (rho r (coord \Phi (rho r p)));
  commit (rho r (coord \Phi (rho r p)))]
⇒ P
shows P
proof —
let ?cfg = rho r
let ?cfg' = rho (Suc r)
let ?crd p = coord \Phi (?cfg p)
let ?dec' = decide (?cfg' p)

Except for the assertion about the commit field, the assertion can be proved directly from the next-state relation.

have 1: step r = 3
  ∧ ?dec' = Some (the (vote (?cfg (?crd p))))
  ∧ ready (?cfg (?crd p))
(is ?Q p r)
proof (rule LV-Suc [OF run, where P=\?Q])
— for step 3, we prove the thesis by expanding the relevant definitions
assume next3 r p (?cfg p) (HOrcvdMsgs LV-M r p (HOs r p) ?cfg)
  (coords (Suc r) p) (?cfg' p)
  and step r = 3
with dec show ?thesis
  by (auto simp: next3-def send3-def isVote-def LV-CHOMachine-def
                 HOrcvdMsgs-def LV-sendMsg-def)

next
— the other steps don’t change the decision
assume next0 r p (?cfg p) (HOrcvdMsgs LV-M r p (HOs r p) ?cfg)
  (coords (Suc r) p) (?cfg' p)
with dec show ?thesis by (auto simp: next0-def)

next
assume next1 r p (?cfg p) (HOrcvdMsgs LV-M r p (HOs r p) ?cfg)
  (coords (Suc r) p) (?cfg' p)
with dec show ?thesis by (auto simp: next1-def)

next
assume next2 r p (?cfg p) (HOrcvdMsgs LV-M r p (HOs r p) ?cfg)
  (coords (Suc r) p) (?cfg' p)
with dec show ?thesis by (auto simp: next2-def)

qed
hence ready (?cfg (?crd p)) by blast
Because the coordinator is ready, there is a majority of processes that consider it to be the coordinator and that have a current timestamp.

with run
have card \{ q . \?crd q = \?crd p \land \text{timestamp} (\?cfg q) = \text{Suc} (\text{phase} r)\} > N \div 2 \text{ by (rule readyE)}
— Hence there is at least one such process . . .

hence card \{ q . \?crd q = \?crd p \land \text{timestamp} (\?cfg q) = \text{Suc} (\text{phase} r)\} \neq 0 \text{ by arith}
then obtain q where \?crd q = \?crd p \text{ and timestamp} (\?cfg q) = \text{Suc} (\text{phase} r)
by auto
— . . . and by a previous lemma the coordinator must have committed.

with run have \text{commit} (\?cfg (\?crd p))
by (auto elim: currentTimestampE)
with I show \?thesis by (blast intro: conds)
qed

7.7 Proof of Integrity

Integrity is proved using a standard invariance argument that asserts that only values present in the initial state appear in the relevant fields.

lemma lv-integrityInvariant:
assumes run: \text{CHORun LV-M rho HOs coords}
and inv: \[
\text{range} (x \circ (\rho n)) \subseteq \text{range} (x \circ (\rho 0)); \\
\text{range} (\text{vote} \circ (\rho n)) \subseteq \{\text{None}\} \cup \text{Some '\{?x0\}}; \\
\text{range} (\text{decide} \circ (\rho n)) \subseteq \{\text{None}\} \cup \text{Some '\{?x0\}}
\]
\(\Rightarrow A\)
says A
proof —
let \(?x0 = \text{range} (x \circ (\rho 0))\)
let \(?x0opt = \{\text{None}\} \cup \text{Some '\{?x0\}}\)

have range (x \circ (\rho n)) \subseteq \?x0
\land range (\text{vote} \circ (\rho n)) \subseteq \?x0opt
\land range (\text{decide} \circ (\rho n)) \subseteq \?x0opt
(is \?Inv n is \?X n \land \?Vote n \land \?Decide n)

proof (induct n)
from run show \?Inv 0
by (auto simp: CHORun-eq CHOinitConfig-def LV-CHOMachine-def LV-initState-def)

next
fix n
assume ih: \?Inv n thus \?Inv (Suc n)

proof ( clarify)
assume x: \?X n and vt: \?Vote n and dec: \?Decide n

Proof of first conjunct
have x': \?X (Suc n)
proof (clarsimp)
fix p

80
from run
show \( x (\rho (\text{Suc } n) p) \in \text{range } (\lambda q. x (\rho 0 q)) \) (is ?P p n)

proof (rule LV-Suc[\text{where } P=?P])
— only \textit{step1} is of interest
assume \textit{stp}: \textit{step } n = 1
and \textit{nxt}: \textit{next1 } n p (\rho n p)
\begin{align*}
(H\text{Or}cv\text{dM}gs \text{LV-M } n p \ (H\text{Os } n p \ (\rho n)) \\
(coords \ (\text{Suc } n) p \ (\rho (\text{Suc } n) p))
\end{align*}

show \(?\text{thesis}

proof (cases \( \rho (\text{Suc } n) p = \rho n p \))

next

assume \textit{step} \( n = 0 \)
with run have \( x (\rho (\text{Suc } n) p) = x (\rho n p) \)
by (simp add: notStep1EqualX)

with \( x \) show \(?\text{thesis} \) by auto

next

assume \( \textit{step} \ n = 2 \)
with run have \( x (\rho (\text{Suc } n) p) = x (\rho n p) \)
by (simp add: notStep1EqualX)

with \( x \) show \(?\text{thesis} \) by auto

next

assume \( \textit{step} \ n = 3 \)
with run have \( x (\rho (\text{Suc } n) p) = x (\rho n p) \)
by (simp add: notStep1EqualX)

with \( x \) show \(?\text{thesis} \) by auto

qed

qed

Proof of second conjunct

have \( \textit{vt}' : \ ?\text{Vote } (\text{Suc } n) \)
proof (clarsimp simp: image-def)
fix p v
assume v: vote (rho (Suc n) p) = Some v
from run have vote (rho (Suc n) p) = Some v ⟹ v ∈ ?x0 (is ?P p n)
proof (rule LV-Suc'[where P=?P'])
  — here only step0 is of interest
  assume stp: step n = 0
  and nxt: next0 n p (rho n p)
  (HOrcvdMsgs LV-M n p (HOs n p) (rho n))
  (coords (Suc n) p) (rho (Suc n) p)
  show ?thesis
  proof (cases rho (Suc n) p = rho n p)
    case True
    from vt have vote (rho n p) ∈ ?x0opt
      by (auto simp: image-def)
    with True show ?thesis by auto
  next
    case False
    from nxt stp False v obtain q where v = x (rho n q)
      by (auto simp: next0-def send0-def LV-CHOMachine-def
      HOrcvdMsgs-def LV-sendMsg-def)
    with x show ?thesis by (auto simp: image-def)
  qed
  — the other cases don’t change the vote
next
  assume step n = 1
  with run have vote (rho (Suc n) p) = vote (rho n p)
    by (simp add: notStep0EqualVote)
  moreover
  from vt have vote (rho n p) ∈ ?x0opt
    by (auto simp: image-def)
  ultimately
  show ?thesis by auto
next
  assume step n = 2
  with run have vote (rho (Suc n) p) = vote (rho n p)
    by (simp add: notStep0EqualVote)
  moreover
  from vt have vote (rho n p) ∈ ?x0opt
    by (auto simp: image-def)
  ultimately
  show ?thesis by auto
next
  assume step n = 3
  with run have vote (rho (Suc n) p) = vote (rho n p)
    by (simp add: notStep0EqualVote)
  moreover
  from vt have vote (rho n p) ∈ ?x0opt

82
by \( (\text{auto simp: image-def}) \)
ultimately
show \(?\text{thesis by auto} \)
qed
with \( v \) show \( \exists q. \ v = x \ (\rho 0 \ q) \) by \( \text{auto} \)
qed
Proof of third conjunct

have \( \text{dec'}: \ ?\text{Decide (Suc \ n)} \)
proof \( (\text{clarsimp simp: image-def}) \)
fix \( p \ v \)
assume \( v: \ \text{decide (rho (Suc \ n) \ p) = Some \ v} \)
show \( \exists q. \ v = x \ (\rho 0 \ q) \)
proof \( (\text{cases decide (rho (Suc \ n) \ p) = decide (rho \ n \ p)}) \)
\text{case True}
with \( \text{dec True} \ v \) show \( ?\text{thesis by (auto simp: image-def)} \)
next
\text{case False}
let \( \text{?crd = coordΦ (rho \ n \ p)} \)
from False run
have \( \text{d': decide (rho (Suc \ n) \ p) = Some (the (vote (rho \ n \ ?crd)))} \)
\( \text{and cmt: commit (rho \ n \ ?crd)} \)
by \( (\text{auto elim: decisionE}) \)
from \( \text{vt} \)
have \( \text{vtc: vote (rho \ n \ ?crd) ∈ \ ?x0opt} \)
by \( (\text{auto simp: image-def}) \)
from run \( \text{cmt} \) have \( \text{vote (rho \ n \ ?crd) ≠ None} \)
by \( (\text{rule commitE}) \)
with \( \text{d' \ v \ vtc} \) show \( ?\text{thesis by auto} \)
qed
qed
from \( x' \ \text{vt'} \ \text{dec'} \) show \( ?\text{thesis by simp} \)
qed
qed
with \( \text{inv} \) show \( ?\text{thesis by simp} \)
qed

Integrity now follows immediately.

\textbf{theorem lv-integrity:}
\textbf{assumes run: CHORun LV-M rho HOs coords}
\textbf{and dec: decide (rho \ n \ p) = Some \ v}
\textbf{shows} \( \exists q. \ v = x \ (\rho 0 \ q) \)
\textbf{proof –}
from \( \text{run} \) have \( \text{decide (rho \ n \ p) ∈ \ {None} \cup \ ?x0opt} \)
\( \text{range (x ◦ (rho \ 0))} \)
by \( (\text{rule lv-integrityInvariant}) \) \( (\text{auto simp: image-def}) \)
with \( \text{dec} \) show \( ?\text{thesis by (auto simp: image-def)} \)
qed
7.8 Proof of Agreement and Irrevocability

The following lemmas closely follow a hand proof provided by Bernadette Charron-Bost.

If a process decides, then a majority of processes have a current timestamp.

**Lemma decisionThenMajorityBeyondTS:**

**Assumes** run: CHORun LV-M rho HOs coords

and dec: decide (rho (Suc r) p) ≠ decide (rho r p)

**Shows** card (procsBeyondTS (Suc (phase r)) (rho r)) > N div 2

**Using** run dec **proof** (rule decisionE)

Lemma decisionE tells us that we are at step 3 and that the coordinator is ready.

**Let** ?crd = coordΦ (rho r p)

**Let** ?qs = { q . coordΦ (rho r q) = ?crd ∧ timestamp (rho r q) = Suc (phase r) }

**Assume** stp: step r = 3 and rdy: ready (rho r ?crd)

Now, lemma readyE implies that a majority of processes have a recent timestamp.

**From** run rdy **have** card ?qs > N div 2 **by** (rule readyE)

**Moreover**

**From** stp LV-timestamp-bounded[OF run, where n=r] **have** ∀ q. timestamp (rho r q) ≤ Suc (phase r) **by** auto

**Hence** ?qs ⊆ procsBeyondTS (Suc (phase r)) (rho r)

**By** (auto simp: procsBeyondTS-def)

**Hence** card ?qs ≤ card (procsBeyondTS (Suc (phase r)) (rho r))

**By** (intro card-mono) auto

**Ultimately show** ?thesis **by** simp

**Qed**

No two different processes have their commit flag set at any state.

**Lemma committedProcsEqual:**

**Assumes** run: CHORun LV-M rho HOs coords

and cmt: commt (rho r p) and cmt': commt (rho r p')

**Shows** p = p'

**Proof**

**From** run cmt have card { q . coordΦ (rho r q) = p} > N div 2

**By** (blast elim: commtE)

**Moreover**

**From** run cmt' have card { q . coordΦ (rho r q) = p'} > N div 2

**By** (blast elim: commtE)

**Ultimately obtain** q where coordΦ (rho r q) = p and p' = coordΦ (rho r q)

**By** (auto elim: majoritiesE')

**Thus** ?thesis **by** simp

**Qed**

No two different processes have their ready flag set at any state.

**Lemma readyProcsEqual:**
assumes \( run: \text{CHORun} \ LV-M \ rho \ HOs \ coords \) and \( \text{rdy: ready} \ (\rho \ r \ p) \) and \( \text{rdy\': ready} \ (\rho \ r \ p\') \)
shows \( p = p' \)

proof --
   let \( ?C \ p = \{q . \text{coord}\Phi \ (\rho \ r \ q) = p \land \text{timestamp} \ (\rho \ r \ q) = \text{Suc} \ (\text{phase} \ r)\} \)
from \( \text{run rdy have card} \ (?C \ p) > N \div 2 \)
   by (blast elim: \text{readyE})
moreover
from \( \text{run rdy\' have card} \ (?C \ p\') > N \div 2 \)
   by (blast elim: \text{readyE})
ultimately
obtain \( q \) where \( \text{coord}\Phi \ (\rho \ r \ q) = p \) and \( p' = \text{coord}\Phi \ (\rho \ r \ q) \)
by (auto elim: \text{majoritiesE'})
thus \( ?\text{thesis by simp} \)

qed

The following lemma asserts that whenever a process \( p \) commits at a state where a majority of processes have a timestamp beyond \( ts \), then \( p \) votes for a value held by some process whose timestamp is beyond \( ts \).

**lemma commitThenVoteRecent:**
assumes \( \text{run: CHORun} \ LV-M \ rho \ HOs \ coords \) and \( \text{maj: card} \ (\text{procsBeyondTS} \ ts \ (\rho \ r)) > N \div 2 \) and \( \text{cmt: commit} \ (\rho \ r \ p) \)
shows \( \exists q \in \text{procsBeyondTS} \ ts \ (\rho \ r) . \text{vote} \ (\rho \ r \ p) = \text{Some} \ (x \ (\rho \ r \ q)) \)
(is \( ?Q \ r \))

proof --
   let \( ?\text{bynd} n = \text{procsBeyondTS} \ ts \ (\rho \ n) \)
   have \( \text{card} \ (?\text{bynd} \ r) > N \div 2 \land \text{commit} \ (\rho \ r \ p) \rightarrow ?Q \ r \ (\text{is} \ ?P \ p \ r) \)
   proof (rule \text{LV-induct[OF run]})

\( \text{next}\_0 \) establishes the property

\( \text{fix} \ n \)
assume \( \text{stp: step} \ n = 0 \)
and \( \text{nxt:} \forall q . \text{next}\_0 \ n \ q (\rho \ n \ q) \)
   (\( \text{HOrcvdMsgs} \ LV-M \ n \ q \ (\text{HOs} \ n \ q) \ (\rho \ n) \))
   (\( \text{coords} \ (\text{Suc} \ n) \ q) \)
   (\( \rho \ (\text{Suc} \ n) \ q) \)
   (is \( \forall q . \ ?\text{nxt} \ q) \)
from \( \text{nxt have} \ ?\text{nxt} \ \text{p} \ p \ ) ..
show \( ?P \ p \ (\text{Suc} \ n) \)
proof (clarify)
   assume \( \text{mj: card} \ (?\text{bynd} \ (\text{Suc} \ n)) > N \div 2 \)
   and \( \text{ct: commit} \ (\rho \ (\text{Suc} \ n) \ p) \)
   show \( ?Q \ (\text{Suc} \ n) \)
   proof --
   let \( ?\text{msgs} = \text{HOrcvdMsgs} \ LV-M \ n \ p \ (\text{HOs} \ n \ p) \ (\rho \ n) \)
from \( \text{stp run have} \sim \text{commit} \ (\rho \ n \ p) \) by (auto elim: \text{commitE})
with \( ?\text{xp of obtain} \ q \ v \ where \)
   \( v: ?\text{msgs} \ q = \text{Some} \ (\text{ValStamp} \ v \ (\text{highestStampRcvd} \ ?\text{msgs})) \) and
vote: vote (rho (Suc n) p) = Some v and
rcvd: card (valStampsRcvd ?msgs) > N div 2
by (auto simp: next0-def)

from mj rcvd obtain q' where
q1': q' ∈ ?bynd (Suc n) and q2': q' ∈ valStampsRcvd ?msgs
by (rule majoritiesE')
have timestamp (rho n q') ≤ timestamp (rho n q)
proof –
from q2' obtain v' ts'
  where ts': ?msgs q' = Some (ValStamp v' ts')
  by (auto simp: valStampsRcvd-def)
  hence ts' ≤ highestStampRcvd ?msgs
  by (rule highestStampRcvd-max)
moreover
from ts' stp have timestamp (rho n q') = ts'
  by (auto simp: LV-CHOMachine-def HOrcvdMsgs-def
       LV-sendMsg-def send0-def)
moreover
from v stp have timestamp (rho n q) = highestStampRcvd ?msgs
  by (auto simp: LV-CHOMachine-def HOrcvdMsgs-def
       LV-sendMsg-def send0-def)
ultimately
  show ?thesis by simp
qed
moreover
from run stp
have timestamp (rho (Suc n) q) = timestamp (rho n q)
  by (simp add: notStep1EqualTimestamp)
moreover
from run stp
have timestamp (rho (Suc n) q) = timestamp (rho n q)
  by (simp add: notStep1EqualTimestamp)
moreover
note q1'
ultimately
have q ∈ ?bynd (Suc n)
  by (simp add: procsBeyondTS-def)
moreover
from v vote stp
have vote (rho (Suc n) p) = Some (x (rho n q))
  by (auto simp: LV-CHOMachine-def HOrcvdMsgs-def
       LV-sendMsg-def send0-def)
moreover
from run stp have x (rho (Suc n) q) = x (rho n q)
  by (simp add: notStep1EqualX)
ultimately
  show ?thesis by force
qed
qed

86
next

We now prove that \textit{next1} preserves the property. Observe that \textit{next1} may establish a majority of processes with current timestamps, so we cannot just refer to the induction hypothesis. However, if that happens, there is at least one process with a fresh timestamp that copies the vote of the (only) committed coordinator, thus establishing the property.

\begin{verbatim}
fix n
assume stp: step n = 1
and nxt: \forall q. next1 n q (rho n q)
  (HOrcvdMsgs LV-M n q (HOs n q) (rho n))
  (coords (Suc n) q)
  (rho (Suc n) q)
  (is \forall q. \?nxt q)
and ih: P p n
from nxt have nxp: \?nxt p ..
show P p (Suc n)
proof (clarify)
assume mj': card (?bynd (Suc n)) > N div 2
and ct': commit (rho (Suc n) p)
from run stp ct' have ct: commit (rho n p)
  by (simp add: notStep03EqualCommit)
from run stp have vote': vote (rho (Suc n) p) = vote (rho n p)
  by (simp add: notStep0EqualVote)
show \?Q (Suc n)
proof (cases \exists q \in ?bynd (Suc n). rho (Suc n) q \neq rho n q)
case True
in this case the property holds because q updates its x field to the vote
  then obtain q where
    q1: q \in ?bynd (Suc n) and q2: rho (Suc n) q \neq rho n q ..
  from nxt have \?nxt q ..
  with q2 stp
  have x': x (rho (Suc n) q) = the (vote (rho n (coord\Phi (rho n q))))
    and coord: commit (rho n (coord\Phi (rho n q)))
      by (auto simp: next1-def send1-def LV-CHOMachine-def HOrcvdMsgs-def
        LV-sendMsg-def isVote-def)
  from run ct have vote: vote (rho n p) \neq None
    by (rule commitE)
  from run coord ct have coord\Phi (rho n q) = p
    by (rule committedProcsEqual)
  with q1 x' vote show \?thesis by auto
next
  case False
if no relevant process moves then procsBeyondTS doesn’t change and we invoke the induction hypothesis
  hence bynd: ?bynd (Suc n) = ?bynd n
\end{verbatim}
proof (auto simp: procsBeyondTS-def)
fix r
assume ts: ts ≤ timestamp (rho n r)
from run have timestamp (rho n r) ≤ timestamp (rho (Suc n) r)
  by (simp add: LV-timestamp-monotonic)
with ts show ts ≤ timestamp (rho (Suc n) r) by simp
qed
with mj' have mj: card (?bynd n) > N div 2 by simp
with ct ih obtain q where
  q ∈ ?bynd n and vote (rho n p) = Some (x (rho n q))
  by blast
with vote' bynd False show ?thesis by auto
qed

next

step2 preserves the property, via the induction hypothesis.

fix n
assume stp: step n = 2
  and nxt: ∀ q. next2 n q (rho n q)
    (HOrcvdMsgs LV-M n q (HOs n q) (rho n))
    (coords (Suc n) q)
    (rho (Suc n) q)
  (is ∀ q. ?nxt q)
  and ih: ?P p n
from nxt have nxp: ?nxt p ..
show ?P p (Suc n)
proof (clarify)
assume mj': card (?bynd (Suc n)) > N div 2
  and ct': commit (rho (Suc n) p)
from run stp ct' have ct: commit (rho n p)
  by (simp add: notStep03EqualCommit)
from run stp have vote': vote (rho (Suc n) p) = vote (rho n p)
  by (simp add: notStep0EqualVote)
from run stp have ∀ q. timestamp (rho (Suc n) q) = timestamp (rho n q)
  by (simp add: notStep1EqualTimestamp)
  and q ∈ ?bynd n and vote (rho n p) = Some (x (rho n q))
  by blast
with vote' bynd False show ?thesis by auto
qed

the initial state and the step3 transition are trivial because the commit flag cannot be set.

qed (auto elim: commitE[OF run])
with maj cmt show ?thesis by simp

88
The following lemma gives the crucial argument for agreement: after some process \( p \) has decided, all processes whose timestamp is beyond the timestamp at the point of decision contain the decision value in their \( x \) field.

**Lemma XOfTimestampBeyondDecision:**

**Assumes**

- \( \text{run}: \text{CHORun LV-M rho HOs coords} \)
- \( \text{dec}: \text{decide (rho (Suc r) p)} \neq \text{decide (rho r p)} \)

**Shows**

\[ \forall q \in \text{procsBeyondTS} (\text{Suc} (\text{phase} r)) (\text{rho} (r+k)). \]
\[ x (\text{rho} (r+k) q) = \text{the (decide (rho (Suc r) p))} \]

**Is** \( \forall q \in \text{bynd k}. - = ?v \text{is } ?P p k \)

**Proof** (induct \( k \))

- **Base step**
  
  **Show** \( ?P p 0 \)
  
  **Proof** (clarify)
  
  **Fix** \( q \)
  
  **Assume** \( q \in \text{?bynd} 0 \)

**Use preceding lemmas about the decision value and the \( x \) field of processes with fresh timestamps**

**From** \( \text{run dec} \)

**Have** \( \text{stp: step} r = 3 \)

**And** \( v: \text{decide (rho (Suc r) p)} = \text{Some (the (vote (rho r (coordPhi (rho r p)))))} \)

**And** \( \text{cnt: commt (rho r (coordPhi (rho r p)))} \)

**By** (auto elim: decisionE)

**From** \( \text{stp LV-timestamp-bounded}[OF run, \text{where} n=r] \)

**Have** \( \text{timestamp (rho r q) } \leq \text{Suc (phase r)} \) **by** simp

**With** \( q \) **Have** \( \text{timestamp (rho r q) } = \text{Suc (phase r)} \)

**By** (simp add: procsBeyondTS-def)

**With** \( \text{run} \)

**Have** \( \text{x: x (rho r q) } = \text{the (vote (rho r (coordPhi (rho r q))))} \)

**And** \( \text{cnt': commt (rho r (coordPhi (rho r q)))} \)

**By** (auto elim: currentTimeStampE)

**From** \( \text{run cnt cnt'} \) **Have** \( \text{coordPhi (rho r p) } = \text{coordPhi (rho r q)} \)

**By** (rule committedProcsEqual)

**With** \( x v \) **Show** \( x (\text{rho (r+0) q) } = ?v \) **by** simp

**Qed**

**Next**

- **Induction step**

**Fix** \( k \)

**Assume** \( \text{ih: } ?P p k \)

**Show** \( ?P p (\text{Suc k}) \)

**Proof** (clarify)

**Fix** \( q \)

**Assume** \( q \in \text{bynd} (\text{Suc k}) \)

- **Distinguish the kind of transition—only step1 is interesting**

**Have** \( x (\text{rho (Suc (r+k)) q) } = ?v \) **(is ?X q (r+k))**

**Proof** (rule LV-Suc[OF run, where \( P=?X]]\)

**Assume** \( \text{stp: step} (r+k) = 1 \)

89
and \( \text{nxt: next1 } (r+k) q \text{ (rho (r+k)) } \)
\( (\text{HOrcvdMsgs LV-M } (r+k) q \text{ (HOs (r+k)) } \text{ (rho (r+k))}) \)
\( (\text{coords } (\text{Suc } (r+k)) q) \)
\( (\text{rho } (\text{Suc } (r+k)) q) \)

show \( \text{?thesis} \)
proof (cases \( \text{rho } (\text{Suc } (r+k)) q = \text{rho } (r+k) q \))
case \text{True}
with \( q \) \( \text{ih} \) show \( \text{?thesis} \) by (auto simp: procsBeyondTS-def)
next
case \text{False}
from \( \text{run dec} \) have \( \text{card } (\text{bynd } 0) > N \text{ div } 2 \)
by (simp add: decisionThenMajorityBeyondTS)
moreover
have \( \text{bynd } 0 \subseteq \text{bynd } k \)
by (auto elim: procsBeyondTS-monotonic[OF run])
hence \( \text{card } (\text{bynd } 0) \leq \text{card } (\text{bynd } k) \)
by (auto intro: card-mono)
ultimately
have \( \text{maj: card } (\text{bynd } k) > N \text{ div } 2 \text{ by simp} \)
let \( ?\text{crd} = \text{coord}(\text{Phi } (\text{rho } (r+k) q) \)
from \( \text{False stp nxt have} \)
\( \text{cmt: commit } (\text{rho } (r+k) ?\text{crd}) \text{ and} \)
\( x: x (\text{rho } (\text{Suc } (r+k)) q) = \text{the } (\text{vote } (\text{rho } (r+k) ?\text{crd})) \)
by (auto simp: next1-def LV-CHOMachine-def HOrcvdMsgs-def LV-sendMsg-def send1-def isVote-def)
from \( \text{run maj cmt stp obtain } q' \)
\( \text{where } q_1': q' \in \text{?bynd } k \)
and \( q_2': \text{vote } (\text{rho } (r+k) ?\text{crd}) = \text{Some } (x (\text{rho } (r+k) q')) \)
by (blast dest: commitThenVoteRecent)
with \( x \) \( \text{ih} \) show \( ?\text{thesis} \) by auto
qed
next
— all other steps hold by induction hypothesis
assume \( \text{step } (r+k) = 0 \)
with \( \text{run have} \ x: x (\text{rho } (\text{Suc } (r+k)) q) = x (\text{rho } (r+k) q) \)
and \( ts: \text{timestamp } (\text{rho } (\text{Suc } (r+k)) q) = \text{timestamp } (\text{rho } (r+k) q) \)
by (auto simp: notStep1EqualX notStep1EqualTimestamp)
from \( ts \) \( q \) \( \text{have} \ q \in \text{?bynd } k \)
by (auto simp: procsBeyondTS-def)
with \( x \) \( \text{ih} \) show \( ?\text{thesis} \) by auto
next
assume \( \text{step } (r+k) = 2 \)
with \( \text{run have} \ x: x (\text{rho } (\text{Suc } (r+k)) q) = x (\text{rho } (r+k) q) \)
and \( ts: \text{timestamp } (\text{rho } (\text{Suc } (r+k)) q) = \text{timestamp } (\text{rho } (r+k) q) \)
by (auto simp: notStep1EqualX notStep1EqualTimestamp)
from \( ts \) \( q \) \( \text{have} \ q \in \text{?bynd } k \)
by (auto simp: procsBeyondTS-def)
with \( x \) \( \text{ih} \) show \( ?\text{thesis} \) by auto
next
assume \( \text{step } (r+k) = 3 \)

with run have \( x (\text{rho} \ (r+k)) q = x (\text{rho} \ (r+k)) q \)
and \( ts\): \( \text{timestamp} (\text{rho} \ (r+k)) q = \text{timestamp} (\text{rho} \ (r+k)) q \)
by (auto simp: notStep1EqualX notStep1EqualTimestamp)
from \( ts \ q \) have \( q \in \ ?bynd k \)
by (auto simp: procsBeyondTS-def)
with \( x \ ih \) show \( \?thesis \) by auto
qed

thus \( x (\text{rho} \ (r + \text{Suc} \ k) q) = \?v \) by simp
qed
qed

We are now in position to prove Agreement: if some process decides at step \( r \) and another (or possibly the same) process decides at step \( r+k \) then they decide the same value.

**Lemma laterProcessDecidesSameValue:**

assumes \( \text{run}: \text{CHORun} \ LV-M \ \rhoos \ \text{coords} \)
and \( p\): decide (\( \text{rho} \ (\text{Suc} \ r) \ p \)) \( \neq \) decide (\( \text{rho} \ (r) \ p \))
and \( q\): decide (\( \text{rho} \ (\text{Suc} \ (r+k)) q \)) \( \neq \) decide (\( \text{rho} \ (r+k) q \))
shows decide (\( \text{rho} \ (\text{Suc} \ (r+k)) q \)) = decide (\( \text{rho} \ (\text{Suc} \ r) \ p \))

proof —
let \( \?bynd k = \text{procsBeyondTS} \ (\text{Suc} \ (\text{phase} \ r)) \ (\text{rho} \ (r+k)) \)
let \( ?qcrd = \text{coord} \Phi \ (\text{rho} \ (r+k) q) \)
from \( \text{run} \ p \) have \( \text{notNone}: \text{decide} \ (\text{rho} \ (\text{Suc} \ r) \ p) \neq \text{None} \)
by (auto elim: decisionE)
— process \( q \) decides on the vote of its coordinator
from \( \text{run} \ q \) have \( \text{dec}: \text{decide} \ (\text{rho} \ (\text{Suc} \ (r+k)) q) = \text{Some} \ \text{(the} \ (\text{vote} \ (\text{rho} \ (r+k) \ ?qcrd))) \)
and \( \text{cmt}: \text{commit} \ (\text{rho} \ (r+k) \ ?qcrd) \)
by (auto elim: decisionE)
— that vote is the \( x \) field of some process \( q' \) with a recent timestamp
from \( \text{run} \ p \) have \( \text{card} \ (\?bynd \ 0) > N \div 2 \)
by (simp add: decisionThenMajorityBeyondTS)
moreover
from \( \text{run} \) have \( \?bynd \ 0 \subseteq \?bynd k \)
by (auto elim: procsBeyondTS-monotonic)
hence \( \text{card} \ (\?bynd \ 0) \leq \text{card} \ (\?bynd \ k) \)
by (auto intro: card-mono)
ultimately
have \( \text{maj}: \text{card} \ (\?bynd \ k) > N \div 2 \) by simp
from \( \text{run} \ \text{maj} \ \text{cmt} \) obtain \( q' \)
where \( q'1: q' \in \ ?bynd k \)
and \( q'2: \text{vote} \ (\text{rho} \ (r+k) \ ?qcrd) = \text{Some} \ (x \ (\text{rho} \ (r+k) q')) \)
by (auto dest: commitThenVoteRecent)
— the \( x \) field of process \( q' \) is the value \( p \) decided on
from \( \text{run} \ p \ q'1 \) have \( x \ (\text{rho} \ (r+k) q') = \text{the} \ (\text{decide} \ (\text{rho} \ (\text{Suc} \ r) \ p)) \)
by (auto dest: XOfTimestampBeyondDecision)
— which proves the assertion

91
A process that holds some decision \( v \) has decided \( v \) sometime in the past.

**Lemma** *Decision NonNull Then Decided:

**Assumes**
- \( \text{run} : \text{CHORun LV-M rho HOs coords} \)
- \( \text{dec} : \text{decide (rho n p)} = \text{Some} \ v \)

**Shows**
- \( \exists m < n \). \( \text{decide (rho (Suc m) p)} \neq \text{decide (rho m p)} \)
- \( \wedge \text{decide (rho (Suc m) p)} = \text{Some} \ v \)

**Proof**
- Let \( \forall n. \text{dec} \ (\text{Suc} \ n) \neq \text{dec} \ n \)
- \( \forall m < n. \text{dec} \ (\text{Suc} \ m) \neq \text{dec} \ (\text{Suc} \ m) \)
- \( \wedge \text{dec} \ (\text{Suc} \ n) = \text{Some} \ v \)

**Theorem** *LV Irrevocability:

**Assumes**
- \( \text{run} : \text{CHORun LV-M rho HOs coords} \)
- \( \text{dec} : \text{decide (rho m p)} = \text{Some} \ v \)

**Shows**
- \( \text{decide (rho (Suc (m+k) p)} = \text{Some} \ v \)

**Proof**
- From \( \text{run p} \) obtain \( n \) where
  - \( n1 : n < m \)
  - \( n2 : \text{decide (rho (Suc n) p)} \neq \text{decide (rho n p)} \)
  - \( n3 : \text{decide (rho (Suc n) p)} = \text{Some} \ v \)
- By (auto dest: decisionNonNullThenDecided)
have \( \forall i. \text{decide} \ (\rho \ (\text{Suc} \ (n+i))) \ p) = \text{Some} \ v \ (\text{is} \ \forall \ ?\text{dec} \ i) \\
\text{proof} \\
\text{fix} \ i \ \\
\text{show} \ ?\text{dec} \ i \\
\text{proof} \ (\text{induct} \ i) \\
\text{from} \ n3 \ \text{show} \ ?\text{dec} 0 \ \text{by} \ \text{simp} \\
\text{next} \\
\text{fix} \ j \ \\
\text{assume} \ ih: \ ?\text{dec} \ j \\
\text{show} \ ?\text{dec} \ (\text{Suc} \ j) \\
\text{proof} \ (\text{rule} \ \text{ccontr}) \\
\text{assume} \ ctr: \ \neg \ (\exists \ ?\text{dec} \ (\text{Suc} \ j)) \\
\text{with} \ ih \\
\text{have} \ \text{decide} \ (\rho \ (\text{Suc} \ (n + \text{Suc} \ j))) \ p) = \neg \ \text{decide} \ (\rho \ (n + \text{Suc} \ j)) \ p) \\
\text{by} \ \text{simp} \\
\text{with} \ \text{run} \ n2 \\
\text{have} \ \text{decide} \ (\rho \ (n + \text{Suc} \ j)) p = \text{decide} \ (\rho \ (\text{Suc} \ n)) p \\
\text{by} \ (\text{rule} \ \text{laterProcessDecidesSameValue}) \\
\text{with} \ ctr \ n3 \ \text{show} \ False \ \text{by} \ \text{simp} \\
\text{qed} \\
\text{qed} \\
\text{qed} \\
\text{moreover} \\
\text{from} \ n1 \ \text{obtain} \ j \ \text{where} \ m + k = \text{Suc}(n+j) \\
\text{by} \ (\text{auto} \ \text{dest}: \ \text{less-imp-Suc-add}) \\
\text{ultimately} \\
\text{show} \ ?\text{thesis} \ \text{by} \ \text{auto} \\
\text{qed} \\
\text{theorem} \ \text{lv-agreement}: \\
\text{assumes} \ \text{run: CHORun LV-M \rho HOs coords} \\
\text{and} \ p: \ \text{decide} \ (\rho \ m) = \text{Some} \ v \\
\text{and} \ q: \ \text{decide} \ (\rho \ n) = \text{Some} \ w \\
\text{shows} \ v = w \\
\text{proof} \ \\
\text{from} \ \text{run} \ p \ \text{obtain} \ k \\
\text{where} \ k1: \ \text{decide} \ (\rho \ (\text{Suc} \ k)) p = \neg \ \text{decide} \ (\rho \ k) p \\
\text{and} \ k2: \ \text{decide} \ (\rho \ (\text{Suc} \ k)) p = \text{Some} \ v \\
\text{by} \ (\text{auto} \ \text{dest}: \ \text{decisionNonNullThenDecided}) \\
\text{from} \ \text{run} \ q \ \text{obtain} \ l \\
\text{where} \ l1: \ \text{decide} \ (\rho \ (\text{Suc} \ l)) q = \neg \ \text{decide} \ (\rho \ l) q \\
\text{and} \ l2: \ \text{decide} \ (\rho \ (\text{Suc} \ l)) q = \text{Some} \ w \\
\text{by} \ (\text{auto} \ \text{dest}: \ \text{decisionNonNullThenDecided}) \\
\text{show} \ ?\text{thesis} \\
\text{proof} \ (\text{cases} \ k \leq \ l) \\
\text{case} \ True \\
\text{then} \ \text{obtain} \ m \ \text{where} \ m: \ l = k+m \ \text{by} \ (\text{auto} \ \text{simp: le-iiff-add}) \\
\text{from} \ \text{run} \ k1 \ l1 \ m \\
\text{have} \ \text{decide} \ (\rho \ (\text{Suc} \ l)) q = \text{decide} \ (\rho \ (\text{Suc} \ k)) p \\
\ \text{qed}
7.9 Proof of Termination

The proof of termination relies on the communication predicate, which stipulates the existence of some phase during which there is a single coordinator that (a) receives a majority of messages and (b) is heard by everybody. Therefore, all processes successfully execute the protocol, deciding at step 3 of that phase.

**Theorem lv-termination:**

**Assumes** run: \(\text{CHORun } LV-M \rho HOs coords\)

**And** commG: \(\text{CHOcommGlobal } LV-M HOs coords\)

**Shows** \(\exists r. \forall p. \text{decide } (\rho r p) \neq \text{None}\)

**Proof**

The communication predicate implies the existence of a “successful” phase \(ph\), coordinated by some process \(c\) for all processes.

**Step 1**

**From** commG **obtain** \(ph c\) with \(c: \forall p. \text{coords } (4*ph) p = c\)

**And** maj0: card (HOs \((4*ph) c\)) > \(N \div 2\)

**And** maj2: card (HOs \((4*ph+2) c\)) > \(N \div 2\)

**And** rcv1: \(\forall p. c \in \text{HOs } (4*ph+1) p\)

**And** rcv3: \(\forall p. c \in \text{HOs } (4*ph+3) p\) by (auto simp: LV-CHOMachine-def LV-commGlobal-def)

**Let** \(?r0 = 4*ph\)

**Let** \(?r1 = \text{Suc } ?r0\)

**Let** \(?r2 = \text{Suc } ?r1\)

**Let** \(?r3 = \text{Suc } ?r2\)

**Let** \(?r4 = \text{Suc } ?r3\)

Process \(c\) is the coordinator of all steps of phase \(ph\).

**From** run \(c\) **have** \(c: \forall p. \text{coord}(\rho ?r p) = c\) by (auto simp add: phase-def coordinators)

**With** run **have** \(c1: \forall p. \text{coord}(\rho ?r1 p) = c\) by (auto simp add: step-def mod-Suc notStep3EqualCoord)

**With** run **have** \(c2: \forall p. \text{coord}(\rho ?r2 p) = c\) by (auto simp add: step-def mod-Suc notStep3EqualCoord)
with run have c3: ∀ p. coordΦ (rho ?r3 p) = c by (auto simp add: step-def mod-Suc notStep3EqualCoord)

The coordinator receives ValStamp messages from a majority of processes at step 0 of phase ph and therefore commits during the transition at the end of step 0.

have 1: commit (rho ?r1 c) (is ?P c (4∗ph))
proof (rule LV-Suc′[OF run, where P=?P], auto simp: step-def)
assume next0 ?r c (rho ?r c) (HOrcvdMsgs LV-M ?r c (HOs ?r c) (rho ?r)) (coords (Suc ?r) c) (rho (Suc ?r) c)
with c' maj0 show commit (rho (Suc ?r) c) by (auto simp: step-def next0-def send0-def valStampsRcvd-def LV-CHOMachine-def HOrcvdMsgs-def LV-sendMsg-def)

qed

All processes receive the vote of c at step 1 and therefore update their time stamps during the transition at the end of step 1.

have 2: ∀ p. timestamp (rho ?r2 p) = Suc ph
proof
fix p
let ?msgs = HOrcvdMsgs LV-M ?r1 p (HOs ?r1 p) (rho ?r1)
let ?crd = coordΦ (rho ?r1 p)
from run 1 c1 rcv1
have cnd: ?msgs ?crd ≠ None ∧ isVote (the (?msgs ?crd))
  by (auto elim: commitE simp: step-def LV-CHOMachine-def HOrcvdMsgs-def LV-sendMsg-def send1-def isVote-def)
show timestamp (rho ?r2 p) = Suc ph (is ?P p (Suc (4∗ph)))
proof (rule LV-Suc′[OF run, where P=?P], auto simp: step-def mod-Suc)
assume next1 ?r1 p (rho ?r1 p) (?msgs (coords (Suc ?r1) p) (rho ?r2 p))
with cnd show ?thesis by (auto simp: next1-def phase-def)
qed

qed

The coordinator receives acknowledgements from a majority of processes at step 2 and sets its ready flag during the transition at the end of step 2.

have 3: ready (rho ?r3 c) (is ?P c (Suc (Suc (4∗ph))))
proof (rule LV-Suc′[OF run, where P=?P], auto simp: step-def mod-Suc)
assume next2 ?r2 c (rho ?r2 c)
  (HOrcvdMsgs LV-M ?r2 c (HOs ?r2 c) (rho ?r2)) (coords (Suc ?r2) c (rho ?r3 c))
with 2 c2 maj2 show ?thesis
  by (auto simp: mod-Suc step-def LV-CHOMachine-def HOrcvdMsgs-def LV-sendMsg-def next2-def send2-def acksRcvd-def isAck-def phase-def)

qed

All processes receive the vote of the coordinator during step 3 and decide during the transition at the end of that step.

have 4: ∀ p. decide (rho ?r4 p) ≠ None
proof
let \(?msgs = \text{HOrcvdMsgs LV-M ?r3 p (HOs ?r3 p) (rho ?r3)}\)
let \(?crd = \text{coord} (\text{rho ?r3 p})\)
from \(\text{run 3 c3 rev3}\)
have \(\text{cnd: ?msgs ?crd} \neq \text{None} \land \text{isVote (the (?msgs ?crd))}\)
by (auto elim: \text{readyE}
  simp: \text{step-def mod-Suc LV-CHOMachine-def HOrcvdMsgs-def}
  \text{LV-sendMsg-def send3-def isVote-def numeral-3-eq-3})
show \(\text{decide (rho ?r4 p)} \neq \text{None (is ?P p (Suc (Suc (Suc (4*ph)))))}\)
proof (rule \text{LV-Suc[OF run, where P=?P], auto simp: step-def mod-Suc})
  assume \(\text{next3 ?r3 p (rho ?r3 p) ?msgs (coords (Suc ?r3) p) (rho ?r4 p)}\)
  with \(\text{cnd show } \exists \text{ v. decide (rho ?r4 p) = Some v}\)
  by (auto simp: \text{next3-def})
qed

This immediately proves the assertion.

\(\text{from 4 show } \text{?thesis} \ldots\)

\(\text{qed}\)

7.10 LastVoting Solves Consensus

Summing up, all (coarse-grained) runs of LastVoting for HO collections that
satisfy the communication predicate satisfy the Consensus property.

theorem \text{lv-consensus}:\nassumes \text{run: CHORun LV-M rho HOs coords}
  \text{and commG: CHOcommGlobal LV-M HOs coords}
shows \(\text{consensus (x o (rho 0)) decide rho}\)
proof — the above statement of termination is stronger than what we need
from \text{lv-termination[OF assms]}
obtain \(r\) where \(\forall p. \text{decide (rho r p)} \neq \text{None} \ldots\)
hence \(\forall p. \exists r. \text{decide (rho r p)} \neq \text{None by blast}\)
with \text{lv-integrity[OF run] lv-agreement[OF run]}
show \(\text{?thesis by (auto simp: consensus-def image-def)}\)
qed

By the reduction theorem, the correctness of the algorithm carries over to
the fine-grained model of runs.

theorem \text{lv-consensus-fg}:\nassumes \text{run: fg-run LV-M rho HOs HOs coords}
  \text{and commG: CHOcommGlobal LV-M HOs coords}
shows \(\text{consensus (\lambda p. x (state (rho 0) p)) decide (state o rho)}\)
  \(\text{(is consensus ?inits - -)}\)
proof (rule \text{local-property-reduction[OF run consensus-is-local]})
fix \(\text{crun}\)
assume \text{crun: CSHORun LV-M crun HOs HOs coords}
\text{and} \ \text{init:} \ crun \ 0 = \text{state} (\rho \ 0) \\
\text{from} \ crun \ \text{have} \ \text{CHORun} \ LV-M \ crun \ \text{HOs coords}  \\
\quad \text{by} \ (\text{unfold} \ \text{CHORun-def} \ \text{SHORun-def}) \\
\text{from} \ \text{this} \ commG \ \text{have} \ \text{consensus} \ (x \circ (\text{crun} \ 0)) \ \text{decide crun}  \\
\quad \text{by} \ (\text{rule lv-consensus}) \\
\text{with} \ \text{init} \ \text{show} \ \text{consensus} \ ?\text{inits} \ \text{decide crun}  \\
\quad \text{by} \ (\text{simp add: o-def}) \\
\text{qed} \\
\end{theory}

theory \ UteDefs 
imports ..../HOModel 
begin 

8 Verification of the \(\mathcal{U}_{T,E,\alpha}\) Consensus Algorithm

Algorithm \(\mathcal{U}_{T,E,\alpha}\) is presented in [3]. It is an uncoordinated algorithm that tolerates value (a.k.a. Byzantine) faults, and can be understood as a variant of \textit{UniformVoting}. The parameters \(T\), \(E\), and \(\alpha\) appear as thresholds of the algorithm and in the communication predicates. Their values can be chosen within certain bounds in order to adapt the algorithm to the characteristics of different systems.

We formalize in Isabelle the correctness proof of the algorithm that appears in [3], using the framework of theory \textit{HOModel}.

8.1 Model of the Algorithm

We begin by introducing an anonymous type of processes of finite cardinality that will instantiate the type variable \('proc\) of the generic HO model.

typedecl \ Proc — the set of processes 
axiomatization where \ Proc-finite: OFCLASS(Proc, finite-class) 
instance \ Proc :: finite by (rule Proc-finite)

abbreviation 
\(N \equiv \text{card} \ (\text{UNIV::Proc set}) — \text{number of processes}\)

The algorithm proceeds in \textit{phases} of 2 rounds each (we call \textit{steps} the individual rounds that constitute a phase). The following utility functions compute the phase and step of a round, given the round number.

abbreviation 
\(n\text{Steps} \equiv 2\) 
\text{definition} \ phase \ where \ phase \ (r::nat) \equiv r \ div \ n\text{Steps} 
\text{definition} \ step \ where \ step \ (r::nat) \equiv r \ mod \ n\text{Steps}

lemma \ phase-zero [simp]: \ phase \ 0 = 0 
by (simp add: phase-def)
lemma step-zero [simp]: step 0 = 0
by (simp add: step-def)

lemma phase-step: (phase r * nSteps) + step r = r
by (auto simp add: phase-def step-def)

The following record models the local state of a process.

record 'val pstate =
x :: 'val — current value held by process
vote :: 'val option — value the process voted for, if any
decide :: 'val option — value the process has decided on, if any

Possible messages sent during the execution of the algorithm.

datatype 'val msg =
  Val 'val |
  Vote 'val option

The x field of the initial state is unconstrained, all other fields are initialized appropriately.

definition Ute-initState where
Ute-initState p st ≡ (vote st = None) ∧ (decide st = None)

The following locale introduces the parameters used for the $U_{T,E,\alpha}$ algorithm and their constraints [3].

locale ute-parameters =
fixes α :: nat and T :: nat and E :: nat
assumes majE: $2 \cdot E \geq N + 2 \cdot \alpha$
and majT: $2 \cdot T \geq N + 2 \cdot \alpha$
and EltN: $E < N$
and TltN: $T < N$

begin

Simple consequences of the above parameter constraints.

lemma alpha-lt-N: $\alpha < N$
using EltN majE by auto

lemma alpha-lt-T: $\alpha < T$
using majT alpha-lt-N by auto

lemma alpha-lt-E: $\alpha < E$
using majE alpha-lt-N by auto

We separately define the transition predicates and the send functions for each step and later combine them to define the overall next-state relation.

In step 0, each process sends its current $x$. If it receives the value $v$ more than $T$ times, it votes for $v$, otherwise it doesn’t vote.
In step 1, each process sends its current vote. If it receives more than $\alpha$ votes for a given value $v$, it sets its $x$ field to $v$, else it sets $x$ to a default value.

If the process receives more than $E$ votes for $v$, it decides $v$, otherwise it leaves its decision unchanged.

The overall send function and next-state relation are simply obtained as the composition of the individual relations defined above.
8.2 Communication Predicate for $U_{T,E,\alpha}$

Following [3], we now define the communication predicate for the $U_{T,E,\alpha}$ algorithm to be correct. The round-by-round predicate stipulates the following conditions:

- no process may receive more than $\alpha$ corrupted messages, and
- every process should receive more than $\max(T, N + 2\alpha - E - 1)$ correct messages.

[3] also requires that every process should receive more than $\alpha$ correct messages, but this is implied, since $T > \alpha$ (cf. lemma alpha-lt-T).

definition Ute-commPerRd where

\[
\forall p. \text{card}(\text{HOs} p - \text{SHO}s p) \leq \alpha \\
\land \text{card}(\text{SHO}s p \cap \text{HOs} p) > N + 2\alpha - E - 1 \\
\land \text{card}(\text{SHO}s p \cap \text{HOs} p) > T
\]

The global communication predicate requires there exists some phase $\Phi$ such that:

- all HO and SHO sets of all processes are equal in the second step of phase $\Phi$, i.e. all processes receive messages from the same set of processes, and none of these messages is corrupted,
- every process receives more than $T$ correct messages in the first step of phase $\Phi+1$, and
- every process receives more than $E$ correct messages in the second step of phase $\Phi+1$.

The predicate in the article [3] requires infinitely many such phases, but one is clearly enough.

\[
\exists \Phi. \text{(let } r = \text{Succ} (\text{nSteps} \times \Phi) \text{ in } (\exists \pi, \forall p. \pi = \text{HOs} r p \land \pi = \text{SHO}s r p) \\
\land (\forall p. \text{card}(\text{SHO}s (\text{Succ} r) p \cap \text{HOs} (\text{Succ} r) p) > T) \\
\land (\forall p. \text{card}(\text{SHO}s (\text{Succ} (\text{Succ} r)) p \cap \text{HOs} (\text{Succ} (\text{Succ} r)) p) > E))
\]

8.3 The $U_{T,E,\alpha}$ Heard-Of Machine

We now define the coordinated HO machine for the $U_{T,E,\alpha}$ algorithm by assembling the algorithm definition and its communication-predicate.

definition Ute-SHOMachine where

\[
\text{Ute-SHOMachine} = []
\]

100
CinitState = (λ p st crd. Ute-initState p st),
sendMsg = Ute-sendMsg,
CnextState = (λ r p st msgs st!. Ute-nextState r p st msgs st'),
SHOcommPerRd = Ute-commPerRd,
SHOcommGlobal = Ute-commGlobal
|

abbreviation
Ute-M ≡ (Ute-SHOMachine::(Proc, 'val pstate, 'val msg) SHOMachine)

end — locale ute-parameters

end
theory UteProof
imports UteDefs ../Majorities ../Reduction
begin
context ute-parameters
begin

8.4 Preliminary Lemmas

Processes can make a vote only at first round of each phase.

lemma vote-step:
assumes next: nextState Ute-M r p (rho r p) μ (rho (Suc r) p)
and vote (rho (Suc r) p) ≠ None
shows step r = 0
proof (rule contr)
assume step r ≠ 0
with assms have vote (rho (Suc r) p) = None
   by (auto simp:Ute-SHOMachine-def nextState-def Ute-nextState-def next1-def)
with assms show False by auto
qed

Processes can make a new decision only at second round of each phase.

lemma decide-step:
assumes run: SHORun Ute-M rho HOs SHOs
and d1: decide (rho r p) ≠ Some v
and d2: decide (rho (Suc r) p) = Some v
shows step r ≠ 0
proof
assume sr:step r = 0
from run obtain μ where Ute-nextState r p (rho r p) μ (rho (Suc r) p)
   unfolding Ute-SHOMachine-def nextState-def SHORun-eq SHOnextConfig-eq
by force
with sr have next0 r p (rho r p) μ (rho (Suc r) p)
   unfolding Ute-nextState-def by auto
hence decide (rho r p) = decide (rho (Suc r) p)
   by (auto simp:next0-def)
with d1 d2 show False by auto

qed

lemma unique-majority-E:
  assumes majv: card {qq::Proc. F qq = Some m} > E
  and majw: card {qq::Proc. F qq = Some m'} > E
  shows m = m'
proof –
  from majv majw majE
  have card {qq::Proc. F qq = Some m} > N div 2
    and card {qq::Proc. F qq = Some m'} > N div 2
    by auto
  then obtain qq
    where qq ∈ {qq::Proc. F qq = Some m}
    and qq ∈ {qq::Proc. F qq = Some m'}
    by (rule majoritiesE')
  thus ?thesis by auto
qed

lemma unique-majority-E-α:
  assumes majv: card {qq::Proc. F qq = Some m} > E − α
  and majw: card {qq::Proc. F qq = Some m'} > E − α
  shows m = m'
proof –
  from majE alpha-lt-N majv majw
  have card {qq::Proc. F qq = m} > N div 2
    and card {qq::Proc. F qq = m'} > N div 2
    by auto
  then obtain qq
    where qq ∈ {qq::Proc. F qq = m}
    and qq ∈ {qq::Proc. F qq = m'}
    by (rule majoritiesE')
  thus ?thesis by auto
qed

lemma unique-majority-T:
  assumes majv: card {qq::Proc. F qq = Some m} > T
  and majw: card {qq::Proc. F qq = Some m'} > T
  shows m = m'
proof –
  from majT majv majw
  have card {qq::Proc. F qq = Some m} > N div 2
    and card {qq::Proc. F qq = Some m'} > N div 2
    by auto
  then obtain qq
    where qq ∈ {qq::Proc. F qq = Some m}
    and qq ∈ {qq::Proc. F qq = Some m'}
    by (rule majoritiesE')
  thus ?thesis by auto
No two processes may vote for different values in the same round.

**Lemma** common-vote:
- **assumes** usafe: SHOcommPerRd Ute-M HO SHO
- **and** nxtxp: nextState Ute-M r p (rho r p) mu p (rho (Suc r) p)
- **and** mup: mu p ∈ SHOmsgVectors Ute-M r p (rho r) (HO p) (SHO p)
- **and** nxtq: nextState Ute-M r q (rho r q) mu q (rho (Suc r) q)
- **and** muq: mu q ∈ SHOmsgVectors Ute-M r q (rho r) (HO q) (SHO q)
- **and** vp: vote (rho (Suc r) p) = Some v
- **and** vq: vote (rho (Suc r) q) = Some v

**shows** vp = vq using assms proof

```plaintext
have gtn: card {qq. sendMsg Ute-M r qq p (rho r qq) = Val v} + card {qq. sendMsg Ute-M r qq q (rho r qq) = Val v} > N

proof -
  have card {qq. sendMsg Ute-M r qq p (rho r qq) = Val v} > T - α
  ∧ card {qq. sendMsg Ute-M r qq q (rho r qq) = Val v} > T - α
  (is card ?vrcvdp - ?ahop ⊆ ?vsentp)
  by (auto simp: SHOmsgVectors-def)
  hence card ?vrcvdp - ?ahop ≤ card ?vsentp
  and card (?vrcvdp - ?ahop) ≥ card ?vrcvdp - card ?ahop
  by (auto simp: card-mono diff-card-le-card-Diff)
  hence card ?vsentp ≥ card ?vrcvdp - card ?ahop by auto
moreover
from nxtxp vp have stp:step r = 0 by (auto simp: vote-step)
from mup
have {qq. µp qq = Some (Val v)} - (HO p - SHO p)
  ⊆ {qq. sendMsg Ute-M r qq p (rho r qq) = Val v}
  (is ?vrcvdp - ?ahop ⊆ ?vsentp)
  by (auto simp: SHOmsgVectors-def)
  hence card (?vrcvdp - ?ahop) ≤ card ?vsentp
  and card (?vrcvdp - ?ahop) ≥ card ?vrcvdp - card ?ahop
  by (auto simp: card-mono diff-card-le-card-Diff)
  hence card ?vsentp ≥ card ?vrcvdp - card ?ahop by auto
moreover
from nxtq stp have next0 r p (rho r p) µq (rho (Suc r) p)
  by (auto simp: Ute-SHOMachine-def nextState-def Ute-nextState-def)
  with vp have card ?vrcvdp > T
  unfolding next0-def by auto
moreover
from muq
have {qq. µq qq = Some (Val v)} - (HO q - SHO q)
  ⊆ {qq. sendMsg Ute-M r qq q (rho r qq) = Val v}
  (is ?vrcvdq - ?ahoq ⊆ ?vsentq)
  by (auto simp: SHOmsgVectors-def)
  hence card (?vrcvdq - ?ahoq) ≤ card ?vsentq
  and card (?vrcvdq - ?ahoq) ≥ card ?vrcvdq - card ?ahoq
  by (auto simp: card-mono diff-card-le-card-Diff)
  hence card ?vsentq ≥ card ?vrcvdq - card ?ahoq by auto
moreover
from nxtq stp have next0 r q (rho r q) µq (rho (Suc r) q)
  by (auto simp: Ute-SHOMachine-def nextState-def Ute-nextState-def)
  with vq have card {qq. µq qq = Some (Val v)} > T

qed
```
by (unfold next0-def, auto)
moreover
from usafe have card ?ahop ≤ α and card ?ahoq ≤ α
  by (auto simp: Ute-SHOMachine-def Ute-commPerRd-def)
ultimately
show ?thesis using alpha-lt-T by auto
qed
thus ?thesis using majT by auto
qed

show ?thesis
proof (rule ccontr)
assume vpq:vp ≠ vq
have ∀ qq. sendMsg Ute-M r qq p (rho r qq) = sendMsg Ute-M r qq q (rho r qq)
  by (auto simp: Ute-SHOMachine-def Ute-sendMsg-def step-def send0-def send1-def)
with vpq
have {qq. sendMsg Ute-M r qq p (rho r qq) = Val vp} ∩ {qq. sendMsg Ute-M r qq q (rho r qq) = Val vq} = {}
  by auto
with gtn
have card ({qq. sendMsg Ute-M r qq p (rho r qq) = Val vp} ∪ {qq. sendMsg Ute-M r qq q (rho r qq) = Val vq}) > N
  by (auto simp: card-Un-Int)
moreover
have card ({qq. sendMsg Ute-M r qq p (rho r qq) = Val vp} ∪ {qq. sendMsg Ute-M r qq q (rho r qq) = Val vq}) ≤ N
  by (auto simp: card-mono)
ultimately
show False by auto
qed
qed

No decision may be taken by a process unless it received enough messages
holding the same value.

lemma decide-with-threshold-E:
assumes run: SHORun Ute-M rho HOs SHOs
and usafe: SHOcommPerRd Ute-M (HOs r) (SHOs r)
and d1: decide (rho r p) ≠ Some v
and d2: decide (rho (Suc r) p) = Some v
shows card {q. sendMsg Ute-M r q p (rho r q) = Vote (Some v)} > E − α
proof –
from run obtain µp
where nxt:nextState Ute-M r p (rho r p) µp (rho (Suc r) p)
and ∀ qq. qq ∈ HOs r p ⟷ µp qq ≠ None
and ∀ qq. qq ∈ SHOs r p ∩ HOs r p
  ⟷ µp qq = Some (sendMsg Ute-M r qq p (rho r qq))
unfolding Ute-SHOMachine-def SHORun-eq SHOnextConfig-eq SHOmsgVectors-def by blast

hence \( \{ qq, \mu p qq = Some (Vote (Some v)) \} \subseteq \{ qq, \text{sendMsg} Ute-M r qq p (\rho r qq) = Vote (Some v) \} \)
(is \( \text{?vrcvd}p - \text{?ahop} \subseteq \text{?vsent}p \)) by auto

hence card (\( \text{?vrcvd}p - \text{?ahop} \)) \leq card \( \text{?vsent}p \)
and card (\( \text{?vrcvd}p - \text{?ahop} \)) \geq card \( \text{?vrcvd}p - \text{?ahop} \)
by (auto simp: card-mono diff-card-le-card-Diff)

hence card \( \text{?vsent}p \) \geq card \( \text{?vrcvd}p - \text{?ahop} \)
by auto

moreover
from usafe have card (HOs r p - SHOs r p) \leq \alpha
  by (auto simp: Ute-SHOMachine-def Ute-commPerRd-def)
moreover
from run d1 d2 have step r \neq 0 by (rule decide-step)
with nat have next1 r p (\rho r p) \mu p (\rho (Suc r) p)
  by (auto simp: Ute-SHOMachine-def nextState-def Ute-nextState-def)
with run d1 d2 have card \( \{ qq, \mu p qq = Some (Vote (Some v)) \} > E \)
  unfolding next1-def by auto
ultimately
show \( \text{?thesis} \) using alpha-lt-E by auto
qed

8.5 Proof of Agreement and Validity

If more than \( E - \alpha \) messages holding \( v \) are sent to some process \( p \) at round \( r \), then every process \( pp \) correctly receives more than \( \alpha \) such messages.

\textbf{lemma common-x-argument-1:}
assumes usafe:SHOcommPerRd Ute-M (HOs Suc r) (SHOs Suc r)
and threshold: card \( \{ q, \text{sendMsg} Ute-M (Suc r) q p (\rho (Suc r) q) \) = Vote (Some v) \} > E - \alpha
(is \( \text{card (?msgs p v)} > - \))
shows card (\( \text{?msgs} v \cap (SHOs Suc r) pp \cap HOs Suc r) pp)) \geq \alpha
proof –
  have card (\( \text{?msgs} pp v \) + card (SHOs Suc r) pp \cap HOs Suc r) pp) > N + \alpha
  proof –
  have \( \forall v. \text{sendMsg} Ute-M (Suc r) q p (\rho (Suc r) q) \)
  = sendMsg Ute-M (Suc r) q p (\rho (Suc r) q)
  by (auto simp: Ute-SHOMachine-def Ute-sendMsg-def step-def send0-def send1-def)
moreover
from usafe
have card (SHOs Suc r) pp \cap HOs Suc r) pp) > N + 2*\alpha - E - 1
  by (auto simp: Ute-SHOMachine-def step-def Ute-commPerRd-def)
ultimately
show \( \text{?thesis} \) using threshold by auto
qed

moreover
have card (\( \text{?msgs} pp v \) + card (SHOs Suc r) pp \cap HOs Suc r) pp)
\( = \) card (\( \text{?msgs} pp v \cup (SHOs Suc r) pp \cap HOs Suc r) pp))

105
If more than $E - \alpha$ messages holding $v$ are sent to $p$ at some round $r$, then any process $pp$ will set its $x$ to value $v$ in $r$.

**lemma** common-x-argument-2:

**assumes** run: SHORun Ute-M rho HOs SHOs

and unsafe: $\forall r. \text{SHOcommPerRd} Ute-M (\text{HOs} r) (\text{SHOs} r)$

and nextpp: nextState Ute-M (Suc $r$) $pp$ (rho (Suc $r$) $pp$)

$\mu pp (\text{rho} (\text{Suc} (\text{Suc} r)) pp)$

and mupp: $\mu pp \in \text{SHOmsgVectors} Ute-M (\text{Suc} r) pp (\text{rho} (\text{Suc} r))$

(\text{HOs} (\text{Suc} r) pp) (\text{SHOs} (\text{Suc} r) pp)

and threshold: $\text{card} \{ q. \text{sendMsg} Ute-M (\text{Suc} r) q p (\text{rho} (\text{Suc} r) q) = \text{Vote} (\text{Some} v) \} > E - \alpha$

**shows** $x (\text{rho} (\text{Suc} (\text{Suc} r)) pp) = v$

**proof**

have stp: $\text{step} (\text{Suc} r) \neq 0$

**proof**

assume sr: $\text{step} (\text{Suc} r) = 0$

hence $\forall q. \text{sendMsg} Ute-M (\text{Suc} r) q p (\text{rho} (\text{Suc} r) q)$

$= \text{Val} (x (\text{rho} (\text{Suc} r) q))$

by (auto simp: Ute-SHOMachine-def Ute-sendMsg-def send0-def)

moreover

from threshold obtain qq where

sendMsg Ute-M (Suc $r$) qq p (rho (Suc $r$) qq) = Vote (Some v)

by force

ultimately

show False by simp

qed

have va: $\text{card} \{ qq. \mu pp qq = \text{Some} (\text{Vote} (\text{Some} v)) \} > \alpha$

(is card $(?msgs v) > \alpha$)

**proof**

from mupp

have SHOs (Suc $r$) $pp \cap$ HOs (Suc $r$) $pp$

$\subseteq \{ qq. \mu pp qq = \text{Some} (\text{sendMsg} Ute-M (\text{Suc} r) qq pp (\text{rho} (\text{Suc} r) qq)) \}$

unfolding SHOmsgVectors-def by auto

moreover

hence $(?msgs v) \supseteq (\text{?sent pp v} \cap (\text{SHOs} (\text{Suc} r) pp \cap \text{HOs} (\text{Suc} r) pp))$

by auto

hence $\text{card} (?msgs v)$

$\geq \text{card} ((\text{?sent pp v} \cap (\text{SHOs} (\text{Suc} r) pp \cap \text{HOs} (\text{Suc} r) pp))}$
by (auto intro: card-mono)
moreover
from usafe threshold
have alph: card ((?sent pp v) ∩ (SHOs (Suc r) pp ∩ HOs (Suc r) pp)) > α
  by (blast dest: common-x-argument-1)
ultimately
show ?thesis by auto
qed
moreover
from nxtpp stp
have next1 (Suc r) pp (rho (Suc r) pp) µpp (rho (Suc (Suc r)) pp)
  by (auto simp: Ute-SHOMachine-def nextState-def Ute-nextState-def)
ultimately
obtain w where wa: card (?msgs w) > α and xw: x (rho (Suc (Suc r)) pp) = w
  unfolding next1-def by auto
have v = w
proof –
  note usafe
  moreover
  obtain qv where qv ∈ SHOs (Suc r) pp and µpp qv = Some (Vote (Some v))
  proof –
  have ¬ (?msgs v ⊆ HOs (Suc r) pp – SHOs (Suc r) pp)
    proof
      assume ?msgs v ⊆ HOs (Suc r) pp – SHOs (Suc r) pp
      hence card (?msgs v) ≤ card ((HOs (Suc r) pp) – (SHOs (Suc r) pp))
        by (auto simp: card-mono)
      moreover
      from usafe
      have card (HOs (Suc r) pp – SHOs (Suc r) pp) ≤ α
        by (auto simp: Ute-SHOMachine-def Ute-commPerRd-def)
      moreover
      note va
      ultimately
      show False by auto
    qed
  then obtain qv
    where qv ∉ HOs (Suc r) pp – SHOs (Suc r) pp
    and qv: µpp qv = Some (Vote (Some v))
    by auto
  with mupp have qv ∈ SHOs (Suc r) pp
    unfolding SHOmsgVectors-def by auto
  with qv that show ?thesis by auto
    qed
  with stp mupp have vote (rho (Suc r) qv) = Some v
    by (auto simp: Ute-SHOMachine-def SHOmsgVectors-def
        Ute-sendMsg-def send1-def)
  moreover
  obtain qw where
\(qw \in \text{SHOs} \ (\text{Suc} \ r) \ pp\) and \(\mu_{pp} \ qw = \text{Some} \ (\text{Vote} \ (\text{Some} \ w))\)

**proof** –
- **have** \(- (\text{msgs} \ w \subseteq \text{HOs} \ (\text{Suc} \ r) \ pp - \text{SHOs} \ (\text{Suc} \ r) \ pp)\)
  **proof**
  - **assume** \(\text{msgs} \ w \subseteq \text{HOs} \ (\text{Suc} \ r) \ pp - \text{SHOs} \ (\text{Suc} \ r) \ pp\)
  - **hence** \(\text{card} \ (\text{msgs} \ w) \leq \text{card} \ ((\text{HOs} \ (\text{Suc} \ r) \ pp) - (\text{SHOs} \ (\text{Suc} \ r) \ pp))\)
    by (auto simp: card-mono)
  **moreover**
  - **have** \(\text{card} \ (\text{HOs} \ (\text{Suc} \ r) \ pp - \text{SHOs} \ (\text{Suc} \ r) \ pp) \leq \alpha\)
    by (auto simp: Ute-SHOMachine-def Ute-commPerRd-def)
  **moreover**
  - **note** \(\text{wa}\)
  - **ultimately**
    **show** \(\text{False}\) by auto
  **qed**

then obtain \(qw\)
- **where** \(qw \notin \text{HOs} \ (\text{Suc} \ r) \ pp - \text{SHOs} \ (\text{Suc} \ r) \ pp\)
  **and** \(qw: \mu_{pp} \ qw = \text{Some} \ (\text{Vote} \ (\text{Some} \ w))\)
  by auto
- **with mupp** **have** \(qw \in \text{SHOs} \ (\text{Suc} \ r) \ pp\)
  **by** unfolding SHOmsgVectors-def by auto
  **with** \(qw\) **that show** \(?thesis\) by auto
  **qed**

with \(stp \ mupp\) **have** \(\text{vote} \ ((\text{rho} \ (\text{Suc} \ r)) \ qw) = \text{Some} \ w\)
  by (auto simp: Ute-SHOMachine-def SHOmsgVectors-def Ute-sendMsg-def send1-def)
  **moreover**
  **from run** **obtain** \(\mu_{qv} \ \mu_{qw}\)
  **where** \(\text{nextState} \ Ute-M \ r \ \qv \ ((\text{rho} \ r) \ \qv) \ \mu_{qv} \ (\text{rho} \ (\text{Suc} \ r) \ \qv)\)
  **and** \(\mu_{qv} \in \text{SHOmsgVectors} \ Ute-M \ r \ \qv \ (\text{rho} \ r) \ (\text{HOs} \ r \ \qv) \ (\text{SHOs} \ r \ \qv)\)
  **and** \(\text{nextState} \ Ute-M \ r \ \qw \ ((\text{rho} \ r) \ \qw) \ \mu_{qw} \ (\text{rho} \ (\text{Suc} \ r) \ \qw)\)
  **and** \(\mu_{qw} \in \text{SHOmsgVectors} \ Ute-M \ r \ \qw \ (\text{rho} \ r) \ (\text{HOs} \ r \ \qw) \ (\text{SHOs} \ r \ \qw)\)
  **by** (auto simp: Ute-SHOMachine-def SHORun-eq SHOnextConfig-eq) blast
  **ultimately**
  **show** \(?thesis\) **using** usafe **by** (auto dest: common-vote)
  **qed**

with \(xw\) **show** \(x \ (\text{rho} \ (\text{Suc} \ r)) \ pp) = v\) **by** auto
  **qed**

Inductive argument for the agreement and validity theorems.

**lemma** safety-inductive-argument:
- **assumes** \(\text{run: SHORun} \ Ute-M \ \text{rho} \ \text{HOs} \ \text{SHOs}\)
  **and** \(\text{comm: } \forall \ r. \ \text{SHOcommPerRd} \ Ute-M \ (\text{HOs} \ r) \ (\text{SHOs} \ r)\)
  **and** \(\text{sh: } E - \alpha < \text{card} \ \{q. \ \text{sendMsg} \ Ute-M \ r' q p \ (\text{rho} \ r' q) = \text{Vote} \ (\text{Some} \ v)\}\)
  **and** \(\text{stp1: } \text{step} \ r' = \text{Suc} \ 0\)
  **shows** \(E - \alpha < \text{card} \ \{q. \ \text{sendMsg} \ Ute-M \ (\text{Suc} \ r') \ q p \ (\text{rho} \ (\text{Suc} \ r') \ q) = \text{Vote} \ (\text{Some} \ v)\}\)

108
proof

\begin{proof}
\begin{enumerate}
\item from \texttt{stp1} have \(r' > 0\) by (auto simp: \texttt{step-def})
\item with \texttt{stp1} obtain \(r\) where \(r^\nu \cdot r' = \text{Suc} \ r\) and \texttt{stpr:step (Suc \ r) = Suc \ 0} by (auto dest: \texttt{gr0-implies-Suc})
\end{enumerate}
\end{proof}

\begin{proof}
\begin{enumerate}
\item have \(\forall \ pp. \ x \ (\text{rho} \ (\text{Suc} \ (\text{Suc} \ r)) \ pp) = v\)
\item proof
\item fix \(pp\)
\item from \texttt{run} obtain \(\mu pp\)
\item where \(\mu pp \in \text{SHOmsgVectors \ Ute-M} \ r' \ pp \ (\text{rho} \ r') \ (\text{HOs} \ r' \ pp) \ (\text{SHOs} \ r' \ pp)
\item and \(\text{nextState} \ \text{Ute-M} \ r' \ pp \ (\text{rho} \ r' \ pp) \ \mu pp \ (\text{rho} \ (\text{Suc} \ r') \ pp)
\item by (auto simp: \texttt{Ute-SHOMachine-def \ SHORun-eq \ SHOnextConfig-eq})
\item with \texttt{run comm ih rr' show} \(x \ (\text{rho} \ (\text{Suc} \ r)) \ pp) = v\)
\item by (auto dest: \texttt{common-x-argument-2})
\end{enumerate}
\end{proof}

qed

\begin{proof}
\begin{enumerate}
\item with \texttt{run stpr}
\item have \(\forall \ pp \ p. \ \text{sendMsg \ Ute-M} \ (\text{Suc} \ (\text{Suc} \ r)) \ pp \ p \ (\text{rho} \ (\text{Suc} \ (\text{Suc} \ r)) \ pp) = \text{Val} \ v\)
\item by (auto simp: \texttt{Ute-SHOMachine-def \ SHORun-eq \ SHOnextConfig-eq \\
\texttt{Ute-sendMsg-def \ send0-def \ mod-Suc \ step-def})
\item with \(rr'
\item have \(\forall \ pp. \ \mu pp. \ \mu pp' \in \text{SHOmsgVectors \ Ute-M} \ (\text{Suc} \ r') \ p \ (\text{rho} \ (\text{Suc} \ r'))
\item \(\text{HOs} \ (\text{Suc} \ r') \ p \ (\text{SHOs} \ (\text{Suc} \ r') \ p) \subseteq \{ q. \ \mu pp' \ q = \text{Some} \ (\text{Val} \ v)\}\)
\item by (auto simp: \texttt{SHOmsgVectors-def})
\item hence \(\forall \ pp. \ \mu pp. \ \mu pp' \in \text{SHOmsgVectors \ Ute-M} \ (\text{Suc} \ r') \ p \ (\text{rho} \ (\text{Suc} \ r'))
\item \(\text{HOs} \ (\text{Suc} \ r') \ p \ (\text{SHOs} \ (\text{Suc} \ r') \ p) \leq \text{card} \ \{ q. \ \mu pp' \ q = \text{Some} \ (\text{Val} \ v)\}\)
\item by (auto simp: \texttt{card-mono})
\end{enumerate}
\end{proof}

moreover

\begin{proof}
\begin{enumerate}
\item from \texttt{comm} have \(\forall \ p. \ T < \text{card} \ (\text{SHOs} \ (\text{Suc} \ r') \ p \ \cap \ \text{HOs} \ (\text{Suc} \ r') \ p)\)
\item by (auto simp: \texttt{Ute-SHOMachine-def \ Ute-commPerRd-def})
\end{enumerate}
\end{proof}

ultimately

\begin{proof}
\begin{enumerate}
\item have \(\forall \ p. \ \mu pp. \ \mu pp' \in \text{SHOmsgVectors \ Ute-M} \ (\text{Suc} \ r') \ p \ (\text{rho} \ (\text{Suc} \ r'))
\item \(\text{HOs} \ (\text{Suc} \ r') \ p \ (\text{SHOs} \ (\text{Suc} \ r') \ p) \leq \text{card} \ \{ q. \ \mu pp' \ q = \text{Some} \ (\text{Val} \ v)\}\)
\item by (auto dest: \texttt{less-le-trans})
\end{enumerate}
\end{proof}

show \(\text{thesis}\)

\begin{proof}
\begin{enumerate}
\item have \(\forall \ pp. \ \text{vote} \ ((\text{rho} \ (\text{Suc} \ (\text{Suc} \ r')) \ pp) = \text{Some} \ v\)
\item proof
\item fix \(pp\)
\item from \texttt{run} obtain \(\mu pp\)
\item where \texttt{nnxpp: \nextState \ Ute-M} \ (\text{Suc} \ r') \ pp \ (\text{rho} \ (\text{Suc} \ r') \ pp) \ \mu pp \ (\text{rho} \ (\text{Suc} \ (\text{Suc} \ r')) \ pp)
\item and \texttt{mapp: \mu pp \in \text{SHOmsgVectors \ Ute-M} \ (\text{Suc} \ r') \ pp \ (\text{rho} \ (\text{Suc} \ r'))
\end{enumerate}
\end{proof}
A process that holds some decision \( v \) has decided \( v \) sometime in the past.

**Lemma decisionNonNullThenDecided:**

**Assumes**
- \( \text{run}: \text{SHORun} \ Ute-M \ rho \ HOs \ SHOs \) and
- \( \text{dec}: \text{decide} (\rho n p) = \text{Some} v \)

**Shows**
- \( \exists m < n. \ \text{decide} (\rho (Suc m) p) \neq \text{decide} (\rho m p) \)
- \( \land \ \text{decide} (\rho (Suc m) p) = \text{Some} v \)

**Proof**
- Let \( \text{dec} k = \text{decide} ((\rho k) p) \)
- \( \forall m < n. \ \text{dec} (Suc m) \neq \text{dec} (Suc m) \neq \text{Some} v \)
- \( \text{dec} n \neq \text{Some} v \)
- \( \text{is } ?P n \text{ is } ?A n \rightarrow \) -

**Proof (induct n)**
- From \( \text{run} \) show \( ?P 0 \)
- By (auto simp: Ute-SHOMachine-def SHORun-eq HOSnextConfig-eq initState-def Ute-initState-def)

**Next**
- Fix \( n \)
- Assume \( ih: ?P n \) thus \( ?P (Suc n) \) by force

**qed**
- With \( dec \) show \( ?thesis \) by auto

**qed**

110
If process \( p_1 \) has decided value \( v_1 \) and process \( p_2 \) later decides, then \( p_2 \) must decide \( v_1 \).

**lemma laterProcessDecidesSameValue:**

assumes run: SHORun \( Ute-M \) rho \( HOs \) \( SHOs \)

and comm:\( \forall r. \) SHOcommPerRd \( Ute-M \) \( (HOs \ r) \) \( (SHOs \ r) \)

and \( dv1:\text{decide} (\text{rho} (Suc \ r) \ p1) = \text{Some} \ v1 \)

and \( dn2:\text{decide}(\text{rho} (r + k) \ p2) \neq \text{Some} \ v2 \)

and \( dv2:\text{decide} (\text{rho} (Suc (r + k)) \ p2) = \text{Some} \ v2 \)

shows \( v_2 = v_1 \)

**proof**

- from run \( dv1 \) obtain \( r1 \)
  - where \( r1r: r1 < \text{Suc} \ r \)
  - and \( dn1:\text{decide} (\text{rho} (r1 \ p1) \neq \text{Some} \ v1 \)
  - and \( dv1':\text{decide} (\text{rho} (Suc (r1) \ p1) = \text{Some} \ v1 \)
  - by (auto dest: decisionNonNullThenDecided)

- from \( r1r \) obtain \( s \) where \( r1s: \text{Suc} \ r = \text{Suc} (r1 + s) \)
  - by (auto dest: less-imp-Suc-add)

  then obtain \( k' \) where \( kk'r + k = r1 + k' \)
  - by auto

  with \( dn2 \) \( dv2 \)

  have \( dv2':\text{decide} (\text{rho} (r1 + k') p2) \neq \text{Some} v2 \)
  - and \( dv2':\text{decide} (\text{rho} (Suc (r1 + k')) p2) = \text{Some} v2 \)
  - by auto

- from \( run \) \( dn1 \) \( dv1' \) \( dn2' \) \( dv2' \)

  have \( rs0: \text{step} \ r1 = \text{Suc} 0 \) and \( rks0: \text{step} (r1 + k') = \text{Suc} 0 \)
  - by (auto simp: mod-Suc step-def dest: decide-step)

  have \( \text{step} (r1 + k') = \text{step} (\text{step} r1 + k') \)
  - unfolding step-def by (simp add: mod-add-left-eq)

  with \( rs0 \) \( rks0 \) have \( \text{step} k' = 0 \) by (auto simp: step-def mod-Suc)

  then obtain \( k'' \) where \( k'' = k''*\text{nSteps} \)
  - by (auto simp: step-def)

  with \( dn2' \) \( dv2' \)

  have \( dn2'':\text{decide} (\text{rho} (r1 + k''*\text{nSteps}) p2) \neq \text{Some} v2 \)
  - and \( dv2'':\text{decide} (\text{rho} (Suc (r1 + k''*\text{nSteps})) p2) = \text{Some} v2 \)
  - by auto

- from \( rs0 \) have \( stp:\text{step} (r1 + k''*\text{nSteps}) = \text{Suc} 0 \)
  - unfolding step-def by auto

  have \( inv: \text{card}\{q, \text{sendMsg} Ute-M (r1 + k''*\text{nSteps}) q \ p1 (\text{rho} (r1 + k''*\text{nSteps}) q) = \text{Vote} (\text{Some} v1)\} > E - \alpha \)

  **proof** (induct \( k'' \))

  - from \( stp \) have \( \text{step} (r1 + 0*\text{nSteps}) = \text{Suc} 0 \)
    - by (auto simp: step-def)

  - from \( run \) \( comm \) \( dn1 \) \( dv1' \)

  show \( \text{card}\{q, \text{sendMsg} Ute-M (r1 + 0*\text{nSteps}) q \ p1 (\text{rho} (r1 + 0*\text{nSteps}) q) = \text{Vote} (\text{Some} v1)\} > E - \alpha \)
= Vote (Some v1)} > E − α

by (intro decide-with-threshold-E) auto

next

fix k''

assume ih: E − α < 

card {q. sendMsg Ute-M (r1 + k'' * nSteps) q p1 (rho (r1 + k'' * nSteps)) q} = Vote (Some v1)}

from rs0 have steps: step (r1 + k'' * nSteps) = Suc 0

by (auto simp: step-def)

with run comm ih

have E − α < 

card {q. sendMsg Ute-M (Suc (Suc (r1 + k'' * nSteps))) q p1 (rho (Suc (Suc (r1 + k'' * nSteps))) q) = Vote (Some v1)}

by (rule safety-inductive-argument)

thus E − α < 

card {q. sendMsg Ute-M (r1 + Suc k'' * nSteps) q p1 (rho (r1 + Suc k'' * nSteps) q) = Vote (Some v1)}

by auto

qed

moreover

from run have ∀ q. sendMsg Ute-M (r1 + k'' * nSteps) q p1 (rho (r1 + k'' * nSteps)) q) = sendMsg Ute-M (r1 + k'' * nSteps) q p2 (rho (r1 + k'' * nSteps) q)

by (auto simp: Ute-SHOMachine-def Ute-sendMsg-def step-def send0-def send1-def)

moreover

from run comm dn2'' dv2''

have E − α < 

card {q. sendMsg Ute-M (r1 + k'' * nSteps) q p2 (rho (r1 + k'' * nSteps) q) = Vote (Some v2)}

by (auto dest: decide-with-threshold-E)

ultimately

show v2 = v1 by (auto dest: unique-majority-E-α)

qed

The Agreement property is an immediate consequence of the two preceding lemmas.

**Theorem ute-agreement:**

assumes run: SHORun Ute-M rho HOs SHOs

and comm: ∀ r. SHOcommPerRd Ute-M (HOs r) (SHOs r)

and p: decide (rho m p) = Some v

and q: decide (rho n q) = Some w

shows v = w

proof –

from run p obtain k

where k1: decide (rho (Suc k) p) ≠ decide (rho k p)
and \( k_2 \): decide \((\rho (\text{Suc } k) p) = \text{Some } v\) by (auto dest: decisionNonNullThenDecided)

from \( \text{run } q \) obtain \( l \)
where \( l_1 \): decide \((\rho (\text{Suc } l) q) \neq \text{decide } (\rho l q)\)
and \( l_2 \): decide \((\rho (\text{Suc } l) q) = \text{Some } w\) by (auto dest: decisionNonNullThenDecided)

show \(?\text{thesis}\)
proof (cases \( k \leq l \))
  case True
  then obtain \( m \) where \( m : \text{decide } (\rho (\text{Suc } l) q) \neq = \text{decide } (\rho l q)\)
  from \( \text{run comm } k_2 l_1 l_2 m \) have \( w = v \) by (auto elim!: laterProcessDecidesSameValue)
  thus \(?\text{thesis}\) by simp
next
  case False
  hence \( l \leq k \) by simp
  then obtain \( m \) where \( m : \text{decide } (\rho (\text{Suc } l) q) = \text{Some } w\)
  from \( \text{run comm } k_1 k_2 m \) show \(?\text{thesis}\)
  by (auto elim!: laterProcessDecidesSameValue)
qed

qed

Main lemma for the proof of the Validity property.

lemma \( \text{validity-argument}:\)
assumes \( \text{run: } \text{SHORun } \text{Ute-M } \rho \text{ HOs}\text{SHOs }\)
and \( \text{comm: } \forall r. \text{SHOcommPerRd } \text{Ute-M } (\text{HOs } r) (\text{SHOs } r)\)
and \( \text{init: } \forall p. x (\text{((} \rho 0 p) = v\text{))}\)
and \( \text{dw: } \text{decide } (\rho r p) = \text{Some } w\)
and \( \text{stp: step } r' = \text{Suc } 0\)
shows \( \text{card } \{ q. \text{sendMsg } \text{Ute-M } r' q p (\rho r' q) = \text{Vote } (\text{Some } v)\} > E - \alpha\)
proof –
  define \( k \) where \( k = r' \div \text{nSteps}\)
  with \( \text{stp} \) have \( \text{stp: } r' = \text{Suc } 0 + k \ast \text{nSteps}\)
  using \( \text{div-mult-mod-eq } [\text{of } r' \text{nSteps}]\)
  by (simp add: step-def)
  moreover
  have \( E - \alpha < \)
  card \( \{ q. \text{sendMsg } \text{Ute-M } (\text{Suc } 0 + k\ast\text{nSteps}) q p ((\rho (\text{Suc } 0 + k\ast\text{nSteps})) q)\) = \text{Vote } (\text{Some } v)\}
  proof (induct \( k \))
  have \( \forall pp. \text{vote } (\text{((} \rho (\text{Suc } 0)) pp) = \text{Some } v\)
  proof
  fix \( pp \)
  from \( \text{run obtain } \mu pp\)
  where \( \text{nextState } \text{Ute-M } 0 pp (\rho 0 pp) \mu pp (\rho (\text{Suc } 0) pp)\)
  and \( \text{msgVectors } \mu pp \in \text{SHOmsgVectors } \text{Ute-M } 0 pp (\rho 0) (\text{HOs } 0 pp) (\text{SHOs } 0 pp)\)
  by (auto simp: \text{Ute-SHOMachine-def } \text{SHORun-eq } \text{SHOnextConfig-eq})

113
have majv: card \{ q. \mu pp q = \text{Some} (\text{Val} v)\} > T

proof –
from run init have \forall q. \text{sendMsg} Ute-M 0 q pp (\rho 0 q) = \text{Val} v
  by (auto simp: Ute-SHOMachine-def SHORun-eq SHOnextConfig-eq
       Ute-sendMsg-def send0-def step-def)
moreover
from comm have shoT: card (SHOs 0 pp \cap HOs 0 pp) > T
  by (auto simp: Ute-SHOMachine-def Ute-commPerRd-def)
moreover
from mupp have SHOs 0 pp \cap HOs 0 pp \subseteq \{ q. \mu pp q = \text{Some} (\text{sendMsg} Ute-M 0 q pp (\rho 0 q))\}
  by (auto simp: SHOmsgVectors-def)
hence card (SHOs 0 pp \cap HOs 0 pp) \leq card \{ q. \mu pp q = \text{Some} (\text{sendMsg} Ute-M 0 q pp (\rho 0 q))\}
  by (auto simp: card-mono)
ultimately
show ?thesis by (auto simp: less-le-trans)
qed
moreover
from nxtpp have next0 0 pp ((\rho 0) pp) \mu pp (\rho (Suc 0) pp)
  by (auto simp: Ute-SHOMachine-def nextState-def Ute-nextState-def step-def)
ultimately
obtain w where majw: card \{ q. \mu pp q = \text{Some} (\text{Val} w)\} > T
  and votew: vote (\rho (Suc 0) pp) = \text{Some} w
  by (auto simp: next0-def)
from majv majw have v = w by (auto dest: unique-majority-T)
with votew show vote (\rho (Suc 0) pp) = \text{Some} v by simp
qed
with run
have card \{ q. \text{sendMsg} Ute-M (Suc 0) q p (\rho (Suc 0) q) = \text{Vote} (\text{Some} v)\} = N
  by (auto simp: Ute-SHOMachine-def SHORun-eq SHOnextConfig-eq
       Ute-sendMsg-def send0-def send1-def)
thus E - \alpha <
card \{ q. \text{sendMsg} Ute-M (Suc 0 + 0 * nSteps) q p (\rho (Suc 0 + 0 * nSteps) q)
  = \text{Vote} (\text{Some} v)\}
  using majE EltN by auto
next
fix k
assume ih: E - \alpha <
card \{ q. \text{sendMsg} Ute-M (Suc 0 + k * nSteps) q p (\rho (Suc 0 + k * nSteps) q)
  = \text{Vote} (\text{Some} v)\}
have step (Suc 0 + k * nSteps) = Suc 0
  by (auto simp: mod-Suc step-def)
from run comm ih this
have $E - \alpha <$
\[
\text{card } \{q.\text{sendMsg } Ute-M (\text{Suc (Suc } 0 + k * \text{nSteps})) q p
\]
\[
(rho (\text{Suc (Suc } 0 + k * \text{nSteps})) q)
\]
\[
= \text{Vote (Some } v)\}
\]
by (rule safety-inductive-argument)
thus $E - \alpha <$
\[
\text{card } \{q.\text{sendMsg } Ute-M (\text{Suc } 0 + \text{Suc } k * \text{nSteps}) q p
\]
\[
(rho (\text{Suc } 0 + \text{Suc } k * \text{nSteps}) q)
\]
\[
= \text{Vote (Some } v)\}
\]
by simp
qed
ultimately
show ?thesis by simp
qed

The following theorem shows the Validity property of algorithm $U_{T,E,\alpha}$.

**theorem** ute-validity:
assumes run: SHORun Ute-M rho HOs SHOs
and comm: \forall r. SHOcommPerRd Ute-M (HOs r) (SHOs r)
and init: \forall p. x (rho 0 p) = v
and dw: decide (rho r p) = Some w
shows v = w

**proof**–
from run dw obtain r1
where dnr1: decide ((rho r1) p) \neq Some w
and dwr1: decide ((rho \text{Suc r1}) p) = Some w
by (force dest: decisionNonNullThenDecided)
with run have step r1 \neq 0 by (rule decide-step)
hence step r1 = \text{Suc 0} by (simp add: step-def mod-Suc)
with assms
have $E - \alpha <$
\[
\text{card } \{q.\text{sendMsg } Ute-M r1 q p (rho r1 q) = \text{Vote (Some } v)\}
\]
by (rule validity-argument)
moreover
from run comm dnr1 dwr1
have card $\{q.\text{sendMsg } Ute-M r1 q p (rho r1 q) = \text{Vote (Some } w)\} > E - \alpha$
by (auto dest: decide-with-threshold-E)
ultimately
show v = w by (auto dest: unique-majority-E-\alpha)
qed

**8.6 Proof of Termination**

At the second round of a phase that satisfies the conditions expressed in the global communication predicate, processes update their $x$ variable with the value $v$ they receive in more than $\alpha$ messages.

**lemma** set-x-from-vote:
assumes run: SHORun Ute-M rho HOs SHOs
and comm: SHOcommPerRd Ute-M (HOs r) (SHOs r)
and \(stp\): step \((Suc\ r)\) = \(Suc\ 0\)
and \(\pi\): \(\forall p.\ HOs\ (Suc\ r)\ p = SHOs\ (Suc\ r)\ p\)
and \(\text{nxt}\): nextState \(Ute-M\ (Suc\ r)\ p\ (\rho (Suc\ r)\ p)\ \mu\ (\rho (Suc\ (Suc\ r))\ p)\)
and \(\text{mu}\): \(\mu\in SHOmsgVectors\ Ute-M\ (Suc\ r)\ p\ (\rho (Suc\ r))\)
and \(wp\): \(\alpha < \text{card}\ \{qq.\ \mu qq = Some\ (Vote\ (Some\ v))\}\)
shows \(x (\rho (Suc\ (Suc\ r)))\)\)

proof –
from \(nxt\ \text{stp} \ \text{wp} \ \text{obtain} \ \text{wp}\ where\ \text{xwp} \ \alpha < \text{card} \ \{qq.\ \mu qq = Some\ (Vote\ (Some\ wp))\}\)
and \(\text{xp} : x (\rho (Suc\ (Suc\ r)))\)\) = \(wp\)
by \((auto\ simp: Ute-SHOMachine-def nextState-def Ute-nextState-def next1-def)\)

have \(wp = v\)
proof –
from \(xwp\ \text{obtain} \ pp\ where\ \text{smw} \ \mu pp = Some\ (Vote\ (Some\ wp))\)
by force
have \(vote\ (\rho (Suc\ r)\ pp) = Some\ wp\)
proof –
from \(smw\ \text{mu}\ \pi\)
have \(\mu pp = Some\ (sendMsg\ Ute-M\ (Suc\ r)\ pp\ p\ (\rho\ (Suc\ r)\ pp))\)
unfolding \(SHOmsgVectors-def\) by force
with \(\text{stp}\) have \(\mu pp = Some\ (Vote\ (\text{vote}(\rho\ (Suc\ r)\ pp)))\)
by \((auto\ simp: Ute-SHOMachine-def Ute-sendMsg-def send1-def)\)
with \(smw\) show \(\text{thesis}\) by \(auto\)
qed
moreover
from \(wp\ \text{obtain} \ qq\ where\ \text{smv} \ \mu qq = Some\ (Vote\ (Some\ v))\)
by force
have \(vote\ (\rho (Suc\ r)\ qq) = Some\ v\)
proof –
from \(smv\ \text{mu}\ \pi\)
have \(\mu qq = Some\ (sendMsg\ Ute-M\ (Suc\ r)\ qq\ p\ (\rho\ (Suc\ r)\ qq))\)
unfolding \(SHOmsgVectors-def\) by force
with \(\text{stp}\) have \(\mu qq = Some\ (Vote\ (\text{vote}(\rho\ (Suc\ r)\ qq)))\)
by \((auto\ simp: Ute-SHOMachine-def Ute-sendMsg-def send1-def)\)
with \(smw\) show \(\text{thesis}\) by \(auto\)
qed
moreover
from \(\text{run}\ \text{obtain} \ \mu pp\ \mu qq\ where\ nextState\ Ute-M\ r\ pp\ (\rho\ r\ pp)\ \mu pp\ (\rho\ (Suc\ r)\ pp)\)
and \(\mu pp\in SHOmsgVectors\ Ute-M\ r\ pp\ (\rho\ r)\ (HOs\ r\ pp)\ (SHOs\ r\ pp)\)
and \(\text{nextState}\ Ute-M\ r\ qq\ ((\rho\ r)\ qq)\ \mu qq\ (\rho\ (Suc\ r)\ qq)\)
and \(\mu qq\in SHOmsgVectors\ Ute-M\ r\ qq\ (\rho\ r)\ (HOs\ r\ qq)\ (SHOs\ r\ qq)\)
unfolding \(Ute-SHOMachine-def\ \text{SHORun-eq SHOnextConfig-eq}\) by \text{blast}
ultimately
show \(\text{thesis}\) using \(\text{comm}\) by \((auto\ dest: common-vote)\)
qed
with \(xp\) show \(\text{thesis}\) by \(simp\)

116
Assume that HO and SHO sets are uniform at the second step of some phase. Then at the subsequent round there exists some value $v$ such that any received message which is not corrupted holds $v$.

**lemma** termination-argument-1:

**assumes** run: SHORun Ute-M rho HOs SHOs

**and** comm: SHOcommPerRd Ute-M (HOs r) (SHOs r)

**and** stp: step (Suc r) = Suc 0

**and** $\pi$: $\forall$ p. $\pi 0 = HOs (Suc r)$ p $\land \pi 0 = SHOs (Suc r)$ p

obtains $v$ where

$\forall p \mu p' q.
\begin{array}{l}
[ q \in SHOs (Suc (Suc r)) p \cap HOs (Suc (Suc r)) p;
\mu p' \in SHOmsgVectors Ute-M (Suc (Suc r)) p (rho (Suc (Suc r)))

(HOs (Suc (Suc r)) p) (SHOs (Suc (Suc r)) p)
\end{array}
\implies \mu p' q = (Some (Val v))$

**proof**

from $\pi$ have hosho: $\forall$ p. SHOs (Suc r) p = SHOs (Suc r) p $\cap$ HOs (Suc r) p

by simp

have $\forall p q. x (rho (Suc (Suc r)) p) = x (rho (Suc (Suc r)) q)$

**proof**

fix $p q$

from run obtain $\mu p$

where nxt: nextState Ute-M (Suc r) p (rho (Suc r) p)

$\mu p$ (rho (Suc (Suc r)) p)

and mu: $\mu p \in SHOmsgVectors Ute-M (Suc r) p (rho (Suc r))$

(HOs (Suc r) p) (SHOs (Suc r) p)

by (auto simp: Ute-SHOMachine-def SHORun-eq SHOnextConfig-eq)

from run obtain $\mu q$

where nxtq: nextState Ute-M (Suc r) q (rho (Suc r) q)

$\mu q$ (rho (Suc (Suc r)) q)

and muq: $\mu q \in SHOmsgVectors Ute-M (Suc r) q (rho (Suc r))$

(HOs (Suc r) q) (SHOs (Suc r) q)

by (auto simp: Ute-SHOMachine-def SHORun-eq SHOnextConfig-eq)

have $\forall qq. \mu p qq = \mu q qq$

**proof**

fix $qq$

show $\mu p qq = \mu q qq$

**proof**

(case cases $\mu p qq = None$)

case False

with mu $\pi$ have 1: $qq \in SHOs (Suc r)$ p and 2: $qq \in SHOs (Suc r)$ q

unfolding SHOmsgVectors-def by auto

from mu $\pi$ 1

have $\mu p qq = Some (sendMsg Ute-M (Suc r) qq p (rho (Suc r) qq))$

unfolding SHOmsgVectors-def by auto

moreover
from \( \mu q \pi 2 \)

have \( \mu q q = \text{Some (sendMsg Ute-M (Suc r) q q (rho (Suc r) q q))} \)

unfolding \( \text{SHOmsgVectors-def} \) by auto

ultimately

show \(?\text{thesis}\) by (auto simp: \( \text{Ute-SHOMachine-def Ute-sendMsg-def step-def send0-def send1-def} \))

next
case True

with \( \mu \) have \( qq \notin \text{HOs (Suc r) p} \) unfolding \( \text{SHOmsgVectors-def} \) by auto

with \( \pi \mu q \) have \( \mu q q = \text{None} \) unfolding \( \text{SHOmsgVectors-def} \) by auto

with True show \(?\text{thesis}\) by simp

qed

qed

hence \( \text{vsets:} \& \forall q. \{ qq. \mu p qq = \text{Some (Vote (Some v))} \} \)

\( = \{ qq. \mu q qq = \text{Some (Vote (Some v))} \} \)

by auto


show \( x (\rho (\text{Suc (Suc r)}) p) = x (\rho (\text{Suc (Suc r)}) q) \)

proof (cases \( \exists v. \alpha < \text{card} \{ qq. \mu p qq = \text{Some (Vote (Some v))} \} \), clarify)

fix \( v \)

assume vp: \( \alpha < \text{card} \{ qq. \mu p qq = \text{Some (Vote (Some v))} \} \)

with run comm stp \( \pi \text{nxt} \mu \) have \( x (\rho (\text{Suc (Suc r)}) p) = v \)

by (auto dest: set-x-from-vote)

moreover

from \( \text{vsets vp} \)

have \( \alpha < \text{card} \{ qq. \mu q qq = \text{Some (Vote (Some v))} \} \) by auto

with run comm stp \( \pi \text{nxt}q \muq \) have \( x (\rho (\text{Suc (Suc r)}) q) = v \)

by (auto dest: set-x-from-vote)

ultimately

show \( x (\rho (\text{Suc (Suc r)}) p) = x (\rho (\text{Suc (Suc r)}) q) \)

by auto

next

assume nov: \( \neg (\exists v. \alpha < \text{card} \{ qq. \mu q qq = \text{Some (Vote (Some v))} \}) \)

with \( \text{nxt}q \text{stp} \) have \( x (\rho (\text{Suc (Suc r)}) q) = \text{undefined} \)

by (auto simp: \( \text{Ute-SHOMachine-def nextState-def Ute-nextState-def next1-def} \))

moreover

from \( \text{vsets nov} \)

have \( \neg (\exists v. \alpha < \text{card} \{ qq. \mu q qq = \text{Some (Vote (Some v))} \}) \) by auto

with \( \text{nxt}q \text{stp} \) have \( x (\rho (\text{Suc (Suc r)}) q) = \text{undefined} \)

by (auto simp: \( \text{Ute-SHOMachine-def nextState-def Ute-nextState-def next1-def} \))

ultimately

show \(?\text{thesis}\) by simp

qed

then obtain \( v \) where \( \& q. x (\rho (\text{Suc (Suc r)}) q) = v \) by blast
moreover

from \( \text{stp} \) have \( \text{step} \ (Suc \ (Suc \ r)) = 0 \)
by (auto simp: step-def mod-Suc)

hence \( \land p \mu p' q \).

[ \( q \in \text{SHOs} \ (Suc \ (Suc \ r)) \) \( \cap \text{HOs} \ (Suc \ (Suc \ r)) \) \( p \);
\( \mu p' \in \text{SHOmsgVectors Ute-M} \ (Suc \ (Suc \ r)) \) \( p \) \( \rho (Suc \ (Suc \ r)) \))
\( (\text{HOs} \ (Suc \ (Suc \ r)) \) \( p \) \( (\text{SHOs} \ (Suc \ (Suc \ r)) \) \( p \)) \)

\( ] \implies \mu p' q = \text{Some} \ (Val \ (x \ (\rho \ (Suc \ (Suc \ r)) \ q))) \)
by (auto simp: Ute-SHOMachine-def SHOmsgVectors-def Ute-sendMsg-def send0-def)

ultimately

have \( \land p \mu p' q \).

[ \( q \in \text{SHOs} \ (Suc \ (Suc \ r)) \) \( \cap \text{HOs} \ (Suc \ (Suc \ r)) \) \( p \);
\( \mu p' \in \text{SHOmsgVectors Ute-M} \ (Suc \ (Suc \ r)) \) \( p \) \( \rho (Suc \ (Suc \ r)) \))
\( (\text{HOs} \ (Suc \ (Suc \ r)) \) \( p \) \( (\text{SHOs} \ (Suc \ (Suc \ r)) \) \( p \)) \)

\( ] \implies \mu p' q = (\text{Some} \ (Val \ v)) \)
by auto

with that show thesis by blast

qed

If a process \( p \) votes \( v \) at some round \( r \), then all messages received by \( p \) in \( r \)
that are not corrupted hold \( v \).

lemma termination-argument-2:

assumes \( \text{map:} \mu p \in \text{SHOmsgVectors Ute-M} \ (Suc \ r) \) \( p \) \( \rho (Suc \ r) \)
\( (\text{HOs} \ (Suc \ r)) \) \( p \) \( (\text{SHOs} \ (Suc \ r)) \) \( p \)

and \( \text{nxtq: nextState Ute-M r q} \) \( (\rho \ r \ q) \) \( \mu q \ rho (Suc \ r) \ q \)

and \( \text{vq: vote (\rho \ (Suc \ r) \ q) = Some v} \)

and \( \text{qsho: q} \in \text{SHOs} \ (Suc \ r) \) \( p \) \( \cap \text{HOs} \ (Suc \ r) \) \( p \)

shows \( \mu p q = (\text{Some} \ (\text{Vote} \ (\text{Some} \ v))) \)
proof –

from \( \text{nxtq vq} \) have \( \text{step} \ r = 0 \) by (auto simp: vote-step)

with \( \text{map qsho have} \mu p q = \text{Some} \ (\text{Vote} \ (\text{vote} \ (\rho \ (Suc \ r) \ q))) \)

by (auto simp: Ute-SHOMachine-def SHOmsgVectors-def Ute-sendMsg-def send0-def mod-Suc)

with \( \text{vq} \) show \( \mu p q = \text{Some} \ (\text{Vote} \ (\text{Some} \ v)) \) by auto

qed

We now prove the Termination property.

theorem ute-termination:

assumes \( \text{run: SHORun Ute-M rho HOs SHOs} \)
and \( \text{commR: \forall r. SHOcommPerRd Ute-M (HOs r) (SHOs r)} \)
and \( \text{commG: SHOcommGlobal Ute-M HOs SHOs} \)

shows \( \exists r v. \ \text{decide} \ (\rho \ r \ p) = \text{Some v} \)

proof –

from \( \text{commG} \)

go \( \Phi \ pi r0 \)
where \( \text{rr: r0 = Suc (nSteps * \Phi)} \)

and \( \pi: \forall p. \pi = \text{HOs r0 p} \land \pi = \text{SHOs r0 p} \)

and \( t: \forall p. \text{card} \ (\text{SHOs} \ (Suc \ r0) \ p) \cap \text{HOs} \ (Suc \ r0) \ p > T \)

119
and e: ∀ p. card (SHOs (Suc (Suc r0))) p ∩ HOs (Suc (Suc r0)) p > E
by (auto simp: Ute-SHOMachine-def Ute-commGlobal-def Let-def)
from rr have stp: step r0 = Suc 0 by (auto simp: step-def)

obtain w where votew: ∀ p. (vote (rho (Suc (Suc r0))) p) = Some w
proof –
have abc: ∀ p. ∃ w. vote (rho (Suc (Suc r0))) p = Some w
proof
fix p
from run stp obtain µp
where nxt: nextState Ute-M (Suc r0) p (rho (Suc r0)) p µp
(rho (Suc (Suc r0))) p
and map: µp ∈ SHOmsgVectors Ute-M (Suc r0) p (rho (Suc r0))
(HOs (Suc r0) p) (SHOs (Suc r0) p)
by (auto simp: Ute-SHOMachine-def SHORun-eq SHOnextConfig-eq)

have ∃ v. T < card { qq. µpqq = Some (Val v) }
proof –
from t have card (SHOs (Suc r0)) p ∩ HOs (Suc r0) p > T ..
moreover
from run commR stp π rr
obtain v where
\[ ∀ p. µp' q. ∃ q ∈ SHOs (Suc r0) p ∩ HOs (Suc r0) p; µp' ∈ SHOmsgVectors Ute-M (Suc r0) p (rho (Suc r0))
(HOs (Suc r0) p) (SHOs (Suc r0) p) \]
⇒ µp' q = Some (Val v)
using termination-argument-1 by blast

with map obtain v where
\[ ∀ qq. qq ∈ SHOs (Suc r0) p ∩ HOs (Suc r0) p \implies µp qq = Some (Val v) \]
by auto

hence SHOs (Suc r0) p ∩ HOs (Suc r0) p ⊆ { qq. µp qq = Some (Val v) } by auto
hence card (SHOs (Suc r0)) p ∩ HOs (Suc r0) p ≤ card { qq. µp qq = Some (Val v) } by (auto intro: card-mono)
ultimately
have T < card { qq. µp qq = Some (Val v) } by auto

thus ?thesis by auto
qed

with stp nxt show ∃ w. vote ((rho (Suc (Suc r0))) p) = Some w
by (auto simp: Ute-SHOMachine-def nextState-def Ute-nextState-def
step-def mod-Suc next0-def)

qed

then obtain qq w where qqw: vote (rho (Suc (Suc r0))) qq = Some w
by blast
have ∀ pp. vote (rho (Suc (Suc r0))) pp = Some w

120
proof
  fix pp
  from abc obtain wp where pwp:vote ((\rho (Suc (Suc r0))) pp) = Some wp
    by blast
  from run obtain \mu pp \mu qq
    where ntxp: nextState Ute-M (Suc r0) pp (\rho (Suc r0)) pp
        \mu pp (\rho (Suc (Suc r0))) pp
    and map: \mu pp \in SHOmsgVectors Ute-M (Suc r0) pp (\rho (Suc r0))
        (HOs (Suc r0) pp) (SHOs (Suc r0) pp)
    and ntxq: nextState Ute-M (Suc r0) qq (\rho (Suc r0)) qq
        \mu qq (\rho (Suc (Suc r0))) qq
    and map: \mu qq \in SHOmsgVectors Ute-M (Suc r0) qq (\rho (Suc r0))
        (HOs (Suc r0) qq) (SHOs (Suc r0) qq)
  unfolding Ute-SHOMachine-def SHORun-eq SHOnextConfig-eq by blast
  from commR this pwp qqw have wp = w
    by (auto dest: common-vote)
  with pwp show vote ((\rho (Suc (Suc r0))) pp) = Some w
    by auto
  qed
  with that show \thesis by auto
  qed

from run obtain \mu p'
  where ntxp: nextState Ute-M (Suc r0) (Suc (Suc r0)) p (\rho (Suc (Suc r0))) p
        \mu p' (\rho (Suc (Suc (Suc r0)))) p
    and map': \mu p' \in SHOmsgVectors Ute-M (Suc r0) (Suc (Suc r0)) p (\rho (Suc (Suc r0)))
        (HOs (Suc r0) p) (SHOs (Suc r0) p)
    by (auto simp: Ute-SHOMachine-def SHORun-eq SHOnextConfig-eq)
  have \mu pp' qq = Some (Vote (Some w))
    by (auto
      simp: Ute-SHOMachine-def SHORun-eq SHOnextConfig-eq)
  proof
    fix qq
    assume qqsho: qq \in SHOseq (Suc (Suc r0)) p \intersection HOseq (Suc (Suc r0)) p
    from run obtain \mu qq where
      ntxqq: nextState Ute-M (Suc r0) qq (\rho (Suc r0)) qq
        \mu qq (\rho (Suc (Suc r0))) qq
      by (auto simp: Ute-SHOMachine-def SHORun-eq SHOnextConfig-eq)
    from commR map' ntxqq votew qqsho show \mu p' qq = Some (Vote (Some w))
      by (auto dest: termination-argument-2)
  qed

hence SHOseq (Suc (Suc r0)) p \intersection HOseq (Suc (Suc r0)) p
    \subseteq {qq. \mu pp' qq = Some (Vote (Some w))}
  by auto

hence vsho: card (SHOseq (Suc (Suc r0)) p \intersection HOseq (Suc (Suc r0)) p)
    \subseteq card {qq. \mu pp' qq = Some (Vote (Some w))}
  by (auto simp: card-mono)

from stp have step (Suc r0) = Suc 0
unfolding step-def by auto
with nxtp have next1 \((\text{Suc} (\text{Suc} r0))\) \(\rho\) \((\text{Suc} (\text{Suc} r0))\) \(\mu\)

by \((\text{auto simp: Ute-SHOMachine-def nextState-def Ute-nextState-def})\)
moreover
from \(e\) have \(E < \text{card} (\text{SHOs} (\text{Suc} (\text{Suc} r0)) \cap \text{HOs} (\text{Suc} (\text{Suc} r0))\) \(\rho\)
by auto
with \(w\) have \(\text{majv:card} \{qq. \mu\}
\(\text{rho} (\text{Suc} (\text{Suc} (\text{Suc} r0))))\)
by auto
ultimately
show \(?\text{thesis}\) by \((\text{auto simp: next1-def})\)
qed

8.7 \(U_{T,E,\alpha}\) Solves Weak Consensus

Summing up, all (coarse-grained) runs of \(U_{T,E,\alpha}\) for HO and SHO collections that satisfy the communication predicate satisfy the Weak Consensus property.

\textbf{theorem} ute-weak-consensus:
\textbf{assumes} run: SHORun Ute-M \(\rho\) HOs SHOs
\text{and} commR: \(\forall r. \text{SHOcommPerRd Ute-M} (\text{HOs} r) (\text{SHOs} r)\)
\text{and} commG: \(\text{SHOcommGlobal Ute-M} \text{HOs} \text{SHOs}\)
\textbf{shows} weak-consensus \((x \circ (\rho \text{ 0}))\) decide \(\rho\)
\textbf{unfolding} weak-consensus-def
\textbf{using} ute-validity[OF run commR]
\text{ute-agreement}[OF run commR]
\text{ute-termination}[OF run commR commG]
\textbf{by} auto

By the reduction theorem, the correctness of the algorithm carries over to the fine-grained model of runs.

\textbf{theorem} ute-weak-consensus-fg:
\textbf{assumes} run: fg-run Ute-M \(\rho\) HOs SHOs \((\lambda q. \text{undefined})\)
\text{and} commR: \(\forall r. \text{SHOcommPerRd Ute-M} (\text{HOs} r) (\text{SHOs} r)\)
\text{and} commG: \(\text{SHOcommGlobal Ute-M} \text{HOs} \text{SHOs}\)
\textbf{shows} weak-consensus \((\lambda p.x \circ (\rho \text{ 0}))\) decide \((\text{state} \circ \rho)\)
\(\text{is weak-consensus ?inits -}\)
\textbf{proof} \((\text{rule local-property-reduction[OF run weak-consensus-is-local!]})\)
\text{fix crun}
\textbf{assume} crun: CSHORun Ute-M crun HOs SHOs \((\lambda r.q. \text{undefined})\)
\text{and} init: crun 0 = state \((\rho \text{ 0})\)
from crun have SHORun Ute-M crun HOs SHOs \textbf{by} \((\text{unfold SHORun-def})\)
from this commR commG
\textbf{have} weak-consensus \((x \circ (\text{crun} 0))\) decide crun
\textbf{by} \((\text{rule ute-weak-consensus})\)
\textbf{with} init show weak-consensus ?inits decide crun
\textbf{by} \((\text{simp add: o-def})\)
qed
9 Verification of the $A_{T,E,\alpha}$ Consensus algorithm

Algorithm $A_{T,E,\alpha}$ is presented in [3]. Like $U_{T,E,\alpha}$, it is an uncoordinated algorithm that tolerates value faults, and it is parameterized by values $T$, $E$, and $\alpha$ that serve a similar function as in $U_{T,E,\alpha}$, allowing the algorithm to be adapted to the characteristics of different systems. $A_{T,E,\alpha}$ can be understood as a variant of OneThirdRule tolerating Byzantine faults.

We formalize in Isabelle the correctness proof of the algorithm that appears in [3], using the framework of theory $HOModel$.

9.1 Model of the Algorithm

We begin by introducing an anonymous type of processes of finite cardinality that will instantiate the type variable $'proc$ of the generic HO model.

```
typedcl Proc — the set of processes
axiomatization where Proc-finite: OFCLASS(Proc, finite-class)
instance Proc :: finite by (rule Proc-finite)
```

abbreviation

$N \equiv \text{card} (UNIV::Proc set)$ — number of processes

The following record models the local state of a process.

```
record $'val$ pstate =
  $x$ :: $'val$ — current value held by process
  decide :: $'val$ option — value the process has decided on, if any
```

The $x$ field of the initial state is unconstrained, but no decision has yet been taken.

```
definition Ate-initState where
  Ate-initState p st \equiv (decide st = None)
```

The following locale introduces the parameters used for the $A_{T,E,\alpha}$ algorithm and their constraints [3].

```
locale ate-parameters =
  fixes $\alpha$::nat and $T$::nat and $E$::nat
  assumes \(TNaE:T \geq 2*(N + 2*\alpha - E)\)
  and \(TltN:T < N\)
```
and \( \text{EltN} : E < N \)

begin

The following are consequences of the assumptions on the parameters.

**lemma** \( \text{majE} : 2 \cdot (E - \alpha) \geq N \)

**using** \( \text{TNaE TltN by auto} \)

**lemma** \( \text{Eglt} : E > \alpha \)

**using** \( \text{majE EltN by auto} \)

**lemma** \( \text{Tge2a} : T \geq 2 \cdot \alpha \)

**using** \( \text{TNaE EltN by auto} \)

At every round, each process sends its current \( x \). If it received more than \( T \) messages, it selects the smallest value and store it in \( x \). As in algorithm \( \text{OneThirdRule} \), we therefore require values to be linearly ordered.

If more than \( E \) messages holding the same value are received, the process decides that value.

**definition** \( \text{mostOftenRcvd} \) where

\[
\text{mostOftenRcvd} (\text{msgs} : \text{Proc} \Rightarrow \text{'val option}) \equiv \{ v. \forall w. \text{card} \{ qq. \text{msgs} qq = \text{Some} w \} \leq \text{card} \{ qq. \text{msgs} qq = \text{Some} v \} \}
\]

**definition** \( \text{Ate-sendMsg} :: \text{nat} \Rightarrow \text{Proc} \Rightarrow \text{Proc} \Rightarrow \text{'val pstate} \Rightarrow \text{'val} \)

**where**

\( \text{Ate-sendMsg } r \ p \ q \ st \ \equiv \ x \ st \)

**definition** \( \text{Ate-nextState} :: \text{nat} \Rightarrow \text{Proc} \Rightarrow (\text{'val:linorder}) \text{pstate} \Rightarrow (\text{Proc} \Rightarrow \text{'val option}) \Rightarrow \text{'val pstate} \Rightarrow \text{bool} \)

**where**

\( \text{Ate-nextState } r \ p \ s \ t \ m s g s \ s t' \ \equiv \)

\[
\text{if card} \{ q. \text{msgs} q \neq \text{None} \} > T \]

\[
\quad \text{then } x \ s t' = \text{Min} (\text{mostOftenRcvd} \text{msgs})
\]

\[
\quad \text{else } x \ s t' = x \ s t
\]

\[
\land ( (\exists v. \text{card} \{ q. \text{msgs} q = \text{Some} v \} > E \land \text{decide} s t' = \text{Some} v)
\]

\[
\lor \neg (\exists v. \text{card} \{ q. \text{msgs} q = \text{Some} v \} > E)
\]

\[
\land \text{decide} s t' = \text{decide} s t)
\]

**9.2 Communication Predicate for \( A_{T,E,\alpha} \)**

Following [3], we now define the communication predicate for the \( A_{T,E,\alpha} \) algorithm. The round-by-round predicate requires that no process may receive more than \( \alpha \) corrupted messages at any round.

**definition** \( \text{Ate-commPerRd} \) where

\( \text{Ate-commPerRd } H O r s \ S H O r s \ \equiv \)
∀ p. \text{card} (\text{HOrs} p - \text{SHOrs} p) \leq \alpha

The global communication predicate stipulates the three following conditions:

- for every process \( p \) there are infinitely many rounds where \( p \) receives more than \( T \) messages,
- for every process \( p \) there are infinitely many rounds where \( p \) receives more than \( E \) uncorrupted messages,
- and there are infinitely many rounds in which more than \( E - \alpha \) processes receive uncorrupted messages from the same set of processes, which contains more than \( T \) processes.

definition \( \text{Ate-commGlobal} \) where
\[
\text{Ate-commGlobal} \text{HOrs SHOrs} \equiv
(\forall r \, p. \exists r' > r. \text{card} (\text{HOrs} r' p) > T)
\land (\forall r \, p. \exists r' > r. \text{card} (\text{SHOrs} r' p \cap \text{HOs} r' p) > E)
\land (\forall r \, p. \exists r' > r. \exists \pi_1 \pi_2.
\text{card} \pi_1 > E - \alpha
\land \text{card} \pi_2 > T
\land (\forall p \in \pi_1. \text{HOs} r' p = \pi_2 \land \text{SHOs} r' p \cap \text{HOs} r' p = \pi_2))
\]

9.3 The \( \mathcal{A}_{T,E,\alpha} \) Heard-Of Machine

We now define the non-coordinated SHO machine for the \( \mathcal{A}_{T,E,\alpha} \) algorithm by assembling the algorithm definition and its communication-predicate.

definition \( \text{Ate-SHOMachine} \) where
\[
\text{Ate-SHOMachine} = (\lambda \pi p st crd. \text{Ate-initState} p (\text{st::('val::linorder pstate)'}))
\land \text{sendMsg} = \text{Ate-sendMsg},
\text{CnextState} = (\lambda r p st msgs crd st'. \text{Ate-nextState} r p st msgs st'),
\text{SHOcommPerRd} = (\text{Ate-commPerRd:: Proc HO \Rightarrow Proc HO \Rightarrow bool}),
\text{SHOcommGlobal} = \text{Ate-commGlobal}
\]

abbreviation \( \text{Ate-M} \equiv (\text{Ate-SHOMachine::(Proc, 'val::linorder pstate, 'val) SHOMachine})
\]

end — locale ate-parameters

end
theory \( \text{AteProof} \)
imports \( \text{AteDefs ..} \)/Reduction
begin

context ate-parameters
begin

end
9.4 Preliminary Lemmas

If a process newly decides value \( v \) at some round, then it received more than \( E - \alpha \) messages holding \( v \) at this round.

**Lemma** decide-sent-msgs-threshold:

**Assumes** run: SHORun Ate-M rho HOs SHOs

**And** comm: SHOcommPerRd Ate-M (HOs r) (SHOs r)

**And** wp: decide (rho r p) \( \neq \) Some v

**Shows** card \( \{ qq. sendMsg Ate-M r qq p (rho r qq) = v \} > E - \alpha \)

**Proof**

- From run obtain \( \mu_p \)
  - Where \( \mu_p : \mu_p \in SHOmsgVectors Ate-M r p (rho r) (HOs r p) (SHOs r p) \)
  - And nxt: nextState Ate-M r p (rho r p) \( \mu_p \) (rho (Suc r) p)
    - By (auto simp: SHORun-eq SHOnextConfig-eq)
  - From \( \mu_p \) have \( \{ qq. \mu_p qq = Some v \} \subseteq \{ qq. sendMsg Ate-M r qq p (rho r qq) = v \} \)
    - By (auto simp: SHOmsgVectors-def)
  - Hence card \( ?vrcvd - \?ahop \leq \?vsentp \)
    - And card \( ?vrcvd - \?ahop \geq \text{card} \ ?vrcvd - \text{card} \ ?ahop \)
      - By (auto simp: card-mono diff-card-le-card-Diff)
  - Hence card \( \?vsentp \geq \text{card} \ ?vrcvd - \text{card} \ ?ahop \) by auto
- Moreover
  - From nxt wp wp have card \( \?vrcvd > E \)
    - By (auto simp: Ate-SHOMachine-def nextState-def Ate-nextState-def)
  - Moreover
    - From \( \text{comm} \) have card \( (\text{HOs} r p - \text{SHOs} r p) \leq \alpha \)
      - By (auto simp: Ate-SHOMachine-def Ate-commPerRd-def)
- Ultimately
  - Show \( \?thesis \) using Egta by auto

**QED**

If more than \( E - \alpha \) processes sends a value \( v \) to some process \( q \) at some round, then \( q \) will receive at least \( N + 2*\alpha - E \) messages holding \( v \) at this round.

**Lemma** other-values-received:

**Assumes** comm: SHOcommPerRd Ate-M (HOs r) (SHOs r)

**And** nxt: nextState Ate-M r q (rho r q) \( \mu_q \) ((rho (Suc r)) q)

**And** muq: \( \mu_q \in SHOmsgVectors Ate-M r q (rho r) (HOs r q) (SHOs r q) \)

**And** sent: card \( \{ qq. sendMsg Ate-M r qq q (rho r qq) = v \} > E - \alpha \)
  - (is card \?sent > -)
**Shows** card \( \{ qq. \mu_q qq \neq Some v \} \cap \text{HOs} r q \leq N + 2*\alpha - E \)

**Proof**

- From nxt muq
  - Have \( \{ qq. \mu_q qq \neq Some v \} \cap \text{HOs} r q \) \( - (\text{HOs} r q - \text{SHOs} r q) \)
    - \( \subseteq \{ qq. sendMsg Ate-M r qq q (rho r qq) \neq v \} \)
      - (is \text{not}vrcvd - \text{?ahop} \subseteq \text{?notsent}
unfolding $\text{SHOmsgVectors-def}$ by auto

hence $\text{card \ ?notvsent} \geq \text{card (\ ?notvrcvd \ - \ ?aho)}$

and $\text{card (\ ?notvrcvd \ - \ ?aho)} \geq \text{card \ ?notvrcvd \ - \ card \ ?aho}$

by (auto simp: card-mono diff-card-le-card-Diff)

moreover

from comm have $\text{card \ ?aho} \leq \alpha$

by (auto simp: Ate-SHOMachine-def Ate-commPerRd-def)

moreover

have $1: \text{card \ ?notvsent} + \text{card \ ?vsent} = \text{card (\ ?notvrcvd \ \cup \ ?vsent)}$

by (subst card-Un-Int) auto

have $\text{card (\ ?notvrcvd \ \cup \ ?vsent)} = N$

by simp

ultimately

show $\text{?thesis}$ using EltN Egta by auto

qed

If more than $E - \alpha$ processes send a value $v$ to some process $q$ at some round $r$, and if $q$ receives more than $T$ messages in $r$, then $v$ is the most frequently received value by $q$ in $r$.

**lemma** mostOftenRcvd-v:

assumes comm: $\text{SHOcommPerRd \ Ate-M \ (HOs \ r) \ (SHOs \ r)}$

and nxt: $\text{nextState \ Ate-M \ r \ q \ (rho \ r \ q) \ \mu (\rho \ \text{(Suc \ r)}) \ q}$

and muq: $\mu \in \text{SHOmsgVectors \ Ate-M \ r \ q \ (rho \ r \ q) \ (SHOs \ r \ q)}$

and threshold-T: $\text{card \ \{qq. \ \mu q \ qq \neq \text{None}\} > T}$

and threshold-E: $\text{card \ \{qq. \ sendMsg \ Ate-M \ r \ qq \ q \ (rho \ r \ qq) = v\} > E - \alpha}$

shows $\text{mostOftenRcvd} \ \mu q = \{v\}$

proof –

from muq have hodef:$\text{HOs \ r \ q} = \{\text{qq.} \ \mu q \ \text{qq} \neq \text{None}\}$

unfolding $\text{SHOmsgVectors-def}$ by auto

from comm nxt muq threshold-E

have $\text{card (\ \{qq. \ \mu q \ qq \neq \text{Some \ v}\} \ \cap \ \text{HOs \ r \ q}) \leq N + 2*\alpha - E}$

(is $\text{card \ ?heardnotv} \leq -$)

by (rule other-values-received)

moreover

have $\text{card \ ?heardnotv} \leq T + 1 - \text{card \ \{qq. \ \mu q \ qq = \text{Some \ v}\}$

proof –

from muq

have $\text{?heardnotv} = (\text{HOs \ r \ q}) - \{qq. \ \mu q \ qq = \text{Some \ v}\}$

and $\{qq. \ \mu q \ qq = \text{Some \ v}\} \subseteq \text{HOs \ r \ q}$

unfolding $\text{SHOmsgVectors-def}$ by auto

hence $\text{card \ ?heardnotv} = \text{card (\ HOs \ r \ q)} - \text{card \ \{qq. \ \mu q \ qq = \text{Some \ v}\}$

by (auto simp: card-Diff-subset)

with hodef threshold-T show $\text{?thesis}$ by auto

qed

ultimately

have $\text{card \ \{qq. \ \mu q \ qq = \text{Some \ v}\} > card \ ?heardnotv}$

using TNaE by auto

127
moreover

\{ 
  \text{fix } w \\
  \text{assume } w: w \neq v \\
  \text{with } hodef \text{ have } \{ qq, \mu qq = \text{Some } w \} \subseteq \text{?heardnotv by auto} \\
  \text{hence } \text{card } \{ qq, \mu qq = \text{Some } w \} \leq \text{card } \text{?heardnotv by (auto simp: card-mono)} \\
\}

ultimately

\text{have } \{ w, \text{card } \{ qq, \mu qq = \text{Some } w \} \geq \text{card } \{ qq, \mu qq = \text{Some } v \} \} = \{ v \} \\
\text{by force} \\
\text{thus } \text{?thesis unfolding mostOftenRcvd-def by auto} \\
\text{qed}

If at some round more than $E - \alpha$ processes have their $x$ variable set to $v$, then this is also true at next round.

\text{lemma common-x-induct:} \\
\text{assumes ran: } \text{SHORun } Ate-M rho HOs SHOs \\
\text{and comm: } \text{SHOcommPerRd } Ate-M (HOs (r+k)) (SHOs (r+k)) \\
\text{and } ih: \text{card } \{ qq, x (rho (r+k) qq) = v \} > E - \alpha \\
\text{shows } \text{card } \{ qq, x (rho (r + Suc k) qq) = v \} > E - \alpha \\
\text{proof –} \\
\text{from } ih \\
\text{have } \text{thrE:y pp. card } \{ qq, \text{sendMsg } Ate-M (r+k) qq pp (rho (r+k) qq) = v \} > E - \alpha \\
\text{by (auto simp: Ate-SHOMachine-def Ate-sendMsg-def)}

\{ 
  \text{fix } qq \\
  \text{assume } kv:x (rho (r+k) qq) = v \\
  \text{from } ran \text{ obtain } \mu qq \\
  \text{where } \text{nxt: } \text{nextState } Ate-M (r+k) qq (rho (r+k) qq) \mu qq ((rho (Suc (r + k))) qq) \\
  \text{and } \mu qq: \mu qq \in \text{SHOmsgVectors } Ate-M (r+k) qq (rho (r+k)) (HOs (r+k) qq) (SHOs (r+k) qq) \\
  \text{by (auto simp: SHORun-eq SHOnextConfig-eq)}
\}

\text{have } x (rho (r + Suc k) qq) = v \\
\text{proof (cases card } \{ pp, \mu qq pp \neq None \} > T) \\
\text{case True} \\
\text{with } \text{comm nxt muq thrE have } \text{mostOftenRcvd } \mu qq = \{ v \} \\
\text{by (auto dest: mostOftenRcvd-v)} \\
\text{with } \text{nxt True show } x (rho (r + Suc k) qq) = v \\
\text{by (auto simp: Ate-SHOMachine-def nextState-def Ate-nextState-def)} \\
\text{next} \\
\text{case False} \\
\text{with } \text{nxt have } x (rho (r + Suc k) qq) = x (rho (r+k) qq) \\
\text{by (auto simp: Ate-SHOMachine-def nextState-def Ate-nextState-def)} \\
\text{with } \text{kv show } x (rho (r + Suc k) qq) = v \text{ by simp} \\
\text{qed}
Whenever some process newly decides value \( v \), then any process that updates its \( x \) variable will set it to \( v \).

**Lemma common-x:**

**Assumes** run: SHORun \( Ate-M \) rho HOs SHOs

**And** comm: \( \forall \ r. \ SHOcommPerRd \ (Ate-M::(Proc, 'val::linorder pstate, 'val) SHOMachine) \)

\[ (Hos \ r) \ (SHOs \ r) \]

**And** d1: decide \((rho \ r \ p) \neq \text{Some} \ v\)

**And** d2: decide \((rho \ (Suc \ r) \ p) = \text{Some} \ v\)

**Shows** \( x \ (rho \ (r + Suc \ k) \ q) \neq x \ (rho \ (r + k) \ q) \)

**Proof:**

**From** comm

**Have** SHOcommPerRd \( (Ate-M::(Proc, 'val::linorder pstate, 'val) SHOMachine) \)

\[ (Hos \ (r+k)) \ (SHOs \ (r+k)) \ .. \]

**Moreover**

**From** run obtain \( \mu q \)

**Where** nxt: nextState \( Ate-M \ (r+k) \ q \ (rho \ (r+k) \ q) \ \mu q \ (rho \ (r + Suc \ k) \ q) \)

**And** muq: \( \mu q \in \text{SHOmsgVectors} \ Ate-M \ (r+k) \ q \ (rho \ (r+k)) \)

\[ (Hos \ (r+k)) \ (SHOs \ (r+k) \ q) \]

**By** \( \text{auto simp: SHORun-eq SHOnextConfig-eq} \)

**Moreover**

**From** nxt qupdate

**Have** threshold-T: \( \text{card} \ {qq. \ \mu q \ qq \neq \text{None}} \ > T \)

**And** xsmall: \( x \ (rho \ (r + Suc \ k) \ q) = \text{Min} \ (\text{mostOftenRcvd} \ \mu q) \)

**By** \( \text{auto simp: Ate-SHOMachine-def nextState-def Ate-nextState-def} \)

**Moreover**

**Have** \( E - \alpha < \text{card} \ {qq. \ x \ (rho \ (r + k) \ qq) = v} \)

**Proof:** (induct \( k \))

**From** run comm d1 d2

**Have** \( E - \alpha < \text{card} \ {qq. \ sendMsg \ Ate-M \ r \ qq \ (rho \ r \ qq) = v} \)

**By** \( \text{auto dest: decide-sent-msgs-threshold} \)

**Thus** \( E - \alpha < \text{card} \ {qq. \ x \ (rho \ (r + 0) \ qq) = v} \)

**By** \( \text{auto simp: Ate-SHOMachine-def Ate-sendMsg-def} \)

**Next**

**Fix** \( k \)

**Assume** \( E - \alpha < \text{card} \ {qq. \ x \ (rho \ (r + k) \ qq) = v} \)

**With** run comm show \( E - \alpha < \text{card} \ {qq. \ x \ (rho \ (r + Suc \ k) \ qq) = v} \)

**By** \( \text{auto dest: common-x-induct} \)

\text{qed}
with run
have \( E - \alpha < \text{card} \{qq, sendMsg Ate-M (r+k) qq q (rho (r+k) qq) = v\} \)
by (auto simp: Ate-SHOMachine-def Ate-sendMsg-def SHORun-eq SHOnextConfig-eq)
ultimately
have mostOftenRcvd \( \mu q = \{v\} \) by (auto dest: mostOftenRcvd-v)
with xsmall show ?thesis by auto
qed

A process that holds some decision \( v \) has decided \( v \) sometime in the past.

lemma decisionNonNullThenDecided:
assumes run: SHORun Ate-M rho HOs SHOs
and dec: decide (rho n p) = Some v
obtains m where m < n
and decide (rho m p) \( \neq \) Some v
and decide (rho (Suc m) p) = Some v
proof
let \( ?\text{dec} k = \text{decide} (rho k p) \)
have \( (\forall m<n. ?\text{dec} (Suc m) \neq ?\text{dec} m \rightarrow ?\text{dec} (Suc m) \neq Some v) \rightarrow ?\text{dec} n \neq Some v \)
(is \( ?\text{P} n \) is \( ?\text{A} n \rightarrow -) \)
proof (induct n)
from run show \( ?\text{P} 0 \)
by (auto simp: Ate-SHOMachine-def SHORun-eq HOinitConfig-eq
initState-def Ate-initState-def)
next
fix n
assume ih: \( ?\text{P} n \) thus \( ?\text{P} (Suc n) \) by force
qed
with dec that show ?thesis by auto
qed

9.5 Proof of Validity

Validity asserts that if all processes were initialized with the same value,
then no other value may ever be decided.

theorem ate-validity:
assumes run: SHORun Ate-M rho HOs SHOs
and comm: \( \forall r. \text{SHOcommPerRd} Ate-M (HOs r) (SHOs r) \)
and initv: \( \forall q. x (rho 0 q) = v \)
and dp: decide (rho r p) = Some w
shows w = v
proof
\{ 
fix r
have \( \forall qq. sendMsg Ate-M r qq p (rho r qq) = v \)
proof (induct r)
from run initv show \( \forall qq. sendMsg Ate-M 0 qq p (rho 0 qq) = v \)
by (auto simp: SHORun-eq SHOnextConfig-eq Ate-SHOMachine-def Ate-sendMsg-def)
next
fix r
assume ih:∀ qq. sendMsg Ate-M r qq p (rho r qq) = v

have ∀ qq. x (rho (Suc r) qq) = v
proof
fix qq
from run obtain µqq
  where nxt: nextState Ate-M r qq (rho r qq) µqq (rho (Suc r) qq)
  and mu: µqq ∈ SHOmsgVectors Ate-M r qq (rho r qq) (SHOs r qq)
by (auto simp: SHORun-eq SHOnextConfig-eq)
from nxt
have (card \{ pp. µqq pp ≠ None \} > T ∧ x (rho (Suc r) qq) = Min (mostOftenRcvd µqq))
  ∨ (card \{ pp. µqq pp ≠ None \} ≤ T ∧ x (rho (Suc r) qq) = x (rho r qq))
by (auto simp: Ate-SHOMachine-def nextState-def Ate-nextState-def)
thus x (rho (Suc r) qq) = v
proof safe
assume x (rho (Suc r) qq) = x (rho r qq)
with ih show ?thesis
by (auto simp: Ate-SHOMachine-def Ate-sendMsg-def)

next
assume threshold-T: T < card \{ pp. µqq pp ≠ None \}
  and xsmall: x (rho (Suc r) qq) = Min (mostOftenRcvd µqq)

have card \{ pp. ∃ w. w ≠ v ∧ µqq pp = Some w \} ≤ T div 2
proof –
from comm have 1:card (HOs r qq − SHOs r qq) ≤ α
  by (auto simp: Ate-SHOMachine-def Ate-commPerRd-def)
moreover
from mu ih
have SHOs r qq ∩ HOs r qq ⊆ \{ pp. µqq pp = Some v \}
  and HOs r qq = \{ pp. µqq pp ≠ None \}
by (auto simp: SHOmsgVectors-def Ate-SHOMachine-def Ate-sendMsg-def)
hence \{ pp. µqq pp ≠ None \} − \{ pp. µqq pp = Some v \}
  ⊆ HOs r qq − SHOs r qq
by auto
hence card (\{ pp. µqq pp ≠ None \} − \{ pp. µqq pp = Some v \})
  ≤ card (HOs r qq − SHOs r qq)
by (auto simp: card-mono)
ultimately
have \{ pp. µqq pp ≠ None \} − \{ pp. µqq pp = Some v \} ≤ T div 2
  using Tge2a by auto
moreover
have \{ pp. µqq pp ≠ None \} − \{ pp. µqq pp = Some v \}
  = \{ pp. w. w ≠ v ∧ µqq pp = Some w \} by auto
ultimately
show ?thesis by simp
moreover
have \{pp, \mu qq pp \neq None\} = \{pp, \mu qq pp = Some v\} \cup \{pp, \exists w, w \neq v \land \mu qq pp = Some w\}
and \{pp, \mu qq pp = Some v\} \cap \{pp, \exists w, w \neq v \land \mu qq pp = Some w\} = \{
by auto
hence card \{pp, \mu qq pp \neq None\} = card \{pp, \mu qq pp = Some v\} + card \{pp, \exists w, w \neq v \land \mu qq pp = Some w\}

moreover
note threshold-T
ultimately
have card \{pp, \mu qq pp = Some v\} \geq card \{pp, \exists w, w \neq v \land \mu qq pp = Some w\}
by auto

moreover
\{fix w\}
assume w \neq v
hence \{pp, \mu qq pp = Some w\} \subseteq \{pp, \exists w, w \neq v \land \mu qq pp = Some w\}
by auto
hence card \{pp, \mu qq pp = Some w\} \leq card \{pp, \exists w, w \neq v \land \mu qq pp = Some w\}
by (auto simp: card-mono)

ultimately
have zz:\(\forall w, w \neq v \Rightarrow card \{pp, \mu qq pp = Some w\} < card \{pp, \mu qq pp = Some v\}\)
by force
hence \(\forall w, card \{pp, \mu qq pp = Some v\} \leq card \{pp, \mu qq pp = Some w\}\)
\(\Rightarrow w = v\)
by force
with zz have mostOftenRcvd \mu qq = \{v\}
by (force simp: mostOftenRcvd-def)
with xsml show x (rho (Suc r) qq) = v by auto
qed

qed

thus \(\forall qq, sendMsg Ate-M (Suc r) qq p (rho (Suc r) qq) = v\)
by (auto simp: Ate-SHOMachine-def Ate-sendMsg-def)

qed

}

note P = this

from run dp obtain rp
where rp: rp < r decide (rho rp p) \neq Some w
decide (rho (Suc rp) p) = Some w
by (rule decisionNonNullThenDecided)
from run obtain \( \mu p \)
where \( \text{nxt: nextState Ate-M \( \rho \) p (\( \rho \) Suc \( \rho \)) p} \)
and \( \mu; \mu \in \text{SHOmsgVectors Ate-M \( \rho \) p (\( \rho \) \( \rho \)) \( \text{SHOs} \) \( \text{rp} \) p) (\( \text{SHOs} \) \( \text{rp} \) p) \)
by (auto simp: SHORun-eq SHOnextConfig-eq)

\{
fix \( w \)
assume \( w: w \neq v \)
from \text{comm} have \( \text{card (\( \text{Hos} \) \( \rho \) \( p \) - \( \text{SHOs} \) \( \rho \) \( p \))} \leq \alpha \)
by (auto simp: Ate-SHOMachine-def Ate-commPerRd-def)
moreover
from \( \mu \) \( P \) have \( \text{SHOs} \) \( \rho \) \( p \rangle \cap \text{Hos} \) \( \rho \) \( p \rangle \subseteq \{pp. \mu p pp = \text{Some} v\} \)
and \( \text{Hos} \) \( \rho \) \( p \rangle = \{pp. \mu p pp \neq \text{None}\} \)
by (auto simp: SHOmsgVectors-def)
hence \( \{pp. \mu p pp \neq \text{None}\} - \{pp. \mu p pp = \text{Some} v\} \)
\( \subseteq \text{Hos} \) \( \rho \) \( p \rangle - \text{SHOs} \) \( \rho \) \( p \rangle \)
by auto
hence \( \text{card (\{pp. \mu p pp \neq \text{None}\} - \{pp. \mu p pp = \text{Some} v\})} \)
\( \leq \text{card (\( \text{Hos} \) \( \rho \) \( p \rangle - \text{SHOs} \) \( \rho \) \( p \rangle \))} \)
by (auto simp: card-mono)
ultimately
have \( \text{card (\{pp. \mu p pp \neq \text{None}\} - \{pp. \mu p pp = \text{Some} v\})} < E \)
using Egta by auto
moreover
from \( w \) have \( \{pp. \mu p pp = \text{Some} w\} \)
\( \subseteq \{pp. \mu p pp \neq \text{None}\} - \{pp. \mu p pp = \text{Some} v\} \)
by auto
hence \( \text{card (\{pp. \mu p pp = \text{Some} w\})} \)
\( \leq \text{card (\{pp. \mu p pp \neq \text{None}\} - \{pp. \mu p pp = \text{Some} v\})} \)
by (auto simp: card-mono)
ultimately
have \( \text{card (\{pp. \mu p pp = \text{Some} w\})} < E \) by simp
\}
hence \( \text{PP: } \forall w. \text{card (\{pp. \mu p pp = \text{Some} w\})} \geq E \implies w = v \) by force
from \( \text{rp \ nxt \ mu} \) have \( \text{card (q. \mu p q = \text{Some} w)} > E \)
by (auto simp: SHOmsgVectors-def Ate-SHOMachine-def
nextState-def Ate-nextState-def)
with \( \text{PP} \) show \( \mu \text{thesis} \) by auto
qed

9.6 Proof of Agreement
If two processes decide at the same round, they decide the same value.

lemma common-decision:
assumes run: \( \text{SHORun Ate-M} \rho \text{Hos} \text{SHOs} \)
and \text{comm:} \( \text{SHOcommPerRd Ate-M} (\text{Hos} \text{r}) (\text{SHOs} \text{r}) \)

133
and \(\text{nvp}: \text{decide}(\rho r p) \neq \text{Some } v\)
and \(\text{vp}: \text{decide}(\rho (\text{Suc } r) p) = \text{Some } v\)
and \(\text{nwp}: \text{decide}(\rho r q) \neq \text{Some } w\)
and \(\text{wp}: \text{decide}(\rho (\text{Suc } r) q) = \text{Some } w\)

\(\text{shows } w = v\)

\textbf{proof –}

\textbf{proof –}

\begin{itemize}
  \item have \(\text{gtn}: \text{card}\ \{ qq. \text{sendMsg} Ate-M r qq p (\rho r qq) = v\} + \text{card}\ \{ qq. \text{sendMsg} Ate-M r qq q (\rho r qq) = w\} > N\)
  \item \(\text{proof –}
  \begin{itemize}
    \item from \(\text{run comm nvp vp}\)
    \item have \(\text{card}\ \{ qq. \text{sendMsg} Ate-M r qq p (\rho r qq) = v\} > E - \alpha\)
      \item by (rule decide-sent-msgs-threshold)
    \item moreover
    \item from \(\text{run comm nwq wq}\)
    \item have \(\text{card}\ \{ qq. \text{sendMsg} Ate-M r qq q (\rho r qq) = w\} > E - \alpha\)
      \item by (rule decide-sent-msgs-threshold)
    \item ultimately
    \item show \(\text{thesis using majE by auto}\)
  \end{itemize}
  \end{itemize}
\end{itemize}

\textbf{qed}

\begin{itemize}
  \item \textbf{show ?thesis}
  \item \textbf{proof (rule ccontr)}
    \item assume \(\text{ww: } w \neq v\)
    \item have \(\forall qq. \text{sendMsg} Ate-M r qq p (\rho r qq) = \text{sendMsg} Ate-M r qq q (\rho r qq)\)
      \item by (auto simp: Ate-SHOMachine-def Ate-sendMsg-def)
    \item with \(\text{ww}\)
    \item have \(\{ qq. \text{sendMsg} Ate-M r qq p (\rho r qq) = v\} \cap \{ qq. \text{sendMsg} Ate-M r qq q (\rho r qq) = w\} = {}\)
      \item by auto
    \item with \(\text{gtn}\)
    \item have \(\text{card}\ \{ qq. \text{sendMsg} Ate-M r qq p (\rho r qq) = v\} \cup \{ qq. \text{sendMsg} Ate-M r qq q (\rho r qq) = w\} > N\)
      \item by (auto simp: card-Un-Int)
    \item moreover
    \item have \(\text{card}\ \{ qq. \text{sendMsg} Ate-M r qq p (\rho r qq) = v\} \cup \{ qq. \text{sendMsg} Ate-M r qq q (\rho r qq) = w\} \leq N\)
      \item by (auto simp: card-mono)
    \item ultimately
    \item show \(\text{False by auto}\)
  \end{itemize}
\textbf{qed}

\textbf{qed}

If process \(p\) decides at step \(r\) and process \(q\) decides at some later step \(r+k\) then \(p\) and \(q\) decide the same value.

\textbf{lemma laterProcessDecidesSameValue :}

\textbf{assumes run: SHORun Ate-M rho HOs SHOs}\n
\textbf{and comm: }\forall r. \text{SHOcommPerRd Ate-M (HOs r) (SHOs r)}\n
\textbf{and nd1: }\text{decide}(\rho r p) \neq \text{Some } v\n
\begin{itemize}
  \item \(\text{nd1: } \text{decide}(\rho r p) \neq \text{Some } v\)
and \( d1 \): decide \((\rho (\text{Suc} \ r) \ p) = \text{Some} \ v\)

and \( nd2 \): decide \((\rho (r+k) \ q) \neq \text{Some} \ w\)

and \( d2 \): decide \((\rho (\text{Suc} (r+k)) \ q) = \text{Some} \ w\)

shows \( w = v \)

**proof (rule ccontr)**

assume \( \text{vdifw}: w \neq v \)

have \( kgt0: k > 0 \)

proof (rule ccontr)

assume \( \neg k > 0 \)

hence \( k = 0 \) by auto

with \( \text{run comm nd1 d1 nd2 d2} \) have \( w = v \)

by (auto dest: \text{common-decision})

with \( \text{vdifw} \) show \( \text{False} \).

qed

have \( 1: \{ qq. \text{sendMsg Ate-M r qq p (\rho r qq)} = v \} \)

and \( \{ qq. \text{sendMsg Ate-M (r+k) qq q (\rho (r+k) qq)} = w \} = \{ \} \)

(is \( \text{?sentv} \cap \text{?sentw} = \{ \} \))

proof (rule ccontr)

assume \( \neg \text{?thesis} \)

then obtain \( qq \)

where \( \text{xxr}: x (\rho r qq) = v \) and \( \text{xxw}: x (\rho (r+k) qq) = w \)

by (auto simp: \text{Ate-SHOMachine-def Ate-sendMsg-def})

have \( \exists k'<k. x (\rho (r+k') qq) \neq w \land x (\rho (r + \text{Suc} k') qq) = w \)

proof (rule ccontr)

assume \( \neg f: \neg \text{?thesis} \)

\{
  fix \( k' \)

  assume \( \text{kk':k'} < k \) hence \( x (\rho (r+k') qq) \neq w \)

  proof (induct \( k' \))

  from \( \text{xxr} \) \( \text{vdifw} \) show \( x (\rho (r + 0) qq) \neq w \) by simp

  next

  fix \( k' \)

  assume \( \text{ih':k'} < k \implies x (\rho (r+k') qq) \neq w \)

  and \( \text{kk':Suc} k' < k \)

  from \( \text{kk'} \) have \( k' < k \) by simp

  with \( \text{ih'} \) show \( x (\rho (r + \text{Suc} k') qq) \neq w \) by auto

  qed
\}

with \( f \) have \( \forall k' < k. x (\rho (r + \text{Suc} k') qq) \neq w \) by auto

moreover

from \( kgt0 \) have \( k - 1 < k \) and \( \text{kk':Suc} (k - 1) = k \) by auto

ultimately

have \( x (\rho (r + \text{Suc} (k - 1)) qq) \neq w \) by blast

with \( \text{xxw} \) \( \text{kk} \) show \( \text{False} \) by simp

qed

then obtain \( k' \)

where \( k' < k \)
and $w$: $x\ (\rho\ (r\ +\ Suc\ k')\ qq) = w$
and $\mathit{qupdatex}$: $x\ (\rho\ (r\ +\ Suc\ k')\ qq) \neq x\ (\rho\ (r\ +\ k')\ qq)$
by auto
from run comm nd1 d1 $\mathit{qupdatex}$
have $x\ (\rho\ (r\ +\ Suc\ k')\ qq) = v$ by (rule $\mathit{common-x}$)
with $w$ show $\mathit{False}$ by simp
qed
from run comm nd1 d1 have $\mathit{sentv}$: $\mathit{card}\ ?\mathit{sentv} > E - \alpha$
by (auto dest: $\mathit{decide-sent-msgs-threshold}$)
from run comm nd2 d2 have $\mathit{sentw}$: $\mathit{card}\ ?\mathit{sentw} > E - \alpha$
by (auto dest: $\mathit{decide-sent-msgs-threshold}$)
with $\mathit{sentv}\ \mathit{majE}$ have $(\mathit{card}\ ?\mathit{sentv}) + (\mathit{card}\ ?\mathit{sentw}) > N$
by simp
with 1 $\mathit{vdifw}$ have 2: $\mathit{card}\ (\mathit{?sentv} \cup ?\mathit{sentw}) > N$
by (auto simp: $\mathit{card-Un-Int}$)
have $\mathit{card}\ (\mathit{?sentv} \cup ?\mathit{sentw}) \leq N$
by (auto simp: $\mathit{card-mono}$)
with 2 show $\mathit{False}$ by simp
qed
The Agreement property is now an immediate consequence.

theorem $\mathit{ate-agreement}$:
assumes run: $\mathit{SHORun}\ Ate-M\ \rho\ \mathit{HOs}\ \mathit{SHOs}$
and $\mathit{comm}$: $\forall r.\ \mathit{SHOcommPerRd}\ Ate-M\ (\mathit{HOs}\ r)\ (\mathit{SHOs}\ r)$
and $p$: $\mathit{decide}\ (\rho\ m\ p) = \mathit{Some}\ v$
and $q$: $\mathit{decide}\ (\rho\ n\ q) = \mathit{Some}\ w$
shows $w = v$
proof
from run $p$ obtain $k$ where
$k$: $k < m$ $\mathit{decide}\ (\rho\ k\ p) \neq \mathit{Some}\ v$ $\mathit{decide}\ (\rho\ (Suc\ k)\ p) = \mathit{Some}\ v$
by (rule $\mathit{decisionNonNullThenDecided}$)
from run $q$ obtain $l$ where
$l$: $l < n$ $\mathit{decide}\ (\rho\ l\ q) \neq \mathit{Some}\ w$ $\mathit{decide}\ (\rho\ (Suc\ l)\ q) = \mathit{Some}\ w$
by (rule $\mathit{decisionNonNullThenDecided}$)
show $\mathit{thesis}$
proof (cases $k \leq l$)
case True
then obtain $i$ where $l = k + i$ by (auto simp add: le_iff_add)
with run comm $k\ l$ show $\mathit{thesis}$
by (auto dest: $\mathit{laterProcessDecidesSameValue}$)
next
case False
hence $l \leq k$ by simp
then obtain $i$ where $m$: $k = l + i$ by (auto simp add: le_iff_add)
with run comm $k\ l$ show $\mathit{thesis}$
by (auto dest: $\mathit{laterProcessDecidesSameValue}$)
qed
qed
9.7 Proof of Termination

We now prove that every process must eventually decide, given the global and round-by-round communication predicates.

**theorem ate-termination:**

- **assumes** `run`: `SHORun Ate-M rho HOs SHOs`
- and `commR`: `∀ r. (SHOcommPerRd::((Proc, 'val::linorder pstate, 'val) SHOMachine) => (Proc HO) => (Proc HO) => bool)`
- and `commG`: `SHOcommGlobal Ate-M HOs SHOs`

**shows** `∃ r v. decide (rho r p) = Some v`

**proof**

- from `commG` obtain `r' π1 π2`
  - where `πea`: `card π1 > E - α`
  - and `πt`: `card π2 > T`
  - and `hosho`: `∀ p ∈ π1. (HOs r' p = π2 ∧ SHOs r' p ∩ HOs r' p = π2)`
  - by `(auto simp: Ate-SHOMachine-def Ate-commGlobal-def)`

obtain `v` where

- `P1`: `∀ pp. card `{qq. sendMsg Ate-M (Suc r') qq pp (rho (Suc r') qq) = v} > E - α`

**proof**

- have `∀ p ∈ π1. ∀ q ∈ π1. x (rho (Suc r') p) = x (rho (Suc r') q)`
- proof `(clarify)`
  - fix `p q`
  - assume `p: p ∈ π1` and `q: q ∈ π1`

- from `run` obtain `µp`
  - where `nxtp: nextState Ate-M r' p (rho r' p) µp (rho (Suc r') p)`
  - and `mup: µp ∈ SHOmsgVectors Ate-M r' p (rho r' p) (SHOs r' p)`
  - by `(auto simp: SHORun-eq SHOnextConfig-eq)`

- from `run` obtain `µq`
  - where `nxtg: nextState Ate-M r' q (rho r' q) µq (rho (Suc r') q)`
  - and `muq: µq ∈ SHOmsgVectors Ate-M r' q (rho r' q) (SHOs r' q)`
  - by `(auto simp: SHORun-eq SHOnextConfig-eq)`

- from `mup muq p q`
  - have `{qq. µq qq ≠ None} = HOs r' q`
  - and `2:{qq. µq qq = Some (sendMsg Ate-M r' qq q (rho r' qq))} ⊇ SHOs r' q ∩ HOs r' q`
  - and `{qq. µp qq ≠ None} = HOs r' p`
  - and `4:{qq. µp qq = Some (sendMsg Ate-M r' qq p (rho r' qq))} ⊇ SHOs r' p ∩ HOs r' p`
  - by `(auto simp: SHOmsgVectors-def)`

with `p q hosho`
have \( \pi_2 = \{ qq. \mu qq \neq \text{None} \} \)
and \( \pi_2 = \{ qq. \mu qq \neq \text{None} \} \) by auto
from \( p \ q \) hosto 2
have \( bb : \{ qq. \mu qq = \text{Some } (\text{sendMsg Ate-M r' qq q (rho r' qq)}) \} \supseteq \pi_2 \)
by auto
from \( p \ q \) hosto 4
have \( dd : \{ qq. \mu qq = \text{Some } (\text{sendMsg Ate-M r' qq p (rho r' qq)}) \} \supseteq \pi_2 \)
by auto
have Min (mostOftenRcvd \( \mu qq \)) = Min (mostOftenRcvd \( \mu qq \))
proof
have \( \forall qq. \text{sendMsg Ate-M r' qq p (rho r' qq)} = \text{sendMsg Ate-M r' qq q (rho r' qq)} \)
by (auto simp: Ate-SHOMachine-def Ate-sendMsg-def)
moreover
from \( aa \ bb \ cc \ dd \) have \( \forall qq. \mu qq \neq \text{None} \rightarrow \mu qq = \mu qq \)
by force
ultimately
have \( \forall qq. \mu qq = \mu qq \) by blast
thus \( \psi \)thesis by (auto simp: mostOftenRcvd-def)
qed
with \( \pi t \ aa \ nxtq \ pi t \ cc \ nxtp \)
show \( x (\rho \ (\text{Suc } r')) = x (\rho \ (\text{Suc } r')) \)
by (auto simp: Ate-SHOMachine-def nextState-def Ate-nextState-def)
qed
then obtain \( v \) where \( P v : \forall p \in \pi 1. x (\rho \ (\text{Suc } r')) = v \) by blast
{ fix \( pp \) from \( P v \) have \( \forall p \in \pi 1. \text{sendMsg Ate-M (Suc r') p pp (rho (Suc r')) = v} \)
by (auto simp: Ate-SHOMachine-def Ate-sendMsg-def)
hence \( \text{card } \pi 1 \leq \text{card } \{ qq. \text{sendMsg Ate-M (Suc r') qq pp (rho (Suc r') qq)} = v \} \)
by (auto intro: card-mono)
with \( \pi ea \)
have \( E - \alpha < \text{card } \{ qq. \text{sendMsg Ate-M (Suc r') qq pp (rho (Suc r') qq)} = v \} \)
by simp
}
with that show \( \psi \)thesis by blast
qed

{ fix \( k \ pp \)
have \( E - \alpha < \text{card } \{ qq. \text{sendMsg Ate-M (Suc r' + k) qq pp (rho (Suc r' + k) qq)} = v \} \)
(is \( ?P k \))
proof (induct k)
  from P1 show ?P 0 by simp
next
  fix k
  assume ih: ?P k
  from commR
  have (SHOcommPerRd::((Proc, 'val::linorder pstate, 'val) SHOMachine)
    ⇒ (Proc HO) ⇒ (Proc HO) ⇒ bool)
    Ate-M (HOs (Suc r' + k)) (SHOs (Suc r' + k)) ..
  moreover
  from ih have E − α < card \{qq. x (rho (Suc r' + k) qq) = v\}
    by (auto simp: Ate-SHOMachine-def Ate-sendMsg-def)
  ultimately
  have E − α < card \{qq. x (rho (Suc r' + Suc k) qq) = v\}
    by (rule common-x-induct[OF run])
  thus ?P (Suc k)
    by (auto simp: Ate-SHOMachine-def Ate-sendMsg-def)
  qed
}
ote P2 = this
{
  fix k pp
  assume ppupdate: x (rho (Suc r' + Suc k) pp) ≠ x (rho (Suc r' + k) pp)
  from commR
  have (SHOcommPerRd::((Proc, 'val::linorder pstate, 'val) SHOMachine)
    ⇒ (Proc HO) ⇒ (Proc HO) ⇒ bool)
    Ate-M (HOs (Suc r' + k)) (SHOs (Suc r' + k)) ..
  moreover
  from run obtain µpp
    where nxt:nextState Ate-M (Suc r' + k) pp (rho (Suc r' + k) pp) µpp
      (rho (Suc r' + Suc k) pp)
    and mu: µpp ∈ SHOmsgVectors Ate-M (Suc r' + k) pp (rho (Suc r' + k))
      (HOs (Suc r' + k) pp) (SHOs (Suc r' + k) pp)
    by (auto simp: SHORun-eq SHOnextConfig-eq)
  moreover
  from nxt ppupdate
  have threshold-T: card \{qq. µpp qq \neq None\} > T
    and xsmall: x (rho (Suc r' + Suc k) pp) = Min (mostOftenRcvd µpp)
    by (auto simp: Ate-SHOMachine-def nextState-def Ate-nextState-def)
  moreover
  from P2
  have E − α < card \{qq. sendMsg Ate-M (Suc r' + k) qq pp (rho (Suc r' + k) qq) = v\}.
    ultimately
  have mostOftenRcvd µpp = \{v\} by (auto dest!: mostOftenRcvd-v)
    with xsmall
  have x (rho (Suc r' + Suc k) pp) = v by simp
have $P_4 \forall pp. \exists k. x (\rho (\text{Suc } r' + \text{Suc } k) pp) = v$
proof
  fix pp
  from \textit{commG} have $\exists r'' > r'. \text{card } (\text{HOs } r'' \ pp) > T$
    by (auto simp: \textit{Ate-SHOMachine-def} \textit{Ate-commGlobal-def})
then obtain $k$ where $\text{Suc } r' + k > r'$ \textit{and t:card } (\text{HOs } (\text{Suc } r' + k) pp) > T
  by (auto dest: less-imp-Suc-add)
moreover
from \textit{run} obtain $\mu$ where
  next: $\text{nextState } \textit{Ate-M } (\text{Suc } r' + k) \ pp (\rho (\text{Suc } r' + k) pp) \mu$ $pp$
and mu: $\mu \ pp \in \textit{SHOmsgVectors } \textit{Ate-M } (\text{Suc } r' + k) \ pp (\rho (\text{Suc } r' + k))$
 (HOs (Suc r' + k) pp) (SHOs (Suc r' + k) pp)
  by (auto simp: \textit{SHORun-\textit{eq} SHO\textit{nextConfig-\textit{eq}}})
moreover
have $x (\rho (\text{Suc } r' + \text{Suc } k) pp) = v$
proof
  from \textit{commR} have ($\text{SHOcommPerRd::}(\text{Proc}, \text{val::linorder pstate} , \text{val::linorder}) \textit{SHOMachine}$)
    $\Rightarrow (\text{Proc } \text{HO}) \Rightarrow (\text{Proc } \text{HO}) \Rightarrow \text{bool}$
    $\textit{Ate-M } (\text{HOs } (\text{Suc } r' + k)) (\text{SHOs } (\text{Suc } r' + k))$ ..
moreover
from mu have $\text{HOs } (\text{Suc } r' + k) \ pp = \{ q. \mu \ pp \ q \neq \text{None} \}$
  by (auto simp: \textit{SHOmsgVectors-def})
with next t
  have threshold-T: $\text{card } \{ q. \mu \ pp \ q \neq \text{None} \} > T$
and xsmall: $x (\rho (\text{Suc } r' + \text{Suc } k) pp) = \text{Min } (\textit{mostOftenRcvd } \mu \ pp)$
  by (auto simp: \textit{Ate-SHOMachine-def} \textit{nextState-def Ate-nextState-def})
moreover
from $P_2$ have $E - \alpha < \text{card } \{ qq. \text{sendMsg } \textit{Ate-M } (\text{Suc } r' + k) \ qq \ pp (\rho (\text{Suc } r' + k) \ qq) = v \}$ ..
ultimately
  have $\textit{mostOftenRcvd } \mu \ pp = \{ v \}$
  using next mu by (auto dest!: \textit{mostOftenRcvd-v})
with xsmall show ?thesis by auto
qed
thus $\exists k. x (\rho (\text{Suc } r' + \text{Suc } k) pp) = v$ ..
qed

have $P_5 a: \forall pp. \exists rr. \forall k. x (\rho (\text{rr } + k) \ pp) = v$
proof
  fix pp
from $P_4$ obtain $rk$ where
  $\textit{xtvn: } x (\rho (\text{Suc } r' + \text{Suc } rk) \ pp) = v$ (is $x (\rho (\text{rr } \ pp) = v)$
by blast
have \( \forall k. x (\rho (\varr + k) \text{pp}) = v \)
proof
fix \( k \)
show \( x (\rho (\varr + k) \text{pp}) = v \)
proof (induct \( k \))
  from \( x \text{rrv} \) show \( x (\rho (\varr + 0) \text{pp}) = v \) by simp
next
fix \( k \)
assume \( \text{ih} \): \( x (\rho (\varr + k) \text{pp}) = v \)
obtain \( k' \) where \( \text{rrk} \): \( \text{Suc} \ r' + k' = \varr + k \) by auto
show \( x (\rho (\varr + \text{Suc} k) \text{pp}) = v \)
proof (rule \( \text{ccontr} \))
  assume \( \text{nv} \): \( x (\rho (\varr + \text{Suc} k) \text{pp}) \neq v \)
  with \( \text{rrk} \) \( \text{ih} \) have \( x (\rho (\text{Suc} r' + \text{Suc} k') \text{pp}) \neq x (\rho (\text{Suc} r' + k') \text{pp}) \)
    by (simp add: \( \text{ac-simps} \))
  hence \( x (\rho (\text{Suc} r' + \text{Suc} k') \text{pp}) = v \) by (rule \( \text{P3} \))
  with \( \text{rrk} \) \( \text{nv} \) show \( \text{False} \) by (simp add: \( \text{ac-simps} \))
qed
qed
qed
thus \( \exists \text{rr}. \forall k. x (\rho (\text{rr} + k) \text{pp}) = v \) by blast
qed

from \( \text{P5a} \) have \( \exists F. \forall \text{pp} \ k. x (\rho (F \text{pp} + k) \text{pp}) = v \) by (rule \( \text{choice} \))
then obtain \( R ::(\text{Proc} \Rightarrow \text{nat}) \)
  where \( \text{imgR} \): \( R' \text{ (UNIV::Proc } \text{set) \neq } \{ \} \)
    and \( R :: \forall \text{pp} \ k. x (\rho (R \text{pp} + k) \text{pp}) = v \)
  by blast

define \( \text{rr} \) where \( \text{rr} = \text{Max} (R' \text{ UNIV}) \)

have \( \text{P5}: \forall r' > \text{rr}. \forall \text{pp}. x (\rho r' \text{pp}) = v \)
proof (clarify)
fix \( r' \text{pp} \)
assume \( r': r' > \text{rr} \)
hence \( r' > R \text{pp} \) by (auto simp: \( \text{rr-def} \))
then obtain \( i \) where \( r' = R \text{pp} + i \)
  by (auto dest: less-imp-Suc-add)
with \( R \) show \( x (\rho r' \text{pp}) = v \) by auto
qed

from \( \text{commG} \) have \( \exists r' > \text{rr}. \text{card} (\text{SHOs} r' \text{p} \cap \text{HOs} r' \text{p}) > E \)
  by (auto simp: \( \text{Ate-SHOMachine-def} \) \( \text{Ate-commGlobal-def} \))

with \( \text{P5} \) obtain \( r' \)
  where \( r' > \text{rr} \)
    and \( \text{card} (\text{SHOs} r' \text{p} \cap \text{HOs} r' \text{p}) > E \)
    and \( \forall \text{pp}. \text{sendMsg} \text{Ate-M} r' \text{pp} p (\rho \text{r'pp}) = v \)
  by (auto simp: \( \text{Ate-SHOMachine-def} \) \( \text{Ate-sendMsg-def} \))
moreover
from run obtain \( \mu p \)
where \( n x t : n x t S t a t e A t e-M r' p (r h o r' p) \mu p (r h o (S u c r') p) \)
and \( m u : \mu p \in S H O m s g V e c t o r s A t e-M r' p (r h o r') (H O s r' p) (S H O s r' p) \)
by (auto simp: SHORun-eq SHOnextConfig-eq)
from \( m u \)
have \( c a r d (S H O s r' p \cap H O s r' p) \leq c a r d \{ q . \mu p q = S o m e (s e n d M s g A t e-M r' q p (r h o r' q))\} \)
by (auto simp: SHOmsgVectors-def intro: card-mono)
ultimately
have \( t h r e s h o l d-E : c a r d \{ q . \mu p q = S o m e v \} > E \) by auto
with \( n x t s h o w ? t h e s i s \)
by (auto simp: Ate-SHOMachine-def nextState-def Ate-nextState-def)
qed

9.8 \( A_{T,E,\alpha} \) Solves Weak Consensus

Summing up, all (coarse-grained) runs of \( A_{T,E,\alpha} \) for HO and SHO collections that satisfy the communication predicate satisfy the Weak Consensus property.

**Theorem ate-weak-consensus:**

assumes run: \( S H O R u n A t e-M r h o H O s S H O s \)
and commR: \( \forall r . \ S H O c o m m P e r R d A t e-M (H O s r) (S H O s r) \)
and commG: \( \ S H O c o m m G l o b a l A t e-M H O s S H O s \)
shows \( w e a k - c o n s e n s u s (x \circ (r h o 0)) d e c i d e r h o \)
unfolding \( w e a k - c o n s e n s u s - d e f \) using assms
by (auto elim: ate-validity ate-agreement ate-termination)

By the reduction theorem, the correctness of the algorithm carries over to the fine-grained model of runs.

**Theorem ate-weak-consensus-fg:**

assumes run: \( f g - r u n A t e-M r h o H O s S H O s (\lambda r q . u n d e f i n e d) \)
and commR: \( \forall r . \ S H O c o m m P e r R d A t e-M (H O s r) (S H O s r) \)
and commG: \( \ S H O c o m m G l o b a l A t e-M H O s S H O s \)
shows \( w e a k - c o n s e n s u s (\lambda p . x (s t a t e (r h o 0) p)) d e c i d e (s t a t e \circ r h o) \)
(is \( w e a k - c o n s e n s u s \ ? i n i t s - - ) \)
proof (rule local-property-reduction[OF run weak-consensus-is-local!])
fix crun
assume crun: \( C S H O R u n A t e-M c r u n H O s S H O s (\lambda r q . u n d e f i n e d) \)
and init: \( c r u n 0 = s t a t e (r h o 0) \)
from crun have \( S H O R u n A t e-M c r u n H O s S H O s \) by (unfold SHORun-def)
from this commR commG
have \( w e a k - c o n s e n s u s (x \circ (c r u n 0)) d e c i d e c r u n \)
by (rule ate-weak-consensus)
with init show \( w e a k - c o n s e n s u s \ ? i n i t s d e c i d e c r u n \)
by (simp add: o-def)
qed
10 Verification of the \textit{EIGByz}_f Consensus Algorithm

Lynch \cite{Lynch12} presents \textit{EIGByz}_f, a version of the exponential information gathering algorithm tolerating Byzantine faults, that works in $f$ rounds, and that was originally introduced in \cite{Lynch91}.

We begin by introducing an anonymous type of processes of finite cardinality that will instantiate the type variable $\text{Proc}$ of the generic HO model.

\begin{verbatim}
typedef Proc — the set of processes
axiomatization where Proc-finite: OFCLASS(Proc, finite-class)
instance Proc :: finite by (rule Proc-finite)
\end{verbatim}

\textbf{abbreviation}

\begin{verbatim}
N ≡ card (UNIV::Proc set) — number of processes
\end{verbatim}

The algorithm is parameterized by $f$, which represents the number of rounds and the height of the tree data structure (see below).

\begin{verbatim}
axiomatization f::nat
where f: f < N
\end{verbatim}

\section{10.1 Tree Data Structure}

The algorithm relies on propagating information about the initially proposed values among all the processes. This information is stored in trees whose branches are labeled by lists of (distinct) processes. For example, the interpretation of an entry \([p,q] \mapsto \text{Some } v\) is that the current process heard from process $q$ that it had heard from process $p$ that its proposed value is $v$. The value initially proposed by the process itself is stored at the root of the tree.

We introduce the type of \textit{labels}, which encapsulate lists of distinct process identifiers and whose length is at most $f+1$.

\begin{verbatim}
definition Label = {xs::Proc list. length xs ≤ Suc f ∧ distinct xs}
typedef Label = Label
  by (auto simp: Label-def intro: exI[where x= []]) — the empty list is a label
\end{verbatim}

There is a finite number of different labels.

\begin{verbatim}
lemma finite-Label: finite Label
\end{verbatim}
proof  
  have Label ⊆ {xs. set xs ⊆ (UNIV::Proc set) ∧ length xs ≤ Suc f}  
    by (auto simp: Label-def)  
  moreover  
  have finite {xs. set xs ⊆ (UNIV::Proc set) ∧ length xs ≤ Suc f}  
    by (rule finite-lists-length-le) auto  
  ultimately  
  show ?thesis by (auto elim: finite-subset)  
qed

lemma finite-UNIV-Label: finite (UNIV::Label set)  
proof  
  from finite-Label have finite (Abs-Label ` Label) by simp  
  moreover  
  {  
    fix l::Label  
    have l ∈ Abs-Label ` Label  
      by (rule Abs-Label-cases) auto  
  }  
  hence (UNIV::Label set) = (Abs-Label ` Label) by auto  
  ultimately show ?thesis by simp  
qed

lemma finite-Label-set [iff]: finite (S :: Label set)  
  using finite-UNIV-Label by (auto intro: finite-subset)

Utility functions on labels.

definition root-node where  
  root-node ≡ Abs-Label []

definition length-lbl where  
  length-lbl l ≡ length (Rep-Label l)

lemma length-lbl [intro]: length-lbl l ≤ Suc f  
  unfolding length-lbl-def using Label-def Rep-Label by auto

definition is-leaf where  
  is-leaf l ≡ length-lbl l = Suc f

definition last-lbl where  
  last-lbl l ≡ last (Rep-Label l)

definition butlast-lbl where  
  butlast-lbl l ≡ Abs-Label (butlast (Rep-Label l))

definition set-lbl where  
  set-lbl l = set (Rep-Label l)

The children of a non-leaf label are all possible extensions of that label.
\textbf{definition} children where
\begin{align*}
\text{children } l & \equiv \\
\text{if is-leaf } l & \text{ then } \{ \} \\
\text{else } & \{ \text{Abs-Label (Rep-Label } l \& [p]) | p . p \notin \text{ set-lbl } l \} \\
\end{align*}

\section*{10.2 Model of the Algorithm}

The following record models the local state of a process.

\textbf{record} 'val pstate = \\
vals :: Label \Rightarrow 'val option \\
newvals :: Label \Rightarrow 'val \\
deckide :: 'val option

Initially, no values are assigned to non-root labels, and an arbitrary value is assigned to the root: that value is interpreted as the initial proposal of the process. No decision has yet been taken, and the \textit{newvals} field is unconstrained.

\textbf{definition} EIG-initState where
\begin{align*}
\text{EIG-initState } p & \text{ st } \equiv \\
(\forall l. (\text{vals } st \ l = \text{None}) = (l \neq \text{root-node})) \\
\wedge & \text{decide } st = \text{None}
\end{align*}

\textbf{type-synonym} 'val Msg = Label \Rightarrow 'val option

At every round, every process sends its current \textit{vals} tree to all processes. In fact, only the level of the tree corresponding to the round number is used (cf. definition of \textit{extend-vals} below).

\textbf{definition} EIG-sendMsg where
\begin{align*}
\text{EIG-sendMsg } r & \ p \ q \ st \equiv \text{vals } st
\end{align*}

During the first \(f - 1\) rounds, every process extends its tree \textit{vals} according to the values received in the round. No decision is taken.

\textbf{definition} extend-vals where
\begin{align*}
\text{extend-vals } r & \ p \ st \ msgs \ st' \equiv \\
\text{vals } st' & = (\lambda l. \\
\text{if length-lbl } l = \text{Suc } r \wedge & \text{msgs } (\text{last-lbl } l) \neq \text{None} \\
\text{then } & (\text{the } (\text{msgs } (\text{last-lbl } l))) \ (\text{butlast-lbl } l) \\
\text{else if length-lbl } l = & \text{Suc } r \wedge \text{msgs } (\text{last-lbl } l) = \text{None} \text{ then None} \\
\text{else } & \text{vals } st \ l)
\end{align*}

\textbf{definition} next-main where
\begin{align*}
\text{next-main } r & \ p \ st \ msgs \ st' \equiv \text{extend-vals } r \ p \ st \ msgs \ st' \wedge \text{decide } st' = \text{None}
\end{align*}

In the final round, in addition to extending the tree as described previously, processes construct the tree \textit{newvals}, starting at the leaves. The values at the leaves are copied from \textit{vals}, except that missing values \textit{None} are replaced
by the default value `undefined`. Moving up, if there exists a majority value
among the children, it is assigned to the parent node, otherwise the parent
node receives the default value `undefined`. The decision is set to the value
computed for the root of the tree.

```latex
fun fixupval :: `'val option ⇒ `'val where
  fixupval None = undefined
  | fixupval (Some v) = v
```

```latex
definition has-majority :: `'val ⇒ (′a ⇒ `'val) ⇒ ′a set ⇒ bool where
  has-majority v g S ≡ card \{ e ∈ S. g e = v\} > (card S) div 2
```

```latex
definition check-newvals :: ′val pstate ⇒ bool where
  check-newvals st ≡ ∀ l. is-leaf l ∧ newvals st l = fixupval (vals st l)
  \lor (is-leaf l) ∧
  ( (∃ w. has-majority w (newvals st) (children l) ∧ newvals st l = w)
  \lor (¬ (∃ w. has-majority w (newvals st) (children l))
    ∧ newvals st l = undefined))
```

```latex
definition next-end where
  next-end r p st msgs st' ≡
  extend-vals r p st msgs st'
  ∧ check-newvals st'
  ∧ decide st' = Some (newvals st' root-node)
```

The overall next-state relation is defined such that every process applies
`nextMain` during rounds 0, . . . , `f − 1`, and applies `nextEnd` during round `f`.
After that, the algorithm terminates and nothing changes anymore.

```latex
definition EIG-nextState where
  EIG-nextState r ≡
  if r < `f then next-main r
  else if r = `f then next-end r
  else (λp st msgs st'. st' = st)
```

### 10.3 Communication Predicate for `EIGByz`

The secure kernel `SKr` w.r.t. given HO and SHO collections consists of the
process from which every process receives the correct message.

```latex
definition SKr :: Proc HO ⇒ Proc HO ⇒ Proc set where
  SKr HO SHO ≡ \{ q . ∀ p. q ∈ HO p ∩ SHO p\}
```

The secure kernel `SK` of an entire execution (i.e., for sequences of HO and
SHO collections) is the intersection of the secure kernels for all rounds. Ob-
viously, only the first `f` rounds really matter, since the algorithm terminates
after that.

```latex
definition SK :: (nat ⇒ Proc HO) ⇒ (nat ⇒ Proc HO) ⇒ Proc set where
  SK HOs SHOs ≡ \{ q. ∀ r. q ∈ SKr (HOs r) (SHOs r)\}
```
The round-by-round predicate requires that the secure kernel at every round contains more than \((N+f) \div 2\) processes.

**definition EIG-commPerRd where**

\[
EIG\text{-commPerRd} \text{ HO SHO} \equiv \text{card } (SKr \text{ HO SHO}) > (N + f) \div 2
\]

The global predicate requires that the secure kernel for the entire execution contains at least \(N - f\) processes. Messages from these processes are always correctly received by all processes.

**definition EIG-commGlobal where**

\[
EIG\text{-commGlobal} \text{ HOs SHOs} \equiv \text{card } (SK \text{ HOs SHOs}) \geq N - f
\]

The above communication predicates differ from Lynch’s presentation of \(EIGByzf\). In fact, the algorithm was originally designed for synchronous systems with reliable links and at most \(f\) faulty processes. In such a system, every process receives the correct message from at least the non-faulty processes at every round, and therefore the global predicate \(EIG\text{-commGlobal}\) is satisfied. The standard correctness proof assumes that \(N > 3f\), and therefore \(N - f > (N + f) \div 2\). Since moreover, for any \(r\), we obviously have

\[
\left( \bigcap_{p \in \Pi, r' \in \mathbb{N}} SHO(p, r') \right) \subseteq \left( \bigcap_{p \in \Pi} SHO(p, r) \right),
\]

it follows that any execution of \(EIGByzf\) where \(N > 3f\) also satisfies \(EIG\text{-commPerRd}\) at any round. The standard correctness hypotheses thus imply our communication predicates.

However, our proof shows that \(EIGByzf\) can indeed tolerate more transient faults than the standard bound can express. For example, consider the case where \(N = 5\) and \(f = 2\). Our predicates are satisfied in executions where two processes exhibit transient faults, but never fail simultaneously. Indeed, in such an execution, every process receives four correct messages at every round, hence \(EIG\text{-commPerRd}\) always holds. Also, \(EIG\text{-commGlobal}\) is satisfied because there are three processes from which every process receives the correct messages at all rounds. By our correctness proof, it follows that \(EIGByzf\) then achieves Consensus, unlike what one could expect from the standard correctness predicate. This observation underlines the interest of expressing assumptions about transient faults, as in the HO model.

### 10.4 The \(EIGByzf\) Heard-Of Machine

We now define the non-coordinated SHO machine for \(EIGByzf\) by assembling the algorithm definition and its communication-predicate.

**definition EIG-SHOMachine where**

\[
EIG\text{-SHOMachine} = \emptyset
\]

\[
\text{CinitState} = (\lambda p \, \text{st \ crd. EIG-initState} \ p \ \text{st}),
\]
abbreviation EIG-M ≡ (EIG-SHOMachine::{Proc, 'val pstate, 'val Msg} SHOMachine)

end

theory EigbyzProof
imports EigbyzDefs ../Majorities ../Reduction
begin

10.5 Preliminary Lemmas

Some technical lemmas about labels and trees.

lemma not-leaf-length:
  assumes l: ¬(is-leaf l)
  shows length-lbl l ≤ f
  using l length-lbl[of l] by (simp add: is-leaf-def)

lemma nil-is-Label: [] ∈ Label
  by (auto simp: Label-def)

lemma card-set-lbl: card (set-lbl l) = length-lbl l
  unfolding set-lbl-def length-lbl-def
  using Rep-Label[of l, unfolded Label-def]
  by (auto elim: distinct-card)

lemma Rep-Label-root-node [simp]: Rep-Label root-node = []
  using nil-is-Label by (simp add: root-node-def Abs-Label-inverse)

lemma root-node-length [simp]: length-lbl root-node = 0
  by (simp add: length-lbl-def)

lemma root-node-not-leaf: ¬(is-leaf root-node)
  by (simp add: is-leaf-def)

Removing the last element of a non-root label gives a label.

lemma butlast-rep-in-label:
  assumes l:l ≠ root-node
  shows butlast (Rep-Label l) ∈ Label

proof
  have Rep-Label l ≠ []

proof
  assume Rep-Label l = []
  hence Rep-Label l = Rep-Label root-node by simp

with l show False by (simp only: Rep-Label-inject)
The label of a child is well-formed.

**Lemma Rep-Label-append:**

**Assumes** \( l : \neg (\text{is-leaf } l) \)

**Shows** \((\text{Rep-Label } l @ [p] \in \text{Label}) = (p \notin \text{set-lbl } l)\)

**Proof**

\[ \text{assume } \text{lhs} \rightarrow \text{rhs} \]

\[ \text{by (auto simp: Label-def elim: distinct-butlast)} \]

**Qed**

The label of a child is well-formed.

The label of any child node is one longer than the label of its parent.

**Lemma children-length:**

**Assumes** \( l \in \text{children } h \)

**Shows** \( \text{length-lbl } l = \text{Suc} (\text{length-lbl } h) \)

**Using** \( \text{label-children} \)

**Proof**

\[ \text{by (auto simp: length-lbl-def)} \]

**Qed**

The root node is never a child.

**Lemma children-not-root:**

**Assumes** \( \text{root-node } \in \text{children } l \)

**Shows** \( P \)
by (auto simp: root-node-def)

The label of a child with the last element removed is the label of the parent.

lemma children-butlast-lbl:
assumes c ∈ children l
shows butlast-lbl c = l
using label-children[OF assms]
by (auto simp: butlast-lbl-def Rep-Label-inverse)

The root node is not a child, and it is the only such node.

lemma root-iff-no-child: (l = root-node) = (∀ l'. l ∈ children l')
proof
assume l = root-node
thus ∀ l'. l ∈ children l' by (auto elim: children-not-root)
next
assume rhs: ∀ l'. l ∈ children l'
show l = root-node
proof (rule rev-exhaust[of Rep-Label l])
assume Rep-Label l = []
hence Rep-Label l = Rep-Label root-node by simp
thus ?thesis by (simp only: Rep-Label-inject)
next
fix l' q
assume l': Rep-Label l = l' @ [q]
let ?l' = Abs-Label l'
from Rep-Label[of l] l' have l' ∈ Label by (simp add: Label-def)
hence repl': Rep-Label ?l' = l' by (rule Abs-Label-inverse)
from Rep-Label[of l] l' have l' @ [q] ∈ Label by (simp add: Label-def)
with l' have Rep-Label l = Rep-Label (Abs-Label (l' @ [q]))
  by (simp add: Abs-Label-inverse)
hence l = Abs-Label (l' @ [q]) by (simp add: Rep-Label-inject)
moreover
from Rep-Label[of l] l' have length l' < Suc f q ∉ set l'
  by (auto simp: Label-def)
moreover
note repl'
ultimately have l ∈ children ?l'
  by (auto simp: children-def is-leaf-def length-lbl-def set-lbl-def)
with rhs show ?thesis by blast
qed

qed

If some label l is not a leaf, then the set of processes that appear at the end of the labels of its children is the set of all processes that do not appear in l.

lemma children-last-set:
assumes l: ¬(is-leaf l)
shows last-lbl ' (children l) = UNIV - set-lbl l

150
proof

show \( \text{last-lbl} \ (\text{children } l) \subseteq \text{UNIV} - \text{set-lbl } l \)
by (auto dest: label-children simp: last-lbl-def)

next

show \( \text{UNIV} - \text{set-lbl } l \subseteq \text{last-lbl} \ (\text{children } l) \)
proof (auto simp: image-def)

fix \( p \)
assume \( p: p \notin \text{set-lbl } l \)
with \( l \) have \( c: \text{Abs-Label} \ (\text{Rep-Label } l \ @ \ [p]) \in \text{children } l \)
by (auto simp: children-def)

with \( \text{Rep-Label-append(OF } l \ p \) show \( \exists c \in \text{children } l. \ p = \text{last-lbl } c \)
by (force simp: last-lbl-def Abs-Label-inverse)

qed

qed

The function returning the last element of a label is injective on the set of children of some given label.

lemma last-lbl-inj-on-children: inj-on \( \text{last-lbl} \ (\text{children } l) \)
proof (auto simp: inj-on-def)

fix \( c \ c' \)
assume \( c: c \in \text{children } l \) and \( c': c' \in \text{children } l \)
and eq: \( \text{last-lbl } c = \text{last-lbl } c' \)
from \( c \ c' \) obtain \( p \ p' \)
where \( p: \text{Rep-Label } c = \text{Rep-Label } l \ @ \ [p] \)
and \( p': \text{Rep-Label } c' = \text{Rep-Label } l \ @ \ [p'] \)
by (auto dest!: label-children)

from \( p \ p' \) eq have \( p = p' \) by (simp add: last-lbl-def)
with \( p \ p' \) have \( \text{Rep-Label } c = \text{Rep-Label } c' \) by simp
thus \( c = c' \) by (simp add: Rep-Label-inject)

qed

The number of children of any non-leaf label \( l \) is the number of processes that do not appear in \( l \).

lemma card-children:
assumes \( \sim(\text{is-leaf } l) \)
shows \( \text{card} \ (\text{children } l) = N - (\text{length-lbl } l) \)
proof

from assms have \( \text{last-lbl} \ (\text{children } l) = \text{UNIV} - \text{set-lbl } l \)
by (rule children-last-set)

moreover have \( \text{card} \ (\text{UNIV} - \text{set-lbl } l) = \text{card} \ (\text{UNIV::Proc set}) - \text{card} \ (\text{set-lbl } l) \)
by (auto simp: card-Diff-subset-Int)

moreover from last-lbl-inj-on-children have \( \text{card} \ (\text{children } l) = \text{card} \ (\text{last-lbl} \ (\text{children } l)) \)
by (rule sym[OF card-image])

moreover
note \text{card-set-lbl}[\text{of } l]
ultimately
show \text{?thesis by auto}
qed

Suppose a non-root label $l'$ of length $r+1$ ending in $q$, and suppose that $q$ is well heard by process $p$ in round $r$. Then the value with which $p$ decorates $l$ is the one that $q$ associates to the parent of $l$.

\textbf{lemma sho-correct-vls:}
\begin{itemize}
\item \textbf{assumes} \text{run: } \text{SHORun EIG-M } \rho \text{ HOs SHOs}
\item \text{and } l' \in \text{children } l
\item \text{and } \text{shop: } \text{last-lbl } l' \in \text{SHOs (length-lbl } l) \cap \text{HOs (length-lbl } l)
\item \text{is } q \in \text{SHOs (len } l \cap -)
\end{itemize}
shows vals (rho (?len l') p) l' = vals (rho (?len l) ?q) l

\textbf{proof} --
\begin{itemize}
\item let \ $?r = ?len l$
\item from \text{run} obtain \mu p
\item where \text{nxt: } \text{nextState EIG-M } ?r p \text{ (rho } ?r p) \mu p \text{ (rho (Suc } ?r p))
\item and \text{mu: } \mu p \in \text{SHOmsgVectors EIG-M } \rho \text{ p (rho } ?r) \text{ (HOs } ?r p) \text{ (SHOs } ?r p)
\item by \text{(auto simp: EIG-SHOMachine-def SHORun-eq SHOnextConfig-eq)}
\item with \text{shop}
\item have \text{msl:} \mu p \ ?q = \text{Some (vals (rho } ?r ?q))
\item by \text{(auto simp: EIG-SHOMachine-def EIG-sendMsg-def SHOmsgVectors-def)}
\item from \text{nxt length-lbl}[\text{of } l'] \text{children-length}[\text{OF } l']
\item have \text{extend-vals } ?r p \text{ (rho } ?r p) \mu p \text{ (rho (Suc } ?r p))
\item by \text{(auto simp: EIG-SHOMachine-def nextState-def EIG-nextState-def next-main-def next-end-def extend-vals-def)}
\end{itemize}
with \text{msl l' show } \text{?thesis}
\item by \text{(auto simp: extend-vals-def children-length children-butlast-lbl)}
\end{itemize}
\textbf{qed}

A process fixes the value \text{vals } l of a label at state \text{length-lbl } l, and then never modifies the value.

\textbf{lemma keep-vls:}
\begin{itemize}
\item \textbf{assumes} \text{run: } \text{SHORun EIG-M } \rho \text{ HOs SHOs}
\item \text{shows vals (rho (length-lbl } l + n) \text{ p) } l = \text{vals (rho (length-lbl } l) \text{ p) } l
\item \text{is } ?v n = ?vl
\item \textbf{proof} \text{(induct } n)
\item show \text{?v } 0 = ?vl \text{ by simp}
\item next
\item fix \text{n}
\item assume \text{ih: } ?v n = ?vl
\item let \ ?r = \text{length-lbl } l + n
\item from \text{run} obtain \mu p
\item where \text{nxt: } \text{nextState EIG-M } ?r p \text{ (rho } ?r p) \mu p \text{ (rho (Suc } ?r p))
\item by \text{(auto simp: EIG-SHOMachine-def SHORun-eq SHOnextConfig-eq)}
\item with \text{ih} \text{ show } ?v (Suc \text{ n}) = ?vl
\item by \text{(auto simp: EIG-SHOMachine-def nextState-def EIG-nextState-def next-main-def next-end-def extend-vals-def)}
\end{itemize}
10.6  Lynch’s Lemmas and Theorems

If some process is safely heard by all processes at round \( r \), then all processes agree on the value associated to labels of length \( r+1 \) ending in that process.

**Lemma lynch-6-15:**
- Assumes \( \text{run}: \text{SHORun EIG-M rho HOs SHOs} \)
- And \( l', l'' \in \text{children } l \)
- Shows \( \text{vals (rho (length-lbl } l') p) l' = \text{vals (rho (length-lbl } l'') q) l'' \)
- Using \( \text{assms unfolding SKr-def by (auto simp: sho-correct-vals)} \)

Suppose that \( l \) is a non-root label whose last element was well heard by all processes at round \( r \), and that \( l' \) is a child of \( l \) corresponding to process \( q \) that is also well heard by all processes at round \( r+1 \). Then the values associated with \( l \) and \( l' \) by any process \( p \) are identical.

**Lemma lynch-6-16-a:**
- Assumes \( \text{run}: \text{SHORun EIG-M rho HOs SHOs} \)
- And \( l, l'' \in \text{children } t \)
- And \( \text{skrl}: \text{last-lbl } l \in \text{SKr (HOs (length-lbl } l)} \) (SHOs (length-lbl l))
- And \( \text{skrl'}: \text{last-lbl } l' \in \text{SKr (HOs (length-lbl l)} \) (SHOs (length-lbl l))
- Shows \( \text{vals (rho (length-lbl } l'') p) l' = \text{vals (rho (length-lbl l) p) l} \)
- Using \( \text{assms by (auto simp: SKr-def sho-correct-vals)} \)

For any non-leaf label \( l \), more than half of its children end with a process that is well heard by everyone at round \( \text{length-lbl } l \).

**Lemma lynch-6-16-c:**
- Assumes \( \text{commR: EIG-commPerRd (HOs (length-lbl l)} \) (SHOs (length-lbl l))
- And \( l, l'' \in \text{children } l \)
- Shows \( \text{card \{l' \in \text{children } l. \text{last-lbl } l' \in \text{SKr (HOs } \text{?r}) \} (SHOs ?r)\) > card (\text{children } l) div 2 \)
- 
**Proof**

- Let \( ?skr = \text{SKr (HOs } \text{?r}) \) (SHOs ?r)

  - Have \( \text{last-lbl } l, ?lhs = ?skr - \text{set-lbl } l \)
  - Proof
    - From \text{children-last-set[OF l]}
    - Show \( \text{last-lbl } l, ?lhs \subseteq ?skr - \text{set-lbl } l \)
    - By (auto simp: children-length)
  - Next
    - Fix \( p \)
    - Assume \( p: p \in ?skr p \notin \text{set-lbl } l \)
    - With \text{children-last-set[OF l]}

qed
have \( p \in \text{last-lbl ' children } l \) \text{ by } auto

with \( p \) have \( p \in \text{last-lbl ' ?lhs} \) by (auto simp: image-def children-length)

\}

thus \( \text{?skr - set-lbl } l \subseteq \text{last-lbl ' ?lhs} \) \text{ by } auto

qed

moreover

from \( \text{last-lbl-inj-on-children[of } l \) have \( \text{inj-on last-lbl ?lhs} \) \text{ by } (auto simp: inj-on-def)

ultimately

have \( \text{card ?lhs} \geq \text{card (children } l \) \text{ div 2} \) \text{ by } simp

finally show \( \text{thesis} \) \text{ by } simp

moreover

from \text{commR} have \( \text{card ?skr} > (N + f) \) \text{ div 2} \text{ by } (auto simp: EIG-commPerRd-def)

with \( \text{not-leaf-length[of } l \) have \( \text{card ?skr} = ?r > (N - ?r) \) \text{ div 2} \text{ by } auto

with \( \text{card-children[of } l \) have \( \text{card ?skr} = ?r > \text{card (children } l \) \text{ div 2} \text{ by } simp

ultimately show \( \text{thesis} \) \text{ by } simp

qed

If \( l \) is a non-leaf label such that all of its children corresponding to well-heard processes at round \( \text{length-lbl } l \) have a uniform newvals decoration at round \( f + 1 \), then \( l \) itself is decorated with that same value.

\textbf{lemma newvals-skr-uniform:}

assumes \( \text{run: SHORun EIG-M rho HOs SHOs} \)

and \( \text{commR: EIG-commPerRd (HOs (length-lbl } l \) (SHOs (length-lbl } l \))} \)

and \( \text{notleaf: } \neg(\text{is-leaf } l \) \)

and \( \text{unif: } \forall l'. \forall l' \in \text{children } l; \)

\( \text{last-lbl } l' \in \text{SKr (HOs (length-lbl } l \) (SHOs (length-lbl } l \)) \)

\( \implies \text{newvals (rho (Suc } f \) p} l' = v \)

shows \( \text{newvals (rho (Suc } f \) p} l = v \)

\textbf{proof –}

from \( \text{unif} \)

have \( \text{card } \{ l' \in \text{children } l; \text{last-lbl } l' \in \text{SKr (HOs ?r) (SHOs ?r)} \} \leq \text{card } \{ l' \in \text{children } l; \text{newvals (rho (Suc } f \) p} l' = v \}

by (auto intro: card-mono)

with \text{lynch-6-16-c[of HOs l SHOs, OF commR notleaf]} have \( \text{maj: has-majority } v \text{ (newvals (rho (Suc } f \) p} ) \text{ (children } l \)

by (simp add: has-majority-def)

from \text{run} have \( \text{check-newvals (rho (Suc } f \) p} \)
by (auto simp: EIG-SHOMachine-def SHORun-eq SHOnextConfig-eq
  nextState-def EIG-nextState-def next-end-def)

with maj notleaf obtain w
  where wmaj: has-majority w (newvals (rho (Suc f) p)) (children l)
    and wupd: newvals (rho (Suc f) p) l = w
  by (auto simp: check-newvals-def)
from maj wmaj have w = v
  by (auto simp: has-majority-def elim: abs-majoritiesE')
with wupd show ?thesis by simp
qed

A node whose label $l$ ends with a process which is well heard at round $length-lbl l$ will have its newvals field set (at round $f+1$) to the “fixed-up” value given by $vals$.

**Lemma lynch-6-16-d:**

assumes run: SHORun EIG-M rho HOs SHOs
  and commR: $\forall r. \ EIG-commPerRd (HOs r) (SHOs r)$
  and notroot: $l \in \text{children } t$
  and skr: last-lbl l $\in \text{SKr (HOs (} \text{len } t \text{)) (SHOs (} \text{len } t \text{))}$
    (is - $\in \text{SKr (} \text{HOs (} \text{len } t \text{))}$)
shows newvals (rho (Suc f) p) l = fixupval (vals (rho (} \text{len } l \text{) p) l)
  (is ?P l)
using notroot skr proof (induct Suc f $-$ ?len l arbitrary: l t)
fix l t
  assume 0 $=$ Suc f $-$ ?len l
with length-lbl[of l] have leaf: is-leaf l by (simp add: is-leaf-def)

from run have check-newvals (rho (Suc f) p)
  by (auto simp: EIG-SHOMachine-def SHORun-eq SHOnextConfig-eq
  nextState-def EIG-nextState-def next-end-def)
with leaf show ?P l
  by (auto simp: check-newvals-def is-leaf-def)
next
fix k l t
assume ih: $\exists l' t'.
  [k = Suc f $-$ length-lbl l'; l' $\in \text{children } t';
    last-lbl l' $\in \text{SKr (} \text{HOs (} \text{len } l' \text{)) (SHOs (} \text{len } t' \text{))}]$
  $\implies ?P l'$
and flk: Suc k $=$ Suc f $-$ ?len l
and notroot: l $\in \text{children } t$
and skr: last-lbl l $\in \text{SKr (} \text{HOs (} \text{len } t \text{)) (SHOs (} \text{len } t \text{))}$
let $?v = \text{fixupval (} \text{vals (rho (} \text{len } l \text{) p) l)}$
from flk have notlf: $\neg$is-leaf l by (simp add: is-leaf-def)

{ fix l'
  assume l': l' $\in \text{children } l$
    and skr': last-lbl l' $\in \text{SKr (} \text{HOs (} \text{len } l \text{)) (SHOs (} \text{len } l \text{))}$

}\
from run notroot skr l′ skr′

have eavs (ρ (?len l′) p) l′ = eavs (ρ (?len l) p) l
  by (rule lynch-6-16-a)

moreover
from flk l′ have k = Suc f − ?len l′ by (simp add: children-length)
from this l′ skr′ have ?P l′ by (rule ih)

ultimately
have newvals (ρ (Suc f) p) l′ = ?v
  using notroot l′ by (simp add: children-length)
}

with run commR notlf show ?P l by (auto intro: newvals-skr-uniform)

qed

Following Lynch [12], we introduce some more useful concepts for reasoning about the data structure.

A label is *common* if all processes agree on the final value it is decorated with.

**definition common where**

common ρ l ≡ ∀ p q. newvals (ρ (Suc f) p) l = newvals (ρ (Suc f) q) l

The subtrees of a given label are all its possible extensions.

**definition subtrees where**

subtrees h ≡ { l . ∃ t. Rep-Label l = (Rep-Label h) @ t }

**lemma children-in-subtree:**

assumes l ∈ children h

shows l ∈ subtrees h

using label-children[of assms] by (auto simp: subtrees-def)

**lemma subtrees-refl [iff]:** l ∈ subtrees l

by (auto simp: subtrees-def)

**lemma subtrees-root [iff]:** l ∈ subtrees root-node

by (auto simp: subtrees-def)

**lemma subtrees-trans:**

assumes l'' ∈ subtrees l′ and l′ ∈ subtrees l

shows l'' ∈ subtrees l

using assms by (auto simp: subtrees-def)

**lemma subtrees-antisym:**

assumes l ∈ subtrees l′ and l′ ∈ subtrees l

shows l′ = l

using assms by (auto simp: subtrees-def Rep-Label-inject)

**lemma subtrees-tree:**
assumes $l': l \in \text{subtrees } l'$ and $l'': l \in \text{subtrees } l''$

shows $l' \in \text{subtrees } l'' \vee l'' \in \text{subtrees } l'$

using assms proof (auto simp: subtrees-def append-eq-append-conv2)

fix $xs$

assume $\text{Rep-Label } l' @ xs = \text{Rep-Label } l'$

hence $\text{Rep-Label } l' = \text{Rep-Label } l'' @ xs$ by (rule sym)

thus $\exists ys. \text{Rep-Label } l' = \text{Rep-Label } l'' @ ys$ ..

qed

lemma subtrees-cases:

assumes $l'. l' \in \text{subtrees } l$

and self: $l' = l \Longrightarrow P$

and child: $\forall c. c \in \text{children } l; l' \in \text{subtrees } c \Longrightarrow P$

shows $P$

proof

- from $l'$ obtain $t$ where $t: \text{Rep-Label } l' = (\text{Rep-Label } l) @ t$

  by (auto simp: subtrees-def)

  have $l' = l \vee (\exists c \in \text{children } l. l' \in \text{subtrees } c)$

proof (cases $t$)

  assume $t = []$

  with $t$ show $\theta$thesis by (simp add: Rep-Label-inject)

next

fix $p t'$

assume cons: $t = p \# t'$

from $\text{Rep-Label}[of } l'\rangle t$ have $\text{length } (\text{Rep-Label } l @ t) \leq \text{Suc } f$

by (auto simp: Label-def)

with cons have notleaf: $\neg (\text{is-leaf } l)$

by (auto simp: is-leaf-def length-lbl-def)

let $?c = \text{Abs-Label } (\text{Rep-Label } l @ [p])$

from $t$ cons $\text{Rep-Label}[of } l'\rangle$ have $p: p \notin \text{set-lbl } l$

by (auto simp: Label-def set-lbl-def)

with notleaf have $c: (?c \in \text{children } l)$

by (auto simp: children-def)

moreover

from notleaf $p$ have $\text{Rep-Label } l @ [p] \in \text{Label}$

by (simp add: Rep-Label-append)

hence $\text{Rep-Label } ?c = (\text{Rep-Label } l @ [p])$

by (simp add: Abs-Label-inverse)

with cons $t$ have $l' \in \text{subtrees } ?c$

by (auto simp: subtrees-def)

ultimately show $\theta$thesis by blast

qed

thus $\theta$thesis by (auto elim!: self child)

qed

lemma subtrees-leaf:

assumes $l: \text{is-leaf } l$ and $l': l' \in \text{subtrees } l$

shows $l' = l$

157
using $l'$ proof (rule subtrees-cases)
fix $c$
assume $c \in \text{children} \ l$ — impossible
with $l$ show $?\text{thesis}$ by (simp add: children-def)
qed

lemma children-subtrees-equal:
assumes $c: c \in \text{children} \ l$ and $c': c' \in \text{children} \ l$
and $\text{sub}: c' \in \text{subtrees} \ c$
shows $c' = c$
proof –
from assms have $\text{Rep-Label} \ c' = \text{Rep-Label} \ c$
by (auto simp: subtrees-def dest: !: label-children)
thus $?\text{thesis}$ by (simp add: Rep-Label-inject)
qed

A set $C$ of labels is a subcovering w.r.t. label $l$ if for all leaf subtrees $s$ of $l$
there exists some label $h \in C$ such that $s$ is a subtree of $h$ and $h$ is a subtree
of $l$.

definition subcovering where
subcovering $C \ l \equiv$
$\forall s \in \text{subtrees} \ l. \text{is-leaf} \ s \longrightarrow (\exists h \in C. \ h \in \text{subtrees} \ l \land s \in \text{subtrees} \ h)$

A covering is a subcovering w.r.t. the root node.

abbreviation covering where
covering $C \equiv \text{subcovering} \ C \ \text{root-node}$

The set of labels whose last element is well heard by all processes throughout
the execution forms a covering, and all these labels are common.

lemma lynch-6-18-a:
assumes $\text{SHORun} \ EIG-M \ \rho \ \text{HOs} \ \text{SHOs}$
and $\forall r. \ EIG\text{-commPerRd} (\text{HOs} \ r) (\text{SHOs} \ r)$
and $l \in \text{children} \ t$
and $\text{last-lbl} \ t \in \text{SKr} (\text{HOs} (\text{length-lbl} \ t)) (\text{SHOs} (\text{length-lbl} \ t))$
shows $\text{common} \ \rho \ l$
using assms
by (auto simp: common-def lynch-6-16-d lynch-6-15
intro: arg-cong[where $f=\text{fixupval}$])

lemma lynch-6-18-b:
assumes $\text{run}: \text{SHORun} \ EIG-M \ \rho \ \text{HOs} \ \text{SHOs}$
and $\text{commG}: \ EIG\text{-commGlobal} \ \text{HOs} \ \text{SHOs}$
and $\text{commR}: \forall r. \ EIG\text{-commPerRd} (\text{HOs} \ r) (\text{SHOs} \ r)$
shows $\text{covering} \ \{l. \exists t. \ l \in \text{children} \ t \land \text{last-lbl} \ l \in (\text{SK} \ \text{HOs} \ \text{SHOs})\}$
proof (clarsimp simp: subcovering-def)
fix $l$
assume $\text{is-leaf} \ l$
with $\text{card-set-lbl}[\text{of} \ l]$ have $\text{card} (\text{set-lbl} \ l) = \text{Suc} \ f$

by (simp add: is-leaf-def)
with commG have $N < \operatorname{card}(SK\ HOs\ SHOs) + \operatorname{card}(\text{set-lbl}\ l)$
  by (simp add: EIG-commGlobal-def)
hence $\exists q \in \text{set-lbl}\ l.\ q \in SK\ HOs\ SHOs$
  by (auto dest: majorities-intersect)
then obtain $l_1\ q\ l_2$ where
  $l : \text{Rep-Label}\ l = (l_1 @ [q]) @ l_2$ and $q \in SK\ HOs\ SHOs$
unfolding set-lbl-def by (auto intro: split-list-propE)
let $?h = \text{Abs-Label}\ (l_1 @ [q])$
from Rep-Label[of l] l have $l_1 @ [q] \in \text{Label}$ by (simp add: Label-def)
hence $\text{length-lbl}\ ?h \neq 0$ by (simp add: length-lbl-inverse)
hence $?h \neq \text{root-node}$ by auto
then obtain $t$ where $t : ?h \in \text{children}\ t$
  by (auto simp: root-iff-no-child)
moreover
from reph q have $\text{last-lbl}\ ?h \in SK\ HOs\ SHOs$ by (simp add: last-lbl-def)
moreover
from reph l have $l \in \text{subtrees}\ ?h$ by (simp add: subtrees-def)
ultimately
show $\exists h.\ (\exists t.\ h \in \text{children}\ t) \land \text{last-lbl}\ h \in SK\ HOs\ SHOs \land l \in \text{subtrees}\ h$
  by blast
qed

If $C$ covers the subtree rooted at label $l$ and if $l \notin C$ then $C$ also covers
subtrees rooted at $l$’s children.

lemma lynch-6-19-a:
  assumes $\text{cov} : \text{subcovering}\ C\ l$
  and $l : l \notin C$
  and $e : e \in \text{children}\ l$
  shows $\text{subcovering}\ C\ e$
proof (clarsimp simp: subcovering-def)
  fix $s$
  assume $s : s \in \text{subtrees}\ e$ and leaf: $\text{is-leaf}\ s$
  from $s\ \text{children-in-subtree}[\text{OF}\ e]$ have $s \in \text{subtrees}\ l$
    by (rule subtrees-trans)
  with leaf cov obtain $h$ where $h : h \in C\ h \in \text{subtrees}\ l\ s \in \text{subtrees}\ h$
    by (auto simp: subcovering-def)
  with $l$ obtain $e'$ where $e' : e' \in \text{children}\ l\ h \in \text{subtrees}\ e'$
    by (auto elim: subtrees-cases)
  from $s \in \text{subtrees}\ h$ have $h \in \text{subtrees}\ e'\ s \in \text{subtrees}\ e'$
    by (rule subtrees-trans)
  with $s$ have $e \in \text{subtrees}\ e' \lor e' \in \text{subtrees}\ e$
    by (rule subtrees-tree)
  with $e\ e'$ have $e' = e$
    by (auto dest: children-subtrees-equal)
  with $e'\ h$ show $\exists h : h \in \text{subtrees}\ e \land s \in \text{subtrees}\ h$ by blast
qed

159
If there is a subcovering $C$ for a label $l$ such that all labels in $C$ are common, then $l$ itself is common as well.

**Lemma lynch-6-19-b:**

**Assumes**
- $\text{run}: \text{SHORun EIG-M \rho HOs SHOs}$
- $\text{cov}: \text{subcovering } C \ l$
- $\text{com}: \forall l' \in C. \text{common } \rho l'$

**Shows** $\text{common } \rho l$

**Using** $\text{cov}$ **Proof** (induct $\text{Suc } f - \text{length-lbl } l$ arbitrary: $l$)

**Fix** $l$

**Assume** $0: 0 = \text{Suc } f - \text{length-lbl } l$
- $\text{and } C: \text{subcovering } C \ l$

**From** $0 \text{ length-lbl}[\text{of } l]$ **have** $\text{is-leaf } l$
- **by** ($\text{simp add: is-leaf-def}$)

**With** $C$ **obtain** $h$ **where**
- $h: h \in C \ h \in \text{subtrees } l \ l \in \text{subtrees } h$
- **by** ($\text{auto simp: subcovering-def}$)

**Hence** $l \in C$ **by** ($\text{auto dest: subtrees-antisym}$)

**With** $\text{com}$ **show** $\text{common } \rho l$ ..

**Next**

**Fix** $k \ l$

**Assume** $k: \text{Suc } k = \text{Suc } f - \text{length-lbl } l$
- $\text{and } C: \text{subcovering } C \ l$

**And** $ih: \forall l'. \ [k = \text{Suc } f - \text{length-lbl } l'; \text{subcovering } C l'] \implies \text{common } \rho l'$

**Show** $\text{common } \rho l$

**Proof** (cases $l \in C$)
- **Case** $\text{True}$
  - **With** $\text{com}$ **show** $?\text{thesis} ..$

**Next**

- **Case** $\text{False}$
  - **With** $C$ **have** $\forall e \in \text{children } l. \text{subcovering } C e$
    - **by** ($\text{blast intro: lynch-6-19-a}$)

**Moreover**

- **From** $k$ **have** $\forall e \in \text{children } l. \ k = \text{Suc } f - \text{length-lbl } e$
  - **by** ($\text{auto simp: children-length}$)

**Ultimately**

- **Have** $\text{com-ch}: \forall e \in \text{children } l. \text{common } \rho e$
  - **by** ($\text{blast intro: } ih$)

**Show** $?\text{thesis}$

**Proof** ($\text{clarsimp simp: common-def}$)

**Fix** $p \ q$

- **From** $k$ **have** $\text{notleaf}: \neg(\text{is-leaf } l)$ **by** ($\text{simp add: is-leaf-def}$)

**Let** $?r = \text{Suc } f$

**From** $\text{com-ch}$

- **Have** $\forall e \in \text{children } l. \text{newvals } (\rho \ ?r \ p) \ e = \text{newvals } (\rho \ ?r \ q) \ e$
  - **by** ($\text{auto simp: common-def}$)

**Hence** $\forall w. \{e \in \text{children } l. \text{newvals } (\rho \ ?r \ p) \ e = w\}$

- $\{e \in \text{children } l. \text{newvals } (\rho \ ?r \ q) \ e = w\}$
  - **by** $\text{auto}$

**Moreover**
from run
have check-newvals (rho ?r p) check-newvals (rho ?r q)
by (auto simp: EIG-SHOMachine-def SHORun-eq SHOnextConfig-eq nextState-def
    EIG-nextState-def next-end-def)

with notleaf have
(∃ w. has-majority w (newvals (rho ?r p)) (children l)
    ∧ newvals (rho ?r p) l = w)
∨ ¬(∃ w. has-majority w (newvals (rho ?r p)) (children l))
    ∧ newvals (rho ?r p) l = undefined
(∃ w. has-majority w (newvals (rho ?r q)) (children l)
    ∧ newvals (rho ?r q) l = w)
∨ ¬(∃ w. has-majority w (newvals (rho ?r q)) (children l))
    ∧ newvals (rho ?r q) l = undefined
by (auto simp: check-newvals-def)
ultimately show newvals (rho ?r p) l = newvals (rho ?r q) l
by (auto simp: has-majority-def elim: abs-majoritiesE')
qued
qed
qed

The root of the tree is a common node.

lemma lynch-6-20:
assumes run: SHORun EIG-M rho HOs SHOs
    and commG: EIG-commGlobal HOs SHOs
    and commR: ∀ r. EIG-commPerRd (HOs r) (SHOs r)
shows common rho root-node

using run lynch-6-18-b[OF assms]
proof (rule lynch-6-19-b, clarify)
  fix l t
  assume l ∈ children t last-lbl l ∈ SK HOs SHOs
  thus common rho l by (auto simp: SK-def elim: lynch-6-18-a[OF run commR])
qued

A decision is taken only at state f + 1 and then stays stable.

lemma decide:
assumes run: SHORun EIG-M rho HOs SHOs
shows decide (rho r p) =
    (if r < Suc f then None
    else Some (newvals (rho (Suc f) p) root-node))
(is ?P r)

proof (induct r)
  from run show ?P 0
  by (auto simp: EIG-SHOMachine-def SHORun-eq HOinitConfig-eq
      initState-def EIG-initState-def)

next
  fix r
  assume ih: ?P r
  from run obtain µp
    where EIG-nextState r p (rho r p) µp (rho (Suc r) p)

  from run show µp = (newvals (rho (Suc r) p) (Suc r) µp)
    by (auto simp: EIG-SHOMachine-def SHORun-eq HOinitConfig-eq
        initState-def EIG-initState-def)

  ultimately show True
  by (auto simp: check-newvals-def)
qued
qed
10.7 Proof of Agreement, Validity, and Termination

The Agreement property is an immediate consequence of lemma lynch-6-20.

**Theorem Agreement:**

**Assumes** run: SHORun EIG-M rho HOs SHOs
and commG: EIG-commGlobal HOs SHOs
and commR: \( \forall r.\) EIG-commPerRd (HOs r) (SHOs r)
and \( p: \) decide (rho m p) = Some v
and \( q: \) decide (rho n q) = Some w

**Shows** v = w

**Using** p q lynch-6-20[OF run commG commR]
by (auto simp: decide[OF run] common-def)

We now show the Validity property: if all processes initially propose the same value \( v \), then no other value may be decided.

By lemma sho-correct-vals, value \( v \) must propagate to all children of the root that are well heard at round 0, and lemma lynch-6-16-d implies that \( v \) is the value assigned to all these children by newvals. Finally, lemma newvals-skr-uniform lets us conclude.

**Theorem Validity:**

**Assumes** run: SHORun EIG-M rho HOs SHOs
and commR: \( \forall r.\) EIG-commPerRd (HOs r) (SHOs r)
and initv: \( \forall q.\) the (vals (rho 0 q) root-node) = v
and dp: decide (rho r p) = Some w

**Shows** v = w

**Proof** –

have v: \( \forall q.\) vals (rho 0 q) root-node = Some v
proof
fix q
from run have vals (rho 0 q) root-node \( \neq \) None
by (auto simp: EIG-SHOMachine-def SHORun-eq HOinitConfig-eq
initState-def EIG-initState-def)
then obtain w where w: vals (rho 0 q) root-node = Some w
by auto
from initv have the (vals (rho 0 q) root-node) = v ...
with w show vals (rho 0 q) root-node = Some v by simp
Theorem Termination:

\begin{align*}
\text{assumes } & \SHORun \EIG-M \rho \HOs \SHOs \\
\text{shows } & \exists r. \text{ decide } (\rho r p) = \Some v \\
\text{using } & \assms \text{ by (auto simp: decide)}
\end{align*}

10.8 \EIGByz_f Solves Weak Consensus

Summing up, all (coarse-grained) runs of \EIGByz_f for HO and SHO collections that satisfy the communication predicate satisfy the Weak Consensus property.

Theorem \eig-weak-consensus:

\begin{align*}
\text{assumes } & \run. \SHORun \EIG-M \rho \HOs \SHOs \\
\text{and } & \commR. \forall r. \text{ EIG-commPerRd } (\HOs r) (\SHOs r) \\
\text{and } & \commG. \text{ EIG-commGlobal } \HOs \SHOs \\
\text{shows } & \weak-consensus \ (\lambda p. \text{ the } (\vals (\rho 0 p) \text{ root-node})) \text{ decide } \rho \\
\text{unfolding } & \weak-consensus-def \\
\text{using } & \Validity[\OF \run \commR] \\
& \Agreement[\OF \run \commG \commR] \\
& \Termination[\OF \run] \\
\text{by } & \text{auto}
\end{align*}

By the reduction theorem, the correctness of the algorithm carries over to
the fine-grained model of runs.

**Theorem** eig-weak-consensus-fg:

assumes \( \text{run}: \text{fg-run EIG-M rho HOs SHOs (}\lambda r q. \text{undefined}) \)
and \( \text{commR}: \forall r. \text{EIG-commPerRd (HOs r) (SHOs r)} \)
and \( \text{commG}: \text{EIG-commGlobal HOs SHOs} \)

shows weak-consensus (\( \lambda p. \text{the (vals (state (rho 0) p) root-node)}) \)
\( \text{decide (state o rho)} \)

(is weak-consensus ?inits - -)

**Proof** (rule local-property-reduction[OF run weak-consensus-is-local])

fix crun

assume crun: CSHORun EIG-M crun HOs SHOs (\( \lambda r q. \text{undefined}) \)
and init: crun 0 = state (rho 0)

from crun have SHORun EIG-M crun HOs SHOs by (unfold SHORun-def)

from this commR commG

have weak-consensus (\( \lambda p. \text{the (vals (crun 0 p) root-node)}) \) decide crun
by (rule eig-weak-consensus)

with init show weak-consensus ?inits decide crun
by (simp add: o-def)

qed

end

11 Conclusion

In this contribution we have formalized the Heard-Of model in the proof assistant Isabelle/HOL. We have established a formal framework, in which fault-tolerant distributed algorithms can be represented, and that caters for different variants (benign or malicious faults, coordinated and uncoordinated algorithms). We have formally proved a reduction theorem that relates fine-grained (asynchronous) interleaving executions and coarse-grained executions, in which an entire round constitutes the unit of atomicity. As a corollary, many correctness properties, including Consensus, can be transferred from the coarse-grained to the fine-grained representation.

We have applied this framework to give formal proofs in Isabelle/HOL for six different Consensus algorithms known from the literature. Thanks to the reduction theorem, it is enough to verify the algorithms over coarse-grained runs, and this keeps the effort manageable. For example, our LastVoting algorithm is similar to the DiskPaxos algorithm verified in [10], but our proof here is an order of magnitude shorter, although we prove safety and liveness properties, whereas only safety was considered in [10].

We also emphasize that the uniform characterization of fault assumptions via communication predicates in the HO model lets us consider the effects of transient failures, contrary to standard models that consider only permanent failures. For example, our correctness proof for the EIGByz algorithm
establishes a stronger result than that claimed by the designers of the algorithm. The uniform presentation also paves the way towards comparing assumptions of different algorithms.

The encoding of the HO model as Isabelle/HOL theories is quite straightforward, and we find our Isar proofs quite readable, although they necessarily contain the full details that are often glossed over in textbook presentations. We believe that our framework allows algorithm designers to study different fault-tolerant distributed algorithms, their assumptions, and their proofs, in a clear, rigorous and uniform way.

References


