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

Henri Debrat[1] and Stephan Merz[2]
[1] Université de Lorraine & LORIA
[2] Inria Nancy Grand-Est & LORIA
Villers-lès-Nancy, France

April 19, 2020

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

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.

A preliminary report on the formalization of the *LastVoting* algorithm in the HO model appeared in [6]. The paper [4] contains a paper-and-pencil proof of the reduction theorem relating coarse-grained and fine-grained executions, and [5] reports on the formal verification of the $\mathcal{U}_{T,E,\alpha}$, $\mathcal{A}_{T,E,\alpha}$, and *EIGByz*$_f$ algorithms.

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 ω-sequences of configurations (vectors of process states). Hence, a run is modeled as a function of type $nat \Rightarrow {}'proc \Rightarrow {}'pst$ where type variables $'proc$ and $'pst$ 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 $dec{::}'pst \Rightarrow val\ option$ 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
  $('proc,'pst)\ run = nat \Rightarrow 'proc \Rightarrow 'pst$

definition
  $consensus :: ('proc \Rightarrow 'val) \Rightarrow ('pst \Rightarrow 'val\ option) \Rightarrow ('proc,'pst)\ run \Rightarrow bool$
where
  $consensus\ vals\ dec\ rho \equiv$
    $(\forall n\ p\ v.\ dec\ (rho\ n\ p) = Some\ v \longrightarrow v \in range\ vals)$
  $\wedge (\forall m\ n\ p\ q\ v\ w.\ dec\ (rho\ m\ p) = Some\ v \wedge dec\ (rho\ n\ q) = Some\ w$
      $\longrightarrow v = w)$
  $\wedge (\forall p.\ \exists n.\ dec\ (rho\ n\ p) \neq None)$

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 *weak-consensus* **where**
  $weak\text{-}consensus\ vals\ dec\ rho \equiv$
    $(\forall v.\ (\forall p.\ vals\ p = v) \longrightarrow (\forall n\ p\ w.\ dec\ (rho\ n\ p) = Some\ w \longrightarrow w = v))$
  $\wedge (\forall m\ n\ p\ q\ v\ w.\ dec\ (rho\ m\ p) = Some\ v \wedge dec\ (rho\ n\ q) = Some\ w$
      $\longrightarrow v = w)$
  $\wedge (\forall p.\ \exists n.\ dec\ (rho\ n\ p) \neq None)$

Clearly, *consensus* implies *weak-consensus*.

lemma *consensus-then-weak-consensus*:
  assumes *consensus vals dec rho*
  shows *weak-consensus vals dec rho*
  using *assms* **by** (*auto simp*: *consensus-def weak-consensus-def image-def*)

Over Boolean values ("binary Consensus"), *weak-consensus* implies *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., $card\ (range\ vals) \leq 2$).

lemma *binary-weak-consensus-then-consensus*:
  assumes *bc*: $weak\text{-}consensus\ (vals{::}'proc \Rightarrow bool)\ dec\ rho$
  shows *consensus vals dec rho*
proof −
  { — Show the Integrity property, the other conjuncts are the same.
    fix *n p v*

assume *dec*: *dec* (*rho n p*) = *Some v*
have $v \in$ *range vals*
proof (*cases* $\exists\, w.\ \forall\, p.$ *vals p* = *w*)
case *True*
then obtain *w* **where** *w*: $\forall\, p.$ *vals p* = *w* **..**
with *bc* **have** *dec* (*rho n p*) $\in$ {*Some w*, *None*} **by** (*auto simp*: *weak-consensus-def*)
with *dec w* **show** *?thesis* **by** (*auto simp*: *image-def*)
next
case *False*
— In this case both possible values occur in *vals*, and the result is trivial.
thus *?thesis* **by** (*auto simp*: *image-def*)
qed
} **note** *integrity* = *this*
from *bc* **show** *?thesis*
unfolding *consensus-def weak-consensus-def* **by** (*auto elim*!: *integrity*)
qed

The algorithms that we are going to verify solve the Consensus or weak Consensus problem, under different hypotheses about the kinds and number of faults.

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" $HO(p, r)$ 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 $HO(p, r)$ 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 $HO(p, r)$ 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 $SHO(p, r) \subseteq HO(p, r)$ 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 $HO(p, r) \setminus SHO(p, r)$ 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.

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*

For non-coordinated HO algorithms, the coordinator argument of functions *CinitState* and *CnextState* is irrelevant, and we define utility functions that omit that argument.

definition *isNCAlgorithm* **where**
$$
\begin{aligned}
&\mathit{isNCAlgorithm\ alg} \equiv \\
&\quad (\forall\, p\ st\ crd\ crd'.\ \mathit{CinitState\ alg\ p\ st\ crd} = \mathit{CinitState\ alg\ p\ st\ crd'}) \\
&\wedge (\forall\, r\ p\ st\ msgs\ crd\ crd'\ st'.\ \mathit{CnextState\ alg\ r\ p\ st\ msgs\ crd\ st'} \\
&\qquad\qquad\qquad = \mathit{CnextState\ alg\ r\ p\ st\ msgs\ crd'\ st'})
\end{aligned}
$$

definition *initState* **where**
$$\mathit{initState\ alg\ p\ st} \equiv \mathit{CinitState\ alg\ p\ st\ undefined}$$

definition *nextState* **where**
$$\mathit{nextState\ alg\ r\ p\ st\ msgs\ st'} \equiv \mathit{CnextState\ alg\ r\ p\ st\ msgs\ undefined\ st'}$$

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

type-synonym
$$\mathit{'proc\ HO} = \mathit{'proc} \Rightarrow \mathit{'proc\ set}$$

type-synonym
$$\mathit{'proc\ coord} = \mathit{'proc} \Rightarrow \mathit{'proc}$$

An execution of an HO algorithm is defined with respect to HO and SHO assignments that indicate, for every round r and every process p, from which sender processes p receives messages (resp., uncorrupted messages) at round 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 *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. This definition of executions may appear surprising for an asynchronous distributed system, but it simplifies system verification, compared to a "fine-grained" execution model that records individual events such as message sending and reception or local transitions. We will justify later why the "coarse-grained" model is sufficient for verifying interesting correctness properties of HO algorithms.

The predicate *CSHOinitConfig* describes the possible initial configurations for algorithm A (remember that a configuration is a function that assigns local states to every process).

definition *CHOinitConfig* **where**
$$\mathit{CHOinitConfig\ A\ cfg\ (coord{::}'proc\ coord)} \equiv \forall\, p.\ \mathit{CinitState\ A\ p\ (cfg\ p)\ (coord\ p)}$$

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* **where**
$SHOmsgVectors\ A\ r\ p\ cfg\ HOp\ SHOp \equiv$
$\{\mu.\ (\forall\, q.\ q \in HOp \longleftrightarrow \mu\ q \neq None)$
$\wedge\ (\forall\, q.\ q \in SHOp \cap HOp \longrightarrow \mu\ q = Some\ (sendMsg\ A\ r\ q\ p\ (cfg\ q)))\}$

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* **where**
$CSHOnextConfig\ A\ r\ cfg\ HO\ SHO\ coord\ cfg' \equiv$
$\forall\, p.\ \exists\, \mu \in SHOmsgVectors\ A\ r\ p\ cfg\ (HO\ p)\ (SHO\ p).$
$CnextState\ A\ r\ p\ (cfg\ p)\ \mu\ (coord\ p)\ (cfg'\ p)$

definition *CSHORun* **where**
$CSHORun\ A\ rho\ HOs\ SHOs\ coords \equiv$
$CHOinitConfig\ A\ (rho\ 0)\ (coords\ 0)$
$\wedge\ (\forall\, r.\ CSHOnextConfig\ A\ r\ (rho\ r)\ (HOs\ r)\ (SHOs\ r)\ (coords\ (Suc\ r))$
$(rho\ (Suc\ r)))$

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

lemma *HOinitConfig-eq*:
$HOinitConfig\ A\ cfg = (\forall\, p.\ initState\ A\ p\ (cfg\ p))$
by (*auto simp*: *HOinitConfig-def CHOinitConfig-def initState-def*)

definition *SHOnextConfig* **where**
$SHOnextConfig\ A\ r\ cfg\ HO\ SHO\ cfg' \equiv$
$CSHOnextConfig\ A\ r\ cfg\ HO\ SHO\ (\lambda q.\ undefined)\ cfg'$

lemma *SHOnextConfig-eq*:
$SHOnextConfig\ A\ r\ cfg\ HO\ SHO\ cfg' =$
$(\forall\, p.\ \exists\, \mu \in SHOmsgVectors\ A\ r\ p\ cfg\ (HO\ p)\ (SHO\ p).$
$nextState\ A\ r\ p\ (cfg\ p)\ \mu\ (cfg'\ p))$
by (*auto simp*: *SHOnextConfig-def CSHOnextConfig-def SHOmsgVectors-def nextState-def*)

definition *SHORun* **where**
SHORun A rho HOs SHOs ≡
CSHORun A rho HOs SHOs (*λr q. undefined*)

lemma *SHORun-eq*:
SHORun A rho HOs SHOs =
(*HOinitConfig A* (*rho 0*)
∧ (∀ *r. SHOnextConfig A r* (*rho r*) (*HOs r*) (*SHOs r*) (*rho* (*Suc r*))))
by (*auto simp*: *SHORun-def CSHORun-def HOinitConfig-def 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 *HOrcvdMsgs* **where**
HOrcvdMsgs A r p HO cfg ≡
λq. if q ∈ *HO then Some* (*sendMsg A r q p* (*cfg q*)) *else None*

lemma *SHOmsgVectors-HO*:
SHOmsgVectors A r p cfg HO HO = {*HOrcvdMsgs A r p HO cfg*}
unfolding *SHOmsgVectors-def HOrcvdMsgs-def* **by** *auto*

With coordinators

definition *CHOnextConfig* **where**
CHOnextConfig A r cfg HO coord cfg′ ≡
CSHOnextConfig A r cfg HO HO coord cfg′

lemma *CHOnextConfig-eq*:
CHOnextConfig A r cfg HO coord cfg′ =
(∀ *p. CnextState A r p* (*cfg p*) (*HOrcvdMsgs A r p* (*HO p*) *cfg*)
(*coord p*) (*cfg′ p*))
by (*auto simp*: *CHOnextConfig-def CSHOnextConfig-def SHOmsgVectors-HO*)

definition *CHORun* **where**
CHORun A rho HOs coords ≡ *CSHORun A rho HOs HOs coords*

lemma *CHORun-eq*:
CHORun A rho HOs coords =
(*CHOinitConfig A* (*rho 0*) (*coords 0*)
∧ (∀ *r. CHOnextConfig A r* (*rho r*) (*HOs r*) (*coords* (*Suc r*)) (*rho* (*Suc r*))))
by (*auto simp*: *CHORun-def CSHORun-def CHOinitConfig-def CHOnextConfig-def*)

Without coordinators

definition *HOnextConfig* **where**

HOnextConfig A r cfg HO cfg' ≡ *SHOnextConfig A r cfg HO HO cfg'*

lemma *HOnextConfig-eq*:
HOnextConfig A r cfg HO cfg' =
(∀ *p*. *nextState A r p* (*cfg p*) (*HOrcvdMsgs A r p* (*HO p*) *cfg*) (*cfg' p*))
by (*auto simp*: *HOnextConfig-def SHOnextConfig-eq SHOmsgVectors-HO*)

definition *HORun* **where**
HORun A rho HOs ≡ *SHORun A rho HOs HOs*

lemma *HORun-eq*:
HORun A rho HOs =
(*HOinitConfig A* (*rho 0*)
∧ (∀ *r*. *HOnextConfig A r* (*rho r*) (*HOs 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 ⋀*cfg*. *CHOinitConfig A cfg* (*coords 0*) ⟹ *P cfg*
shows *P* (*rho 0*)
using *assms* **unfolding** *CHORun-eq* **by** *blast*

lemma *CHORun-Suc*:
assumes *CHORun A rho HOs coords*
and ⋀*r*. *CHOnextConfig A r* (*rho r*) (*HOs r*) (*coords* (*Suc r*)) (*rho* (*Suc r*))
⟹ *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*) ⟹ *P 0*
and *step*: ⋀*r*. ⟦ *P r*; *CHOnextConfig A r* (*rho r*) (*HOs r*) (*coords* (*Suc r*))
(*rho* (*Suc r*)) ⟧ ⟹ *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 *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* +

$HOcommPerRd :: 'proc\ HO \Rightarrow bool$
$HOcommGlobal :: (nat \Rightarrow 'proc\ HO) \Rightarrow bool$

record $('proc, 'pst, 'msg)\ CHOMachine = ('proc, 'pst, 'msg)\ CHOAlgorithm\ +$
$CHOcommPerRd :: nat \Rightarrow 'proc\ HO \Rightarrow 'proc\ coord \Rightarrow bool$
$CHOcommGlobal :: (nat \Rightarrow 'proc\ HO) \Rightarrow (nat \Rightarrow 'proc\ coord) \Rightarrow bool$

record $('proc, 'pst, 'msg)\ SHOMachine = ('proc, 'pst, 'msg)\ CHOAlgorithm\ +$
$SHOcommPerRd :: ('proc\ HO) \Rightarrow ('proc\ HO) \Rightarrow bool$
$SHOcommGlobal :: (nat \Rightarrow 'proc\ HO) \Rightarrow (nat \Rightarrow 'proc\ HO) \Rightarrow bool$

record $('proc, 'pst, 'msg)\ CSHOMachine = ('proc, 'pst, 'msg)\ CHOAlgorithm\ +$
$CSHOcommPerRd :: ('proc\ HO) \Rightarrow ('proc\ HO) \Rightarrow 'proc\ coord \Rightarrow bool$
$CSHOcommGlobal :: (nat \Rightarrow 'proc\ HO) \Rightarrow (nat \Rightarrow 'proc\ HO)$
$\Rightarrow (nat \Rightarrow 'proc\ coord) \Rightarrow bool$

end — theory HOModel
theory *Reduction*
imports *HOModel Stuttering-Equivalence.StutterEquivalence*
begin

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 easier 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 ω-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.

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*

In an initial configuration for an algorithm, the local state of every process satisfies the algorithm's initial-state predicate, and all other components have obvious default values.

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

$$
\begin{aligned}
&\quad \mathit{round\ config} = (\lambda p.\ 0)\\
&\wedge (\forall p.\ \mathit{CinitState\ A\ p\ (state\ config\ p)\ (coord\ p)})\\
&\wedge \mathit{sent\ config} = (\lambda p.\ \{\})\\
&\wedge \mathit{rcvd\ config} = (\lambda p\ q.\ \mathit{None})\\
&\wedge \mathit{network\ config} = \{\}
\end{aligned}
$$

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 *sendMsg* function. The effect of the transition is to add q to the *sent* component of the configuration and the message quadruple to the *network* component.

definition *fg-send-msg* **where**

$$
\begin{aligned}
&\mathit{fg\text{-}send\text{-}msg\ A\ p\ q\ config\ config'} \equiv\\
&\quad q \notin (\mathit{sent\ config\ p})\\
&\wedge \mathit{config'} = \mathit{config}\ (\!|\\
&\qquad \mathit{sent} := (\mathit{sent\ config})(p := (\mathit{sent\ config\ p}) \cup \{q\}),\\
&\qquad \mathit{network} := \mathit{network\ config} \cup\\
&\qquad\qquad \{(p,\ \mathit{round\ config\ p},\ q,\\
&\qquad\qquad\quad \mathit{sendMsg\ A\ (round\ config\ p)\ p\ q\ (state\ config\ p)})\}\ |\!)
\end{aligned}
$$

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 *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 SHO set (parameter *SHO*), then an arbitrary value m' is received instead of m.

definition *fg-rcv-msg* **where**

$$
\begin{aligned}
&\mathit{fg\text{-}rcv\text{-}msg\ p\ q\ HO\ SHO\ config\ config'} \equiv\\
&\exists\, m\ m'.\ (q,\ (\mathit{round\ config\ p}),\ p,\ m) \in \mathit{network\ config}\\
&\quad \wedge q \in \mathit{HO}\\
&\quad \wedge \mathit{config'} = \mathit{config}\ (\!|\\
&\qquad \mathit{rcvd} := (\mathit{rcvd\ config})(p := (\mathit{rcvd\ config\ p})(q :=\\
&\qquad\qquad \mathit{if}\ q \in \mathit{SHO}\ \mathit{then\ Some\ m\ else\ Some\ m'})),\\
&\qquad \mathit{network} := \mathit{network\ config} - \{(q,\ (\mathit{round\ config\ p}),\ p,\ m)\}\ |\!)
\end{aligned}
$$

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 *CnextState* 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**
$$
\begin{array}{l}
\quad \textit{fg-local } A\ p\ HO\ crd\ config\ config' \equiv \\
\qquad \textit{sent config } p = UNIV \\
\quad \wedge\ \textit{dom } (\textit{rcvd config } p) = HO \\
\quad \wedge\ (\exists\, s.\ \textit{CnextState } A\ (\textit{round config } p)\ p\ (\textit{state config } p)\ (\textit{rcvd config } p)\ crd\ s \\
\qquad\quad \wedge\ config' = config\ (\!| \\
\qquad\qquad\quad round := (\textit{round config})(p := Suc\ (\textit{round config } p)), \\
\qquad\qquad\quad state := (\textit{state config})(p := s), \\
\qquad\qquad\quad sent := (\textit{sent config})(p := \{\}), \\
\qquad\qquad\quad rcvd := (\textit{rcvd config})(p := \lambda q.\ None)\ |\!))
\end{array}
$$

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

definition *fg-next-config* **where**
$$
\begin{array}{l}
\quad \textit{fg-next-config } A\ p\ HO\ SHO\ crd\ config\ config' \equiv \\
\qquad (\exists\, q.\ \textit{fg-send-msg } A\ p\ q\ config\ config') \\
\quad \vee\ (\exists\, q.\ \textit{fg-rcv-msg } p\ q\ HO\ SHO\ config\ config') \\
\quad \vee\ \textit{fg-local } A\ p\ HO\ crd\ config\ config'
\end{array}
$$

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**
$$
\begin{array}{l}
\quad \textit{fg-run } A\ rho\ HOs\ SHOs\ coords \equiv \\
\qquad \textit{fg-init-config } A\ (rho\ 0)\ (coords\ 0) \\
\quad \wedge\ (\forall\, i.\ \exists\, p.\ \textit{fg-next-config } A\ p \\
\qquad\qquad\qquad\qquad (HOs\ (round\ (rho\ i)\ p)\ p) \\
\qquad\qquad\qquad\qquad (SHOs\ (round\ (rho\ i)\ p)\ p) \\
\qquad\qquad\qquad\qquad (coords\ (round\ (rho\ (Suc\ i))\ p)\ p) \\
\qquad\qquad\qquad\qquad (rho\ i)\ (rho\ (Suc\ i))) \\
\quad \wedge\ (\forall\, p\ r.\ \exists\, n.\ round\ (rho\ n)\ p = r)
\end{array}
$$

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**
$$\quad \textit{fg-start-round } rho\ p\ r \equiv LEAST\ (n{::}nat).\ round\ (rho\ n)\ p = r$$

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** ∃ *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* ≤ *m*
  shows *round* (*rho n*) *p* ≤ *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* ≤ *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** ∃ *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* ≤ *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 −
  {
    fix *k i*
    have *round* (*rho* (*k*+*i*)) *p* = *round* (*rho k*) *p*
        ⟹ *state* (*rho* (*k*+*i*)) *p* = *state* (*rho k*) *p*
      (**is** *?R i* ⟹ *?S i*)
    proof (*induct i*)
      show *?S 0* **by** *simp*
    next
      fix *j*
      assume *ih*: *?R j* ⟹ *?S j*
        and *r*: *round* (*rho* (*k* + *Suc j*)) *p* = *round* (*rho k*) *p*
      from *rho* **have** *1*: *round* (*rho k*) *p* ≤ *round* (*rho* (*k*+*j*)) *p*
        by (*auto elim*: *fg-round-numbers-increase*)
      from *rho* **have** *2*: *round* (*rho* (*k*+*j*)) *p* ≤ *round* (*rho* (*k* + *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* (*Suc* (*k*+*j*))) *p* = *round* (*rho* (*k*+*j*)) *p* **by** *simp*
      with *rho* **have** *state* (*rho* (*Suc* (*k*+*j*))) *p* = *state* (*rho* (*k*+*j*)) *p*
        by (*auto elim*: *fg-state-change*)
      with *3 ih* **show** *?S* (*Suc j*) **by** *simp*
    qed
  }
  note *aux* = *this*
  show *?thesis*
  proof (*cases n* ≤ *m*)
    case *True*
    then obtain *k* **where** *m* = *n*+*k* **by** (*auto simp*: *le-iff-add*)
    with *rd* **show** *?thesis* **by** (*auto simp*: *aux*)
  next
    case *False*
    hence *m* ≤ *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
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 *fg-run A rho HOs SHOs coords*
shows *round* (*rho* (*fg-start-round rho p r*)) *p* = *r*
using *assms* **by** (*auto simp*: *fg-run-def fg-start-round-def intro*: *LeastI-ex*)

lemma *fg-start-round-smallest*:
assumes *round* (*rho k*) *p* = *r*
shows *fg-start-round rho p r* ≤ (*k*::*nat*)
using *assms* **unfolding** *fg-start-round-def* **by** (*rule Least-le*)

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

lemma *fg-start-round-0*:
assumes *rho*: *fg-run A rho HOs SHOs coords*
shows *fg-start-round rho p 0* = *0*
proof −
from *rho* **have** *round* (*rho 0*) *p* = *0* **by** (*auto simp*: *fg-run-def fg-init-config-def*)
hence *fg-start-round rho p 0* ≤ *0* **by** (*rule fg-start-round-smallest*)
thus *?thesis* **by** *simp*
qed

lemma *fg-start-round-strict-mono*:
assumes *rho*: *fg-run A rho HOs SHOs coords*
shows *strict-mono* (*fg-start-round rho p*)
proof
fix *r r'*
assume *r*: (*r*::*nat*) < *r'*
from *rho* **have** *round* (*rho* (*fg-start-round rho p r*)) *p* = *r* **by** (*rule fg-start-round*)
from *rho this r* **show** *fg-start-round rho p r* < *fg-start-round rho p r'*
by (*rule 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 *fg-same-round-same-state*, this implies that the local state of process p is the same at all these configurations.

lemma *fg-round-between-start-rounds*:
assumes *rho*: *fg-run A rho HOs SHOs coords*
and *1*: *fg-start-round rho p r* ≤ *n*
and *2*: *n* < *fg-start-round rho p* (*Suc r*)

shows *round* (*rho n*) *p* = *r* (**is** *?rd* = *r*)
proof (*rule antisym*)
  from *1* **have** *round* (*rho* (*fg-start-round rho p r*)) *p* ≤ *?rd*
    by (*rule fg-round-numbers-increase*[*OF rho*])
  thus *r* ≤ *?rd* **by** (*simp add*: *fg-start-round*[*OF rho*])
next
  show *?rd* ≤ *r*
  proof (*rule ccontr*)
    assume ¬ *?thesis*
    hence *Suc r* ≤ *?rd* **by** *simp*
    hence *fg-start-round rho p* (*Suc r*) ≤ *fg-start-round rho p ?rd*
      by (*rule rho*[*THEN fg-start-round-strict-mono*, *THEN strict-mono-mono*,
            THEN monoD])
    also have ... ≤ *n* **by** (*auto intro*: *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 *fg-local-transition-from-round*:
assumes *rho*: *fg-run A rho HOs SHOs coords*
obtains *n* **where** *round* (*rho n*) *p* = *r*
    and *fg-start-round rho p* (*Suc r*) = *Suc n*
    and *fg-local A p* (*HOs r p*) (*coords* (*Suc r*) *p*) (*rho n*) (*rho* (*Suc n*))
proof −
  have *fg-start-round rho p* (*Suc r*) ≠ *0* (**is** *?start* ≠ *0*)
  proof
    assume *contr*: *?start* = *0*
    from *rho* **have** *round* (*rho ?start*) *p* = *Suc r* **by** (*rule fg-start-round*)
    with *contr rho* **show** *False* **by** (*auto simp*: *fg-run-def fg-init-config-def*)
  qed
  then obtain *n* **where** *n*: *?start* = *Suc n* **by** (*auto simp*: *gr0-conv-Suc*)
  with *fg-start-round*[*OF rho*, *of p Suc r*]
  have *0*: *round* (*rho* (*Suc n*)) *p* = *Suc r* **by** *simp*
  have *1*: *round* (*rho n*) *p* = *r*
  proof (*rule fg-round-between-start-rounds*[*OF rho*])
    have *fg-start-round rho p r* < *fg-start-round rho p* (*Suc r*)
      by (*rule fg-start-round-strict-mono*[*OF rho*, *THEN strict-monoD*]) *simp*
    with *n* **show** *fg-start-round rho p r* ≤ *n* **by** *simp*
  next
    from *n* **show** *n* < *?start* **by** *simp*
  qed
  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*)
hence *fg-local A p* (*HOs r p*) (*coords* (*Suc r*) *p*) (*rho n*) (*rho* (*Suc n*))
proof (*auto simp*: *fg-next-config-def*)
fix *q*
assume *fg-send-msg A p' q ?cfg ?cfg'*
— impossible because round changes
with *0 1* **show** *?thesis* **by** (*auto simp*: *fg-send-msg-def*)
next
fix *q*
assume *fg-rcv-msg p' q ?HO ?SHO ?cfg ?cfg'*
— impossible because round changes
with *0 1* **show** *?thesis* **by** (*auto simp*: *fg-rcv-msg-def*)
next
assume *fg-local A p' ?HO ?crd ?cfg ?cfg'*
with *0 1* **show** *?thesis* **by** (*cases p' = p*) (*auto simp*: *fg-local-def*)
qed
with *1 n that* **show** *?thesis* **by** *auto*
qed

We now prove two invariants asserted in [4]. The first one states that any message m in transit from process p to process q for round r corresponds to the message computed by p for q, given p's state at its rth local transition.

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'*
assume *send*: *fg-send-msg A p' q' ?cfg ?cfg'*
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'
    assume *fg-rcv-msg* p' q' *?HO ?SHO ?cfg* $?cfg'$
    with m' **have** *?msg n* **by** (*auto simp*: *fg-rcv-msg-def*)
    with *ih* **show** *?thesis* **.**
  next
    assume *fg-local A* p' *?HO ?crd ?cfg* $?cfg'$
    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 $p \in$ *HOs* (*round* (*rho n*) *q*) *q*
  (**is** $p \in$ *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* $\Longrightarrow$ *?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-rcv-msg* steps of process q, the proof is immediately reduced to the induction hypothesis.

fix q'
assume *rcvmsg*: *fg-rcv-msg* p' q' *?HO ?SHO ?cfg ?cfg'*
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 rd* **show** *?thesis* **by** *simp*
next
case *False*
with *rcvd rcvmsg rd* **show** *?thesis* **by** (*auto simp*: *fg-rcv-msg-def*)
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 *ih* **show** *?P* (*Suc n*) **by** *simp*
next
assume *fg-local A* p' *?HO ?crd ?cfg ?cfg'*
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
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'*
assume *rcvmsg*: *fg-rcv-msg p' q' ?HO ?SHO ?cfg ?cfg'*
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
assume *fg-local A p' ?HO ?crd ?cfg ?cfg'*
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
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 where
  coarse-run rho r p ≡ 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
      (is CSHORun - ?cr - - -)
proof (auto simp: CSHORun-def)
  from rho show CHOinitConfig A (?cr 0) (coords 0)
    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[THEN fg-local-transition-from-round] obtain n
      where n: round (rho n) p = r
        and start: fg-start-round rho p (Suc r) = Suc n (is ?start = -)
        and loc: fg-local A p (HOs r p) (coords (Suc r) p) (rho n) (rho (Suc n))
            (is fg-local - - ?HO ?crd ?cfg ?cfg')
      by blast
    have cfg: ?cr r p = state ?cfg p
      unfolding coarse-run-def proof (rule fg-same-round-same-state[OF rho])
      from n show round (rho (fg-start-round rho p r)) p = round ?cfg p
        by (simp add: fg-start-round[OF rho])
    qed
    from start have cfg': ?cr (Suc r) p = state ?cfg' p
      by (simp add: coarse-run-def)
    have rcvd: rcvd ?cfg p ∈ SHOmsgVectors A r p (?cr r) ?HO (SHOs r p)
    proof (auto simp: SHOmsgVectors-def)
      fix q
      assume q ∈ ?HO
      with n loc show ∃m. rcvd ?cfg p q = Some m by (auto simp: fg-local-def)
    next
      fix q m
      assume rcvd ?cfg p q = Some m
      with rho n show q ∈ ?HO by (auto simp: fg-invariant2a)
    next
```

fix q
assume *sho*: $q \in$ *SHOs r p* **and** *ho*: $q \in$ *?HO*
from *ho n loc* **obtain** m **where** *rcvd ?cfg p q* = *Some m*
by (*auto simp*: *fg-local-def*)
with *rho n sho* **show** *rcvd ?cfg p q* = *Some* (*sendMsg A r q p* (*?cr r q*))
by (*auto simp*: *fg-invariant2b coarse-run-def*)
qed
with *n loc cfg cfg'*
show $\exists\, \mu \in$ *SHOmsgVectors A r p* (*?cr r*) *?HO* (*SHOs r p*).
CnextState A r p (*?cr r p*) μ *?crd* (*?cr* (*Suc r*) *p*)
by (*auto simp*: *fg-local-def*)
qed
qed

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**
locally-similar (σ::*nat* $\Rightarrow$ *'proc* $\Rightarrow$ *'pst*) $\tau \equiv$
$\forall\, p$::*'proc*. ($\lambda n.\ \sigma\ n\ p$) $\approx$ ($\lambda n.\ \tau\ n\ p$)

definition *local-property* **where**
local-property $P \equiv$
$\forall\, \sigma\ \tau$. *locally-similar* $\sigma\ \tau \longrightarrow P\ \sigma \longrightarrow P\ \tau$

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* $\sigma\ \tau \Longrightarrow$ *locally-similar* $\tau\ \sigma$
by (*simp add*: *locally-similar-def stutter-equiv-sym*)

lemma *locally-similar-trans* [*trans*]:
locally-similar $\varrho\ \sigma \Longrightarrow$ *locally-similar* $\sigma\ \tau \Longrightarrow$ *locally-similar* $\varrho\ \tau$
by (*force simp add*: *locally-similar-def elim*: *stutter-equiv-trans*)

lemma *local-property-eq*:
local-property P = ($\forall\, \sigma\ \tau$. *locally-similar* $\sigma\ \tau \longrightarrow P\ \sigma = P\ \tau$)
by (*auto simp*: *local-property-def dest*: *locally-similar-sym*)

Consider any fine-grained run *rho*. The projection of *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* ∘ *rho*) (*coarse-run rho*)
proof (*auto simp*: *locally-similar-def*)
  fix *p*
  show (λn. *state* (*rho n*) *p*) ≈ (λn. *coarse-run rho n p*) (**is** *?fgr* ≈ *?cgr*)
  proof (*rule stutter-equivI*)
    show *stutter-sampler* (*fg-start-round rho p*) *?fgr*
    proof (*auto simp*: *stutter-sampler-def*)
      from *rho* **show** *fg-start-round rho p 0* = *0*
        by (*rule fg-start-round-0*)
    next
      show *strict-mono* (*fg-start-round rho p*)
        by (*rule fg-start-round-strict-mono*[*OF rho*])
    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 p 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 p r*)) *p*
        by (*rule fg-same-round-same-state*)
    qed
  next
    show *stutter-sampler id ?cgr*
      by (*rule id-stutter-sampler*)
  next
    show *?fgr* ∘ *fg-start-round rho p* = *?cgr* ∘ *id*
      by (*auto simp*: *coarse-run-def*)
  qed
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*:
      $\bigwedge$ *crho*. ⟦ *CSHORun A crho HOs SHOs coords*; *crho 0* = *state* (*rho 0*)⟧
        ⟹ *P crho*
  shows *P* (*state* ∘ *rho*)
proof −
  have *coarse-run rho 0* = *state* (*rho 0*)
    by (*rule ext*, *simp add*: *coarse-run-def fg-start-round-0*[*OF rho*])

from *rho*[*THEN reduction*] *this*
have P (*coarse-run rho*) **by** (*rule coarse-correct*)
with *coarse-run-locally-similar*[*OF rho*] P
show *?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 *local-property-reduction*.

lemma *integrity-is-local*:
assumes *sim*: *locally-similar* σ τ
and *val*: $\bigwedge n.\ dec\ (\sigma\ n\ p) = Some\ v \Longrightarrow v \in range\ vals$
and *dec*: $dec\ (\tau\ n\ p) = Some\ v$
shows $v \in range\ vals$
proof –
from *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 *sym*[*OF this*] *dec* **show** *?thesis* **by** (*auto elim*: *val*)
qed

lemma *validity-is-local*:
assumes *sim*: *locally-similar* σ τ
and *val*: $\bigwedge n.\ dec\ (\sigma\ n\ p) = Some\ w \Longrightarrow w = v$
and *dec*: $dec\ (\tau\ n\ p) = Some\ w$
shows $w = v$
proof –
from *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 *sym*[*OF this*] *dec* **show** *?thesis* **by** (*auto elim*: *val*)
qed

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

lemma *termination-is-local*:
assumes *sim*: *locally-similar* σ τ
and *trm*: $dec\ (\sigma\ m\ p) = Some\ v$
shows $\exists\, n.\ dec\ (\tau\ n\ p) = Some\ v$

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 $\forall n\ p\ v.\ dec\ (\sigma\ n\ p) = Some\ v \longrightarrow v \in range\ vals$
  and $dec\ (\tau\ n\ p) = Some\ v$
  thus $v \in range\ vals$ **by** (*blast intro*: *integrity-is-local*)
next
  fix $\sigma\ \tau\ m\ n\ p\ q\ v\ w$
  assume *locally-similar* $\sigma\ \tau$
  and $\forall m\ n\ p\ q\ v\ w.\ dec\ (\sigma\ m\ p) = Some\ v \wedge dec\ (\sigma\ n\ q) = Some\ w \longrightarrow v = w$
  and $dec\ (\tau\ m\ p) = Some\ v$ **and** $dec\ (\tau\ n\ q) = Some\ w$
  thus $v = w$ **by** (*blast intro*: *agreement-is-local*)
next
  fix $\sigma\ \tau\ p$
  assume *locally-similar* $\sigma\ \tau$
  and $\forall p.\ \exists m\ v.\ dec\ (\sigma\ m\ p) = Some\ v$
  thus $\exists n\ w.\ dec\ (\tau\ n\ p) = Some\ w$ **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 $\forall n\ p\ w.\ dec\ (\sigma\ n\ p) = Some\ w \longrightarrow w = v$
  and $dec\ (\tau\ n\ p) = Some\ w$
  thus $w = v$ **by** (*blast intro*: *validity-is-local*)
next
  fix $\sigma\ \tau\ m\ n\ p\ q\ v\ w$
  assume *locally-similar* $\sigma\ \tau$
  and $\forall m\ n\ p\ q\ v\ w.\ dec\ (\sigma\ m\ p) = Some\ v \wedge dec\ (\sigma\ n\ q) = Some\ w \longrightarrow v = w$
  and $dec\ (\tau\ m\ p) = Some\ v$ **and** w: $dec\ (\tau\ n\ q) = Some\ w$
  thus $v = w$ **by** (*blast intro*: *agreement-is-local*)
next
  fix $\sigma\ \tau\ p$
  assume *locally-similar* $\sigma\ \tau$
  and $\forall p.\ \exists m\ v.\ dec\ (\sigma\ m\ p) = Some\ v$
  thus $\exists n\ w.\ dec\ (\tau\ n\ p) = Some\ w$ **by** (*blast dest*: *termination-is-local*)
qed

end
theory *Majorities*

imports *Main*
begin

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 crd: card E < card S + card T
        and s: S ⊆ E and t: T ⊆ E and e: finite E
    shows S ∩ T ≠ {}
proof (clarify)
  assume contra: S ∩ T = {}
  from s t e have finite S and finite T by (auto simp: finite-subset)
  with crd contra have card E < card (S ∪ T) by (auto simp add: card-Un-Int)
  moreover
  from s t e have card (S ∪ T) ≤ card E by (simp add: card-mono)
  ultimately
  show False by simp
qed

lemma abs-majoritiesE:
  assumes crd: card E < card S + card T
      and s: S ⊆ E and t: T ⊆ E and e: finite E
  obtains p where p ∈ S and p ∈ T
proof −
  from assms have S ∩ T ≠ {} by (rule abs-majorities-intersect)
  then obtain p where p ∈ S ∩ T by blast
  with that show ?thesis by auto
qed
```

Special case: both sets S and T are majorities.

```
lemma abs-majoritiesE':
  assumes Smaj: card S > (card E) div 2 and Tmaj: card T > (card E) div 2
      and s: S ⊆ E and t: T ⊆ E and e: finite E
  obtains p where p ∈ S and p ∈ T
proof (rule abs-majoritiesE[OF - s t e])
  from Smaj Tmaj show card E < card S + 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 majorities-intersect:
```

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

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

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

end
theory *OneThirdRuleDefs*
imports ../*HOModel*
begin

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
$N \equiv$ *card* (*UNIV*::*Proc set*)

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

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 *HO* **obtain** *q* **where** *q*: $q \in$ *HOs m p*
  by *auto*
with *v* **have** *HOV ?msgs* (*the* (*?msgs q*)) $\neq$ {}
  by (*auto simp*: *HOV-def HOrcvdMsgs-def*)
hence *HOp*: *0* < *card* (*HOV ?msgs* (*the* (*?msgs q*)))
  by *auto*
also from *v* **have** ... $\leq$ *card* (*HOV ?msgs v*)
  by (*simp add*: *MFR-def*)
finally have *HOV ?msgs v* $\neq$ {}
  by *auto*
thus *?thesis*
  by (*auto simp*: *HOV-def HOrcvdMsgs-def*)
qed

TwoThirds msgs v holds if value v has been received from more than 2/3 of all processes.

definition *TwoThirds* **where**
  TwoThirds msgs v $\equiv$ (*2*$*$*N*) *div 3* < *card* (*HOV msgs v*)

The next-state relation of algorithm *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 *Some* is the constructor of the option datatype, whereas ϵ 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.

definition *OTR-nextState* **where**
  OTR-nextState r p (*st*::(*'val*::*linorder*) *pstate*) *msgs st'* $\equiv$
   if (*2*$*$*N*) *div 3* < *card* {*q*. *msgs q* $\neq$ *None*}
   then st' = (| *x* = *Min* {*v* . *MFR msgs v*},
         decide = (*if* ($\exists$ *v*. *TwoThirds msgs v*)
                 then Some (ϵ *v*. *TwoThirds msgs v*)
                 else decide st) |)
   else st' = *st*

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.

definition *OTR-sendMsg* **where**
  OTR-sendMsg r p q st $\equiv$ *x st*

5.2 Communication Predicate for *One-Third Rule*

We now define the communication predicate for the *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 Π of processes where Π

contains more than two thirds of all processes. The "per-round" part of the communication predicate is trivial.

definition *OTR-commPerRd* **where**
OTR-commPerRd HOrs $\equiv$ *True*

definition *OTR-commGlobal* **where**
OTR-commGlobal HOs $\equiv$
$\forall r.\ \exists r0\ \Pi.\ r0 \geq r \wedge (\forall p.\ HOs\ r0\ p = \Pi) \wedge card\ \Pi > (2*N)\ div\ 3$

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 *crd* arguments of the initial- and next-state predicates are unused.

definition *OTR-HOMachine* **where**
OTR-HOMachine =
(| *CinitState* = (λ *p st crd*. *OTR-initState p st*),
sendMsg = *OTR-sendMsg*,
CnextState = (λ *r p st msgs crd st'*. *OTR-nextState r p st msgs st'*),
HOcommPerRd = *OTR-commPerRd*,
HOcommGlobal = *OTR-commGlobal* |)

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 *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 *VInv* **where**
VInv rho n $\equiv$
let xinit = (*range* ($x \circ$ (*rho* 0)))
in range ($x \circ$ (*rho n*)) $\subseteq$ *xinit*
$\wedge$ *range* (*decide* $\circ$ (*rho n*)) $\subseteq$ {*None*} $\cup$ (*Some* ` *xinit*)

```
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 0))
  have range (x ∘ (rho (Suc m))) ⊆ ?xinit
  proof (clarsimp cong del: image-cong-simp)
    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
```

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

Integrity is an immediate consequence.

theorem *OTR-integrity*:
assumes *run*:*HORun OTR-M rho HOs* **and** *dec*: *decide* (*rho n p*) = *Some v*
shows $\exists q.$ *v* = *x* (*rho 0 q*)
proof −
let *?xinit* = *range* (*x* $\circ$ (*rho 0*))
from *run* **have** *VInv rho n* **by** (*rule vinv-invariant*)
hence *range* (*decide* $\circ$ (*rho n*)) $\subseteq$ {*None*} $\cup$ (*Some* ‘ *?xinit*)
by (*auto simp*: *VInv-def Let-def*)

hence *decide* ((*rho n*) *p*) ∈ {*None*} ∪ (*Some* ‘ *?xinit*)
  by (*auto simp*: *image-def*)
with *dec* **show** *?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 HORun OTR-M rho HOs
     and ¬ ((2*N) div 3
          < card {q. (HOrcvdMsgs OTR-M n p (HOs n p) (rho n)) q ≠ None})
  shows rho (Suc n) p = rho n p
  using assms
  by (auto simp: HORun-eq HOnextConfig-eq OTR-HOMachine-def
              nextState-def OTR-nextState-def)

lemma nextState-decide:
  assumes run:HORun OTR-M rho HOs
  and chg: decide (rho (Suc n) p) ≠ decide (rho n p)
  shows TwoThirds (HOrcvdMsgs OTR-M n p (HOs n p) (rho n))
                (the (decide (rho (Suc n) p)))
proof −
  from run
  have OTR-nextState n p (rho n p)
                 (HOrcvdMsgs OTR-M n p (HOs n p) (rho n)) (rho (Suc n) p)
   by (simp add: HORun-eq HOnextConfig-eq OTR-HOMachine-def nextState-def)
  with chg show ?thesis by (auto simp: OTR-nextState-def elim: someI)
qed

lemma A1:
  assumes run:HORun OTR-M rho HOs
  and dec: decide (rho (Suc n) p) = Some v
  and chg: decide (rho (Suc n) p) ≠ decide (rho n p) (is decide ?st' ≠ decide ?st)
  shows (2*N) div 3 < card { q . x (rho n q) = v }
proof −
  from run chg
  have TwoThirds (HOrcvdMsgs OTR-M n p (HOs n p) (rho n))
                (the (decide ?st'))
    (is TwoThirds ?msgs -)
    by (rule nextState-decide)
  with dec have TwoThirds ?msgs v by simp
  hence (2*N) div 3 < card { q . ?msgs q = Some v }
    by (simp add: TwoThirds-def HOV-def)
  moreover
```

have { *q* . *?msgs q* = *Some v* } ⊆ { *q* . *x* (*rho n q*) = *v* }
  by (*auto simp*: *OTR-HOMachine-def OTR-sendMsg-def HOrcvdMsgs-def*)
hence *card* { *q* . *?msgs q* = *Some v* } ≤ *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*: *HORun OTR-M rho HOs*
  and *HO*: (*2∗N*) *div 3*
      < *card* { *q* . *HOrcvdMsgs OTR-M n p* (*HOs n p*) (*rho n*) *q* ≠ *None* }
  and *maj*: (*2∗N*) *div 3* < *card* { *q* . *x* (*rho n q*) = *v* }
  shows *x* (*rho* (*Suc n*) *p*) = *v*
proof −
  from *run*
  have *nxt*: *OTR-nextState n p* (*rho n p*)
        (*HOrcvdMsgs OTR-M n p* (*HOs n p*) (*rho n*))
        (*rho* (*Suc n*) *p*)
    (**is** *OTR-nextState - - ?st ?msgs ?st'*)
   by (*simp add*: *HORun-eq HOnextConfig-eq OTR-HOMachine-def nextState-def*)
  let *?HOVothers* = ⋃ { *HOV ?msgs w* | *w* . *w* ≠ *v*}
  — processes from which p received values different from v

  have *w*: *card ?HOVothers* ≤ *N div 3*
  proof −
    have *card ?HOVothers* ≤ *card* (*UNIV* − { *q* . *x* (*rho n q*) = *v* })
     by (*auto simp*: *HOV-def HOrcvdMsgs-def OTR-HOMachine-def OTR-sendMsg-def*

        intro: *card-mono*)
    also have . . . = *N* − *card* { *q* . *x* (*rho n q*) = *v* }
      by (*auto simp*: *card-Diff-subset*)
    also from *maj* **have** . . . ≤ *N div 3* **by** *auto*
    finally show *?thesis* **.**
  qed

  have *hov*: *HOV ?msgs v* = { *q* . *?msgs q* ≠ *None* } − *?HOVothers*
    by (*auto simp*: *HOV-def*) *blast*

  have *othHO*: *?HOVothers* ⊆ { *q* . *?msgs q* ≠ *None* }
    by (*auto simp*: *HOV-def*)

Show that v has been received from more than $N/3$ processes.

  from *HO* **have** *N div 3* < *card* { *q* . *?msgs q* ≠ *None* } − (*N div 3*)
    by *auto*
  also from *w HO* **have** . . . ≤ *card* { *q* . *?msgs q* ≠ *None* } − *card ?HOVothers*

by *auto*
also from *hov othHO* **have** ... = *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 ∀ *w*. *w* ≠ *v* ⟶ *card* (*HOV ?msgs w*) ≤ *card ?HOVothers*
by (*force intro*: *card-mono*)
with *w* **have** *cardw*: ∀ *w*. *w* ≠ *v* ⟶ *card* (*HOV ?msgs w*) ≤ *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 ∀ *w*. *w* ≠ *v* ⟶ ¬(*MFR ?msgs w*)
proof (*auto simp*: *MFR-def not-le*)
fix *w*
assume *w* ≠ *v*
with *cardw HOV* **have** *card* (*HOV ?msgs w*) < *card* (*HOV ?msgs v*) **by** *auto*
thus ∃ *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* } ≤ *card* { *q* . *?msgs q* ≠ *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 *run*:*HORun OTR-M rho HOs*
and *n*: (*2*∗*N*) *div 3* < *card* { *q* . *x* (*rho n q*) = *v* } (**is** *?twothird n*)
shows *?twothird* (*n*+*k*)
proof (*induct k*)
from *n* **show** *?twothird* (*n*+*0*) **by** *simp*
next
fix *m*
assume *m*: *?twothird* (*n*+*m*)
have ∀ *q*. *x* (*rho* (*n*+*m*) *q*) = *v* ⟶ *x* (*rho* (*n* + *Suc m*) *q*) = *v*
proof (*rule*+)
fix *q*
assume *q*: *x* ((*rho* (*n*+*m*)) *q*) = *v*
let *?msgs* = *HOrcvdMsgs OTR-M* (*n*+*m*) *q* (*HOs* (*n*+*m*) *q*) (*rho* (*n*+*m*))
show *x* (*rho* (*n* + *Suc m*) *q*) = *v*
proof (*cases* (*2*∗*N*) *div 3* < *card* { *q* . *?msgs q* ≠ *None* })
case *True*

```
      from m have (2*N) div 3 < card { q . x (rho (n+m) q) = v } by simp
      with True run show ?thesis by (auto elim: A2)
    next
      case False
      with run q show ?thesis by (auto dest: nextState-change)
    qed
  qed
  hence card {q. x (rho (n+m) q) = v} ≤ card {q. x (rho (n + Suc m) q) = v}
    by (auto intro: card-mono)
  with m show ?twothird (n + 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 dec: decide (rho n p) = Some v (is ?dec n)
  shows ∀ k. (2*N) div 3 < card { q . x (rho (n+k) q) = v }
        (is ∀ k. ?twothird (n+k))
using dec proof (induct n)
  — The base case is trivial since no process has decided
  assume ?dec 0 with run show ∀ k. ?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: ?dec m ⟹ ∀ k. ?twothird (m+k) and m: ?dec (Suc m)
  show ∀ k. ?twothird ((Suc m) + k)
  proof
    fix k
    have ?twothird (m + 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 ?dec m)
      case True with ih show ?thesis by blast
    next
      case False
      with run m have ?twothird m by (auto elim: A1)
      with run show ?thesis by (blast dest: A3)
    qed
    thus ?twothird ((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∗N) div 3 < card {q. x (rho (n+k) q) = v} (is - < card ?V)
    by (blast dest: A4)
  moreover
  from run p'
  have (2∗N) div 3 < card {q. x (rho ((n+k)+0) q) = w} (is - < card ?W)
    by (blast dest: A4)
  ultimately
  have N < card ?V + card ?W by auto
  then obtain proc where proc ∈ ?V ∩ ?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 ≤ 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 ≤ 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 run: HORun OTR-M rho HOs
      and commG: HOcommGlobal OTR-M HOs
  shows ∃ r v. decide (rho r p) = Some v
proof −
  from commG obtain r0 Π where
    pi: ∀ q. HOs r0 q = Π and pic: card Π > (2∗N) div 3
    by (auto simp: OTR-HOMachine-def OTR-commGlobal-def)
  let ?msgs q r = HOrcvdMsgs OTR-M r q (HOs r q) (rho r)

  from run pi have ∀ p q. ?msgs q r0 = ?msgs p r0
   by (auto simp: HORun-eq OTR-HOMachine-def HOrcvdMsgs-def OTR-sendMsg-def)
  then obtain μ where ∀ q. ?msgs q r0 = μ by auto
  moreover
  from pi pic have ∀ p. (2∗N) div 3 < card {q. ?msgs p r0 q ≠ None}
    by (auto simp: HORun-eq HOnextConfig-eq HOrcvdMsgs-def)
  with run have ∀ q. x (rho (Suc r0) q) = Min {v . MFR (?msgs q r0) v}
    by (auto simp: HORun-eq HOnextConfig-eq OTR-HOMachine-def
                nextState-def OTR-nextState-def)
  ultimately
  have ∀ q. x (rho (Suc r0) q) = Min {v . MFR μ v} by auto
  then obtain v where v:∀ q. x (rho (Suc r0) q) = v by auto

  have P:∀ k. ∀ q. x (rho (Suc r0+k) q) = v
  proof
    fix k
    show ∀ q. x (rho (Suc r0+k) q) = v
    proof (induct k)
      from v show ∀ q. x (rho (Suc r0+0) q) = v by simp
    next
      fix k
      assume ih:∀ q. x (rho (Suc r0 + k) q) = v
      show ∀ q. x (rho (Suc r0 + Suc k) q) = v
      proof
        fix q
        show x (rho (Suc r0 + Suc k) q) = v
        proof (cases (2∗N) div 3 < card { p . ?msgs q (Suc r0 + k) p ≠ None })
          case True
          have N > 0 by (rule finite-UNIV-card-ge-0) simp
          with ih
          have (2∗N) div 3 < card { p . x (rho (Suc r0 + k) p) = v } by auto
          with True run show ?thesis by (auto elim: A2)
        next
          case False
          with run ih show ?thesis by (auto dest: nextState-change)
        qed
      qed
```

qed
qed

from *commG* **obtain** $r0'$ Π'
where $r0'$: $r0' \geq Suc\ r0$
and pi': $\forall q.\ HOs\ r0'\ q = \Pi'$
and pic': $card\ \Pi' > (2{*}N)\ div\ 3$
by (*force simp*: *OTR-HOMachine-def OTR-commGlobal-def*)
from $r0'$ *P* **have** v':$\forall q.\ x\ (rho\ r0'\ q) = v$ **by** (*auto simp*: *le-iff-add*)

from *run*
have *OTR-nextState* $r0'$ *p* (*rho* $r0'$ *p*) (*?msgs p* $r0'$) (*rho* (*Suc* $r0'$) *p*)
by (*simp add*: *HORun-eq HOnextConfig-eq OTR-HOMachine-def nextState-def*)
moreover
from pi' pic' **have** $(2{*}N)\ div\ 3 < card\ \{q.\ (?msgs\ p\ r0')\ q \neq None\}$
by (*auto simp*: *HOrcvdMsgs-def OTR-sendMsg-def*)
moreover
from pi' pic' v' **have** *TwoThirds* (*?msgs p* $r0'$) *v*
by (*simp add*: *TwoThirds-def HOrcvdMsgs-def OTR-HOMachine-def OTR-sendMsg-def HOV-def*)
ultimately
have *decide* (*rho* (*Suc* $r0'$) *p*) = *Some* (ϵ *v*. *TwoThirds* (*?msgs p* $r0'$) *v*)
by (*auto simp*: *OTR-nextState-def*)
thus *?thesis* **by** *blast*
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* ($x \circ$ (*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* ($\lambda r\ q.$ *undefined*)
and *commG*: *HOcommGlobal OTR-M HOs*
shows *consensus* ($\lambda p.\ x$ (*state* (*rho 0*) *p*)) *decide* (*state* $\circ$ *rho*)
(**is** *consensus ?inits - -*)
proof (*rule local-property-reduction*[*OF run consensus-is-local*])
fix *crun*

assume *crun*: *CSHORun OTR-M crun HOs HOs* ($\lambda r\ q.\ undefined$)
  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* ($x \circ$ (*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 algorithm [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
  $N \equiv$ *card* (*UNIV*::*Proc set*) — number of processes

The algorithm proceeds in *phases* of 2 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.

abbreviation *nSteps* $\equiv$ *2*

definition *phase* **where** *phase* (*r*::*nat*) $\equiv$ *r div nSteps*

definition *step* **where** *step* (*r*::*nat*) $\equiv$ *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** $isValVote\ m \equiv \exists\, z\ v.\ m = ValVote\ z\ v$

definition *isVal* **where** $isVal\ m \equiv \exists\, v.\ m = Val\ v$

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**
  $UV\text{-}initState\ p\ st \equiv (vote\ st = None) \wedge (decide\ st = None)$

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
  $msgRcvd\ (msgs{::}\ Proc \rightharpoonup 'val\ msg) = \{q\ .\ msgs\ q \neq None\}$

definition *smallestValRcvd* **where**
  $smallestValRcvd\ (msgs{::}Proc \rightharpoonup ('val{::}linorder)\ msg) \equiv$
   $Min\ \{v.\ \exists\, q.\ msgs\ q = Some\ (Val\ v)\}$

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**
  $send0\ r\ p\ q\ st \equiv Val\ (x\ st)$

definition *next0* **where**
  $next0\ r\ p\ st\ (msgs{::}Proc \rightharpoonup ('val{::}linorder)\ msg)\ st' \equiv$
    $(\exists\, v.\ (\forall\, q \in msgRcvd\ msgs.\ msgs\ q = Some\ (Val\ v))$
      $\wedge\ st' = st\ (\!|\ vote := Some\ v,\ x := smallestValRcvd\ msgs\ |\!))$
  $\vee\ \neg(\exists\, v.\ \forall\, q \in msgRcvd\ msgs.\ msgs\ q = Some\ (Val\ v))$
    $\wedge\ st' = st\ (\!|\ x := smallestValRcvd\ msgs\ |\!)$

In step 1, each process sends its current x and *vote* values.

definition *send1* **where**
$send1\ r\ p\ q\ st \equiv ValVote\ (x\ st)\ (vote\ st)$

definition *valVoteRcvd* **where**
— processes from which values and votes were received
$valVoteRcvd\ (msgs :: Proc \rightharpoonup 'val\ msg) \equiv$
$\{q\ .\ \exists\, z\ v.\ msgs\ q = Some\ (ValVote\ z\ v)\}$

definition *smallestValNoVoteRcvd* **where**
$smallestValNoVoteRcvd\ (msgs::Proc \rightharpoonup ('val::linorder)\ msg) \equiv$
$Min\ \{v.\ \exists\, q.\ msgs\ q = Some\ (ValVote\ v\ None)\}$

definition *someVoteRcvd* **where**
— set of processes from which some vote was received
$someVoteRcvd\ (msgs :: Proc \rightharpoonup 'val\ msg) \equiv$
$\{\ q\ .\ q \in msgRcvd\ msgs \wedge isValVote\ (the\ (msgs\ q)) \wedge getvote\ (the\ (msgs\ q)) \neq None\ \}$

definition *identicalVoteRcvd* **where**
$identicalVoteRcvd\ (msgs :: Proc \rightharpoonup 'val\ msg)\ v \equiv$
$\forall\, q \in msgRcvd\ msgs.\ isValVote\ (the\ (msgs\ q)) \wedge getvote\ (the\ (msgs\ q)) = Some\ v$

definition *x-update* **where**
$x\text{-}update\ st\ msgs\ st' \equiv$
$(\exists\, q \in someVoteRcvd\ msgs\ .\ x\ st' = the\ (getvote\ (the\ (msgs\ q))))$
$\vee\ someVoteRcvd\ msgs = \{\} \wedge x\ st' = smallestValNoVoteRcvd\ msgs$

definition *dec-update* **where**
$dec\text{-}update\ st\ msgs\ st' \equiv$
$(\exists\, v.\ identicalVoteRcvd\ msgs\ v \wedge decide\ st' = Some\ v)$
$\vee\ \neg(\exists\, v.\ identicalVoteRcvd\ msgs\ v) \wedge decide\ st' = decide\ st$

definition *next1* **where**
$next1\ r\ p\ st\ msgs\ st' \equiv$
$x\text{-}update\ st\ msgs\ st'$
$\wedge\ dec\text{-}update\ st\ msgs\ st'$
$\wedge\ vote\ st' = None$

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

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

definition *UV-nextState* **where**
$UV\text{-}nextState\ r \equiv if\ step\ r = 0\ then\ next0\ r\ else\ next1\ r$

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**
$UV\text{-}commPerRd\ HOrs \equiv \forall\, p\ q.\ \exists\, pq.\ pq \in HOrs\ p \cap HOrs\ q$

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**
$UV\text{-}commGlobal\ HOs \equiv \exists\, r.\ \forall\, p\ q.\ HOs\ r\ p = HOs\ r\ q$

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**
$UV\text{-}HOMachine = (\!|$
$CinitState = (\lambda p\ st\ crd.\ UV\text{-}initState\ p\ st),$
$sendMsg = UV\text{-}sendMsg,$
$CnextState = (\lambda r\ p\ st\ msgs\ crd\ st'.\ UV\text{-}nextState\ r\ p\ st\ msgs\ st'),$
$HOcommPerRd = UV\text{-}commPerRd,$
$HOcommGlobal = UV\text{-}commGlobal$
$|\!)$

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

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 ∃pq. pq ∈ msgRcvd (HOrcvdMsgs UV-M r p (HOs r p) (rho r))
           ∧ pq ∈ msgRcvd (HOrcvdMsgs UV-M r q (HOs r q) (rho r))
           ∧ (HOrcvdMsgs UV-M r p (HOs r p) (rho r)) pq
             = (HOrcvdMsgs UV-M r q (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 q) (rho r))
        ∈ {v. ∃p. (HOrcvdMsgs UV-M r q (HOs r q) (rho r)) p = Some (Val v)}
        (is smallestValRcvd ?msgs ∈ ?vals)
unfolding smallestValRcvd-def proof (rule Min-in)
  have ?vals ⊆ getval ` ((the ∘ ?msgs) ` (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 ∈ msgRcvd ?msgs by blast
  with s0 show ?vals ≠ {}
    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 ≠ 0
  and nov: someVoteRcvd (HOrcvdMsgs UV-M r q (HOs r q) (rho r)) = {}
  shows smallestValNoVoteRcvd (HOrcvdMsgs UV-M r q (HOs r q) (rho r))
         ∈ {v . ∃p. (HOrcvdMsgs UV-M r q (HOs r q) (rho r)) p
                   = Some (ValVote v None)}
        (is smallestValNoVoteRcvd ?msgs ∈ ?vals)
unfolding smallestValNoVoteRcvd-def proof (rule Min-in)
  have ?vals ⊆ getval ` ((the ∘ ?msgs) ` (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 ∈ msgRcvd ?msgs by blast
  with s1 nov show ?vals ≠ {}
    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
    by (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)
   by (auto simp: UV-HOMachine-def HORun-eq HOnextConfig-eq nextState-def)
  from s0 sucr have step r = 1 by (auto simp: step-def mod-Suc)
  with nxt sucr show ?thesis
    by (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: ∀ r. HOcommPerRd 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 show ?thesis — no vote in initial state
    by (auto simp: UV-HOMachine-def HORun-eq HOinitConfig-eq
               initState-def UV-initState-def)
next
  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)
    by (auto simp: HORun-eq HOnextConfig-eq nextState-def)
  with vote r
  have nxt0: next0 r′ p (rho r′ p) ?msgs (rho r p) and s0: step r′ = 0
   by (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: ∀ q ∈ msgRcvd ?msgs. ?msgs q = Some (Val v)
    and x: x (rho r p) = smallestValRcvd ?msgs
    by (auto simp: next0-def)
  moreover
  from com obtain q where q ∈ msgRcvd ?msgs
```

by (*force dest*: *some-common-msg*)
with *idv* **have** $\{x \,.\, \exists\, qq.\ ?msgs\ qq = Some\ (Val\ x)\} = \{v\}$
by (*auto simp*: *msgRcvd-def*)
hence *smallestValRcvd ?msgs* = *v*
by (*auto simp*: *smallestValRcvd-def*)
ultimately
show *?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 *run*: *HORun UV-M rho HOs*
and *decide*: *decide* (*rho* (*Suc r*) *p*) $\neq$ *decide* (*rho r p*)
shows *step r* = *1*
proof −
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*)
by (*auto simp*: *HORun-eq HOnextConfig-eq nextState-def*)
with *decide* **show** *?thesis*
by (*auto simp*: *nextState-def UV-HOMachine-def UV-nextState-def*
next0-def step-def)
qed

No process ever decides *None*.

lemma *decide-nonnull*:
assumes *run*: *HORun UV-M rho HOs*
and *decide*: *decide* (*rho* (*Suc r*) *p*) $\neq$ *decide* (*rho r p*)
shows *decide* (*rho* (*Suc r*) *p*) $\neq$ *None*
proof −
let *?msgs* = *HOrcvdMsgs UV-M r p* (*HOs r p*) (*rho r*)
from *assms* **have** *s1*: *step r* = *1* **by** (*rule decide-step*)
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)
with *decide* **show** *?thesis*
by (*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 *run*: *HORun UV-M rho HOs*
and *vote*: *vote* (*rho* (*Suc r*) *p*) = *Some v*
and *q*: $q \in$ *msgRcvd* (*HOrcvdMsgs UV-M r p* (*HOs r p*) (*rho r*))
(**is** - $\in$ *msgRcvd ?msgs*)
shows *getval* (*the* (*?msgs q*)) = *v*

proof −
  have *s0*: *step r* = *0*
  proof (*rule ccontr*)
    assume *step r* ≠ *0*
    hence *step* (*Suc r*) = *0* **by** (*simp add*: *step-def mod-Suc*)
    with *run vote* **show** *False* **by** (*auto simp*: *reset-vote*)
  qed
  with *run* **have** *novote*: *vote* (*rho r p*) = *None* **by** (*auto simp*: *reset-vote*)
  from *run* **have** *nextState UV-M r p* (*rho r p*) *?msgs* (*rho* (*Suc r*) *p*)
    by (*auto simp*: *HORun-eq HOnextConfig-eq nextState-def*)
  with *s0* **have** *nxt*: *next0 r p* (*rho r p*) *?msgs* (*rho* (*Suc r*) *p*)
    by (*auto simp*: *UV-HOMachine-def nextState-def UV-nextState-def*)
  with *novote vote q* **show** *?thesis* **by** (*auto simp*: *next0-def*)
qed

Any two processes can only vote for the same value.

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

If a process decides value v then all processes must have v in their x fields.

lemma *decide-equals-x*:
  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*) *q*) = *v*
proof −
  let *?msgs p′* = *HOrcvdMsgs UV-M r p′* (*HOs r p′*) (*rho r*)
  from *run decide* **have** *s1*: *step r* = *1* **by** (*rule decide-step*)
  from *run* **have** *nextState UV-M r p* (*rho r p*) (*?msgs p*) (*rho* (*Suc r*) *p*)
    by (*auto simp*: *HORun-eq HOnextConfig-eq nextState-def*)
  with *s1* **have** *nxtp*: *next1 r p* (*rho r p*) (*?msgs p*) (*rho* (*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* (*Suc r*) *q*)
    by (*auto simp*: *HORun-eq HOnextConfig-eq nextState-def*)
  with *s1* **have** *nxtq*: *next1 r q* (*rho r q*) (*?msgs q*) (*rho* (*Suc r*) *q*)
    by (*auto simp*: *UV-HOMachine-def nextState-def UV-nextState-def*)

  from *com* **obtain** *pq* **where**
    pq: *pq* ∈ *msgRcvd* (*?msgs p*) *pq* ∈ *msgRcvd* (*?msgs q*)
        (*?msgs p*) *pq* = (*?msgs q*) *pq*
    by (*force dest*: *some-common-msg*)
  with *decide decval nxtp*
  have *vote*: *isValVote* (*the* (*?msgs p pq*))
            getvote (*the* (*?msgs p pq*)) = *Some v*
    by (*auto simp*: *next1-def dec-update-def identicalVoteRcvd-def*)
  with *nxtq pq* **obtain** *q′* **where**
    q′: *q′* ∈ *someVoteRcvd* (*?msgs q*)
        x (*rho* (*Suc r*) *q*) = *the* (*getvote* (*the* (*?msgs q q′*)))
    by (*auto simp*: *next1-def x-update-def someVoteRcvd-def*)
  with *s1 pq vote* **show** *?thesis*
   by (*auto simp*: *HOrcvdMsgs-def 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*: *HORun UV-M rho HOs*
    and *comm*: ∀ *r*. *HOcommPerRd UV-M* (*HOs r*)
    and *x*: ∀ *p*. *x* (*rho r p*) = *v*
  shows *x* (*rho* (*Suc r*) *q*) = *v*
proof −
  let *?msgs* = *HOrcvdMsgs UV-M r q* (*HOs r q*) (*rho r*)
  from *comm* **obtain** *p* **where** *p*: *p* ∈ *msgRcvd ?msgs*
    by (*force dest*: *some-common-msg*)
  from *run* **have** *nextState UV-M r q* (*rho r q*) *?msgs* (*rho* (*Suc r*) *q*)
    by (*auto simp*: *HORun-eq HOnextConfig-eq nextState-def*)
  hence *next0 r q* (*rho r q*) *?msgs* (*rho* (*Suc r*) *q*) ∧ *step r* = *0*
       ∨ *next1 r q* (*rho r q*) *?msgs* (*rho* (*Suc r*) *q*) ∧ *step r* ≠ *0*
    (**is** *?nxt0* ∨ *?nxt1*)
    by (*auto simp*: *UV-HOMachine-def nextState-def UV-nextState-def*)

```
  thus ?thesis
  proof
    assume nxt0: ?nxt0
    hence x (rho (Suc r) q) = smallestValRcvd ?msgs
      by (auto simp: next0-def)
    moreover
    from nxt0 x have ∀ p ∈ msgRcvd ?msgs. ?msgs p = Some (Val v)
      by (auto simp: UV-HOMachine-def HOrcvdMsgs-def UV-sendMsg-def
                     msgRcvd-def send0-def)
    from this p have {x . ∃ p. ?msgs p = Some (Val x)} = {v}
      by (auto simp: msgRcvd-def)
    hence smallestValRcvd ?msgs = v
      by (auto simp: smallestValRcvd-def)
    ultimately
    show ?thesis by simp
  next
    assume nxt1: ?nxt1
    show ?thesis
    proof (cases someVoteRcvd ?msgs = {})
      case True
      with nxt1 have x (rho (Suc r) q) = smallestValNoVoteRcvd ?msgs
        by (auto simp: next1-def x-update-def)
      moreover
      from nxt1 x True
      have ∀ p ∈ msgRcvd ?msgs. ?msgs p = Some (ValVote v None)
        by (auto simp: UV-HOMachine-def HOrcvdMsgs-def UV-sendMsg-def
                       msgRcvd-def send1-def someVoteRcvd-def isValVote-def)
      from this p have {x . ∃ p. ?msgs p = Some (ValVote x None)} = {v}
        by (auto simp: msgRcvd-def)
      hence smallestValNoVoteRcvd ?msgs = v
        by (auto simp: smallestValNoVoteRcvd-def)
      ultimately show ?thesis by simp
    next
      case False
      with nxt1 obtain p' v' where
        p': p' ∈ msgRcvd ?msgs isValVote (the (?msgs p'))
            getvote (the (?msgs p')) = Some v'x (rho (Suc r) q) = v'
        by (auto simp: someVoteRcvd-def next1-def x-update-def)
      with nxt1 have x (rho (Suc r) q) = x (rho r p')
        by (auto simp: UV-HOMachine-def HOrcvdMsgs-def UV-sendMsg-def
                       msgRcvd-def send1-def isValVote-def
                       x-vote-eq[OF run comm])
      with x show ?thesis by auto
    qed
  qed
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*: $\forall$ *r*. *HOcommPerRd UV-M* (*HOs r*)
    and *decide*: *decide* (*rho* (*Suc r*) *p*) $\neq$ *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 $\bigwedge$*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 sometime 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*) $\neq$ *decide* (*rho m p*)
      $\wedge$ *decide* (*rho* (*Suc m*) *p*) = *Some v*
proof −
  let *?dec k* = *decide* (*rho k p*)
  have ($\forall$ *m*<*n*. *?dec* (*Suc m*) $\neq$ *?dec m* $\longrightarrow$ *?dec* (*Suc m*) $\neq$ *Some v*)
      $\longrightarrow$ *?dec n* $\neq$ *Some v*
    (**is** *?P n* **is** *?A n* $\longrightarrow$ -)
  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** *?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: ⋀p'. ∃ q. x (rho r p') = x (rho 0 q)
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)
  by (auto simp: HORun-eq HOnextConfig-eq nextState-def)
hence next0 r p (rho r p) ?msgs (rho (Suc r) p) ∧ step r = 0
      ∨ next1 r p (rho r p) ?msgs (rho (Suc r) p) ∧ step r ≠ 0
  (is ?nxt0 ∨ ?nxt1)
  by (auto simp: UV-HOMachine-def nextState-def UV-nextState-def)
thus ∃ q. x (rho (Suc r) p) = x (rho 0 q)
proof
  assume nxt0: ?nxt0
  hence x (rho (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 nxt0 have ... = { x (rho 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 (rho (Suc r) p) = x (rho r q) by auto
  with ih show ?thesis by auto
next
  assume nxt1: ?nxt1
  show ?thesis
  proof (cases someVoteRcvd ?msgs = {})
    case True
    with nxt1 have x (rho (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 nxt1 True
    have ... = { x (rho 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 (rho (Suc r) p) = x (rho r q) by auto
    with ih show ?thesis by auto
  next
    case False
    with nxt1 obtain q where
      q ∈ someVoteRcvd ?msgs
      x (rho (Suc r) p) = the (getvote (the (?msgs q)))
      by (auto simp: next1-def x-update-def)
    with nxt1 have vote (rho r q) = Some (x (rho (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 (rho (Suc r) p) = x (rho r q)
      by (rule x-vote-eq)
    with ih show ?thesis by auto
```

qed
qed
qed

theorem *uv-integrity*:
assumes *run*: *HORun UV-M rho HOs*
and *com*: $\forall$ *r. HOcommPerRd UV-M* (*HOs r*)
and *dec*: *decide* (*rho r p*) = *Some v*
shows $\exists$ *q. v* = *x* (*rho 0 q*)
proof −
from *run dec* **obtain** *k* **where**
decide (*rho* (*Suc k*) *p*) $\neq$ *decide* (*rho k p*)
decide (*rho* (*Suc k*) *p*) = *Some v*
by (*auto dest*: *decided-then-past-decision*)
with *run com* **have** *x* (*rho* (*Suc k*) *p*) = *v*
by (*rule decide-equals-x*)
with *run com* **show** *?thesis*
by (*auto dest*: *x-values-initial*)
qed

We now turn to Agreement.

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

theorem *uv-agreement*:
assumes *run*: *HORun UV-M rho HOs*
and *com*: $\forall$ *r. HOcommPerRd UV-M* (*HOs 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**
k: *decide* (*rho* (*Suc k*) *p*) $\neq$ *decide* (*rho k p*)
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*) $\neq$ *decide* (*rho l q*)
decide (*rho* (*Suc l*) *q*) = *Some w*
by (*auto dest*: *decided-then-past-decision*)
show *?thesis*
proof (*cases k* $\leq$ *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* $\leq$ *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*: $\forall$ *r*. *HOcommPerRd 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*: $\forall$ *r*. *HOcommPerRd UV-M* (*HOs r*)
and *hoeq*: *HOs r p* = *HOs r q*

```
  shows x (rho (Suc r) p) = x (rho (Suc r) q)
proof –
  let ?msgs p = 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 ∀ p. 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 ≠ 0
    with run
    have ∀ p. 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-update (rho r p) (?msgs p) (rho (Suc r) p)
          x-update (rho r q) (?msgs q) (rho (Suc r) q)
      by (auto simp: next1-def)
    with msgeq have
      x': x-update (rho r p) (?msgs p) (rho (Suc r) p)
          x-update (rho r q) (?msgs p) (rho (Suc r) q)
      by auto
    show ?thesis
    proof (cases someVoteRcvd (?msgs p) = {})
      case True
      with x' show ?thesis
        by (auto simp: x-update-def)
    next
      case False
      with x' stp obtain qp qq where
        vote (rho r qp) = Some (x (rho (Suc r) p)) and
        vote (rho r qq) = Some (x (rho (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
qed
```

We now prove that *UniformVoting* terminates.

theorem *uv-termination*:

assumes *run*: *HORun UV-M rho HOs*
  and *commR*: $\forall r$. *HOcommPerRd UV-M* (*HOs r*)
  and *commG*: *HOcommGlobal UV-M HOs*
shows $\exists r\ v$. *decide* (*rho r p*) = *Some v*
proof −

First obtain a round where all x values agree.

from *commG* **obtain** *r0* **where** *r0*: $\forall q$. *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*: $\forall q$. *x* (*rho* (*Suc 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′*: $\forall q$. *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* (*Suc r0*)) *q*) = *?v*
    by (*rule same-x-stable*)
qed
have *vote′*: $\forall q$. *vote* (*rho* (*Suc r′*) *q*) = *Some ?v*
proof
  fix *q*
  let *?msgs* = *HOrcvdMsgs UV-M r′ q* (*HOs r′ q*) (*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** $\forall q' \in$ *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* $\neq$ {}
    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** *identicalVoteRcvd ?msgs' ?v*
by (*auto simp*: *UV-HOMachine-def HOrcvdMsgs-def UV-sendMsg-def send1-def identicalVoteRcvd-def isValVote-def msgRcvd-def*)
moreover
from *commR* **have** *msgRcvd ?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 identicalVoteRcvd-def msgRcvd-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* (λ*r q*. *undefined*)
and *commR*: $\forall$ *r*. *HOcommPerRd UV-M* (*HOs r*)
and *commG*: *HOcommGlobal UV-M HOs*
shows *consensus* (λ*p*. *x* (*state* (*rho 0*) *p*)) *decide* (*state* $\circ$ *rho*)
(**is** *consensus ?inits - -*)
proof (*rule local-property-reduction*[*OF run consensus-is-local*])
fix *crun*
assume *crun*: *CSHORun UV-M crun HOs HOs* (λ*r q*. *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*)

with *init* **show** *consensus ?inits decide crun*
by (*simp add*: *o-def*)
qed

end
theory *LastVotingDefs*
imports *../HOModel*
begin

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.

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

abbreviation
$N \equiv card\ (UNIV::Proc\ set)$ — 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.

definition *phase* **where** $phase\ (r::nat) \equiv r\ div\ 4$

definition *step* **where** $step\ (r::nat) \equiv r\ mod\ 4$

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*)

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
  $commt$ :: $bool$ — 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
  $coord\Phi$ :: $Proc$ — coordinator for current phase

Possible messages sent during the execution of the algorithm.

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

Characteristic predicates on messages.

definition *isValStamp* **where** $isValStamp\ m \equiv \exists\, v\ ts.\ m = ValStamp\ v\ ts$

definition *isVote* **where** $isVote\ m \equiv \exists\, v.\ m = Vote\ v$

definition *isAck* **where** $isAck\ m \equiv m = Ack$

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**
  $LV\text{-}initState\ p\ st\ crd \equiv$
    $vote\ st = None$
  $\wedge \neg(commt\ st)$
  $\wedge \neg(ready\ st)$
  $\wedge\ timestamp\ st = 0$
  $\wedge\ decide\ st = None$
  $\wedge\ coord\Phi\ st = crd$

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**
$valStampsRcvd\ (msgs :: Proc \rightharpoonup {}'val\ msg) \equiv$
$\{q\ .\ \exists\, v\ ts.\ msgs\ q = Some\ (ValStamp\ v\ ts)\}$

definition *highestStampRcvd* **where**
$highestStampRcvd\ msgs \equiv$
$Max\ \{ts\ .\ \exists\, q\ v.\ (msgs::Proc \rightharpoonup {}'val\ msg)\ q = Some\ (ValStamp\ v\ ts)\}$

In step 0, each process sends its current x and *timestamp* values to its coordinator.

A process that considers itself to be a coordinator updates its *vote* field if it has received messages from a majority of processes. It then sets its *commt* field to true.

definition *send0* **where**
$send0\ r\ p\ q\ st \equiv$
$if\ q = coord\Phi\ st\ then\ ValStamp\ (x\ st)\ (timestamp\ st)\ else\ Null$

definition *next0* **where**
$next0\ r\ p\ st\ msgs\ crd\ st' \equiv$
$if\ p = coord\Phi\ st \wedge card\ (valStampsRcvd\ msgs) > N\ div\ 2$
$then\ (\exists\, p\ v.\ msgs\ p = Some\ (ValStamp\ v\ (highestStampRcvd\ msgs))$
$\wedge\ st' = st\ (\!|\ vote := Some\ v,\ commt := True\ |\!)\ )$
$else\ st' = st$

In step 1, coordinators that have committed send their vote to all processes.

Processes update their x and *timestamp* fields if they have received a vote from their coordinator.

definition *send1* **where**
$send1\ r\ p\ q\ st \equiv$
$if\ p = coord\Phi\ st \wedge commt\ st\ then\ Vote\ (the\ (vote\ st))\ else\ Null$

definition *next1* **where**
$next1\ r\ p\ st\ msgs\ crd\ st' \equiv$
$if\ msgs\ (coord\Phi\ st) \neq None \wedge isVote\ (the\ (msgs\ (coord\Phi\ st)))$
$then\ st' = st\ (\!|\ x := val\ (the\ (msgs\ (coord\Phi\ st))),\ timestamp := Suc(phase\ r)\ |\!)$
$else\ st' = st$

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

A coordinator sets its *ready* field to true if it receives a majority of acknowledgements.

definition *send2* **where**
$send2\ r\ p\ q\ st \equiv$
$if\ timestamp\ st = Suc(phase\ r) \wedge q = coord\Phi\ st\ then\ Ack\ else\ Null$

— processes from which an acknowledgement was received
definition *acksRcvd* **where**

$acksRcvd\ (msgs :: Proc \rightharpoonup {'val}\ msg) \equiv$
$\{\ q\ .\ msgs\ q \neq None \wedge isAck\ (the\ (msgs\ q))\ \}$

definition *next2* **where**
$next2\ r\ p\ st\ msgs\ crd\ st' \equiv$
$if\ p = coord\Phi\ st \wedge card\ (acksRcvd\ msgs) > N\ div\ 2$
$then\ st' = st\ (\!|\ ready := True\ |\!)$
$else\ st' = st$

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 *ready* and *commt* fields to false. All processes reset the coordinators as indicated by the parameter of the operator.

definition *send3* **where**
$send3\ r\ p\ q\ st \equiv$
$if\ p = coord\Phi\ st \wedge ready\ st\ then\ Vote\ (the\ (vote\ st))\ else\ Null$

definition *next3* **where**
$next3\ r\ p\ st\ msgs\ crd\ st' \equiv$
$(if\ msgs\ (coord\Phi\ st) \neq None \wedge isVote\ (the\ (msgs\ (coord\Phi\ st)))$
$then\ decide\ st' = Some\ (val\ (the\ (msgs\ (coord\Phi\ st))))$
$else\ decide\ st' = decide\ st)$
$\wedge\ (if\ p = coord\Phi\ st$
$then\ \neg(ready\ st') \wedge \neg(commt\ st')$
$else\ ready\ st' = ready\ st \wedge commt\ st' = commt\ st)$
$\wedge\ x\ st' = x\ st$
$\wedge\ vote\ st' = vote\ st$
$\wedge\ timestamp\ st' = timestamp\ st$
$\wedge\ coord\Phi\ st' = crd$

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

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

definition
$LV\text{-}nextState :: nat \Rightarrow Proc \Rightarrow {'val}\ pstate \Rightarrow (Proc \rightharpoonup {'val}\ msg)$
$\Rightarrow Proc \Rightarrow {'val}\ pstate \Rightarrow bool$
where
$LV\text{-}nextState\ r \equiv$
$if\ step\ r = 0\ then\ next0\ r$
$else\ if\ step\ r = 1\ then\ next1\ r$
$else\ if\ step\ r = 2\ then\ next2\ r$
$else\ next3\ r$

7.2 Communication Predicate for *LastVoting*

We now define the communication predicate that will be assumed for the correctness proof of the *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.)

definition
  LV-commPerRd **where**
  LV-commPerRd r (HO::Proc HO) (coord::Proc coord) ≡ *True*

definition
  LV-commGlobal **where**
  LV-commGlobal HOs coords ≡
    $\exists\, ph{::}nat.\ \exists\, c{::}Proc.$
      $(\forall\, p.\ coords\ (4{*}ph)\ p = c)$
     $\wedge\ card\ (HOs\ (4{*}ph)\ c) > N\ div\ 2$
     $\wedge\ card\ (HOs\ (4{*}ph{+}2)\ c) > N\ div\ 2$
     $\wedge\ (\forall\, p.\ c \in HOs\ (4{*}ph{+}1)\ p \cap HOs\ (4{*}ph{+}3)\ p)$

7.3 The *LastVoting* Heard-Of Machine

We now define the coordinated HO machine for the *LastVoting* algorithm by assembling the algorithm definition and its communication-predicate.

definition *LV-CHOMachine* **where**
  LV-CHOMachine ≡
   (| *CinitState = LV-initState*,
    sendMsg = LV-sendMsg,
    CnextState = LV-nextState,
    CHOcommPerRd = LV-commPerRd,
    CHOcommGlobal = LV-commGlobal |)

abbreviation
  LV-M ≡ (*LV-CHOMachine::(Proc, 'val pstate, 'val msg) CHOMachine*)

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*:
  finite $\{ts\ .\ \exists\, q\ v.\ (msgs{::}Proc \rightharpoonup {}'val\ msg)\ q = Some\ (ValStamp\ v\ ts)\}$
  (**is** *finite ?ts*)
proof −
  have *?ts* = *stamp ‘ the ‘ msgs ‘ (valStampsRcvd msgs)*
    by (*force simp add*: *valStampsRcvd-def image-def*)
  thus *?thesis* **by** *auto*
qed

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

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

lemma *phase-Suc*:
  phase (Suc r) = (if step r = 3 then Suc (phase r)
                  else phase r)
  unfolding *step-def phase-def* **by** *presburger*

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*:
  assumes *run*: *CHORun LV-M rho HOs coords*
  and *init*: $\forall\, p.$ *CinitState LV-M p (rho 0 p) (coords 0 p)* $\Longrightarrow$ *P 0*

and *step0*: $\bigwedge r$.
$[\![$ *step r = 0*; *P r*; *phase* (*Suc r*) = *phase r*; *step* (*Suc r*) = *1*;
$\forall$ *p*. *next0 r p* (*rho r p*)
(*HOrcvdMsgs LV-M r p* (*HOs r p*) (*rho r*))
(*coords* (*Suc r*) *p*)
(*rho* (*Suc r*) *p*) $]\!]$
$\Longrightarrow$ *P* (*Suc r*)
and *step1*: $\bigwedge r$.
$[\![$ *step r = 1*; *P r*; *phase* (*Suc r*) = *phase r*; *step* (*Suc r*) = *2*;
$\forall$ *p*. *next1 r p* (*rho r p*)
(*HOrcvdMsgs LV-M r p* (*HOs r p*) (*rho r*))
(*coords* (*Suc r*) *p*)
(*rho* (*Suc r*) *p*) $]\!]$
$\Longrightarrow$ *P* (*Suc r*)
and *step2*: $\bigwedge r$.
$[\![$ *step r = 2*; *P r*; *phase* (*Suc r*) = *phase r*; *step* (*Suc r*) = *3*;
$\forall$ *p*. *next2 r p* (*rho r p*)
(*HOrcvdMsgs LV-M r p* (*HOs r p*) (*rho r*))
(*coords* (*Suc r*) *p*)
(*rho* (*Suc r*) *p*) $]\!]$
$\Longrightarrow$ *P* (*Suc r*)
and *step3*: $\bigwedge r$.
$[\![$ *step r = 3*; *P r*; *phase* (*Suc r*) = *Suc* (*phase r*); *step* (*Suc r*) = *0*;
$\forall$ *p*. *next3 r p* (*rho r p*)
(*HOrcvdMsgs LV-M r p* (*HOs r p*) (*rho r*))
(*coords* (*Suc r*) *p*)
(*rho* (*Suc r*) *p*) $]\!]$
$\Longrightarrow$ *P* (*Suc r*)
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*)
(*coords* (*Suc r*)) (*rho* (*Suc r*))
have *step r* $\in$ {*0*,*1*,*2*,*3*} **by** (*auto simp add*: *step-def*)
thus *P* (*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*)
(*auto simp*: *LV-CHOMachine-def CHOnextConfig-eq*
LV-nextState-def LV-sendMsg-def phase-Suc)
next
assume *stp*: *step r = Suc 0*

```
    hence step (Suc r) = 2
      by (auto simp add: step-def mod-Suc)
    with ih nxt stp show ?thesis
      by (intro step1)
         (auto simp: LV-CHOMachine-def CHOnextConfig-eq
                     LV-nextState-def LV-sendMsg-def phase-Suc)
  next
    assume stp: step r = 2
    hence step (Suc r) = 3
      by (auto simp add: step-def mod-Suc)
    with ih nxt stp show ?thesis
      by (intro step2)
         (auto simp: LV-CHOMachine-def CHOnextConfig-eq
                     LV-nextState-def LV-sendMsg-def phase-Suc)
  next
    assume stp: step r = 3
    hence step (Suc r) = 0
      by (auto simp add: step-def mod-Suc)
    with ih nxt stp show ?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 LV-Suc:
  assumes run: CHORun LV-M rho HOs coords
  and step0: ⟦ step r = 0; step (Suc r) = 1; phase (Suc r) = phase r;
              ∀p. next0 r p (rho r p)
                        (HOrcvdMsgs LV-M r p (HOs r p) (rho r))
                        (coords (Suc r) p) (rho (Suc r) p) ⟧
             ⟹ P r
  and step1: ⟦ step r = 1; step (Suc r) = 2; phase (Suc r) = phase r;
              ∀p. next1 r p (rho r p)
                        (HOrcvdMsgs LV-M r p (HOs r p) (rho r))
                        (coords (Suc r) p) (rho (Suc r) p) ⟧
             ⟹ P r
  and step2: ⟦ step r = 2; step (Suc r) = 3; phase (Suc r) = phase r;
              ∀p. next2 r p (rho r p)
                        (HOrcvdMsgs LV-M r p (HOs r p) (rho r))
                        (coords (Suc r) p) (rho (Suc r) p) ⟧
             ⟹ P r
  and step3: ⟦ step r = 3; step (Suc r) = 0; phase (Suc r) = Suc (phase r);
              ∀p. next3 r p (rho r p)
                        (HOrcvdMsgs LV-M r p (HOs r p) (rho r))
                        (coords (Suc r) p) (rho (Suc r) p) ⟧
             ⟹ P r
```

shows *P r*
proof –
  from *run*
  have *nxt*: *CHOnextConfig LV-M r* (*rho r*) (*HOs r*)
                (*coords* (*Suc r*)) (*rho* (*Suc r*))
    by (*auto simp add*: *CHORun-eq*)
  have *step r* $\in$ {*0*,*1*,*2*,*3*} **by** (*auto simp add*: *step-def*)
  thus *P r*
  proof (*auto*)
    assume *stp*: *step r* = *0*
    hence *step* (*Suc r*) = *1*
      by (*auto simp add*: *step-def mod-Suc*)
    with *nxt stp* **show** *?thesis*
      by (*intro step0*)
        (*auto simp*: *LV-CHOMachine-def CHOnextConfig-eq*
              LV-nextState-def LV-sendMsg-def phase-Suc)
  next
    assume *stp*: *step r* = *Suc 0*
    hence *step* (*Suc r*) = *2*
      by (*auto simp add*: *step-def mod-Suc*)
    with *nxt stp* **show** *?thesis*
      by (*intro step1*)
        (*auto simp*: *LV-CHOMachine-def CHOnextConfig-eq*
              LV-nextState-def LV-sendMsg-def phase-Suc)
  next
    assume *stp*: *step r* = *2*
    hence *step* (*Suc r*) = *3*
      by (*auto simp add*: *step-def mod-Suc*)
    with *nxt stp* **show** *?thesis*
      by (*intro step2*)
        (*auto simp*: *LV-CHOMachine-def CHOnextConfig-eq*
              LV-nextState-def LV-sendMsg-def phase-Suc)
  next
    assume *stp*: *step r* = *3*
    hence *step* (*Suc r*) = *0*
      by (*auto simp add*: *step-def mod-Suc*)
    with *nxt stp* **show** *?thesis*
      by (*intro step3*)
        (*auto simp*: *LV-CHOMachine-def CHOnextConfig-eq*
              LV-nextState-def LV-sendMsg-def phase-Suc)
  qed
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 *LV-induct$'$*:
  assumes *run*: *CHORun LV-M rho HOs coords*

and *init*: *CinitState LV-M p* (*rho 0 p*) (*coords 0 p*) $\Longrightarrow$ *P p 0*
and *step0*: $\bigwedge$*r*. ⟦ *step r* = *0*; *P p r*; *phase* (*Suc r*) = *phase r*; *step* (*Suc r*) = *1*;
next0 r p (*rho r p*)
(*HOrcvdMsgs LV-M r p* (*HOs r p*) (*rho r*))
(*coords* (*Suc r*) *p*) (*rho* (*Suc r*) *p*) ⟧
$\Longrightarrow$ *P p* (*Suc r*)
and *step1*: $\bigwedge$*r*. ⟦ *step r* = *1*; *P p r*; *phase* (*Suc r*) = *phase r*; *step* (*Suc r*) = *2*;
next1 r p (*rho r p*)
(*HOrcvdMsgs LV-M r p* (*HOs r p*) (*rho r*))
(*coords* (*Suc r*) *p*) (*rho* (*Suc r*) *p*) ⟧
$\Longrightarrow$ *P p* (*Suc r*)
and *step2*: $\bigwedge$*r*. ⟦ *step r* = *2*; *P p r*; *phase* (*Suc r*) = *phase r*; *step* (*Suc r*) = *3*;
next2 r p (*rho r p*)
(*HOrcvdMsgs LV-M r p* (*HOs r p*) (*rho r*))
(*coords* (*Suc r*) *p*) (*rho* (*Suc r*) *p*) ⟧
$\Longrightarrow$ *P p* (*Suc r*)
and *step3*: $\bigwedge$*r*. ⟦ *step r* = *3*; *P p r*; *phase* (*Suc r*) = *Suc* (*phase r*); *step* (*Suc r*) = *0*;
next3 r p (*rho r p*)
(*HOrcvdMsgs LV-M r p* (*HOs r p*) (*rho r*))
(*coords* (*Suc r*) *p*) (*rho* (*Suc r*) *p*) ⟧
$\Longrightarrow$ *P p* (*Suc r*)
shows *P p n*
by (*rule LV-induct*[*OF run*])
(*auto intro*: *init step0 step1 step2 step3*)

lemma *LV-Suc'*:
assumes *run*: *CHORun LV-M rho HOs coords*
and *step0*: ⟦ *step r* = *0*; *step* (*Suc r*) = *1*; *phase* (*Suc r*) = *phase r*;
next0 r p (*rho r p*)
(*HOrcvdMsgs LV-M r p* (*HOs r p*) (*rho r*))
(*coords* (*Suc r*) *p*) (*rho* (*Suc r*) *p*) ⟧
$\Longrightarrow$ *P p r*
and *step1*: ⟦ *step r* = *1*; *step* (*Suc r*) = *2*; *phase* (*Suc r*) = *phase r*;
next1 r p (*rho r p*)
(*HOrcvdMsgs LV-M r p* (*HOs r p*) (*rho r*))
(*coords* (*Suc r*) *p*) (*rho* (*Suc r*) *p*) ⟧
$\Longrightarrow$ *P p r*
and *step2*: ⟦ *step r* = *2*; *step* (*Suc r*) = *3*; *phase* (*Suc r*) = *phase r*;
next2 r p (*rho r p*)
(*HOrcvdMsgs LV-M r p* (*HOs r p*) (*rho r*))
(*coords* (*Suc r*) *p*) (*rho* (*Suc r*) *p*) ⟧
$\Longrightarrow$ *P p r*
and *step3*: ⟦ *step r* = *3*; *step* (*Suc r*) = *0*; *phase* (*Suc r*) = *Suc* (*phase r*);
next3 r p (*rho r p*)
(*HOrcvdMsgs LV-M r p* (*HOs r p*) (*rho r*))
(*coords* (*Suc r*) *p*) (*rho* (*Suc r*) *p*) ⟧
$\Longrightarrow$ *P p r*
shows *P p r*

by (*rule LV-Suc*[*OF run*])
(*auto intro*: *step0 step1 step2 step3*)

7.5 Boundedness and Monotonicity of Timestamps

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

lemma *LV-timestamp-bounded*:
assumes *run*: *CHORun LV-M rho HOs coords*
shows *timestamp* (*rho n p*) $\leq$ (*if step n* $<$ *2 then phase n else Suc* (*phase n*))
(**is** *?P p n*)
by (*rule LV-induct′* [*OF run*, **where** *P*=*?P*])
(*auto simp*: *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 *run*: *CHORun LV-M rho HOs coords*
shows *timestamp* (*rho n p*) $\leq$ *timestamp* (*rho* (*Suc n*) *p*)
(**is** *?P p n* **is** *?ts* $\leq$ -)
proof (*rule LV-Suc′*[*OF run*, **where** *P*=*?P*])

The case of *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 *stp*: *step n* = *1*
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*)
from *stp* **have** *?ts* $\leq$ *phase n*
using *LV-timestamp-bounded*[*OF run*, **where** *n*=*n*, **where** *p*=*p*] **by** *auto*
with *nxt* **show** *?thesis* **by** (*auto simp add*: *next1-def*)
qed (*auto simp add*: *next0-def next2-def next3-def*)

lemma *LV-timestamp-monotonic*:
assumes *run*: *CHORun LV-M rho HOs coords* **and** *le*: *m* $\leq$ *n*
shows *timestamp* (*rho m p*) $\leq$ *timestamp* (*rho n p*)
(**is** *?ts m* $\leq$ -)
proof −
from *le* **obtain** *k* **where** *k*: *n* = *m*+*k*
by (*auto simp add*: *le-iff-add*)
have *?ts m* $\leq$ *?ts* (*m*+*k*) (**is** *?P k*)
proof (*induct k*)
case *0* **show** *?P 0* **by** *simp*
next
fix *k*
assume *ih*: *?P k*
from *run* **have** *?ts* (*m*+*k*) $\leq$ *?ts* (*m* + *Suc k*)
by (*auto simp add*: *LV-timestamp-increasing*)

```
    with ih show ?P (Suc k) by simp
  qed
  with k show ?thesis by simp
qed
```

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 p) (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) − 1)
  let ?r1 = (4∗(phase r))
  have coordΦ (rho ?r1 p) = coords ?r1 p
  proof (cases phase r > 0)
    case False
```

```
    hence phase r = 0 by auto
    with run show ?thesis
      by (auto simp: LV-CHOMachine-def CHORun-eq CHOinitConfig-def
                     LV-initState-def)
  next
    case True
    hence step (Suc ?r0) = 0 by (auto simp: step-def)
    hence step ?r0 = 3 by (auto simp: mod-Suc step-def)
    moreover
    from run
    have LV-nextState ?r0 p (rho ?r0 p)
                 (HOrcvdMsgs LV-M ?r0 p (HOs ?r0 p) (rho ?r0))
                 (coords (Suc ?r0) p) (rho (Suc ?r0) p)
      by (auto simp: LV-CHOMachine-def CHORun-eq CHOnextConfig-eq)
    ultimately
    have nxt: next3 ?r0 p (rho ?r0 p)
                    (HOrcvdMsgs LV-M ?r0 p (HOs ?r0 p) (rho ?r0))
                    (coords (Suc ?r0) p) (rho (Suc ?r0) p)
      by (auto simp: LV-nextState-def)
    hence coordΦ (rho (Suc ?r0) p) = coords (Suc ?r0) p
      by (auto simp: next3-def)
    with True show ?thesis by auto
  qed
  moreover
  from run
  have coordΦ (rho (Suc (Suc (Suc ?r1))) p) = coordΦ (rho ?r1 p)
       ∧ coordΦ (rho (Suc (Suc ?r1)) p) = coordΦ (rho ?r1 p)
       ∧ coordΦ (rho (Suc ?r1) p) = coordΦ (rho ?r1 p)
    by (auto simp: notStep3EqualCoord step-def phase-def mod-Suc)
  moreover
  have r ∈ {?r1, Suc ?r1, Suc (Suc ?r1), Suc (Suc (Suc ?r1))}
    by (auto simp: step-def phase-def mod-Suc)
  ultimately
  show ?thesis by auto
qed
```

Votes only change at step 0.

```
lemma notStep0EqualVote [rule-format]:
  assumes run: CHORun LV-M rho HOs coords
  shows step r ≠ 0 ⟶ vote (rho (Suc r) p) = vote (rho r p) (is ?P p r)
  by (rule LV-Suc'[OF run, where P=?P])
     (auto simp: next0-def next1-def next2-def next3-def)
```

Commit status only changes at steps 0 and 3.

```
lemma notStep03EqualCommit [rule-format]:
  assumes run: CHORun LV-M rho HOs coords
  shows step r ≠ 0 ∧ step r ≠ 3 ⟶ commt (rho (Suc r) p) = commt (rho r p)
        (is ?P p r)
  by (rule LV-Suc'[OF run, where P=?P])
```

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

Timestamps only change at step 1.

lemma *notStep1EqualTimestamp* [*rule-format*]:
assumes *run*: *CHORun LV-M rho HOs coords*
shows $step\ r \neq 1 \longrightarrow timestamp\ (rho\ (Suc\ r)\ p) = timestamp\ (rho\ r\ p)$
(**is** *?P p r*)
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 $step\ r \neq 1 \longrightarrow x\ (rho\ (Suc\ r)\ p) = x\ (rho\ r\ p)$ (**is** *?P p r*)
by (*rule LV-Suc'*[*OF run*, **where** *P=?P*])
(*auto simp*: *next0-def next1-def next2-def next3-def*)

A process p has its *commt* 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*: *commt* (*rho r p*)
and *conds*: $[\![\ 1 \leq step\ r;\ coord\Phi\ (rho\ r\ p) = p;\ vote\ (rho\ r\ p) \neq None;$
$card\ \{q\ .\ coord\Phi\ (rho\ r\ q) = p\} > N\ div\ 2$
$]\!] \Longrightarrow A$
shows A
proof –
have $commt\ (rho\ r\ p) \longrightarrow$
$1 \leq step\ r$
$\wedge\ coord\Phi\ (rho\ r\ p) = p$
$\wedge\ vote\ (rho\ r\ p) \neq None$
$\wedge\ card\ \{q\ .\ coord\Phi\ (rho\ r\ q) = p\} > N\ div\ 2$
(**is** *?P p r* **is** - $\longrightarrow$ *?R r*)
proof (*rule LV-induct'*[*OF run*, **where** *P=?P*])
— the only interesting step is step 0
fix *n*
assume *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*)
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: $\neg\ commt\ (rho\ n\ p)$ **by** $simp$
  with $nxt\ cm'$
  have $coord\Phi\ (rho\ n\ p) = p$
      $\wedge\ vote\ (rho\ (Suc\ n)\ p) \neq None$
      $\wedge\ card\ (valStampsRcvd\ (HOrcvdMsgs\ LV\text{-}M\ n\ p\ (HOs\ n\ p)\ (rho\ n)))$
        $> N\ div\ 2$
   by $(auto\ simp\ add:\ next0\text{-}def)$
  moreover
  from stp
  have $valStampsRcvd\ (HOrcvdMsgs\ LV\text{-}M\ n\ p\ (HOs\ n\ p)\ (rho\ n))$
      $\subseteq \{q\ .\ coord\Phi\ (rho\ n\ q) = p\}$
   by $(auto\ simp:\ valStampsRcvd\text{-}def\ LV\text{-}CHOMachine\text{-}def$
        $HOrcvdMsgs\text{-}def\ \ LV\text{-}sendMsg\text{-}def\ send0\text{-}def)$
  hence $card\ (valStampsRcvd\ (HOrcvdMsgs\ LV\text{-}M\ n\ p\ (HOs\ n\ p)\ \ (rho\ n)))$
      $\leq card\ \{q\ .\ coord\Phi\ (rho\ n\ q) = p\}$
   by $(auto\ intro:\ card\text{-}mono)$
  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 $(auto\ simp:\ LV\text{-}CHOMachine\text{-}def\ LV\text{-}initState\text{-}def\ next1\text{-}def\ next2\text{-}def$
    $next3\text{-}def\ notStep3EqualCoord[OF\ run])$
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: $CHORun\ LV\text{-}M\ rho\ HOs\ coords$
  and ts: $timestamp\ (rho\ r\ p) = Suc\ (phase\ r)$
  and $conds$: $[\![\ 2 \leq step\ r;$
      $commt\ (rho\ r\ (coord\Phi\ (rho\ r\ p)));$
      $x\ (rho\ r\ p) = the\ (vote\ (rho\ r\ (coord\Phi\ (rho\ r\ p))))$
    $]\!] \Longrightarrow A$
  shows A
proof −
  let $?ts\ n = timestamp\ (rho\ n\ p)$
  let $?crd\ n = coord\Phi\ (rho\ n\ p)$
  have $?ts\ r = Suc\ (phase\ r) \longrightarrow$
      $2 \leq step\ r$

```
      ∧ 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 notStep03EqualCommit)
    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*: *step n = 2* **and** *stp'*: *step (Suc n) = 3*
and *ph*: *phase (Suc n) = phase n*
and *ih*: *?Q p n*
and *nxt*: *next2 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*
have *vt*: *vote (rho (Suc n) (?crd (Suc n))) = vote (rho n (?crd n))*
by *(auto simp add: notStep3EqualCoord notStep0EqualVote)*
from *run stp*
have *cmt*: *commt (rho (Suc n) (?crd (Suc n))) = commt (rho n (?crd n))*
by *(auto simp add: notStep3EqualCoord notStep03EqualCommit)*
with *vt ts ph stp stp' ih nxt*
show *?R (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-timestamp-bounded*).

fix *n*
assume *stp'*: *step (Suc n) = 0*
with *run LV-timestamp-bounded*[**where** *n=Suc n*]
have *?ts (Suc n)* $\leq$ *phase (Suc n)* **by** *auto*
thus *?Q p (Suc n)* **by** *simp*
qed
with *ts* **show** *?thesis* **by** *(intro conds) auto*
qed

If a process p has its *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 *readyE*:
assumes *run*: *CHORun LV-M rho HOs coords* **and** *rdy*: *ready (rho r p)*
and *conds*: ⟦ *step r = 3*; *coordΦ (rho r p) = p*;
card { q . coordΦ (rho r q) = p
$\wedge$ *timestamp (rho r q) = Suc (phase r) } > N div 2*
⟧ $\Longrightarrow$ *P*
shows *P*
proof −
let *?qs n = { q . coordΦ (rho n q) = p*
$\wedge$ *timestamp (rho n q) = Suc (phase n) }*

```
  have ready (rho r p) −→
          step r = 3
        ∧ coordΦ (rho r p) = p
        ∧ card (?qs r) > N div 2
    (is ?Q p r is - −→ ?R p r)
  proof (rule LV-induct'[OF run, where P=?Q])
    — the interesting case is step 2
    fix n
    assume stp: step n = 2 and stp': step (Suc n) = 3
       and ih: ?Q p n and ph: phase (Suc n) = phase n
       and nxt: next2 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 rdy: ready (rho (Suc n) p)
      from stp ih have nrdy: ¬ ready (rho n p) by simp
      with rdy nxt have coordΦ (rho n p) = p
        by (auto simp: next2-def)
      with run stp have coord: coordΦ (rho (Suc n) p) = p
        by (simp add: notStep3EqualCoord)
      let ?acks = acksRcvd (HOrcvdMsgs LV-M n p (HOs n p) (rho n))
      from nrdy rdy nxt have aRcvd: card ?acks > N div 2
        by (auto simp: next2-def)
      have ?acks ⊆ ?qs (Suc n)
      proof (clarify)
        fix q
        assume q: q ∈ ?acks
        with stp
        have n: coordΦ (rho n q) = p ∧ timestamp (rho n q) = Suc (phase n)
          by (auto simp: LV-CHOMachine-def HOrcvdMsgs-def LV-sendMsg-def
                      acksRcvd-def send2-def isAck-def)
        with run stp ph
        show coordΦ (rho (Suc n) q) = p
              ∧ timestamp (rho (Suc n) q) = Suc (phase (Suc n))
          by (simp add: notStep3EqualCoord notStep1EqualTimestamp)
      qed
      hence card ?acks ≤ card (?qs (Suc n))
        by (intro card-mono) auto
      with stp' coord aRcvd show ?R p (Suc n)
        by auto
    qed
    — the remaining steps are all solved trivially
  qed (auto simp: LV-CHOMachine-def LV-initState-def
                next0-def next1-def next3-def)
  with rdy show ?thesis by (blast intro: conds)
qed
```

A process decides only if the following conditions hold:

- it is at step 3,

- its coordinator votes for the value the process decides on,
- the coordinator has its *ready* and *commt* bits set.

lemma *decisionE*:
  assumes *run*: *CHORun LV-M rho HOs coords*
  and *dec*: *decide* (*rho* (*Suc r*) *p*) $\neq$ *decide* (*rho r p*)
  and *conds*: ⟦
    step r = *3*;
    decide (*rho* (*Suc r*) *p*) = *Some* (*the* (*vote* (*rho r* (*coordΦ* (*rho r p*)))));
    ready (*rho r* (*coordΦ* (*rho r p*))); *commt* (*rho r* (*coordΦ* (*rho r p*)))
  ⟧ $\Longrightarrow$ *P*
  shows *P*
proof –
  let *?cfg* = *rho r*
  let *?cfg′* = *rho* (*Suc r*)
  let *?crd p* = *coordΦ* (*?cfg p*)
  let *?dec′* = *decide* (*?cfg′ p*)

Except for the assertion about the *commt* 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* $\{q\ .\ ?crd\ q = ?crd\ p \wedge timestamp\ (?cfg\ q) = Suc\ (phase\ r)\}$
$> N\ div\ 2$ **by** (*rule readyE*)
— Hence there is at least one such process ...
hence *card* $\{q\ .\ ?crd\ q = ?crd\ p \wedge timestamp\ (?cfg\ q) = Suc\ (phase\ r)\} \neq 0$
by *arith*
then obtain *q* **where** *?crd q = ?crd p* **and** *timestamp (?cfg q) = Suc (phase r)*
by *auto*
— ... and by a previous lemma the coordinator must have committed.
with *run* **have** *commt (?cfg (?crd p))*
by (*auto elim*: *currentTimestampE*)
with *1* **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*: *CHORun LV-M rho HOs coords*
and *inv*: $[\![\ range\ (x \circ (rho\ n)) \subseteq range\ (x \circ (rho\ 0));$
$range\ (vote \circ (rho\ n)) \subseteq \{None\} \cup Some\ `\ range\ (x \circ (rho\ 0));$
$range\ (decide \circ (rho\ n)) \subseteq \{None\} \cup Some\ `\ range\ (x \circ (rho\ 0))$
$]\!] \Longrightarrow A$
shows *A*
proof –
let $?x0 = range\ (x \circ rho\ 0)$
let $?x0opt = \{None\} \cup Some\ `\ ?x0$
have $range\ (x \circ rho\ n) \subseteq ?x0$
$\wedge\ range\ (vote \circ rho\ n) \subseteq ?x0opt$
$\wedge\ range\ (decide \circ rho\ n) \subseteq ?x0opt$
(**is** *?Inv n* **is** $?X\ n \wedge ?Vote\ n \wedge ?Decide\ n$)
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*

```
from run
show x (rho (Suc n) p) ∈ range (λq. x (rho 0 q)) (is ?P p n)
proof (rule LV-Suc'[where P=?P])
  — only step1 is of interest
  assume stp: step n = 1
     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 ?thesis
  proof (cases rho (Suc n) p = rho n p)
    case True
    with x show ?thesis by auto
  next
    case False
    with stp nxt have cmt: commt (rho n (coordΦ (rho n p)))
      and xp: x (rho (Suc n) p) = the (vote (rho n (coordΦ (rho n p))))
    by (auto simp: next1-def LV-CHOMachine-def HOrcvdMsgs-def
               LV-sendMsg-def send1-def isVote-def)
    from run cmt have vote (rho n (coordΦ (rho n p))) ≠ None
      by (rule commitE)
    moreover
    from vt have vote (rho n (coordΦ (rho n p))) ∈ ?x0opt
      by (auto simp add: image-def)
    moreover
    note xp
    ultimately
    show ?thesis by (force simp add: image-def)
  qed
  — the other steps don't change x
next
  assume step n = 0
  with run have x (rho (Suc n) p) = x (rho n p)
    by (simp add: notStep1EqualX)
  with x show ?thesis by auto
next
  assume step n = 2
  with run have x (rho (Suc n) p) = x (rho n p)
    by (simp add: notStep1EqualX)
  with x show ?thesis by auto
next
  assume step n = 3
  with run have x (rho (Suc n) p) = x (rho n p)
    by (simp add: notStep1EqualX)
  with x show ?thesis by auto
qed
qed
```

Proof of second conjunct

```
have vt': ?Vote (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*

```
        by (auto simp: image-def)
      ultimately
      show ?thesis by auto
    qed
    with v show ∃ q. v = x (rho 0 q) by auto
  qed
```

Proof of third conjunct

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

Integrity now follows immediately.

```
theorem lv-integrity:
  assumes run: CHORun LV-M rho HOs coords
      and dec: decide (rho n p) = Some v
  shows ∃ q. v = x (rho 0 q)
proof −
  from run have decide (rho n p) ∈ {None} ∪ Some ‘ (range (x ∘ (rho 0)))
    by (rule lv-integrityInvariant) (auto simp: image-def)
  with dec show ?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: commitE)
  moreover
  from run cmt' have card {q . coordΦ (rho r q) = p'} > N div 2
    by (blast elim: commitE)
  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*: *CHORun LV-M rho HOs coords*
and *rdy*: *ready* (*rho r p*) **and** *rdy'*: *ready* (*rho r p'*)
shows $p = p'$
proof −
let *?C p* = {*q* . *coord*Φ (*rho r q*) = *p* ∧ *timestamp* (*rho r q*) = *Suc* (*phase r*)}
from *run rdy* **have** *card* (*?C p*) > *N div 2*
by (*blast elim*: *readyE*)
moreover
from *run rdy'* **have** *card* (*?C p'*) > *N div 2*
by (*blast elim*: *readyE*)
ultimately
obtain *q* **where** *coord*Φ (*rho r q*) = *p* **and** *p'* = *coord*Φ (*rho r q*)
by (*auto elim*: *majoritiesE'*)
thus *?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 *run*: *CHORun LV-M rho HOs coords*
and *maj*: *card* (*procsBeyondTS ts* (*rho r*)) > *N div 2*
and *cmt*: *commt* (*rho r p*)
shows ∃ *q* ∈ *procsBeyondTS ts* (*rho r*). *vote* (*rho r p*) = *Some* (*x* (*rho r q*))
(**is** *?Q r*)
proof −
let *?bynd n* = *procsBeyondTS ts* (*rho n*)
have *card* (*?bynd r*) > *N div 2* ∧ *commt* (*rho r p*) ⟶ *?Q r* (**is** *?P p r*)
proof (*rule LV-induct*[*OF run*])

next0 establishes the property

fix *n*
assume *stp*: *step n* = *0*
and *nxt*: ∀ *q*. *next0 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*)
from *nxt* **have** *nxp*: *?nxt p* **..**
show *?P p* (*Suc n*)
proof (*clarify*)
assume *mj*: *card* (*?bynd* (*Suc n*)) > *N div 2*
and *ct*: *commt* (*rho* (*Suc n*) *p*)
show *?Q* (*Suc n*)
proof −
let *?msgs* = *HOrcvdMsgs LV-M n p* (*HOs n p*) (*rho n*)
from *stp run* **have** ¬ *commt* (*rho n p*) **by** (*auto elim*: *commitE*)
with *nxp ct* **obtain** *q v* **where**
v: *?msgs q* = *Some* (*ValStamp v* (*highestStampRcvd ?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

next

We now prove that *next1* preserves the property. Observe that *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.

fix n
assume stp: $step\ n = 1$
and nxt: $\forall\, q.\ next1\ n\ q\ (rho\ n\ q)$
$(HOrcvdMsgs\ LV\text{-}M\ n\ q\ (HOs\ n\ q)\ (rho\ n))$
$(coords\ (Suc\ n)\ q)$
$(rho\ (Suc\ n)\ q)$
(**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': $commt\ (rho\ (Suc\ n)\ p)$
from $run\ stp\ ct'$ **have** ct: $commt\ (rho\ n\ p)$
by $(simp\ add\text{: } notStep03EqualCommit)$
from $run\ stp$ **have** $vote'$: $vote\ (rho\ (Suc\ n)\ p) = vote\ (rho\ n\ p)$
by $(simp\ add\text{: } notStep0EqualVote)$
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 $coord$: $commt\ (rho\ n\ (coord\Phi\ (rho\ n\ q)))$
by $(auto\ simp\text{: } next1\text{-}def\ send1\text{-}def\ LV\text{-}CHOMachine\text{-}def\ HOrcvdMsgs\text{-}def$
$LV\text{-}sendMsg\text{-}def\ isVote\text{-}def)$
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 $q1\ x'\ vote\ vote'$ **show** $?thesis$ **by** $auto$
next
case $False$

if no relevant process moves then *procsBeyondTS* doesn't change and we invoke the induction hypothesis

hence $bynd$: $?bynd\ (Suc\ n) =\ ?bynd\ n$

```
      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
  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′: commt (rho (Suc n) p)
    from run stp ct′ have ct: commt (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)
    hence bynd′: ?bynd (Suc n) = ?bynd n
      by (auto simp add: procsBeyondTS-def)
    from run stp have ∀ q. x (rho (Suc n) q) = x (rho n q)
      by (simp add: notStep1EqualX)
    with bynd′ vote′ ct mj′ ih show ?Q (Suc n)
      by auto
  qed
```

the initial state and the *step3* transition are trivial because the *commt* flag cannot be set.

```
qed (auto elim: commitE[OF run])
with maj cmt show ?thesis by simp
```

qed

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 *run*: *CHORun LV-M rho HOs coords*
    and *dec*: *decide* (*rho* (*Suc r*) *p*) ≠ *decide* (*rho r p*)
  shows ∀ *q* ∈ *procsBeyondTS* (*Suc* (*phase r*)) (*rho* (*r*+*k*)).
      x (*rho* (*r*+*k*) *q*) = *the* (*decide* (*rho* (*Suc r*) *p*))
  (**is** ∀ *q* ∈ *?bynd k*. - = *?v* **is** *?P p k*)
proof (*induct k*)
  — base step
  show *?P p 0*
  proof (*clarify*)
    fix *q*
    assume *q*: *q* ∈ *?bynd 0*

use preceding lemmas about the decision value and the x field of processes with fresh timestamps

    from *run dec*
    have *stp*: *step r* = *3*
     and *v*: *decide* (*rho* (*Suc r*) *p*) = *Some* (*the* (*vote* (*rho r* (*coord*Φ (*rho r p*)))))
     and *cmt*: *commt* (*rho r* (*coord*Φ (*rho r p*)))
     by (*auto elim*: *decisionE*)
    from *stp LV-timestamp-bounded*[*OF run*, **where** *n*=*r*]
    have *timestamp* (*rho r q*) ≤ *Suc* (*phase r*) **by** *simp*
    with *q* **have** *timestamp* (*rho r q*) = *Suc* (*phase r*)
     by (*simp add*: *procsBeyondTS-def*)
    with *run*
    have *x*: *x* (*rho r q*) = *the* (*vote* (*rho r* (*coord*Φ (*rho r q*))))
     and *cmt'*: *commt* (*rho r* (*coord*Φ (*rho r q*)))
     by (*auto elim*: *currentTimestampE*)
    from *run cmt cmt'* **have** *coord*Φ (*rho r p*) = *coord*Φ (*rho r q*)
     by (*rule committedProcsEqual*)
    with *x v* **show** *x* (*rho* (*r*+*0*) *q*) = *?v* **by** *simp*
  qed
  next
  — induction step
  fix *k*
  assume *ih*: *?P p k*
  show *?P p* (*Suc k*)
  proof (*clarify*)
    fix *q*
    assume *q*: *q* ∈ *?bynd* (*Suc k*)
    — distinguish the kind of transition—only *step1* is interesting
    have *x* (*rho* (*Suc* (*r* + *k*)) *q*) = *?v* (**is** *?X q* (*r*+*k*))
    proof (*rule LV-Suc'*[*OF run*, **where** *P*=*?X*])
     assume *stp*: *step* (*r* + *k*) = *1*

and *nxt*: *next1* $(r+k)$ *q* (*rho* $(r+k)$ *q*)
(*HOrcvdMsgs LV-M* $(r+k)$ *q* (*HOs* $(r+k)$ *q*) (*rho* $(r+k)$))
(*coords* (*Suc* $(r+k)$) *q*)
(*rho* (*Suc* $(r+k)$) *q*)
show *?thesis*
proof (*cases rho* (*Suc* $(r+k)$) *q* = *rho* $(r+k)$ *q*)
case *True*
with *q ih* **show** *?thesis* **by** (*auto simp*: *procsBeyondTS-def*)
next
case *False*
from *run dec* **have** *card* (*?bynd 0*) > *N div 2*
by (*simp add*: *decisionThenMajorityBeyondTS*)
moreover
have *?bynd 0* ⊆ *?bynd k*
by (*auto elim*: *procsBeyondTS-monotonic*[*OF run*])
hence *card* (*?bynd 0*) ≤ *card* (*?bynd k*)
by (*auto intro*: *card-mono*)
ultimately
have *maj*: *card* (*?bynd k*) > *N div 2* **by** *simp*
let *?crd* = *coord*Φ (*rho* $(r+k)$ *q*)
from *False stp nxt* **have**
cmt: *commt* (*rho* $(r+k)$ *?crd*) **and**
x: *x* (*rho* (*Suc* $(r+k)$) *q*) = *the* (*vote* (*rho* $(r+k)$ *?crd*))
by (*auto simp*: *next1-def LV-CHOMachine-def HOrcvdMsgs-def*
LV-sendMsg-def send1-def isVote-def)
from *run maj cmt stp* **obtain** *q'*
where *q1'*: *q'* ∈ *?bynd k*
and *q2'*: *vote* (*rho* $(r+k)$ *?crd*) = *Some* (*x* (*rho* $(r+k)$ *q'*))
by (*blast dest*: *commitThenVoteRecent*)
with *x ih* **show** *?thesis* **by** *auto*
qed
next
— all other steps hold by induction hypothesis
assume *step* $(r+k) = 0$
with *run* **have** *x*: *x* (*rho* (*Suc* $(r+k)$) *q*) = *x* (*rho* $(r+k)$ *q*)
and *ts*: *timestamp* (*rho* (*Suc* $(r+k)$) *q*) = *timestamp* (*rho* $(r+k)$ *q*)
by (*auto simp*: *notStep1EqualX notStep1EqualTimestamp*)
from *ts q* **have** *q* ∈ *?bynd k*
by (*auto simp*: *procsBeyondTS-def*)
with *x ih* **show** *?thesis* **by** *auto*
next
assume *step* $(r+k) = 2$
with *run* **have** *x*: *x* (*rho* (*Suc* $(r+k)$) *q*) = *x* (*rho* $(r+k)$ *q*)
and *ts*: *timestamp* (*rho* (*Suc* $(r+k)$) *q*) = *timestamp* (*rho* $(r+k)$ *q*)
by (*auto simp*: *notStep1EqualX notStep1EqualTimestamp*)
from *ts q* **have** *q* ∈ *?bynd k*
by (*auto simp*: *procsBeyondTS-def*)
with *x ih* **show** *?thesis* **by** *auto*
next

```
      assume step (r+k) = 3
      with run have x: x (rho (Suc (r+k)) q) = x (rho (r+k) q)
        and ts: timestamp (rho (Suc (r+k)) q) = timestamp (rho (r+k) q)
        by (auto simp: notStep1EqualX notStep1EqualTimestamp)
      from ts q have q ∈ ?bynd k
        by (auto simp: procsBeyondTS-def)
      with x ih show ?thesis by auto
    qed
    thus x (rho (r + 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 run: CHORun LV-M rho HOs coords
  and p: decide (rho (Suc r) p) ≠ decide (rho r p)
  and q: decide (rho (Suc (r+k)) q) ≠ decide (rho (r+k) q)
  shows decide (rho (Suc (r+k)) q) = decide (rho (Suc r) p)
proof −
  let ?bynd k = procsBeyondTS (Suc (phase r)) (rho (r+k))
  let ?qcrd = coordΦ (rho (r+k) q)
  from run p have notNone: decide (rho (Suc r) p) ≠ None
    by (auto elim: decisionE)
  — process q decides on the vote of its coordinator
  from run q
  have dec: decide (rho (Suc (r+k)) q) = Some (the (vote (rho (r+k) ?qcrd)))
   and cmt: commt (rho (r+k) ?qcrd)
    by (auto elim: decisionE)
  — that vote is the x field of some process q′ with a recent timestamp
  from run p have card (?bynd 0) > N div 2
    by (simp add: decisionThenMajorityBeyondTS)
  moreover
  from run have ?bynd 0 ⊆ ?bynd k
    by (auto elim: procsBeyondTS-monotonic)
  hence card (?bynd 0) ≤ card (?bynd k)
    by (auto intro: card-mono)
  ultimately
  have maj: card (?bynd k) > N div 2 by simp
  from run maj cmt obtain q′
    where q′1: q′ ∈ ?bynd k
      and q′2: vote (rho (r+k) ?qcrd) = Some (x (rho (r+k) q′))
    by (auto dest: commitThenVoteRecent)
  — the x field of process q′ is the value p decided on
  from run p q′1
  have x (rho (r+k) q′) = the (decide (rho (Suc r) p))
    by (auto dest: XOfTimestampBeyondDecision)
  — which proves the assertion
```

with *dec q′2 notNone* **show** *?thesis* **by** *auto*
qed

A process that holds some decision v has decided v sometime in the past.

lemma *decisionNonNullThenDecided*:
  assumes *run*: *CHORun LV-M rho HOs coords*
    and *dec*: *decide* (*rho n p*) = *Some v*
  shows ∃ *m*<*n*. *decide* (*rho* (*Suc m*) *p*) ≠ *decide* (*rho m p*)
      ∧ *decide* (*rho* (*Suc m*) *p*) = *Some v*
proof −
  let *?dec k* = *decide* (*rho k p*)
  have (∀ *m*<*n*. *?dec* (*Suc m*) ≠ *?dec m* ⟶ *?dec* (*Suc m*) ≠ *Some v*)
      ⟶ *?dec n* ≠ *Some v*
   (**is** *?P n* **is** *?A n* ⟶ -)
  proof (*induct n*)
   from *run* **show** *?P 0*
    by (*auto simp*: *CHORun-eq LV-CHOMachine-def*
        CHOinitConfig-def LV-initState-def)
  next
   fix *n*
   assume *ih*: *?P n*
   show *?P* (*Suc n*)
   proof (*clarify*)
    assume *p*: *?A* (*Suc n*) **and** *v*: *?dec* (*Suc n*) = *Some v*
    from *p* **have** *?A n* **by** *simp*
    with *ih* **have** *?dec n* ≠ *Some v* **by** *simp*
    moreover
    from *p*
    have *?dec* (*Suc n*) ≠ *?dec n* ⟶ *?dec* (*Suc n*) ≠ *Some v* **by** *simp*
    ultimately
    have *?dec* (*Suc n*) ≠ *Some v* **by** *auto*
    with *v* **show** *False* **by** *simp*
   qed
  qed
  with *dec* **show** *?thesis* **by** *auto*
qed

Irrevocability and Agreement are straightforward consequences of the two preceding lemmas.

theorem *lv-irrevocability*:
  assumes *run*: *CHORun LV-M rho HOs coords*
    and *p*: *decide* (*rho m p*) = *Some v*
  shows *decide* (*rho* (*m*+*k*) *p*) = *Some v*
proof −
  from *run p* **obtain** *n* **where**
   n1: *n* < *m* **and**
   n2: *decide* (*rho* (*Suc n*) *p*) ≠ *decide* (*rho n p*) **and**
   n3: *decide* (*rho* (*Suc n*) *p*) = *Some v*
   by (*auto dest*: *decisionNonNullThenDecided*)

```
  have ∀ i. decide (rho (Suc (n+i)) p) = Some v (is ∀ i. ?dec i)
  proof
    fix i
    show ?dec i
    proof (induct i)
      from n3 show ?dec 0 by simp
    next
      fix j
      assume ih: ?dec j
      show ?dec (Suc j)
      proof (rule ccontr)
        assume ctr: ¬ (?dec (Suc j))
        with ih
        have decide (rho (Suc (n + Suc j)) p) ≠ decide (rho (n + Suc j) p)
          by simp
        with run n2
        have decide (rho (Suc (n + Suc j)) p) = decide (rho (Suc n) p)
          by (rule laterProcessDecidesSameValue)
        with ctr n3 show False by simp
      qed
    qed
  qed
  moreover
  from n1 obtain j where m+k = Suc(n+j)
    by (auto dest: less-imp-Suc-add)
  ultimately
  show ?thesis by auto
qed

theorem lv-agreement:
  assumes run: CHORun LV-M rho HOs coords
      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 k2: decide (rho (Suc k) p) = Some v
    by (auto dest: decisionNonNullThenDecided)
  from run q obtain l
    where l1: decide (rho (Suc l) q) ≠ decide (rho l q)
      and l2: decide (rho (Suc l) q) = Some w
    by (auto dest: decisionNonNullThenDecided)
  show ?thesis
  proof (cases k ≤ l)
    case True
    then obtain m where m: l = k+m by (auto simp: le-iff-add)
    from run k1 l1 m
    have decide (rho (Suc l) q) = decide (rho (Suc k) p)
```

```
    by (auto elim: laterProcessDecidesSameValue)
   with k2 l2 show ?thesis by simp
 next
   case False
   hence l ≤ k by simp
   then obtain m where m: k = l+m by (auto simp: le-iff-add)
   from run l1 k1 m
   have decide (rho (Suc k) p) = decide (rho (Suc l) q)
     by (auto elim: laterProcessDecidesSameValue)
   with l2 k2 show ?thesis by simp
 qed
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: CHORun LV-M rho HOs coords
     and commG:CHOcommGlobal LV-M HOs coords
 shows ∃ r. ∀ p. decide (rho r p) ≠ None
proof −
```

The communication predicate implies the existence of a "successful" phase *ph*, coordinated by some process *c* for all processes.

```
 from commG obtain ph c
   where c: ∀ p. 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: ∀ p. c ∈ HOs (4∗ph+1) p
   and rcv3: ∀ p. c ∈ HOs (4∗ph+3) p
   by (auto simp: LV-CHOMachine-def LV-commGlobal-def)
 let ?r0 = 4∗ph
 let ?r1 = Suc ?r0
 let ?r2 = Suc ?r1
 let ?r3 = Suc ?r2
 let ?r4 = Suc ?r3
```

Process *c* is the coordinator of all steps of phase *ph*.

```
 from run c have c':∀ p. coordΦ (rho ?r p) = c
   by (auto simp add: phase-def coordinators)
 with run have c1: ∀ p. coordΦ (rho ?r1 p) = c
   by (auto simp add: step-def mod-Suc notStep3EqualCoord)
 with run have c2: ∀ p. coordΦ (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: commt (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 commt (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
  fix *p*
  let *?msgs* = *HOrcvdMsgs LV-M ?r3 p* (*HOs ?r3 p*) (*rho ?r3*)
  let *?crd* = *coordΦ* (*rho ?r3 p*)
  from *run 3 c3 rcv3*
  have *cnd*: *?msgs ?crd* ≠ *None* ∧ *isVote* (*the* (*?msgs ?crd*))
    by (*auto elim*: *readyE*
      simp: *step-def mod-Suc LV-CHOMachine-def HOrcvdMsgs-def*
        LV-sendMsg-def send3-def isVote-def numeral-3-eq-3)
  show *decide* (*rho ?r4 p*) ≠ *None* (**is** *?P p* (*Suc* (*Suc* (*Suc* (*4*ph*)))))
  proof (*rule LV-Suc'*[*OF run*, **where** *P*=*?P*], *auto simp*: *step-def mod-Suc*)
    assume *next3 ?r3 p* (*rho ?r3 p*) *?msgs* (*coords* (*Suc ?r3*) *p*) (*rho ?r4 p*)
    with *cnd* **show** ∃ *v*. *decide* (*rho ?r4 p*) = *Some v*
      by (*auto simp*: *next3-def*)
  qed
qed

This immediately proves the assertion.

from *4* **show** *?thesis* **..**
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 *lv-consensus*:
  assumes *run*: *CHORun LV-M rho HOs coords*
    and *commG*: *CHOcommGlobal LV-M HOs coords*
  shows *consensus* (*x* ∘ (*rho 0*)) *decide rho*
proof −
  — the above statement of termination is stronger than what we need
  from *lv-termination*[*OF assms*]
  obtain *r* **where** ∀ *p*. *decide* (*rho r p*) ≠ *None* **..**
  hence ∀ *p*. ∃ *r*. *decide* (*rho r p*) ≠ *None* **by** *blast*
  with *lv-integrity*[*OF run*] *lv-agreement*[*OF run*]
  show *?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 *lv-consensus-fg*:
  assumes *run*: *fg-run LV-M rho HOs HOs coords*
    and *commG*: *CHOcommGlobal LV-M HOs coords*
  shows *consensus* (λ*p*. *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 LV-M crun HOs HOs coords*

and *init*: *crun 0* = *state* (*rho 0*)
from *crun* **have** *CHORun LV-M crun HOs coords*
by (*unfold CHORun-def SHORun-def*)
from *this commG* **have** *consensus* ($x \circ$ (*crun 0*)) *decide crun*
by (*rule lv-consensus*)
with *init* **show** *consensus ?inits decide crun*
by (*simp add*: *o-def*)
qed

end
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 *UniformVoting*. The parameters T, E, and α 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 *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
$N \equiv$ *card* (*UNIV*::*Proc set*) — number of processes

The algorithm proceeds in *phases* of 2 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.

abbreviation
nSteps $\equiv$ *2*
definition *phase* **where** *phase* (*r*::*nat*) $\equiv$ *r div nSteps*
definition *step* **where** *step* (*r*::*nat*) $\equiv$ *r mod nSteps*

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 $\mathcal{U}_{T,E,\alpha}$ algorithm and their constraints [3].

locale *ute-parameters* =
fixes α::*nat* **and** T::*nat* **and** E::*nat*
assumes *majE*: $2*E \geq N + 2*\alpha$
and *majT*: $2*T \geq N + 2*\alpha$
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.

definition
  $send0 :: nat \Rightarrow Proc \Rightarrow Proc \Rightarrow {'val}\ pstate \Rightarrow {'val}\ msg$
where
  $send0\ r\ p\ q\ st \equiv Val\ (x\ st)$

definition
  $next0 :: nat \Rightarrow Proc \Rightarrow {'val}\ pstate \Rightarrow (Proc \Rightarrow {'val}\ msg\ option)$
  $\Rightarrow {'val}\ pstate \Rightarrow bool$
where
  $next0\ r\ p\ st\ msgs\ st' \equiv$
  $(\exists v.\ card\ \{q.\ msgs\ q = Some\ (Val\ v)\} > T \wedge st' = st\ (\!|\ vote := Some\ v\ |\!))$
  $\vee \neg(\exists v.\ card\ \{q.\ msgs\ q = Some\ (Val\ v)\} > T) \wedge st' = st\ (\!|\ vote := None\ |\!)$

In step 1, each process sends its current *vote*.

If it receives more than α 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.

definition
  $send1 :: nat \Rightarrow Proc \Rightarrow Proc \Rightarrow {'val}\ pstate \Rightarrow {'val}\ msg$
where
  $send1\ r\ p\ q\ st \equiv Vote\ (vote\ st)$

definition
  $next1 :: nat \Rightarrow Proc \Rightarrow {'val}\ pstate \Rightarrow (Proc \Rightarrow {'val}\ msg\ option)$
  $\Rightarrow {'val}\ pstate \Rightarrow bool$
where
  $next1\ r\ p\ st\ msgs\ st' \equiv$
  $(\ (\exists v.\ card\ \{q.\ msgs\ q = Some\ (Vote\ (Some\ v))\} > \alpha \wedge x\ st' = v)$
  $\vee \neg(\exists v.\ card\ \{q.\ msgs\ q = Some\ (Vote\ (Some\ v))\} > \alpha)$
  $\wedge\ x\ st' = undefined\ \ )$
  $\wedge\ (\ (\exists v.\ card\ \{q.\ msgs\ q = Some\ (Vote\ (Some\ v))\} > E \wedge decide\ st' = Some\ v)$
  $\vee \neg(\exists v.\ card\ \{q.\ msgs\ q = Some\ (Vote\ (Some\ v))\} > E)$
  $\wedge\ decide\ st' = decide\ st\ \ )$
  $\wedge\ vote\ st' = None$

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

definition
  $Ute\text{-}sendMsg :: nat \Rightarrow Proc \Rightarrow Proc \Rightarrow {'val}\ pstate \Rightarrow {'val}\ msg$
where
  $Ute\text{-}sendMsg\ (r{::}nat) \equiv if\ step\ r = 0\ then\ send0\ r\ else\ send1\ r$

definition
  $Ute\text{-}nextState :: nat \Rightarrow Proc \Rightarrow {'val}\ pstate \Rightarrow (Proc \Rightarrow {'val}\ msg\ option)$
  $\Rightarrow {'val}\ pstate \Rightarrow bool$
where
  $Ute\text{-}nextState\ r \equiv if\ step\ r = 0\ then\ next0\ r\ else\ next1\ r$

8.2 Communication Predicate for $\mathcal{U}_{T,E,\alpha}$

Following [3], we now define the communication predicate for the $\mathcal{U}_{T,E,\alpha}$ algorithm to be correct.

The round-by-round predicate stipulates the following conditions:

- no process may receive more than α 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 α correct messages, but this is implied, since $T > \alpha$ (cf. lemma *alpha-lt-T*).

definition *Ute-commPerRd* **where**
$\quad$ *Ute-commPerRd HOrs SHOrs* $\equiv$
$\quad \forall\, p.\ card\ (HOrs\ p - SHOrs\ p) \le \alpha$
$\quad\quad \wedge\ card\ (SHOrs\ p \cap HOrs\ p) > N + 2{*}\alpha - E - 1$
$\quad\quad \wedge\ card\ (SHOrs\ p \cap HOrs\ p) > T$

The global communication predicate requires there exists some phase Φ such that:

- all HO and SHO sets of all processes are equal in the second step of phase Φ, 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.

definition *Ute-commGlobal* **where**
$\quad$ *Ute-commGlobal HOs SHOs* $\equiv$
$\quad \exists\, \Phi.\ (let\ r = Suc\ (nSteps{*}\Phi)$
$\quad\quad in\ \ (\exists\, \pi.\ \forall\, p.\ \pi = HOs\ r\ p \wedge \pi = SHOs\ r\ p)$
$\quad\quad\quad \wedge\ (\forall\, p.\ card\ (SHOs\ (Suc\ r)\ p \cap HOs\ (Suc\ r)\ p) > T)$
$\quad\quad\quad \wedge\ (\forall\, p.\ card\ (SHOs\ (Suc\ (Suc\ r))\ p \cap HOs\ (Suc\ (Suc\ r))\ p) > E))$

8.3 The $\mathcal{U}_{T,E,\alpha}$ Heard-Of Machine

We now define the coordinated HO machine for the $\mathcal{U}_{T,E,\alpha}$ algorithm by assembling the algorithm definition and its communication-predicate.

definition *Ute-SHOMachine* **where**
$\quad$ *Ute-SHOMachine* $= ($|

CinitState = (λ *p st crd*. *Ute-initState p st*),
sendMsg = *Ute-sendMsg*,
CnextState = (λ *r p st msgs crd 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 *nxt*: *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 ccontr*)
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 $qq \in \{qq::Proc.\ F\ qq = Some\ m\}$
and $qq \in \{qq::Proc.\ F\ qq = Some\ m'\}$
by (*rule majoritiesE'*)
thus *?thesis* **by** *auto*
qed

lemma *unique-majority-E-α*:
assumes *majv*: *card* $\{qq::Proc.\ F\ qq = m\} > E - \alpha$
and *majw*: *card* $\{qq::Proc.\ F\ qq = m'\} > E - \alpha$
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 $qq \in \{qq::Proc.\ F\ qq = m\}$
and $qq \in \{qq::Proc.\ F\ qq = m'\}$
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 $qq \in \{qq::Proc.\ F\ qq = Some\ m\}$
and $qq \in \{qq::Proc.\ F\ qq = Some\ m'\}$
by (*rule majoritiesE'*)
thus *?thesis* **by** *auto*

qed

No two processes may vote for different values in the same round.

lemma *common-vote*:
  assumes *usafe*: *SHOcommPerRd Ute-M HO SHO*
  and *nxtp*: *nextState Ute-M r p* (*rho r p*) μp (*rho* (*Suc r*) *p*)
  and *mup*: $\mu p \in$ *SHOmsgVectors Ute-M r p* (*rho r*) (*HO p*) (*SHO p*)
  and *nxtq*: *nextState Ute-M r q* (*rho r q*) μq (*rho* (*Suc r*) *q*)
  and *muq*: $\mu q \in$ *SHOmsgVectors Ute-M r q* (*rho r*) (*HO q*) (*SHO q*)
  and *vp*: *vote* (*rho* (*Suc r*) *p*) = *Some vp*
  and *vq*: *vote* (*rho* (*Suc r*) *q*) = *Some vq*
  shows *vp* = *vq*
using *assms* **proof** −
  have *gtn*: *card* {*qq*. *sendMsg Ute-M r qq p* (*rho r qq*) = *Val vp*}
      + *card* {*qq*. *sendMsg Ute-M r qq q* (*rho r qq*) = *Val vq*} > *N*
  proof −
    have *card* {*qq*. *sendMsg Ute-M r qq p* (*rho r qq*) = *Val vp*} > *T* − α
      $\wedge$ *card* {*qq*. *sendMsg Ute-M r qq q* (*rho r qq*) = *Val vq*} > *T* − α
     (**is** *card ?vsentp* > - $\wedge$ *card ?vsentq* > -)
    proof −
      from *nxtp vp* **have** *stp*:*step r* = *0* **by** (*auto simp*: *vote-step*)
      from *mup*
      have {*qq*. μp *qq* = *Some* (*Val vp*)} − (*HO p* − *SHO p*)
          $\subseteq$ {*qq*. *sendMsg Ute-M r qq p* (*rho r qq*) = *Val vp*}
          (**is** *?vrcvdp* − *?ahop* $\subseteq$ *?vsentp*)
        by (*auto simp*: *SHOmsgVectors-def*)
      hence *card* (*?vrcvdp* − *?ahop*) $\leq$ *card ?vsentp*
        and *card* (*?vrcvdp* − *?ahop*) $\geq$ *card ?vrcvdp* − *card ?ahop*
        by (*auto simp*: *card-mono diff-card-le-card-Diff*)
      hence *card ?vsentp* $\geq$ *card ?vrcvdp* − *card ?ahop* **by** *auto*
      moreover
      from *nxtp stp* **have** *next0 r p* (*rho r p*) μp (*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 vq*)} − (*HO q* − *SHO q*)
          $\subseteq$ {*qq*. *sendMsg Ute-M r qq q* (*rho r qq*) = *Val vq*}
          (**is** *?vrcvdq* − *?ahoq* $\subseteq$ *?vsentq*)
        by (*auto simp*: *SHOmsgVectors-def*)
      hence *card* (*?vrcvdq* − *?ahoq*) $\leq$ *card ?vsentq*
        and *card* (*?vrcvdq* − *?ahoq*) $\geq$ *card ?vrcvdq* − *card ?ahoq*
        by (*auto simp*: *card-mono diff-card-le-card-Diff*)
      hence *card ?vsentq* $\geq$ *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 vq*)} > *T*

by (*unfold next0-def*, *auto*)
moreover
from *usafe* **have** *card ?ahop* $\leq \alpha$ **and** *card ?ahoq* $\leq \alpha$
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* $\neq$ *vq*
have $\forall$ *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*}
$\cap$ {*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*}
$\cup$ {*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*}
$\cup$ {*qq*. *sendMsg Ute-M r qq q* (*rho r qq*) = *Val vq*}) $\leq$ *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*) $\neq$ *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*)}
> $E - \alpha$
proof $-$
from *run* **obtain** μp
where *nxt*:*nextState Ute-M r p* (*rho r p*) μp (*rho* (*Suc r*) *p*)
and $\forall$ *qq*. *qq* $\in$ *HOs r p* $\longleftrightarrow$ μp *qq* $\neq$ *None*
and $\forall$ *qq*. *qq* $\in$ *SHOs r p* $\cap$ *HOs r p*
$\longrightarrow$ 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 $\{qq.\ \mu p\ qq = Some\ (Vote\ (Some\ v))\} - (HOs\ r\ p - SHOs\ r\ p)$
$\subseteq \{qq.\ sendMsg\ Ute\text{-}M\ r\ qq\ p\ (rho\ r\ qq) = Vote\ (Some\ v)\}$
(**is** $?vrcvdp - ?ahop \subseteq ?vsentp$) **by** *auto*
hence $card\ (?vrcvdp - ?ahop) \leq card\ ?vsentp$
and $card\ (?vrcvdp - ?ahop) \geq card\ ?vrcvdp - card\ ?ahop$
by (*auto simp*: *card-mono diff-card-le-card-Diff*)
hence $card\ ?vsentp \geq card\ ?vrcvdp - card\ ?ahop$ **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 *nxt* **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 *?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 α such messages.

lemma *common-x-argument-1*:
assumes *usafe*:$SHOcommPerRd\ Ute\text{-}M\ (HOs\ (Suc\ r))\ (SHOs\ (Suc\ r))$
and *threshold*: $card\ \{q.\ sendMsg\ Ute\text{-}M\ (Suc\ r)\ q\ p\ (rho\ (Suc\ r)\ q)$
$= Vote\ (Some\ v)\} > E - \alpha$
(**is** $card\ (?msgs\ p\ v) > \text{-}$)
shows $card\ (?msgs\ pp\ v \cap (SHOs\ (Suc\ r)\ pp \cap HOs\ (Suc\ r)\ pp)) > \alpha$
proof −
have $card\ (?msgs\ pp\ v) + card\ (SHOs\ (Suc\ r)\ pp \cap HOs\ (Suc\ r)\ pp) > N + \alpha$
proof −
have $\forall\, q.\ sendMsg\ Ute\text{-}M\ (Suc\ r)\ q\ p\ (rho\ (Suc\ r)\ q)$
$= sendMsg\ Ute\text{-}M\ (Suc\ r)\ q\ pp\ (rho\ (Suc\ r)\ q)$
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 *?thesis* **using** *threshold* **by** *auto*
qed
moreover
have $card\ (?msgs\ pp\ v) + card\ (SHOs\ (Suc\ r)\ pp \cap HOs\ (Suc\ r)\ pp)$
$= card\ (?msgs\ pp\ v \cup (SHOs\ (Suc\ r)\ pp \cap HOs\ (Suc\ r)\ pp))$

+ *card* (*?msgs pp v* ∩ (*SHOs* (*Suc r*) *pp* ∩ *HOs* (*Suc r*) *pp*))
by (*auto intro*: *card-Un-Int*)
moreover
have *card* (*?msgs pp v* ∪ (*SHOs* (*Suc r*) *pp* ∩ *HOs* (*Suc r*) *pp*)) ≤ *N*
by (*auto simp*: *card-mono*)
ultimately
show *?thesis* **by** *auto*
qed

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 *usafe*: ∀ *r*. *SHOcommPerRd Ute-M* (*HOs r*) (*SHOs r*)
and *nxtpp*: *nextState Ute-M* (*Suc r*) *pp* (*rho* (*Suc r*) *pp*)
μpp (*rho* (*Suc* (*Suc r*)) *pp*)
and *mupp*: *μpp* ∈ *SHOmsgVectors Ute-M* (*Suc r*) *pp* (*rho* (*Suc r*))
(*HOs* (*Suc r*) *pp*) (*SHOs* (*Suc r*) *pp*)
and *threshold*: *card* {*q*. *sendMsg Ute-M* (*Suc r*) *q p* (*rho* (*Suc r*) *q*)
= *Vote* (*Some v*)} > *E* − *α*
(**is** *card* (*?sent p v*) > -)
shows *x* (*rho* (*Suc* (*Suc r*)) *pp*) = *v*
proof −
have *stp*:*step* (*Suc r*) ≠ *0*
proof
assume *sr*: *step* (*Suc r*) = *0*
hence ∀ *q*. *sendMsg Ute-M* (*Suc r*) *q p* (*rho* (*Suc r*) *q*)
= *Val* (*x* (*rho* (*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*: *card* {*qq*. *μpp qq* = *Some* (*Vote* (*Some v*))} > *α*
(**is** *card* (*?msgs v*) > *α*)
proof −
from *mupp*
have *SHOs* (*Suc r*) *pp* ∩ *HOs* (*Suc r*) *pp*
⊆ {*qq*. *μpp qq* = *Some* (*sendMsg Ute-M* (*Suc r*) *qq pp* (*rho* (*Suc r*) *qq*))}
unfolding *SHOmsgVectors-def* **by** *auto*
moreover
hence (*?msgs v*) ⊇ (*?sent pp v*) ∩ (*SHOs* (*Suc r*) *pp* ∩ *HOs* (*Suc r*) *pp*)
by *auto*
hence *card* (*?msgs v*)
≥ *card* ((*?sent pp v*) ∩ (*SHOs* (*Suc r*) *pp* ∩ *HOs* (*Suc r*) *pp*))

by (*auto intro*: *card-mono*)
moreover
from *usafe threshold*
have *alph*:*card* ((*?sent pp v*) $\cap$ (*SHOs* (*Suc r*) *pp* $\cap$ *HOs* (*Suc r*) *pp*)) $> \alpha$
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*) μ*pp* (*rho* (*Suc* (*Suc r*)) *pp*)
by (*auto simp*: *Ute-SHOMachine-def nextState-def Ute-nextState-def*)
ultimately
obtain *w* where *wa*:*card* (*?msgs w*) $> \alpha$ 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* $\in$ *SHOs* (*Suc r*) *pp* and μ*pp qv* = *Some* (*Vote* (*Some v*))
proof −
have $\neg$ (*?msgs v* $\subseteq$ *HOs* (*Suc r*) *pp* − *SHOs* (*Suc r*) *pp*)
proof
assume *?msgs v* $\subseteq$ *HOs* (*Suc r*) *pp* − *SHOs* (*Suc r*) *pp*
hence *card* (*?msgs v*) $\leq$ *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*) $\leq \alpha$
by (*auto simp*: *Ute-SHOMachine-def Ute-commPerRd-def*)
moreover
note *va*
ultimately
show *False* by *auto*
qed
then obtain *qv*
where *qv* $\notin$ *HOs* (*Suc r*) *pp* − *SHOs* (*Suc r*) *pp*
and *qsv*:μ*pp qv* = *Some* (*Vote* (*Some v*))
by *auto*
with *mupp* have *qv* $\in$ *SHOs* (*Suc r*) *pp*
unfolding *SHOmsgVectors-def* by *auto*
with *qsv 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 $\in$ *SHOs* (*Suc r*) *pp* **and** μ*pp qw* = *Some* (*Vote* (*Some w*))
proof −
have $\neg$ (*?msgs w* $\subseteq$ *HOs* (*Suc r*) *pp* − *SHOs* (*Suc r*) *pp*)
proof
assume *?msgs w* $\subseteq$ *HOs* (*Suc r*) *pp* − *SHOs* (*Suc r*) *pp*
hence *card* (*?msgs w*) $\leq$ *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*) $\leq \alpha$
by (*auto simp*: *Ute-SHOMachine-def Ute-commPerRd-def*)
moreover
note *wa*
ultimately
show *False* **by** *auto*
qed
then obtain *qw*
where *qw* $\notin$ *HOs* (*Suc r*) *pp* − *SHOs* (*Suc r*) *pp*
and *qsw*: μ*pp qw* = *Some* (*Vote* (*Some w*))
by *auto*
with *mupp* **have** *qw* $\in$ *SHOs* (*Suc r*) *pp*
unfolding *SHOmsgVectors-def* **by** *auto*
with *qsw that* **show** *?thesis* **by** *auto*
qed
with *stp mupp* **have** *vote* (*rho* (*Suc r*) *qw*) = *Some w*
by (*auto simp*: *Ute-SHOMachine-def SHOmsgVectors-def*
Ute-sendMsg-def send1-def)
moreover
from *run* **obtain** μ*qv* μ*qw*
where *nextState Ute-M r qv* ((*rho r*) *qv*) μ*qv* (*rho* (*Suc r*) *qv*)
and μ*qv* $\in$ *SHOmsgVectors Ute-M r qv* (*rho r*) (*HOs r qv*) (*SHOs r qv*)
and *nextState Ute-M r qw* ((*rho r*) *qw*) μ*qw* (*rho* (*Suc r*) *qw*)
and μ*qw* $\in$ *SHOmsgVectors Ute-M r qw* (*rho r*) (*HOs r qw*) (*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* (*rho* (*Suc* (*Suc r*)) *pp*) = *v* **by** *auto*
qed

Inductive argument for the agreement and validity theorems.

lemma *safety-inductive-argument*:
assumes *run*: *SHORun Ute-M rho HOs SHOs*
and *comm*: $\forall$ *r*. *SHOcommPerRd Ute-M* (*HOs r*) (*SHOs r*)
and *ih*: $E - \alpha <$ *card* {*q*. *sendMsg Ute-M r′ q p* (*rho r′ q*) = *Vote* (*Some v*)}
and *stp1*: *step r′* = *Suc 0*
shows $E - \alpha <$
card {*q*. *sendMsg Ute-M* (*Suc* (*Suc r′*)) *q p* (*rho* (*Suc* (*Suc r′*)) *q*)
= *Vote* (*Some v*)}

proof −
  from *stp1* **have** $r' > 0$ **by** (*auto simp*: *step-def*)
  with *stp1* **obtain** *r* **where** *rr'*:$r' = Suc\ r$ **and** *stpr*:*step* (*Suc r*) = *Suc 0*
    by (*auto dest*: *gr0-implies-Suc*)

  have ∀ *pp*. *x* (*rho* (*Suc* (*Suc r*)) *pp*) = *v*
  proof
    fix *pp*
    from *run* **obtain** *μpp*
      where *μpp* ∈ *SHOmsgVectors Ute-M r′ pp* (*rho r′*) (*HOs r′ pp*) (*SHOs r′ pp*)
      and *nextState Ute-M r′ pp* (*rho r′ pp*) *μpp* (*rho* (*Suc r′*) *pp*)
      by (*auto simp*: *Ute-SHOMachine-def SHORun-eq SHOnextConfig-eq*)
    with *run comm ih rr′* **show** *x* (*rho* (*Suc* (*Suc r*)) *pp*) = *v*
      by (*auto dest*: *common-x-argument-2*)
  qed
  with *run stpr*
  have ∀ *pp p*. *sendMsg Ute-M* (*Suc* (*Suc r*)) *pp p* (*rho* (*Suc* (*Suc r*)) *pp*) = *Val v*
    by (*auto simp*: *Ute-SHOMachine-def SHORun-eq SHOnextConfig-eq*
                Ute-sendMsg-def send0-def mod-Suc step-def)
  with *rr′*
  have ⋀*p μp′*. *μp′* ∈ *SHOmsgVectors Ute-M* (*Suc r′*) *p* (*rho* (*Suc r′*))
                                (*HOs* (*Suc r′*) *p*) (*SHOs* (*Suc r′*) *p*)
           ⟹ *SHOs* (*Suc r′*) *p* ∩ *HOs* (*Suc r′*) *p*
              ⊆ {*q*. *μp′ q* = *Some* (*Val v*)}
    by (*auto simp*: *SHOmsgVectors-def*)
  hence ⋀*p μp′*. *μp′* ∈ *SHOmsgVectors Ute-M* (*Suc r′*) *p* (*rho* (*Suc r′*))
                                (*HOs* (*Suc r′*) *p*) (*SHOs* (*Suc r′*) *p*)
           ⟹ *card* (*SHOs* (*Suc r′*) *p* ∩ *HOs* (*Suc r′*) *p*)
              ≤ *card* {*q*. *μp′ q* = *Some* (*Val v*)}
    by (*auto simp*: *card-mono*)
  moreover
  from *comm* **have** ⋀*p*. *T* < *card* (*SHOs* (*Suc r′*) *p* ∩ *HOs* (*Suc r′*) *p*)
    by (*auto simp*: *Ute-SHOMachine-def Ute-commPerRd-def*)
  ultimately
  have *vT*:⋀*p μp′*. *μp′* ∈ *SHOmsgVectors Ute-M* (*Suc r′*) *p* (*rho* (*Suc r′*))
                                (*HOs* (*Suc r′*) *p*) (*SHOs* (*Suc r′*) *p*)
             ⟹ *T* < *card* {*q*. *μp′ q* = *Some* (*Val v*)}
    by (*auto dest*: *less-le-trans*)

  show *?thesis*
  proof −
    have ∀ *pp*. *vote* ((*rho* (*Suc* (*Suc r′*))) *pp*) = *Some v*
    proof
      fix *pp*
      from *run* **obtain** *μpp*
        where *nxtpp*: *nextState Ute-M* (*Suc r′*) *pp* (*rho* (*Suc r′*) *pp*) *μpp*
                                (*rho* (*Suc* (*Suc r′*)) *pp*)
          and *mupp*: *μpp* ∈ *SHOmsgVectors Ute-M* (*Suc r′*) *pp* (*rho* (*Suc r′*))

(*HOs* (*Suc r'*) *pp*) (*SHOs* (*Suc r'*) *pp*)
    by (*auto simp*: *Ute-SHOMachine-def SHORun-eq SHOnextConfig-eq*)
  with *vT* **have** *vT'*:*card* {*q*. *μpp q* = *Some* (*Val v*)} > *T*
    by *auto*
  moreover
  from *stpr rr'* **have** *step* (*Suc r'*) = *0*
    by (*auto simp*: *mod-Suc step-def*)
  with *nxtpp*
  have *next0* (*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 *wT*:*card* {*q*. *μpp q* = *Some* (*Val w*)} > *T*
      and *votew*:*vote* (*rho* (*Suc* (*Suc r'*)) *pp*) = *Some w*
    by (*auto simp*: *next0-def*)
  from *vT' wT* **have** *v* = *w*
    by (*auto dest*: *unique-majority-T*)
  with *votew* **show** *vote* (*rho* (*Suc* (*Suc r'*)) *pp*) = *Some v* **by** *simp*
 qed
 with *run stpr rr'*
 have ∀ *p*. *N* = *card* {*q*. *sendMsg Ute-M* (*Suc* (*Suc* (*Suc r*))) *q p*
     ((*rho* (*Suc* (*Suc* (*Suc r*)))) *q*) = *Vote* (*Some v*)}
  by (*auto simp*: *Ute-SHOMachine-def SHORun-eq SHOnextConfig-eq*
     Ute-sendMsg-def send1-def step-def mod-Suc)
 with *rr' majE EltN* **show** *?thesis* **by** *auto*
qed
qed

A process that holds some decision *v* has decided *v* sometime in the past.

lemma *decisionNonNullThenDecided*:
 assumes *run*:*SHORun Ute-M rho HOs SHOs* **and** *dec*: *decide* (*rho n p*) = *Some v*
 shows ∃ *m*<*n*. *decide* (*rho* (*Suc m*) *p*) ≠ *decide* (*rho m p*)
   ∧ *decide* (*rho* (*Suc m*) *p*) = *Some v*
proof –
 let *?dec k* = *decide* ((*rho k*) *p*)
 have (∀ *m*<*n*. *?dec* (*Suc m*) ≠ *?dec m* ⟶ *?dec* (*Suc m*) ≠ *Some v*)
   ⟶ *?dec n* ≠ *Some v*
  (**is** *?P n* **is** *?A n* ⟶ -)
 proof (*induct n*)
  from *run* **show** *?P 0*
   by (*auto simp*: *Ute-SHOMachine-def SHORun-eq HOinitConfig-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

If process *p1* has decided value *v1* and process *p2* later decides, then *p2* must decide *v1*.

lemma *laterProcessDecidesSameValue*:
  assumes *run*:*SHORun Ute-M rho HOs SHOs*
  and *comm*:$\forall$ *r*. *SHOcommPerRd Ute-M* (*HOs r*) (*SHOs r*)
  and *dv1*:*decide* (*rho* (*Suc r*) *p1*) = *Some v1*
  and *dn2*:*decide* (*rho* (*r* + *k*) *p2*) $\neq$ *Some v2*
  and *dv2*:*decide* (*rho* (*Suc* (*r* + *k*)) *p2*) = *Some v2*
  shows *v2* = *v1*
proof −
  from *run dv1* **obtain** *r1*
    where *r1r*:*r1* < *Suc r*
      and *dn1*:*decide* (*rho r1 p1*) $\neq$ *Some v1*
      and *dv1'*:*decide* (*rho* (*Suc r1*) *p1*) = *Some v1*
    by (*auto dest*: *decisionNonNullThenDecided*)

  from *r1r* **obtain** *s* **where** *rr1*:*Suc r* = *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 *dn2'*: *decide* (*rho* (*r1* + *k'*) *p2*) $\neq$ *Some v2*
    and *dv2'*: *decide* (*rho* (*Suc* (*r1* + *k'*)) *p2*) = *Some v2*
    by *auto*

  from *run dn1 dv1' dn2' dv2'*
  have *rs0*:*step r1* = *Suc 0* **and** *rks0*:*step* (*r1* + *k'*) = *Suc 0*
    by (*auto simp*: *mod-Suc step-def dest*: *decide-step*)

  have *step* (*r1* + *k'*) = *step* (*step r1* + *k'*)
    unfolding *step-def* **by** (*simp add*: *mod-add-left-eq*)
  with *rs0 rks0* **have** *step k'* = *0* **by** (*auto simp*: *step-def mod-Suc*)
  then obtain *k''* **where** *k'* = *k''*∗*nSteps* **by** (*auto simp*: *step-def*)
  with *dn2' dv2'*
  have *dn2''*:*decide* (*rho* (*r1* + *k''*∗*nSteps*) *p2*) $\neq$ *Some v2*
    and *dv2''*:*decide* (*rho* (*Suc* (*r1* + *k''*∗*nSteps*)) *p2*) = *Some v2*
    by *auto*

  from *rs0* **have** *stp*:*step* (*r1* + *k''*∗*nSteps*) = *Suc 0*
    unfolding *step-def* **by** *auto*

  have *inv*:*card* {*q*. *sendMsg Ute-M* (*r1* + *k''*∗*nSteps*) *q p1* (*rho* (*r1* + *k''*∗*nSteps*) *q*)
                = *Vote* (*Some v1*)} > *E* − α
  proof (*induct k''*)
    from *stp* **have** *step* (*r1* + *0*∗*nSteps*) = *Suc 0*
      by (*auto simp*: *step-def*)
    from *run comm dn1 dv1'*
    show *card* {*q*. *sendMsg Ute-M* (*r1* + *0*∗*nSteps*) *q p1* (*rho* (*r1* + *0*∗*nSteps*) *q*)

$= Vote\ (Some\ v1)\} > E - \alpha$
  by (*intro decide-with-threshold-E*) *auto*
next
  fix k''
  assume *ih*: $E - \alpha <$
    $card\ \{q.\ sendMsg\ Ute\text{-}M\ (r1 + k''{*}nSteps)\ q\ p1\ (rho\ (r1 + k''{*}nSteps)\ q)$
      $= Vote\ (Some\ v1)\}$
  from *rs0* **have** *stps*: $step\ (r1 + k''{*}nSteps) = Suc\ 0$
    by (*auto simp*: *step-def*)
  with *run comm ih*
  have $E - \alpha <$
    $card\ \{q.\ sendMsg\ Ute\text{-}M\ (Suc\ (Suc\ (r1 + k''{*}nSteps)))\ q\ p1$
      $(rho\ (Suc\ (Suc\ (r1 + k''{*}nSteps)))\ q)$
      $= Vote\ (Some\ v1)\}$
    by (*rule safety-inductive-argument*)
  thus $E - \alpha <$
    $card\ \{q.\ sendMsg\ Ute\text{-}M\ (r1 + Suc\ k'' * nSteps)\ q\ p1$
      $(rho\ (r1 + Suc\ k'' * nSteps)\ q)$
      $= Vote\ (Some\ v1)\}$
    by *auto*
qed
moreover
from *run*
have $\forall q.\ sendMsg\ Ute\text{-}M\ (r1 + k''{*}nSteps)\ q\ p1\ (rho\ (r1 + k''{*}nSteps)\ q)$
  $= sendMsg\ Ute\text{-}M\ (r1 + k''{*}nSteps)\ q\ p2\ (rho\ (r1 + k''{*}nSteps)\ q)$
  by (*auto simp*: *Ute-SHOMachine-def Ute-sendMsg-def*
    step-def send0-def send1-def)
moreover
from *run comm dn2'' dv2''*
have $E - \alpha <$
  $card\ \{q.\ sendMsg\ Ute\text{-}M\ (r1 + k''{*}nSteps)\ q\ p2\ (rho\ (r1 + k''{*}nSteps)\ q)$
    $= Vote\ (Some\ v2)\}$
  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*: $\forall r.\ SHOcommPerRd\ Ute\text{-}M\ (HOs\ r)\ (SHOs\ r)$
  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*: $decide\ (rho\ (Suc\ k)\ p) \neq decide\ (rho\ k\ p)$

and *k2*: *decide* (*rho* (*Suc k*) *p*) = *Some v*
by (*auto dest*: *decisionNonNullThenDecided*)
from *run q* **obtain** *l*
where *l1*: *decide* (*rho* (*Suc l*) *q*) ≠ *decide* (*rho l q*)
and *l2*: *decide* (*rho* (*Suc l*) *q*) = *Some w*
by (*auto dest*: *decisionNonNullThenDecided*)
show *?thesis*
proof (*cases k* ≤ *l*)
case *True*
then obtain *m* **where** *m*: *l* = *k*+*m* **by** (*auto simp add*: *le-iff-add*)
from *run comm k2 l1 l2 m* **have** *w* = *v*
by (*auto elim*!: *laterProcessDecidesSameValue*)
thus *?thesis* **by** *simp*
next
case *False*
hence *l* ≤ *k* **by** *simp*
then obtain *m* **where** *m*: *k* = *l*+*m* **by** (*auto simp add*: *le-iff-add*)
from *run comm l2 k1 k2 m* **show** *?thesis*
by (*auto elim*!: *laterProcessDecidesSameValue*)
qed
qed

Main lemma for the proof of the Validity property.

lemma *validity-argument*:
assumes *run*: *SHORun Ute-M rho HOs SHOs*
and *comm*: ∀ *r*. *SHOcommPerRd Ute-M* (*HOs r*) (*SHOs r*)
and *init*: ∀ *p*. *x* ((*rho 0*) *p*) = *v*
and *dw*: *decide* (*rho r p*) = *Some w*
and *stp*: *step r′* = *Suc 0*
shows *card* {*q*. *sendMsg Ute-M r′ q p* (*rho r′ q*) = *Vote* (*Some v*)} > *E* − *α*
proof −
define *k* **where** *k* = *r′ div nSteps*
with *stp* **have** *stp*: *r′* = *Suc 0* + *k* ∗ *nSteps*
using *div-mult-mod-eq* [*of r′ nSteps*]
by (*simp add*: *step-def*)
moreover
have *E* − *α* <
card {*q*. *sendMsg Ute-M* (*Suc 0* + *k*∗*nSteps*) *q p* ((*rho* (*Suc 0* + *k*∗*nSteps*)) *q*)
= *Vote* (*Some v*)}
proof (*induct k*)
have ∀ *pp*. *vote* ((*rho* (*Suc 0*)) *pp*) = *Some v*
proof
fix *pp*
from *run* **obtain** *μpp*
where *nxtpp*:*nextState Ute-M 0 pp* (*rho 0 pp*) *μpp* (*rho* (*Suc 0*) *pp*)
and *mupp*:*μpp* ∈ *SHOmsgVectors Ute-M 0 pp* (*rho 0*) (*HOs 0 pp*) (*SHOs 0 pp*)
by (*auto simp*: *Ute-SHOMachine-def SHORun-eq SHOnextConfig-eq*)

have *majv*:*card* $\{q.\ \mu pp\ q = Some\ (Val\ v)\} > T$
proof −
from *run init* **have** $\forall q.\ sendMsg\ Ute\text{-}M\ 0\ q\ pp\ (rho\ 0\ q) = Val\ v$
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 $SHOs\ 0\ pp \cap HOs\ 0\ pp$
$\subseteq \{q.\ \mu pp\ q = Some\ (sendMsg\ Ute\text{-}M\ 0\ q\ pp\ (rho\ 0\ q))\}$
by (*auto simp*: *SHOmsgVectors-def*)
hence *card* $(SHOs\ 0\ pp \cap HOs\ 0\ pp)$
$\leq card\ \{q.\ \mu pp\ q = Some\ (sendMsg\ Ute\text{-}M\ 0\ q\ pp\ (rho\ 0\ q))\}$
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* $\{q.\ \mu pp\ q = Some\ (Val\ w)\} > T$
and *votew*:*vote* $(rho\ (Suc\ 0)\ pp) = Some\ w$
by (*auto simp*: *next0-def*)

from *majv majw* **have** $v = w$ **by** (*auto dest*: *unique-majority-T*)
with *votew* **show** *vote* $((rho\ (Suc\ 0))\ pp) = Some\ v$ **by** *simp*
qed
with *run*
have *card* $\{q.\ sendMsg\ Ute\text{-}M\ (Suc\ 0)\ q\ p\ (rho\ (Suc\ 0)\ q) = Vote\ (Some\ v)\} = N$
by (*auto simp*: *Ute-SHOMachine-def SHORun-eq SHOnextConfig-eq Ute-nextState-def step-def Ute-sendMsg-def send1-def*)
thus $E - \alpha <$
card $\{q.\ sendMsg\ Ute\text{-}M\ (Suc\ 0 + 0 * nSteps)\ q\ p\ (rho\ (Suc\ 0 + 0 * nSteps)\ q)$
$= Vote\ (Some\ v)\}$
using *majE EltN* **by** *auto*
next
fix *k*
assume *ih*:$E - \alpha <$
card $\{q.\ sendMsg\ Ute\text{-}M\ (Suc\ 0 + k * nSteps)\ q\ p\ (rho\ (Suc\ 0 + k * nSteps)\ q)$
$= Vote\ (Some\ v)\}$
have *step* $(Suc\ 0 + k * nSteps) = Suc\ 0$
by (*auto simp*: *mod-Suc step-def*)
from *run comm ih this*

have $E - \alpha <$
card $\{q.$ *sendMsg Ute-M* (*Suc* (*Suc* (*Suc 0* + *k* ∗ *nSteps*))) *q p*
(*rho* (*Suc* (*Suc* (*Suc 0* + *k* ∗ *nSteps*))) *q*)
= *Vote* (*Some v*)$\}$
by (*rule safety-inductive-argument*)
thus $E - \alpha <$
card $\{q.$ *sendMsg Ute-M* (*Suc 0* + *Suc k* ∗ *nSteps*) *q p*
(*rho* (*Suc 0* + *Suc k* ∗ *nSteps*) *q*)
= *Vote* (*Some v*)$\}$ **by** *simp*
qed
ultimately
show *?thesis* **by** *simp*
qed

The following theorem shows the Validity property of algorithm $\mathcal{U}_{T,E,\alpha}$.

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* (*Suc r1*)) *p*) = *Some w*
by (*force dest*: *decisionNonNullThenDecided*)
with *run* **have** *step r1* $\neq$ *0* **by** (*rule decide-step*)
hence *step r1* = *Suc 0* **by** (*simp add*: *step-def mod-Suc*)
with *assms*
have $E - \alpha <$
card $\{q.$ *sendMsg Ute-M r1 q p* (*rho r1 q*) = *Vote* (*Some v*)$\}$
by (*rule validity-argument*)
moreover
from *run comm dnr1 dwr1*
have *card* $\{q.$ *sendMsg Ute-M r1 q p* (*rho r1 q*) = *Vote* (*Some w*)$\} > E - \alpha$
by (*auto dest*: *decide-with-threshold-E*)
ultimately
show *v* = *w* **by** (*auto dest*: *unique-majority-E-*α)
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 α 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 π: ∀ p. HOs (Suc r) p = SHOs (Suc r) p
  and nxt: nextState Ute-M (Suc r) p (rho (Suc r) p) μ (rho (Suc (Suc r)) p)
  and mu: μ ∈ SHOmsgVectors Ute-M (Suc r) p (rho (Suc r))
                              (HOs (Suc r) p) (SHOs (Suc r) p)
  and vp: α < card {qq. μ qq = Some (Vote (Some v))}
  shows x ((rho (Suc (Suc r))) p) = v
proof −
  from nxt stp vp obtain wp
    where xwp:α < card {qq. μ qq = Some (Vote (Some wp))}
      and xp:x (rho (Suc (Suc r)) p) = wp
   by (auto simp: Ute-SHOMachine-def nextState-def Ute-nextState-def next1-def)

  have wp = v
  proof −
    from xwp obtain pp where smw:μ pp = Some (Vote (Some wp))
      by force
    have vote (rho (Suc r) pp) = Some wp
    proof −
      from smw mu π
      have μ pp = Some (sendMsg Ute-M (Suc r) pp p (rho (Suc r) pp))
        unfolding SHOmsgVectors-def by force
      with stp have μ pp = Some (Vote (vote (rho (Suc r) pp)))
        by (auto simp: Ute-SHOMachine-def Ute-sendMsg-def send1-def)
      with smw show ?thesis by auto
    qed
    moreover
    from vp obtain qq where smv:μ qq = Some (Vote (Some v))
      by force
    have vote (rho (Suc r) qq) = Some v
    proof −
      from smv mu π
      have μ qq = Some (sendMsg Ute-M (Suc r) qq p (rho (Suc r) qq))
        unfolding SHOmsgVectors-def by force
      with stp have μ qq = Some (Vote (vote (rho (Suc r) qq)))
        by (auto simp: Ute-SHOMachine-def Ute-sendMsg-def send1-def)
      with smv show ?thesis by auto
    qed
    moreover
    from run obtain μpp μqq
      where nextState Ute-M r pp (rho r pp) μpp (rho (Suc r) pp)
        and μpp ∈ SHOmsgVectors Ute-M r pp (rho r) (HOs r pp) (SHOs r pp)
        and nextState Ute-M r qq ((rho r) qq)  μqq (rho (Suc r) qq)
        and μqq ∈ SHOmsgVectors Ute-M r qq (rho r) (HOs r qq) (SHOs r qq)
      unfolding Ute-SHOMachine-def SHORun-eq SHOnextConfig-eq by blast
    ultimately
    show ?thesis using comm by (auto dest: common-vote)
  qed
  with xp show ?thesis by simp
```

qed

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 π: $\forall$ *p*. π*0* = *HOs* (*Suc r*) *p* $\wedge$ π*0* = *SHOs* (*Suc r*) *p*
  obtains *v* **where**
    $\bigwedge$*p* μ*p'* *q*.
      ⟦ *q* ∈ *SHOs* (*Suc* (*Suc r*)) *p* ∩ *HOs* (*Suc* (*Suc r*)) *p*;
        μ*p'* ∈ *SHOmsgVectors Ute-M* (*Suc* (*Suc r*)) *p* (*rho* (*Suc* (*Suc r*)))
                  (*HOs* (*Suc* (*Suc r*)) *p*) (*SHOs* (*Suc* (*Suc r*)) *p*)
      ⟧ $\Longrightarrow$ μ*p'* *q* = (*Some* (*Val v*))
proof −
  from π **have** *hosho*:$\forall$ *p*. *SHOs* (*Suc r*) *p* = *SHOs* (*Suc r*) *p* ∩ *HOs* (*Suc r*) *p*
    by *simp*

  have $\bigwedge$*p q*. *x* (*rho* (*Suc* (*Suc r*)) *p*) = *x* (*rho* (*Suc* (*Suc r*)) *q*)
  proof −
    fix *p q*
    from *run* **obtain** μ*p*
      where *nxt*: *nextState Ute-M* (*Suc r*) *p* (*rho* (*Suc r*) *p*)
                        μ*p* (*rho* (*Suc* (*Suc r*)) *p*)
        and *mu*: μ*p* ∈ *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** μ*q*
      where *nxtq*: *nextState Ute-M* (*Suc r*) *q* (*rho* (*Suc r*) *q*)
                        μ*q* (*rho* (*Suc* (*Suc r*)) *q*)
       and *muq*: μ*q* ∈ *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*. μ*p qq* = μ*q qq*
    proof
      fix *qq*
      show μ*p qq* = μ*q qq*
      proof (*cases* μ*p qq* = *None*)
        case *False*
        with *mu* π **have** *1*:*qq* ∈ *SHOs* (*Suc r*) *p* **and** *2*:*qq* ∈ *SHOs* (*Suc r*) *q*
          unfolding *SHOmsgVectors-def* **by** *auto*
        from *mu* π *1*
        have μ*p qq* = *Some* (*sendMsg Ute-M* (*Suc r*) *qq p* (*rho* (*Suc r*) *qq*))
          unfolding *SHOmsgVectors-def* **by** *auto*
        moreover

```
    from muq π 2
    have μq qq = Some (sendMsg Ute-M (Suc r) qq q (rho (Suc r) qq))
      unfolding SHOmsgVectors-def by auto
    ultimately
    show ?thesis
      by (auto simp: Ute-SHOMachine-def Ute-sendMsg-def step-def
                send0-def send1-def)
  next
    case True
   with mu have qq ∉ HOs (Suc r) p unfolding SHOmsgVectors-def by auto
    with π muq have μq qq = None unfolding SHOmsgVectors-def by auto
    with True show ?thesis by simp
  qed
qed
hence vsets:⋀v. {qq. μp qq = Some (Vote (Some v))}
            = {qq. μq qq = Some (Vote (Some v))}
  by auto


show x (rho (Suc (Suc r)) p) = x (rho (Suc (Suc r)) q)
proof (cases ∃ v. α < card {qq. μp qq = Some (Vote (Some v))}, clarify)
  fix v
  assume vp: α < card {qq. μp qq = Some (Vote (Some v))}
  with run comm stp π nxt mu have x (rho (Suc (Suc r)) p) = v
    by (auto dest: set-x-from-vote)
  moreover
  from vsets vp
  have α < card {qq. μq qq = Some (Vote (Some v))} by auto
  with run comm stp π nxtq muq have x (rho (Suc (Suc r)) q) = v
    by (auto dest: set-x-from-vote)
  ultimately
  show x (rho (Suc (Suc r)) p) = x (rho (Suc (Suc r)) q)
    by auto
next
  assume nov: ¬ (∃ v. α < card {qq. μp qq = Some (Vote (Some v))})
  with nxt stp have x (rho (Suc (Suc r)) p) = undefined
    by (auto simp: Ute-SHOMachine-def nextState-def
              Ute-nextState-def next1-def)
  moreover
  from vsets nov
  have ¬ (∃ v. α < card {qq. μq qq = Some (Vote (Some v))}) by auto
  with nxtq stp have x (rho (Suc (Suc r)) q) = undefined
    by (auto simp: Ute-SHOMachine-def nextState-def
              Ute-nextState-def next1-def)
  ultimately
  show ?thesis by simp
qed
qed
then obtain v where ⋀q. x (rho (Suc (Suc r)) q) = v by blast
```

moreover
from *stp* **have** *step* (*Suc* (*Suc* *r*)) = *0*
by (*auto simp*: *step-def mod-Suc*)
hence $\bigwedge p\ \mu p'\ q.$
$[\![\ q \in$ *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*)
$]\!] \Longrightarrow \mu p'\ q =$ *Some* (*Val* (*x* (*rho* (*Suc* (*Suc* *r*)) *q*)))
by (*auto simp*: *Ute-SHOMachine-def SHOmsgVectors-def Ute-sendMsg-def send0-def*)
ultimately
have $\bigwedge p\ \mu p'\ q.$
$[\![\ q \in$ *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*)
$]\!] \Longrightarrow \mu p'\ q =$ (*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 *mup*: $\mu p \in$ *SHOmsgVectors Ute-M* (*Suc* *r*) *p* (*rho* (*Suc* *r*))
(*HOs* (*Suc* *r*) *p*) (*SHOs* (*Suc* *r*) *p*)
and *nxtq*: *nextState Ute-M r q* (*rho r q*) μq (*rho* (*Suc* *r*) *q*)
and *vq*: *vote* (*rho* (*Suc* *r*) *q*) = *Some v*
and *qsho*: $q \in$ *SHOs* (*Suc* *r*) *p* $\cap$ *HOs* (*Suc* *r*) *p*
shows $\mu p\ q$ = *Some* (*Vote* (*Some v*))
proof −
from *nxtq vq* **have** *step r* = *0* **by** (*auto simp*: *vote-step*)
with *mup qsho* **have** $\mu p\ q$ = *Some* (*Vote* (*vote* (*rho* (*Suc* *r*) *q*)))
by (*auto simp*: *Ute-SHOMachine-def SHOmsgVectors-def Ute-sendMsg-def step-def send1-def mod-Suc*)
with *vq* **show** $\mu p\ q$ = *Some* (*Vote* (*Some v*)) **by** *auto*
qed

We now prove the Termination property.

theorem *ute-termination*:
assumes *run*: *SHORun Ute-M rho HOs SHOs*
and *commR*: $\forall r.$ *SHOcommPerRd Ute-M* (*HOs r*) (*SHOs r*)
and *commG*: *SHOcommGlobal Ute-M HOs SHOs*
shows $\exists r\ v.$ *decide* (*rho r p*) = *Some v*
proof −
from *commG*
obtain $\Phi\ \pi$ *r0*
where *rr*: *r0* = *Suc* (*nSteps* $*\ \Phi$)
and π: $\forall p.\ \pi =$ *HOs r0 p* $\wedge\ \pi =$ *SHOs r0 p*
and *t*: $\forall p.$ *card* (*SHOs* (*Suc r0*) *p* $\cap$ *HOs* (*Suc r0*) *p*) $> T$

and e: $\forall p.\ card\ (SHOs\ (Suc\ (Suc\ r0))\ p \cap HOs\ (Suc\ (Suc\ r0))\ p) > E$
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*:$\forall p.\ (vote\ (rho\ (Suc\ (Suc\ r0))\ p)) = Some\ w$
proof $-$
have *abc*:$\forall p.\ \exists w.\ vote\ (rho\ (Suc\ (Suc\ r0))\ p) = Some\ w$
proof
fix p
from *run stp* **obtain** μp
where *nxt*:$nextState\ Ute\text{-}M\ (Suc\ r0)\ p\ (rho\ (Suc\ r0)\ p)\ \mu p$
$(rho\ (Suc\ (Suc\ r0))\ p)$
and *mup*:$\mu p \in SHOmsgVectors\ Ute\text{-}M\ (Suc\ r0)\ p\ (rho\ (Suc\ r0))$
$(HOs\ (Suc\ r0)\ p)\ (SHOs\ (Suc\ r0)\ p)$
by (*auto simp*: *Ute-SHOMachine-def SHORun-eq SHOnextConfig-eq*)

have $\exists v.\ T < card\ \{qq.\ \mu p\ qq = Some\ (Val\ v)\}$
proof $-$
from t **have** $card\ (SHOs\ (Suc\ r0)\ p \cap HOs\ (Suc\ r0)\ p) > T$ **..**
moreover
from *run commR stp* π *rr*
obtain v **where**
$\bigwedge p\ \mu p'\ q.$
$[\![\ q \in SHOs\ (Suc\ r0)\ p \cap HOs\ (Suc\ r0)\ p;$
$\mu p' \in SHOmsgVectors\ Ute\text{-}M\ (Suc\ r0)\ p\ (rho\ (Suc\ r0))$
$(HOs\ (Suc\ r0)\ p)\ (SHOs\ (Suc\ r0)\ p)$
$]\!] \Longrightarrow \mu p'\ q = Some\ (Val\ v)$
using *termination-argument-1* **by** *blast*

with *mup* **obtain** v **where**
$\bigwedge qq.\ qq \in SHOs\ (Suc\ r0)\ p \cap HOs\ (Suc\ r0)\ p \Longrightarrow \mu p\ qq = Some\ (Val$
$v)$
by *auto*
hence $SHOs\ (Suc\ r0)\ p \cap HOs\ (Suc\ r0)\ p \subseteq \{qq.\ \mu p\ qq = Some\ (Val\ v)\}$
by *auto*
hence $card\ (SHOs\ (Suc\ r0)\ p \cap HOs\ (Suc\ r0)\ p)$
$\leq card\ \{qq.\ \mu p\ qq = Some\ (Val\ v)\}$
by (*auto intro*: *card-mono*)
ultimately
have $T < card\ \{qq.\ \mu p\ qq = Some\ (Val\ v)\}$ **by** *auto*
thus *?thesis* **by** *auto*
qed
with *stp nxt* **show** $\exists w.\ vote\ ((rho\ (Suc\ (Suc\ r0)))\ p) = Some\ w$
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 $\forall pp.\ vote\ (rho\ (Suc\ (Suc\ r0))\ pp) = Some\ w$

proof
  fix *pp*
  from *abc* **obtain** *wp* **where** *pwp*:*vote* ((*rho* (*Suc* (*Suc r0*))) *pp*) = *Some wp*
    by *blast*
  from *run* **obtain** *μpp μqq*
    where *nxtp*: *nextState Ute-M* (*Suc r0*) *pp* (*rho* (*Suc r0*) *pp*)
                              μpp (*rho* (*Suc* (*Suc r0*)) *pp*)
      and *mup*: *μpp* ∈ *SHOmsgVectors Ute-M* (*Suc r0*) *pp* (*rho* (*Suc r0*))
                              (*HOs* (*Suc r0*) *pp*) (*SHOs* (*Suc r0*) *pp*)
      and *nxtq*: *nextState Ute-M* (*Suc r0*) *qq* (*rho* (*Suc r0*) *qq*)
                              μqq (*rho* (*Suc* (*Suc r0*)) *qq*)
      and *muq*: *μqq* ∈ *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** *μp′*
  where *nxtp*: *nextState Ute-M* (*Suc* (*Suc r0*)) *p* (*rho* (*Suc* (*Suc r0*)) *p*)
                          μp′ (*rho* (*Suc* (*Suc* (*Suc r0*))) *p*)
    and *mup′*: *μp′* ∈ *SHOmsgVectors Ute-M* (*Suc* (*Suc r0*)) *p* (*rho* (*Suc* (*Suc*
r0)))
                          (*HOs* (*Suc* (*Suc r0*)) *p*) (*SHOs* (*Suc* (*Suc r0*)) *p*)
  by (*auto simp*: *Ute-SHOMachine-def SHORun-eq SHOnextConfig-eq*)
have ⋀*qq*. *qq* ∈ *SHOs* (*Suc* (*Suc r0*)) *p* ∩ *HOs* (*Suc* (*Suc r0*)) *p*
        ⟹ *μp′ qq* = *Some* (*Vote* (*Some w*))
proof −
  fix *qq*
  assume *qqsho*:*qq* ∈ *SHOs* (*Suc* (*Suc r0*)) *p* ∩ *HOs* (*Suc* (*Suc r0*)) *p*
  from *run* **obtain** *μqq* **where**
    nxtqq:*nextState Ute-M* (*Suc r0*) *qq* (*rho* (*Suc r0*) *qq*)
                    μqq (*rho* (*Suc* (*Suc r0*)) *qq*)
    by (*auto simp*: *Ute-SHOMachine-def SHORun-eq SHOnextConfig-eq*)
  from *commR mup′ nxtqq votew qqsho* **show** *μp′ qq* = *Some* (*Vote* (*Some w*))
    by (*auto dest*: *termination-argument-2*)
qed
hence *SHOs* (*Suc* (*Suc r0*)) *p* ∩ *HOs* (*Suc* (*Suc r0*)) *p*
        ⊆ {*qq*. *μp′ qq* = *Some* (*Vote* (*Some w*))}
  by *auto*
hence *wsho*: *card* (*SHOs* (*Suc* (*Suc r0*)) *p* ∩ *HOs* (*Suc* (*Suc r0*)) *p*)
            ≤ *card* {*qq*. *μp′ qq* = *Some* (*Vote* (*Some w*))}
  by (*auto simp*: *card-mono*)

from *stp* **have** *step* (*Suc* (*Suc r0*)) = *Suc 0*

unfolding *step-def* **by** *auto*
with *nxtp* **have** *next1* (*Suc* (*Suc r0*)) *p* (*rho* (*Suc* (*Suc r0*)) *p*) *μp′*
(*rho* (*Suc* (*Suc* (*Suc r0*))) *p*)
by (*auto simp*: *Ute-SHOMachine-def nextState-def Ute-nextState-def*)
moreover
from *e* **have** *E* < *card* (*SHOs* (*Suc* (*Suc r0*)) *p* ∩ *HOs* (*Suc* (*Suc r0*)) *p*)
by *auto*
with *wsho* **have** *majv*:*card* {*qq*. *μp′ qq* = *Some* (*Vote* (*Some w*))} > *E*
by *auto*
ultimately
show *?thesis* **by** (*auto simp*: *next1-def*)
qed

8.7 $\mathcal{U}_{T,E,\alpha}$ Solves Weak Consensus

Summing up, all (coarse-grained) runs of $\mathcal{U}_{T,E,\alpha}$ for HO and SHO collections that satisfy the communication predicate satisfy the Weak Consensus property.

theorem *ute-weak-consensus*:
assumes *run*: *SHORun Ute-M rho HOs SHOs*
and *commR*: ∀ *r*. *SHOcommPerRd Ute-M* (*HOs r*) (*SHOs r*)
and *commG*: *SHOcommGlobal Ute-M HOs SHOs*
shows *weak-consensus* (*x* ∘ (*rho 0*)) *decide rho*
unfolding *weak-consensus-def*
using *ute-validity*[*OF run commR*]
ute-agreement[*OF run commR*]
ute-termination[*OF run commR commG*]
by *auto*

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

theorem *ute-weak-consensus-fg*:
assumes *run*: *fg-run Ute-M rho HOs SHOs* (*λr q*. *undefined*)
and *commR*: ∀ *r*. *SHOcommPerRd Ute-M* (*HOs r*) (*SHOs r*)
and *commG*: *SHOcommGlobal Ute-M HOs SHOs*
shows *weak-consensus* (*λp*. *x* (*state* (*rho 0*) *p*)) *decide* (*state* ∘ *rho*)
(**is** *weak-consensus ?inits - -*)
proof (*rule local-property-reduction*[*OF run weak-consensus-is-local*])
fix *crun*
assume *crun*: *CSHORun Ute-M crun HOs SHOs* (*λr q*. *undefined*)
and *init*: *crun 0* = *state* (*rho 0*)
from *crun* **have** *SHORun Ute-M crun HOs SHOs* **by** (*unfold SHORun-def*)
from *this commR commG*
have *weak-consensus* (*x* ∘ (*crun 0*)) *decide crun*
by (*rule ute-weak-consensus*)
with *init* **show** *weak-consensus ?inits decide crun*
by (*simp add*: *o-def*)
qed

end — context *ute-parameters*

end
theory *AteDefs*
imports *../HOModel*
begin

9 Verification of the $\mathcal{A}_{T,E,\alpha}$ Consensus algorithm

Algorithm $\mathcal{A}_{T,E,\alpha}$ is presented in [3]. Like $\mathcal{U}_{T,E,\alpha}$, it is an uncoordinated algorithm that tolerates value faults, and it is parameterized by values T, E, and α that serve a similar function as in $\mathcal{U}_{T,E,\alpha}$, allowing the algorithm to be adapted to the characteristics of different systems. $\mathcal{A}_{T,E,\alpha}$ 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.

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

abbreviation
  $N \equiv card\ (UNIV::Proc\ set)$ — 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 ≡ (*decide st* = *None*)

The following locale introduces the parameters used for the $\mathcal{A}_{T,E,\alpha}$ algorithm and their constraints [3].

locale *ate-parameters* =
  fixes α::*nat* **and** T::*nat* **and** E::*nat*
  assumes *TNaE*: $T \geq 2*(N + 2*\alpha - E)$
    and *TltN*: $T < N$

and *EltN*: $E < N$

begin

The following are consequences of the assumptions on the parameters.

lemma *majE*: $2 * (E - \alpha) \geq N$
using *TNaE TltN* **by** *auto*

lemma *Egta*: $E > \alpha$
using *majE EltN* **by** *auto*

lemma *Tge2a*: $T \geq 2 * \alpha$
using *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 *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 *mostOftenRcvd* **where**
$mostOftenRcvd\ (msgs::Proc \Rightarrow 'val\ option) \equiv$
$\{v.\ \forall w.\ card\ \{qq.\ msgs\ qq = Some\ w\} \leq card\ \{qq.\ msgs\ qq = Some\ v\}\}$

definition
$Ate\text{-}sendMsg :: nat \Rightarrow Proc \Rightarrow Proc \Rightarrow 'val\ pstate \Rightarrow 'val$
where
$Ate\text{-}sendMsg\ r\ p\ q\ st \equiv x\ st$

definition
$Ate\text{-}nextState :: nat \Rightarrow Proc \Rightarrow ('val::linorder)\ pstate \Rightarrow (Proc \Rightarrow 'val\ option)$
$\Rightarrow 'val\ pstate \Rightarrow bool$
where
$Ate\text{-}nextState\ r\ p\ st\ msgs\ st' \equiv$
$(if\ card\ \{q.\ msgs\ q \neq None\} > T$
$then\ x\ st' = Min\ (mostOftenRcvd\ msgs)$
$else\ x\ st' = x\ st)$
$\wedge\ (\quad (\exists v.\ card\ \{q.\ msgs\ q = Some\ v\} > E\ \wedge decide\ st' = Some\ v)$
$\vee \neg (\exists v.\ card\ \{q.\ msgs\ q = Some\ v\} > E)$
$\wedge\ decide\ st' = decide\ st)$

9.2 Communication Predicate for $\mathcal{A}_{T,E,\alpha}$

Following [3], we now define the communication predicate for the $\mathcal{A}_{T,E,\alpha}$ algorithm. The round-by-round predicate requires that no process may receive more than α corrupted messages at any round.

definition *Ate-commPerRd* **where**
$Ate\text{-}commPerRd\ HOrs\ SHOrs \equiv$

$\forall p.\ card\ (HOrs\ p\ -\ SHOrs\ p) \leq \alpha$

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
Ate-commGlobal **where**
Ate-commGlobal HOs SHOs $\equiv$
$(\forall r\ p.\ \exists r' > r.\ card\ (HOs\ r'\ p) > T)$
$\wedge\ (\forall r\ p.\ \exists r' > r.\ card\ (SHOs\ r'\ p \cap HOs\ r'\ p) > E)$
$\wedge\ (\forall r.\ \exists r' > r.\ \exists \pi 1\ \pi 2.$
$card\ \pi 1 > E - \alpha$
$\wedge\ card\ \pi 2 > T$
$\wedge\ (\forall p \in \pi 1.\ HOs\ r'\ p = \pi 2\ \wedge\ SHOs\ r'\ p \cap HOs\ r'\ p = \pi 2))$

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 *Ate-SHOMachine* **where**
Ate-SHOMachine = (|
CinitState = (λ *p st crd. Ate-initState p* (*st*::(*'val*::*linorder*) *pstate*)),
sendMsg = *Ate-sendMsg*,
CnextState = (λ *r p st msgs crd st'. Ate-nextState r p st msgs st'*),
SHOcommPerRd = (*Ate-commPerRd*:: *Proc HO* $\Rightarrow$ *Proc HO* $\Rightarrow$ *bool*),
SHOcommGlobal = *Ate-commGlobal*
|)

abbreviation
Ate-M $\equiv$ (*Ate-SHOMachine*::(*Proc*, *'val*::*linorder pstate*, *'val*) *SHOMachine*)

end — locale *ate-parameters*

end
theory *AteProof*
imports *AteDefs* ../*Reduction*
begin

context *ate-parameters*
begin

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 *nvp*: *decide* (*rho r p*) $\neq$ *Some v*
  and *vp*: *decide* (*rho* (*Suc r*) *p*) = *Some v*
  shows *card* {*qq. sendMsg Ate-M r qq p* (*rho r qq*) = *v*} > *E* − α
proof −
  from *run* **obtain** μ*p*
    where *mu*: μ*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*) μ*p* (*rho* (*Suc r*) *p*)
    by (*auto simp*: *SHORun-eq SHOnextConfig-eq*)
  from *mu*
  have {*qq.* μ*p qq* = *Some v*} − (*HOs r p* − *SHOs r p*)
        $\subseteq$ {*qq. sendMsg Ate-M r qq p* (*rho r qq*) = *v*}
      (**is** *?vrcvdp* − *?ahop* $\subseteq$ *?vsentp*)
    by (*auto simp*: *SHOmsgVectors-def*)
  hence *card* (*?vrcvdp* − *?ahop*) $\leq$ *card ?vsentp*
    and *card* (*?vrcvdp* − *?ahop*) $\geq$ *card ?vrcvdp* − *card ?ahop*
    by (*auto simp*: *card-mono diff-card-le-card-Diff*)
  hence *card ?vsentp* $\geq$ *card ?vrcvdp* − *card ?ahop* **by** *auto*
  moreover
  from *nxt nvp vp* **have** *card ?vrcvdp* > *E*
    by (*auto simp*: *Ate-SHOMachine-def nextState-def Ate-nextState-def*)
  moreover
  from *comm* **have** *card* (*HOs r p* − *SHOs r p*) $\leq$ α
    by (*auto simp*: *Ate-SHOMachine-def Ate-commPerRd-def*)
  ultimately
  show *?thesis* **using** *Egta* **by** *auto*
qed

If more than $E - \alpha$ processes send 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*) μ*q* ((*rho* (*Suc r*)) *q*)
  and *muq*: μ*q* $\in$ *SHOmsgVectors Ate-M r q* (*rho r*) (*HOs r q*) (*SHOs r q*)
  and *vsent*: *card* {*qq. sendMsg Ate-M r qq q* (*rho r qq*) = *v*} > *E* − α
          (**is** *card ?vsent* > -)
  shows *card* ({*qq.* μ*q qq* $\neq$ *Some v*} $\cap$ *HOs r q*) $\leq$ *N* + *2***α − *E*
proof −
  from *nxt muq*
  have ({*qq.* μ*q qq* $\neq$ *Some v*} $\cap$ *HOs r q*) − (*HOs r q* − *SHOs r q*)
        $\subseteq$ {*qq. sendMsg Ate-M r qq q* (*rho r qq*) $\neq$ *v*}
    (**is** *?notvrcvd* − *?aho* $\subseteq$ *?notvsent*)

unfolding *SHOmsgVectors-def* **by** *auto*
hence *card ?notvsent* $\geq$ *card* (*?notvrcvd* $-$ *?aho*)
and *card* (*?notvrcvd* $-$ *?aho*) $\geq$ *card ?notvrcvd* $-$ *card ?aho*
by (*auto simp*: *card-mono diff-card-le-card-Diff*)
moreover
from *comm* **have** *card ?aho* $\leq \alpha$
by (*auto simp*: *Ate-SHOMachine-def Ate-commPerRd-def*)
moreover
have *1*: *card ?notvsent* + *card ?vsent* = *card* (*?notvsent* $\cup$ *?vsent*)
by (*subst card-Un-Int*) *auto*
have *?notvsent* $\cup$ *?vsent* = (*UNIV* ::*Proc set*) **by** *auto*
hence *card* (*?notvsent* $\cup$ *?vsent*) = *N* **by** *simp*
with *1 vsent* **have** *card ?notvsent* $\leq$ $N - (E + 1 - \alpha)$ **by** *auto*
ultimately
show *?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*: *SHOcommPerRd Ate-M* (*HOs r*) (*SHOs r*)
and *nxt*: *nextState Ate-M r q* (*rho r q*) μq ((*rho* (*Suc r*)) *q*)
and *muq*: $\mu q \in$ *SHOmsgVectors Ate-M r q* (*rho r*) (*HOs r q*) (*SHOs r q*)
and *threshold-T*: *card* $\{qq.\ \mu q\ qq \neq None\} > T$
and *threshold-E*: *card* $\{qq.$ *sendMsg Ate-M r qq q* (*rho r qq*) $= v\} > E - \alpha$
shows *mostOftenRcvd* $\mu q = \{v\}$
proof $-$
from *muq* **have** *hodef*:*HOs r q* = $\{qq.\ \mu q\ qq \neq None\}$
unfolding *SHOmsgVectors-def* **by** *auto*

from *comm nxt muq threshold-E*
have *card* ($\{qq.\ \mu q\ qq \neq Some\ v\} \cap$ *HOs r q*) $\leq N + 2{*}\alpha - E$
(**is** *card ?heardnotv* $\leq$ -)
by (*rule other-values-received*)
moreover
have *card ?heardnotv* $\geq T + 1 -$ *card* $\{qq.\ \mu q\ qq = Some\ v\}$
proof $-$
from *muq*
have *?heardnotv* = (*HOs r q*) $-$ $\{qq.\ \mu q\ qq = Some\ v\}$
and $\{qq.\ \mu q\ qq = Some\ v\} \subseteq$ *HOs r q*
unfolding *SHOmsgVectors-def* **by** *auto*
hence *card ?heardnotv* = *card* (*HOs r q*) $-$ *card* $\{qq.\ \mu q\ qq = Some\ v\}$
by (*auto simp*: *card-Diff-subset*)
with *hodef threshold-T* **show** *?thesis* **by** *auto*
qed
ultimately
have *card* $\{qq.\ \mu q\ qq = Some\ v\} >$ *card ?heardnotv*
using *TNaE* **by** *auto*

```
  moreover
  {
    fix w
    assume w: w ≠ v
    with hodef have {qq. μq qq = Some w} ⊆ ?heardnotv by auto
   hence card {qq. μq qq = Some w} ≤ card ?heardnotv by (auto simp: card-mono)
  }
  ultimately
  have {w. card {qq. μq qq = Some w} ≥ card {qq. μq qq = Some v}} = {v}
    by force
  thus ?thesis unfolding mostOftenRcvd-def by auto
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.

```
lemma common-x-induct:
  assumes run: SHORun Ate-M rho HOs SHOs
  and comm: SHOcommPerRd Ate-M (HOs (r+k)) (SHOs (r+k))
  and ih: card {qq. x (rho (r + k) qq) = v} > E − α
  shows card {qq. x (rho (r + Suc k) qq) = v} > E − α
proof −
  from ih
  have thrE:∀ pp. card {qq. sendMsg Ate-M (r + k) qq pp (rho (r + k) qq) = v}
                        > E − α
    by (auto simp: Ate-SHOMachine-def Ate-sendMsg-def)

  {
    fix qq
    assume kv:x (rho (r + k) qq) = v
    from run obtain μqq
      where nxt: nextState Ate-M (r + k) qq (rho (r + k) qq) μqq ((rho (Suc (r
+ k))) qq)
        and muq: μqq ∈ SHOmsgVectors Ate-M (r + k) qq (rho (r + k))
                                    (HOs (r + k) qq) (SHOs (r + k) qq)
      by (auto simp: SHORun-eq SHOnextConfig-eq)

    have x (rho (r + Suc k) qq) = v
    proof (cases card {pp. μqq pp ≠ None} > T)
      case True
      with comm nxt muq thrE have mostOftenRcvd μqq = {v}
        by (auto dest: mostOftenRcvd-v)
      with nxt True show x (rho (r + Suc k) qq) = v
        by (auto simp: Ate-SHOMachine-def nextState-def Ate-nextState-def)
    next
      case False
      with nxt have x (rho (r + Suc k) qq) = x (rho (r + k) qq)
        by (auto simp: Ate-SHOMachine-def nextState-def Ate-nextState-def)
      with kv show x (rho (r + Suc k) qq) = v by simp
    qed
```

}
hence {*qq*. *x* (*rho* (*r* + *k*) *qq*) = *v*} ⊆ {*qq*. *x* (*rho* (*r* + *Suc k*) *qq*) = *v*}
by *auto*
hence *card* {*qq*. *x* (*rho* (*r* + *k*) *qq*) = *v*} ≤ *card* {*qq*. *x* (*rho* (*r* + *Suc k*) *qq*) = *v*}
by (*auto simp*: *card-mono*)
with *ih* **show** *?thesis* **by** *auto*
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*: ∀ *r*. *SHOcommPerRd* (*Ate-M*::(*Proc*, *'val*::*linorder pstate*, *'val*) *SHOMachine*)
(*HOs r*) (*SHOs r*)
and *d1*: *decide* (*rho r p*) ≠ *Some v*
and *d2*: *decide* (*rho* (*Suc r*) *p*) = *Some v*
and *qupdatex*: *x* (*rho* (*r* + *Suc k*) *q*) ≠ *x* (*rho* (*r* + *k*) *q*)
shows *x* (*rho* (*r* + *Suc k*) *q*) = *v*
proof −
from *comm*
have *SHOcommPerRd* (*Ate-M*::(*Proc*, *'val*::*linorder pstate*, *'val*) *SHOMachine*)
(*HOs* (*r*+*k*)) (*SHOs* (*r*+*k*)) **..**
moreover
from *run* **obtain** μq
where *nxt*: *nextState Ate-M* (*r*+*k*) *q* (*rho* (*r*+*k*) *q*) μq (*rho* (*r* + *Suc k*) *q*)
and *muq*: μq ∈ *SHOmsgVectors Ate-M* (*r*+*k*) *q* (*rho* (*r*+*k*))
(*HOs* (*r*+*k*) *q*) (*SHOs* (*r*+*k*) *q*)
by (*auto simp*: *SHORun-eq SHOnextConfig-eq*)
moreover
from *nxt qupdatex*
have *threshold-T*: *card* {*qq*. μq *qq* ≠ *None*} > *T*
and *xsmall*: *x* (*rho* (*r* + *Suc k*) *q*) = *Min* (*mostOftenRcvd* μq)
by (*auto simp*: *Ate-SHOMachine-def nextState-def Ate-nextState-def*)
moreover
have *E* − α < *card* {*qq*. *x* (*rho* (*r* + *k*) *qq*) = *v*}
proof (*induct k*)
from *run comm d1 d2*
have *E* − α < *card* {*qq*. *sendMsg Ate-M r qq p* (*rho r qq*) = *v*}
by (*auto dest*: *decide-sent-msgs-threshold*)
thus *E* − α < *card* {*qq*. *x* (*rho* (*r* + *0*) *qq*) = *v*}
by (*auto simp*: *Ate-SHOMachine-def Ate-sendMsg-def*)
next
fix *k*
assume *E* − α < *card* {*qq*. *x* (*rho* (*r* + *k*) *qq*) = *v*}
with *run comm* **show** *E* − α < *card* {*qq*. *x* (*rho* (*r* + *Suc k*) *qq*) = *v*}
by (*auto dest*: *common-x-induct*)
qed

with *run*
have $E - \alpha < card$ $\{qq.$ *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 *?dec k* $=$ *decide* (*rho k p*)
have ($\forall\, m{<}n.$ *?dec* (*Suc m*) $\neq$ *?dec m* $\longrightarrow$ *?dec* (*Suc m*) $\neq$ *Some v*) $\longrightarrow$ *?dec n* $\neq$ *Some v*
(**is** *?P n* **is** *?A n* $\longrightarrow$ -)
proof (*induct n*)
from *run* **show** *?P 0*
by (*auto simp*: *Ate-SHOMachine-def SHORun-eq HOinitConfig-eq initState-def Ate-initState-def*)
next
fix *n*
assume *ih*: *?P n* **thus** *?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*: $\forall\, r.$ *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) (HOs 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 card ({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
```

qed
moreover
have $\{pp.\ \mu qq\ pp \neq None\}$
$= \{pp.\ \mu qq\ pp = Some\ v\} \cup \{pp.\ \exists\, w.\ w \neq v \wedge \mu qq\ pp = Some\ w\}$
and $\{pp.\ \mu qq\ pp = Some\ v\} \cap \{pp.\ \exists\, w.\ w \neq v \wedge \mu qq\ pp = Some\ w\} = \{\}$
by *auto*
hence $card\ \{pp.\ \mu qq\ pp \neq None\}$
$= card\ \{pp.\ \mu qq\ pp = Some\ v\} + card\ \{pp.\ \exists\, w.\ w \neq v \wedge \mu qq\ pp = Some\ w\}$
by (*auto simp*: *card-Un-Int*)
moreover
note *threshold-T*
ultimately
have $card\ \{pp.\ \mu qq\ pp = Some\ v\} > card\ \{pp.\ \exists\, w.\ w \neq v \wedge \mu qq\ pp = Some\ w\}$
by *auto*
moreover
{
fix w
assume $w \neq v$
hence $\{pp.\ \mu qq\ pp = Some\ w\} \subseteq \{pp.\ \exists\, w.\ w \neq v \wedge \mu qq\ pp = Some\ w\}$
by *auto*
hence $card\ \{pp.\ \mu qq\ pp = Some\ w\} \leq card\ \{pp.\ \exists\, w.\ w \neq v \wedge \mu qq\ pp = Some\ w\}$
by (*auto simp*: *card-mono*)
}
ultimately
have $zz{:}\bigwedge w.\ w \neq v \Longrightarrow$
$card\ \{pp.\ \mu qq\ pp = Some\ w\} < card\ \{pp.\ \mu qq\ pp = Some\ v\}$
by *force*
hence $\bigwedge w.\ card\ \{pp.\ \mu qq\ pp = Some\ v\} \leq card\ \{pp.\ \mu qq\ pp = Some\ w\}$
$\Longrightarrow w = v$
by *force*
with *zz* **have** $mostOftenRcvd\ \mu qq = \{v\}$
by (*force simp*: *mostOftenRcvd-def*)
with *xsmall* **show** $x\ (rho\ (Suc\ r)\ qq) = v$ **by** *auto*
qed
qed
thus $\forall\, qq.\ sendMsg\ Ate\text{-}M\ (Suc\ r)\ qq\ p\ (rho\ (Suc\ r)\ qq) = v$
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** μp
where *nxt*: *nextState Ate-M rp p* (*rho rp p*) μp (*rho* (*Suc rp*) *p*)
and *mu*: $\mu p \in$ *SHOmsgVectors Ate-M rp p* (*rho rp*) (*HOs rp p*) (*SHOs rp p*)
by (*auto simp*: *SHORun-eq SHOnextConfig-eq*)

{
fix *w*
assume *w*: $w \neq v$
from *comm* **have** *card* (*HOs rp p* $-$ *SHOs rp p*) $\leq \alpha$
by (*auto simp*: *Ate-SHOMachine-def Ate-commPerRd-def*)
moreover
from *mu P*
have *SHOs rp p* $\cap$ *HOs rp p* $\subseteq$ {*pp*. μp *pp* = *Some v*}
and *HOs rp p* = {*pp*. μp *pp* $\neq$ *None*}
by (*auto simp*: *SHOmsgVectors-def*)
hence {*pp*. μp *pp* $\neq$ *None*} $-$ {*pp*. μp *pp* = *Some v*}
$\subseteq$ *HOs rp p* $-$ *SHOs rp p*
by *auto*
hence *card* ({*pp*. μp *pp* $\neq$ *None*} $-$ {*pp*. μp *pp* = *Some v*})
$\leq$ *card* (*HOs rp p* $-$ *SHOs rp p*)
by (*auto simp*: *card-mono*)
ultimately
have *card* ({*pp*. μp *pp* $\neq$ *None*} $-$ {*pp*. μp *pp* = *Some v*}) $<$ *E*
using *Egta* **by** *auto*
moreover
from *w* **have** {*pp*. μp *pp* = *Some w*}
$\subseteq$ {*pp*. μp *pp* $\neq$ *None*} $-$ {*pp*. μp *pp* = *Some v*}
by *auto*
hence *card* {*pp*. μp *pp* = *Some w*}
$\leq$ *card* ({*pp*. μp *pp* $\neq$ *None*} $-$ {*pp*. μp *pp* = *Some v*})
by (*auto simp*: *card-mono*)
ultimately
have *card* {*pp*. μp *pp* = *Some w*} $<$ *E* **by** *simp*
}
hence *PP*: $\bigwedge w$. *card* {*pp*. μp *pp* = *Some w*} $\geq$ *E* $\Longrightarrow$ *w* = *v* **by** *force*

from *rp nxt mu* **have** *card* {*q*. μp *q* = *Some w*} $>$ *E*
by (*auto simp*: *SHOmsgVectors-def Ate-SHOMachine-def*
nextState-def Ate-nextState-def)
with *PP* **show** *?thesis* **by** *auto*
qed

9.6 Proof of Agreement

If two processes decide at the some round, they decide the same value.

lemma *common-decision*:
assumes *run*: *SHORun Ate-M rho HOs SHOs*
and *comm*: *SHOcommPerRd Ate-M* (*HOs r*) (*SHOs r*)

and *nvp*: *decide* (*rho r p*) $\neq$ *Some v*
and *vp*: *decide* (*rho* (*Suc r*) *p*) = *Some v*
and *nwq*: *decide* (*rho r q*) $\neq$ *Some w*
and *wq*: *decide* (*rho* (*Suc r*) *q*) = *Some w*
shows *w* = *v*
proof −
have *gtn*: *card* {*qq*. *sendMsg Ate-M r qq p* (*rho r qq*) = *v*}
+ *card* {*qq*. *sendMsg Ate-M r qq q* (*rho r qq*) = *w*} > *N*
proof −
from *run comm nvp vp*
have *card* {*qq*. *sendMsg Ate-M r qq p* (*rho r qq*) = *v*} > *E* − α
by (*rule decide-sent-msgs-threshold*)
moreover
from *run comm nwq wq*
have *card* {*qq*. *sendMsg Ate-M r qq q* (*rho r qq*) = *w*} > *E* − α
by (*rule decide-sent-msgs-threshold*)
ultimately
show *?thesis* **using** *majE* **by** *auto*
qed

show *?thesis*
proof (*rule ccontr*)
assume *vw*:*w* $\neq$ *v*
have $\forall$ *qq*. *sendMsg Ate-M r qq p* (*rho r qq*) = *sendMsg Ate-M r qq q* (*rho r qq*)
by (*auto simp*: *Ate-SHOMachine-def Ate-sendMsg-def*)
with *vw*
have {*qq*. *sendMsg Ate-M r qq p* (*rho r qq*) = *v*}
$\cap$ {*qq*. *sendMsg Ate-M r qq q* (*rho r qq*) = *w*} = {}
by *auto*
with *gtn*
have *card* ({*qq*. *sendMsg Ate-M r qq p* (*rho r qq*) = *v*}
$\cup$ {*qq*. *sendMsg Ate-M r qq q* (*rho r qq*) = *w*}) > *N*
by (*auto simp*: *card-Un-Int*)
moreover
have *card* ({*qq*. *sendMsg Ate-M r qq p* (*rho r qq*) = *v*}
$\cup$ {*qq*. *sendMsg Ate-M r qq q* (*rho r qq*) = *w*}) $\leq$ *N*
by (*auto simp*: *card-mono*)
ultimately
show *False* **by** *auto*
qed
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.

lemma *laterProcessDecidesSameValue* :
assumes *run*: *SHORun Ate-M rho HOs SHOs*
and *comm*: $\forall$ *r*. *SHOcommPerRd Ate-M* (*HOs r*) (*SHOs r*)
and *nd1*: *decide* (*rho r p*) $\neq$ *Some v*

```
  and d1: decide (rho (Suc r) p) = Some v
  and nd2: decide (rho (r+k) q) ≠ Some w
  and d2: decide (rho (Suc (r+k)) q) = Some w
  shows w = v
proof (rule ccontr)
  assume vdifw:w ≠ v
  have kgt0: k > 0
  proof (rule ccontr)
    assume ¬ k > 0
    hence k = 0 by auto
    with run comm nd1 d1 nd2 d2 have w = v
      by (auto dest: common-decision)
    with vdifw show False ..
  qed

  have 1: {qq. sendMsg Ate-M r qq p (rho r qq) = v}
          ∩ {qq. sendMsg Ate-M (r+k) qq q (rho (r+k) qq) = w} = {}
    (is ?sentv ∩ ?sentw = {})
  proof (rule ccontr)
    assume ¬ ?thesis
    then obtain qq
      where xrv: x (rho r qq) = v and rkw: x (rho (r+k) qq) = w
      by (auto simp: Ate-SHOMachine-def Ate-sendMsg-def)
    have ∃ k' < k. x (rho (r + k') qq) ≠ w ∧ x (rho (r + Suc k') qq) = w
    proof (rule ccontr)
      assume f: ¬ ?thesis
      {
        fix k'
        assume kk':k' < k hence x (rho (r + k') qq) ≠ w
        proof (induct k')
          from xrv vdifw
          show x (rho (r + 0) qq) ≠ w by simp
        next
          fix k'
          assume ih:k' < k ⟹ x (rho (r + k') qq) ≠ w
             and ksk':Suc k' < k
          from ksk' have k' < k by simp
          with ih f  show x (rho (r + Suc k') qq) ≠ w by auto
        qed
      }
      with f have ∀ k' < k. x (rho (r + Suc k') qq) ≠ w by auto
      moreover
      from kgt0 have k − 1 < k and kk:Suc (k − 1) = k by auto
      ultimately
      have x (rho (r + Suc (k − 1)) qq) ≠ w by blast
      with rkw kk show False by simp
    qed
    then obtain k'
      where k' < k
```

```
      and w: x (rho (r + Suc k') qq) = w
      and qqupdatex: x (rho (r + Suc k') qq) ≠ x (rho (r + k') qq)
    by auto
  from run comm nd1 d1 qqupdatex
  have x (rho (r + Suc k') qq) = v by (rule common-x)
  with w vdifw show False by simp
qed
from run comm nd1 d1 have sentv: card ?sentv > E − α
  by (auto dest: decide-sent-msgs-threshold)
from run comm nd2 d2 have card ?sentw > E − α
  by (auto dest: decide-sent-msgs-threshold)
with sentv majE have (card ?sentv) + (card ?sentw) > N
  by simp
with 1 vdifw  have 2: card (?sentv ∪ ?sentw) > N
  by (auto simp: card-Un-Int)
have card (?sentv ∪ ?sentw) ≤ N
  by (auto simp: card-mono)
with 2 show False by simp
qed
```

The Agreement property is now an immediate consequence.

```
theorem ate-agreement:
  assumes run: SHORun Ate-M rho HOs SHOs
  and comm: ∀ r. SHOcommPerRd Ate-M (HOs r) (SHOs r)
  and p: decide (rho m p) = Some v
  and q: decide (rho n q) = Some w
  shows w = v
proof −
  from run p obtain k where
    k: k < m decide (rho k p) ≠ Some v decide (rho (Suc k) p) = Some v
    by (rule decisionNonNullThenDecided)
  from run q obtain l where
    l: l < n decide (rho l q) ≠ Some w decide (rho (Suc l) q) = Some w
    by (rule decisionNonNullThenDecided)
  show ?thesis
  proof (cases k ≤ l)
    case True
    then obtain i where l = k+i by (auto simp add: le-iff-add)
    with run comm k l show ?thesis
      by (auto dest: laterProcessDecidesSameValue)
  next
    case False
    hence l ≤ k by simp
    then obtain i where m: k = l+i by (auto simp add: le-iff-add)
    with run comm k l show ?thesis
      by (auto dest: 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*: $\forall r$. (*SHOcommPerRd*::((*Proc*, *'val*::*linorder pstate*, *'val*) *SHOMachine*)
 $\Rightarrow$ (*Proc HO*) $\Rightarrow$ (*Proc HO*) $\Rightarrow$ *bool*)
 Ate-M (*HOs r*) (*SHOs r*)
 and *commG*: *SHOcommGlobal Ate-M HOs SHOs*
 shows $\exists r\ v.$ *decide* (*rho r p*) = *Some v*
proof –
 from *commG* **obtain** $r'\ \pi 1\ \pi 2$
 where πea: *card* $\pi 1 > E - \alpha$
 and πt: *card* $\pi 2 > T$
 and *hosho*: $\forall p \in \pi 1.$ (*HOs* r' $p = \pi 2 \wedge$ *SHOs* r' $p \cap$ *HOs* r' $p = \pi 2$)
 by (*auto simp*: *Ate-SHOMachine-def Ate-commGlobal-def*)

 obtain *v* **where**
 P1: $\forall pp.$ *card* {*qq*. *sendMsg Ate-M* (*Suc* r') *qq pp* (*rho* (*Suc* r') *qq*) = *v*} $> E - \alpha$
 proof –
 have $\forall p \in \pi 1.\ \forall q \in \pi 1.$ *x* (*rho* (*Suc* r') *p*) = *x* (*rho* (*Suc* r') *q*)
 proof (*clarify*)
 fix *p q*
 assume *p*: $p \in \pi 1$ **and** *q*: $q \in \pi 1$

 from *run* **obtain** μp
 where *nxtp*: *nextState Ate-M* r' *p* (*rho* r' *p*) μp (*rho* (*Suc* r') *p*)
 and *mup*: $\mu p \in$ *SHOmsgVectors Ate-M* r' *p* (*rho* r') (*HOs* r' *p*) (*SHOs* r' *p*)
 by (*auto simp*: *SHORun-eq SHOnextConfig-eq*)

 from *run* **obtain** μq
 where *nxtq*: *nextState Ate-M* r' *q* (*rho* r' *q*) μq (*rho* (*Suc* r') *q*)
 and *muq*: $\mu q \in$ *SHOmsgVectors Ate-M* r' *q* (*rho* r') (*HOs* r' *q*) (*SHOs* r' *q*)
 by (*auto simp*: *SHORun-eq SHOnextConfig-eq*)

 from *mup muq p q*
 have {*qq*. μq *qq* $\neq$ *None*} = *HOs* r' *q*
 and *2*:{*qq*. μq *qq* = *Some* (*sendMsg Ate-M* r' *qq q* (*rho* r' *qq*))}
 $\supseteq$ *SHOs* r' $q \cap$ *HOs* r' *q*
 and {*qq*. μp *qq* $\neq$ *None*} = *HOs* r' *p*
 and *4*:{*qq*. μp *qq* = *Some* (*sendMsg Ate-M* r' *qq p* (*rho* r' *qq*))}
 $\supseteq$ *SHOs* r' $p \cap$ *HOs* r' *p*
 by (*auto simp*: *SHOmsgVectors-def*)
 with *p q hosho*

have $aa{:}\pi 2 = \{qq.\ \mu q\ qq \neq None\}$
and $cc{:}\pi 2 = \{qq.\ \mu p\ qq \neq None\}$ **by** *auto*
from *p q hosho 2*
have $bb{:}\{qq.\ \mu q\ qq = Some\ (sendMsg\ Ate\text{-}M\ r'\ qq\ q\ (rho\ r'\ qq))\} \supseteq \pi 2$
by *auto*
from *p q hosho 4*
have $dd{:}\{qq.\ \mu p\ qq = Some\ (sendMsg\ Ate\text{-}M\ r'\ qq\ p\ (rho\ r'\ qq))\} \supseteq \pi 2$
by *auto*
have $Min\ (mostOftenRcvd\ \mu p) = Min\ (mostOftenRcvd\ \mu q)$
proof $-$
have $\forall\, qq.\ sendMsg\ Ate\text{-}M\ r'\ qq\ p\ (rho\ r'\ qq)$
$= sendMsg\ Ate\text{-}M\ r'\ qq\ q\ (rho\ r'\ qq)$
by (*auto simp*: *Ate-SHOMachine-def Ate-sendMsg-def*)
with *aa bb cc dd* **have** $\forall\, qq.\ \mu p\ qq \neq None \longrightarrow \mu p\ qq = \mu q\ qq$
by *force*
moreover
from *aa bb cc dd*
have $\{qq.\ \mu p\ qq \neq None\} = \{qq.\ \mu q\ qq \neq None\}$ **by** *auto*
hence $\forall\, qq.\ \mu p\ qq = None \longleftrightarrow \mu q\ qq = None$ **by** *blast*
hence $\forall\, qq.\ \mu p\ qq = None \longrightarrow \mu p\ qq = \mu q\ qq$ **by** *auto*
ultimately
have $\forall\, qq.\ \mu p\ qq = \mu q\ qq$ **by** *blast*
thus *?thesis* **by** (*auto simp*: *mostOftenRcvd-def*)
qed
with πt *aa nxtq* πt *cc nxtp*
show $x\ (rho\ (Suc\ r')\ p) = x\ (rho\ (Suc\ r')\ q)$
by (*auto simp*: *Ate-SHOMachine-def nextState-def Ate-nextState-def*)
qed
then obtain v **where** $Pv{:}\forall\, p \in \pi 1.\ x\ (rho\ (Suc\ r')\ p) = v$ **by** *blast*
{
fix *pp*
from *Pv* **have** $\forall\, p \in \pi 1.\ sendMsg\ Ate\text{-}M\ (Suc\ r')\ p\ pp\ (rho\ (Suc\ r')\ p) = v$
by (*auto simp*: *Ate-SHOMachine-def Ate-sendMsg-def*)
hence $card\ \pi 1 \leq card\ \{qq.\ sendMsg\ Ate\text{-}M\ (Suc\ r')\ qq\ pp\ (rho\ (Suc\ r')\ qq)$
$= v\}$
by (*auto intro*: *card-mono*)
with πea
have $E - \alpha < card\ \{qq.\ sendMsg\ Ate\text{-}M\ (Suc\ r')\ qq\ pp\ (rho\ (Suc\ r')\ qq) =$
$v\}$
by *simp*
}
with *that* **show** *?thesis* **by** *blast*
qed

{
fix *k pp*
have $E - \alpha < card\ \{qq.\ sendMsg\ Ate\text{-}M\ (Suc\ r' + k)\ qq\ pp\ (rho\ (Suc\ r' +$
$k)\ qq) = v\}$
(**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
  }
  note P2 = this

  {
    fix k pp
    assume ppupdatex: 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 ppupdatex
    have threshold-T: card {qq. μpp qq ≠ 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
```

}
note *P3* = *this*

have *P4*:$\forall$ *pp*. $\exists$ *k*. *x* (*rho* (*Suc* r' + *Suc* *k*) *pp*) = *v*
proof
  fix *pp*
  from *commG* **have** $\exists r'' > r'$. *card* (*HOs* r'' *pp*) > *T*
    by (*auto simp*: *Ate-SHOMachine-def Ate-commGlobal-def*)
  then obtain *k* **where** *Suc* r' + *k* > r' **and** *t*:*card* (*HOs* (*Suc* r' + *k*) *pp*) > *T*
    by (*auto dest*: *less-imp-Suc-add*)
  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*: $\mu pp \in$ *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
  have *x* (*rho* (*Suc* r' + *Suc* *k*) *pp*) = *v*
  proof −
    from *commR*
    have (*SHOcommPerRd*::((*Proc*, '*val*::*linorder pstate*, '*val*::*linorder*) *SHOMa-chine*)
              $\Rightarrow$ (*Proc HO*) $\Rightarrow$ (*Proc HO*) $\Rightarrow$ *bool*)
        Ate-M (*HOs* (*Suc* r' + *k*)) (*SHOs* (*Suc* r' + *k*)) **..**
    moreover
    from *mu* **have** *HOs* (*Suc* r' + *k*) *pp* = {*q*. μpp *q* $\neq$ *None*}
      by (*auto simp*: *SHOmsgVectors-def*)
    with *nxt t*
    have *threshold-T*: *card* {*q*. μpp *q* $\neq$ *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 $E - \alpha <$ *card* {*qq*. *sendMsg Ate-M* (*Suc* r' + *k*) *qq pp* (*rho* (*Suc* r' + *k*) *qq*) = *v*} **.**
    ultimately
    have *mostOftenRcvd* μpp = {*v*}
      using *nxt mu* **by** (*auto dest*!: *mostOftenRcvd-v*)
    with *xsmall* **show** *?thesis* **by** *auto*
  qed
  thus $\exists$ *k*. *x* (*rho* (*Suc* r' + *Suc* *k*) *pp*) = *v* **..**
qed

have *P5a*: $\forall$ *pp*. $\exists$ *rr*. $\forall$ *k*. *x* (*rho* (*rr* + *k*) *pp*) = *v*
proof
  fix *pp*
  from *P4* **obtain** *rk* **where**
    xrrv: *x* (*rho* (*Suc* r' + *Suc* *rk*) *pp*) = *v* (**is** *x* (*rho* *?rr pp*) = *v*)

by *blast*
have $\forall k.\ x\ (rho\ (?rr + k)\ pp) = v$
proof
fix k
show $x\ (rho\ (?rr + k)\ pp) = v$
proof (*induct k*)
from *xrrv* **show** $x\ (rho\ (?rr + 0)\ pp) = v$ **by** *simp*
next
fix k
assume *ih*: $x\ (rho\ (?rr + k)\ pp) = v$
obtain k' **where** *rrk*: $Suc\ r' + k' = ?rr + k$ **by** *auto*
show $x\ (rho\ (?rr + Suc\ k)\ pp) = v$
proof (*rule ccontr*)
assume *nv*: $x\ (rho\ (?rr + Suc\ k)\ pp) \neq v$
with *rrk ih*
have $x\ (rho\ (Suc\ r' + Suc\ k')\ pp) \neq x\ (rho\ (Suc\ r' + k')\ pp)$
by (*simp add*: *ac-simps*)
hence $x\ (rho\ (Suc\ r' + Suc\ k')\ pp) = v$ **by** (*rule P3*)
with *rrk nv* **show** *False* **by** (*simp add*: *ac-simps*)
qed
qed
qed
thus $\exists rr.\ \forall k.\ x\ (rho\ (rr + k)\ pp) = v$ **by** *blast*
qed

from *P5a* **have** $\exists F.\ \forall pp\ k.\ x\ (rho\ (F\ pp + k)\ pp) = v$ **by** (*rule choice*)
then obtain R::($Proc \Rightarrow nat$)
where *imgR*: R ‘ ($UNIV$::$Proc\ set$) $\neq \{\}$
and R: $\forall pp\ k.\ x\ (rho\ (R\ pp + k)\ pp) = v$
by *blast*
define *rr* **where** $rr = Max\ (R$ ‘ $UNIV)$

have *P5*: $\forall r' > rr.\ \forall pp.\ x\ (rho\ r'\ pp) = v$
proof (*clarify*)
fix $r'\ pp$
assume r': $r' > rr$
hence $r' > R\ pp$ **by** (*auto simp*: *rr-def*)
then obtain i **where** $r' = R\ pp + i$
by (*auto dest*: *less-imp-Suc-add*)
with R **show** $x\ (rho\ r'\ pp) = v$ **by** *auto*
qed

from *commG* **have** $\exists r' > rr.\ card\ (SHOs\ r'\ p \cap HOs\ r'\ p) > E$
by (*auto simp*: *Ate-SHOMachine-def Ate-commGlobal-def*)
with *P5* **obtain** r'
where $r' > rr$
and $card\ (SHOs\ r'\ p \cap HOs\ r'\ p) > E$
and $\forall pp.\ sendMsg\ Ate\text{-}M\ r'\ pp\ p\ (rho\ r'\ pp) = v$
by (*auto simp*: *Ate-SHOMachine-def Ate-sendMsg-def*)

moreover
from *run* **obtain** μp
  where *nxt*: *nextState Ate-M* r' *p* (*rho* r' *p*) μp (*rho* (*Suc* r') *p*)
    and *mu*: $\mu p \in$ *SHOmsgVectors Ate-M* r' *p* (*rho* r') (*HOs* r' *p*) (*SHOs* r' *p*)
  by (*auto simp*: *SHORun-eq SHOnextConfig-eq*)
from *mu*
have *card* (*SHOs* r' $p \cap$ *HOs* r' *p*)
    $\leq$ *card* $\{q.\ \mu p\ q =$ *Some* (*sendMsg Ate-M* r' *q p* (*rho* r' *q*))$\}$
  by (*auto simp*: *SHOmsgVectors-def intro*: *card-mono*)
ultimately
have *threshold-E*: *card* $\{q.\ \mu p\ q =$ *Some* $v\} > E$ **by** *auto*
with *nxt* **show** *?thesis*
  by (*auto simp*: *Ate-SHOMachine-def nextState-def Ate-nextState-def*)
qed

9.8 $\mathcal{A}_{T,E,\alpha}$ Solves Weak Consensus

Summing up, all (coarse-grained) runs of $\mathcal{A}_{T,E,\alpha}$ for HO and SHO collections that satisfy the communication predicate satisfy the Weak Consensus property.

theorem *ate-weak-consensus*:
  assumes *run*: *SHORun Ate-M rho HOs SHOs*
    and *commR*: $\forall r$. *SHOcommPerRd Ate-M* (*HOs r*) (*SHOs r*)
    and *commG*: *SHOcommGlobal Ate-M HOs SHOs*
  shows *weak-consensus* ($x \circ$ (*rho 0*)) *decide rho*
  unfolding *weak-consensus-def* **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*: *fg-run Ate-M rho HOs SHOs* ($\lambda r\ q$. *undefined*)
    and *commR*: $\forall r$. *SHOcommPerRd Ate-M* (*HOs r*) (*SHOs r*)
    and *commG*: *SHOcommGlobal Ate-M HOs SHOs*
  shows *weak-consensus* (λp. *x* (*state* (*rho 0*) *p*)) *decide* (*state* $\circ$ *rho*)
  (**is** *weak-consensus ?inits - -*)
proof (*rule local-property-reduction*[*OF run weak-consensus-is-local*])
  fix *crun*
  assume *crun*: *CSHORun Ate-M crun HOs SHOs* ($\lambda r\ q$. *undefined*)
    and *init*: *crun 0* = *state* (*rho 0*)
  from *crun* **have** *SHORun Ate-M crun HOs SHOs* **by** (*unfold SHORun-def*)
  from *this commR commG*
  have *weak-consensus* ($x \circ$ (*crun 0*)) *decide crun*
    by (*rule ate-weak-consensus*)
  with *init* **show** *weak-consensus ?inits decide crun*
    by (*simp add*: *o-def*)
qed

end — context *ate-parameters*

end
theory *EigbyzDefs*
imports *../HOModel*
begin

10 Verification of the $\textit{EIGByz}_f$ Consensus Algorithm

Lynch [12] 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 [1].

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
  $N \equiv$ *card* (*UNIV*::*Proc set*) — number of processes

The algorithm is parameterized by f, which represents the number of rounds and the height of the tree data structure (see below).

axiomatization f::*nat*
where f: $f < N$

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 $[p,q] \mapsto$ *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 *labels*, which encapsulate lists of distinct process identifiers and whose length is at most $f+1$.

definition *Label* = {*xs*::*Proc list*. *length xs* $\leq$ *Suc f* $\wedge$ *distinct xs*}
typedef *Label* = *Label*
  by (*auto simp*: *Label-def intro*: *exI*[**where** x= []]) — the empty list is a label

There is a finite number of different labels.

lemma *finite-Label*: *finite Label*

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.

definition *children* **where**
  children l ≡
   if is-leaf l
   then {}
   else { *Abs-Label* (*Rep-Label l* @ [*p*]) | *p* . *p* ∉ *set-lbl l* }

10.2 Model of the Algorithm

The following record models the local state of a process.

record *'val pstate* =
  vals :: *Label* ⇒ *'val option*
  newvals :: *Label* ⇒ *'val*
  decide :: *'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 *newvals* field is unconstrained.

definition *EIG-initState* **where**
  EIG-initState p st ≡
    (∀ *l*. (*vals st l* = *None*) = (*l* ≠ *root-node*))
   ∧ *decide st* = *None*

type-synonym *'val Msg* = *Label* ⇒ *'val option*

At every round, every process sends its current *vals* tree to all processes. In fact, only the level of the tree corresponding to the round number is used (cf. definition of *extend-vals* below).

definition *EIG-sendMsg* **where**
  EIG-sendMsg r p q st ≡ *vals st*

During the first $f - 1$ rounds, every process extends its tree *vals* according to the values received in the round. No decision is taken.

definition *extend-vals* **where**
  extend-vals r p st msgs st' ≡
   vals st' = (λ *l*.
     if length-lbl l = *Suc r* ∧ *msgs* (*last-lbl l*) ≠ *None*
     then (*the* (*msgs* (*last-lbl l*))) (*butlast-lbl l*)
     else if length-lbl l = *Suc r* ∧ *msgs* (*last-lbl l*) = *None then None*
     else vals st l)

definition *next-main* **where**
  next-main r p st msgs st' ≡ *extend-vals r p st msgs st'* ∧ *decide st'* = *None*

In the final round, in addition to extending the tree as described previously, processes construct the tree *newvals*, starting at the leaves. The values at the leaves are copied from *vals*, except that missing values *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.

fun *fixupval* :: *'val option* ⇒ *'val* **where**
  fixupval None = *undefined*
| *fixupval* (*Some v*) = *v*

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*

definition *check-newvals* :: *'val pstate* ⇒ *bool* **where**
  check-newvals st ≡
   ∀ *l*. *is-leaf l* ∧ *newvals st l* = *fixupval* (*vals st l*)
     ∨ ¬(*is-leaf l*) ∧
       ((∃ *w*. *has-majority w* (*newvals st*) (*children l*) ∧ *newvals st l* = *w*)
       ∨ (¬(∃ *w*. *has-majority w* (*newvals st*) (*children l*))
            ∧ *newvals st l* = *undefined*))

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.

definition *EIG-nextState* **where**
  EIG-nextState r ≡
   if r < *f then next-main r*
   else if r = *f then next-end r*
   else (λ*p st msgs st'*. *st'* = *st*)

10.3 Communication Predicate for $EIGByz_f$

The secure kernel *SKr* w.r.t. given HO and SHO collections consists of the process from which every process receives the correct message.

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. Obviously, only the first *f* rounds really matter, since the algorithm terminates after that.

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-commPerRd HO SHO $\equiv$ *card* (*SKr 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-commGlobal HOs SHOs $\equiv$ *card* (*SK HOs SHOs*) $\geq N - f$

The above communication predicates differ from Lynch's presentation of $EIGByz_f$. 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-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

$$\Big(\bigcap_{p \in \Pi, r' \in \mathbb{N}} SHO(p, r')\Big) \subseteq \Big(\bigcap_{p \in \Pi} SHO(p, r)\Big),$$

it follows that any execution of $EIGByz_f$ where $N > 3f$ also satisfies *EIG-commPerRd* at any round. The standard correctness hypotheses thus imply our communication predicates.

However, our proof shows that $EIGByz_f$ 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-commPerRd* always holds. Also, *EIG-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 $EIGByz_f$ 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 $EIGByz_f$ Heard-Of Machine

We now define the non-coordinated SHO machine for $EIGByz_f$ by assembling the algorithm definition and its communication-predicate.

definition *EIG-SHOMachine* **where**
EIG-SHOMachine = (|
CinitState = (λ *p st crd. EIG-initState p st*),

sendMsg = *EIG-sendMsg*,
CnextState = (λ *r p st msgs crd st'*. *EIG-nextState r p st msgs st'*),
SHOcommPerRd = *EIG-commPerRd*,
SHOcommGlobal = *EIG-commGlobal*
⦈

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*)

qed
with *Rep-Label*[*of l*] **show** *?thesis*
by (*auto simp*: *Label-def elim*: *distinct-butlast*)
qed

The label of a child is well-formed.

lemma *Rep-Label-append*:
assumes *l*: $\neg$(*is-leaf l*)
shows (*Rep-Label l* @ [*p*] $\in$ *Label*) = (*p* $\notin$ *set-lbl l*)
(**is** *?lhs* = *?rhs* **is** (*?l'* $\in$ -) = -)
proof
assume *lhs*: *?lhs* **thus** *?rhs*
by (*auto simp*: *Label-def set-lbl-def*)
next
assume *p*: *?rhs*
from *l*[*THEN not-leaf-length*] **have** *length ?l'* $\leq$ *Suc f*
by (*simp add*: *length-lbl-def*)
moreover
from *Rep-Label*[*of l*] **have** *distinct* (*Rep-Label l*)
by (*simp add*: *Label-def*)
with *p* **have** *distinct ?l'* **by** (*simp add*: *set-lbl-def*)
ultimately
show *?lhs* **by** (*simp add*: *Label-def*)
qed

The label of a child is the label of the parent, extended by a process.

lemma *label-children*:
assumes *c*: *c* $\in$ *children l*
shows $\exists$ *p*. *p* $\notin$ *set-lbl l* $\wedge$ *Rep-Label c* = *Rep-Label l* @ [*p*]
proof −
from *c* **obtain** *p*
where *p*: *p* $\notin$ *set-lbl l* **and** *l*: $\neg$(*is-leaf l*)
and *c*: *c* = *Abs-Label* (*Rep-Label l* @ [*p*])
by (*auto simp*: *children-def*)
with *Rep-Label-append*[*OF l*] **show** *?thesis*
by (*auto simp*: *Abs-Label-inverse*)
qed

The label of any child node is one longer than the label of its parent.

lemma *children-length*:
assumes *l* $\in$ *children h*
shows *length-lbl l* = *Suc* (*length-lbl h*)
using *label-children*[*OF assms*] **by** (*auto simp*: *length-lbl-def*)

The root node is never a child.

lemma *children-not-root*:
assumes *root-node* $\in$ *children l*
shows *P*

using *label-children*[*OF assms*] *Abs-Label-inverse*[*OF nil-is-Label*]
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 $c \in$ *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*) = ($\forall$ *l'*. *l* $\notin$ *children l'*)
proof
assume *l* = *root-node*
thus $\forall$ *l'*. *l* $\notin$ *children l'* **by** (*auto elim*: *children-not-root*)
next
assume *rhs*: $\forall$ *l'*. *l* $\notin$ *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'* $\in$ *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*] $\in$ *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* $\notin$ *set l'*
by (*auto simp*: *Label-def*)
moreover
note *repl'*
ultimately have *l* $\in$ *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*: $\neg$(*is-leaf l*)
shows *last-lbl* ‘ (*children l*) = *UNIV* $-$ *set-lbl l*

proof
  show *last-lbl* ‘ (*children l*) $\subseteq$ *UNIV* − *set-lbl l*
    by (*auto dest*: *label-children simp*: *last-lbl-def*)
next
  show *UNIV* − *set-lbl l* $\subseteq$ *last-lbl* ‘ (*children l*)
  proof (*auto simp*: *image-def*)
    fix *p*
    assume *p*: *p* $\notin$ *set-lbl l*
    with *l* **have** *c*: *Abs-Label* (*Rep-Label l* @ [*p*]) $\in$ *children l*
      by (*auto simp*: *children-def*)
    with *Rep-Label-append*[*OF l*] *p*
    show $\exists$ *c* $\in$ *children l*. *p* = *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 last-lbl* (*children l*)
proof (*auto simp*: *inj-on-def*)
  fix *c c'*
  assume *c*: *c* $\in$ *children l* **and** *c'*: *c'* $\in$ *children l*
    and *eq*: *last-lbl c* = *last-lbl c'*
  from *c c'* **obtain** *p p'*
    where *p*: *Rep-Label c* = *Rep-Label l* @ [*p*]
      and *p'*: *Rep-Label c'* = *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** *Rep-Label c* = *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 $\neg$(*is-leaf l*)
  shows *card* (*children l*) = *N* − (*length-lbl l*)
proof −
  from *assms*
  have *last-lbl* ‘ (*children l*) = *UNIV* − *set-lbl l*
    by (*rule children-last-set*)
  moreover
  have *card* (*UNIV* − *set-lbl l*) = *card* (*UNIV*::*Proc set*) − *card* (*set-lbl l*)
    by (*auto simp*: *card-Diff-subset-Int*)
  moreover
  from *last-lbl-inj-on-children*
  have *card* (*children l*) = *card* (*last-lbl* ‘ *children l*)
    by (*rule sym*[*OF card-image*])
  moreover

note *card-set-lbl*[*of l*]
ultimately
show *?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.

lemma *sho-correct-vals*:
assumes *run*: *SHORun EIG-M rho HOs SHOs*
and l': $l' \in$ *children l*
and *shop*: *last-lbl* $l' \in$ *SHOs* (*length-lbl l*) *p* $\cap$ *HOs* (*length-lbl l*) *p*
(**is** *?q* $\in$ *SHOs* (*?len l*) *p* $\cap$ *-*)
shows *vals* (*rho* (*?len* l') *p*) l' = *vals* (*rho* (*?len l*) *?q*) *l*
proof –
let *?r* = *?len l*
from *run* **obtain** μp
where *nxt*: *nextState EIG-M ?r p* (*rho ?r p*) μp (*rho* (*Suc ?r*) *p*)
and *mu*: $\mu p \in$ *SHOmsgVectors EIG-M ?r p* (*rho ?r*) (*HOs ?r p*) (*SHOs ?r p*)
by (*auto simp*: *EIG-SHOMachine-def SHORun-eq SHOnextConfig-eq*)
with *shop*
have *msl*:μp *?q* = *Some* (*vals* (*rho ?r ?q*))
by (*auto simp*: *EIG-SHOMachine-def EIG-sendMsg-def SHOmsgVectors-def*)
from *nxt length-lbl*[*of* l'] *children-length*[*OF* l']
have *extend-vals ?r p* (*rho ?r p*) μp (*rho* (*Suc ?r*) *p*)
by (*auto simp*: *EIG-SHOMachine-def nextState-def EIG-nextState-def next-main-def next-end-def*)
with *msl* l' **show** *?thesis*
by (*auto simp*: *extend-vals-def children-length children-butlast-lbl*)
qed

A process fixes the value *vals l* of a label at state *length-lbl l*, and then never modifies the value.

lemma *keep-vals*:
assumes *run*: *SHORun EIG-M rho HOs SHOs*
shows *vals* (*rho* (*length-lbl l* + *n*) *p*) *l* = *vals* (*rho* (*length-lbl l*) *p*) *l*
(**is** *?v n* = *?vl*)
proof (*induct n*)
show *?v 0* = *?vl* **by** *simp*
next
fix *n*
assume *ih*: *?v n* = *?vl*
let *?r* = *length-lbl l* + *n*
from *run* **obtain** μp
where *nxt*: *nextState EIG-M ?r p* (*rho ?r p*) μp (*rho* (*Suc ?r*) *p*)
by (*auto simp*: *EIG-SHOMachine-def SHORun-eq SHOnextConfig-eq*)
with *ih* **show** *?v* (*Suc n*) = *?vl*
by (*auto simp*: *EIG-SHOMachine-def nextState-def EIG-nextState-def next-main-def next-end-def extend-vals-def*)

qed

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 run: SHORun EIG-M rho HOs SHOs
  and l': l' ∈ children l
  and skr: last-lbl l' ∈ SKr (HOs (length-lbl l)) (SHOs (length-lbl l))
  shows vals (rho (length-lbl l') p) l' = vals (rho (length-lbl l') q) l'
  using 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 run: SHORun EIG-M rho HOs SHOs
    and l: l ∈ children t
    and skrl: last-lbl l ∈ SKr (HOs (length-lbl t)) (SHOs (length-lbl t))
    and l': l' ∈ children l
    and skrl':last-lbl l' ∈ SKr (HOs (length-lbl l)) (SHOs (length-lbl l))
   shows vals (rho (length-lbl l') p) l' = vals (rho (length-lbl l) p) l
  using 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 *length-lbl l*.

```
lemma lynch-6-16-c:
  assumes commR: EIG-commPerRd (HOs (length-lbl l)) (SHOs (length-lbl l))
            (is EIG-commPerRd (HOs ?r) -)
    and l: ¬(is-leaf l)
  shows card {l' ∈ children l. last-lbl l' ∈ SKr (HOs ?r) (SHOs ?r)}
        > card (children l) div 2
    (is card ?lhs > -)
proof −
  let ?skr = SKr (HOs ?r) (SHOs ?r)

  have last-lbl ` ?lhs = ?skr − set-lbl l
  proof
    from children-last-set[OF l]
    show last-lbl ` ?lhs ⊆ ?skr − set-lbl l
      by (auto simp: children-length)
  next
    {
      fix p
      assume p: p ∈ ?skr p ∉ set-lbl l
      with children-last-set[OF l]
```

    have *p* ∈ *last-lbl* ‘ *children l* **by** *auto*
    with *p* **have** *p* ∈ *last-lbl* ‘ *?lhs*
      by (*auto simp*: *image-def children-length*)
  }
  thus *?skr* − *set-lbl l* ⊆ *last-lbl* ‘ *?lhs* **by** *auto*
qed
moreover
from *last-lbl-inj-on-children*[*of l*]
have *inj-on last-lbl ?lhs* **by** (*auto simp*: *inj-on-def*)
ultimately
have *card ?lhs* = *card* (*?skr* − *set-lbl l*) **by** (*auto dest*: *card-image*)
also have ... ≥ (*card ?skr*) − (*card* (*set-lbl l*))
  by (*simp add*: *diff-card-le-card-Diff*)
finally have *card ?lhs* ≥ (*card ?skr*) − *?r*
  using *card-set-lbl*[*of l*] **by** *simp*

moreover
from *commR* **have** *card ?skr* > (*N* + *f*) *div 2*
  by (*auto simp*: *EIG-commPerRd-def*)
with *not-leaf-length*[*OF l*] *f*
have (*card ?skr*) − *?r* > (*N* − *?r*) *div 2* **by** *auto*
with *card-children*[*OF l*]
have (*card ?skr*) − *?r* > *card* (*children l*) *div 2* **by** *simp*

ultimately show *?thesis* **by** *simp*
qed

If *l* is a non-leaf label such that all of its children corresponding to well-heard processes at round *length-lbl l* have a uniform *newvals* decoration at round *f+1*, then *l* itself is decorated with that same value.

lemma *newvals-skr-uniform*:
  assumes *run*: *SHORun EIG-M rho HOs SHOs*
    and *commR*: *EIG-commPerRd* (*HOs* (*length-lbl l*)) (*SHOs* (*length-lbl l*))
        (**is** *EIG-commPerRd* (*HOs ?r*) -)
    and *notleaf*: ¬(*is-leaf l*)
    and *unif*: ⋀*l′*. ⟦*l′* ∈ *children l*;
          last-lbl l′ ∈ *SKr* (*HOs* (*length-lbl l*)) (*SHOs* (*length-lbl l*))
        ⟧ ⟹ *newvals* (*rho* (*Suc f*) *p*) *l′* = *v*
  shows *newvals* (*rho* (*Suc f*) *p*) *l* = *v*
proof −
  from *unif*
  have *card* {*l′* ∈ *children l*. *last-lbl l′* ∈ *SKr* (*HOs ?r*) (*SHOs ?r*)}
      ≤ *card* {*l′* ∈ *children l*. *newvals* (*rho* (*Suc f*) *p*) *l′* = *v*}
    by (*auto intro*: *card-mono*)
  with *lynch-6-16-c*[*of HOs l SHOs*, *OF commR notleaf*]
  have *maj*: *has-majority v* (*newvals* (*rho* (*Suc f*) *p*)) (*children l*)
    by (*simp add*: *has-majority-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 *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*: ∀ *r*. *EIG-commPerRd* (*HOs r*) (*SHOs r*)
and *notroot*: *l* ∈ *children t*
and *skr*: *last-lbl l* ∈ *SKr* (*HOs* (*length-lbl t*)) (*SHOs* (*length-lbl t*))
(**is** - ∈ *SKr* (*HOs* (*?len t*)) -)
shows *newvals* (*rho* (*Suc f*) *p*) *l* = *fixupval* (*vals* (*rho* (*?len l*) *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*: $\bigwedge$ *l' t'*.
⟦*k* = *Suc f* − *length-lbl l'*; *l'* ∈ *children t'*;
last-lbl l' ∈ *SKr* (*HOs* (*?len t'*)) (*SHOs* (*?len t'*))⟧
⟹ *?P l'*
and *flk*: *Suc k* = *Suc f* − *?len l*
and *notroot*: *l* ∈ *children t*
and *skr*: *last-lbl l* ∈ *SKr* (*HOs* (*?len t*)) (*SHOs* (*?len t*))

let *?v* = *fixupval* (*vals* (*rho* (*?len l*) *p*) *l*)
from *flk* **have** *notlf*: ¬(*is-leaf l*) **by** (*simp add*: *is-leaf-def*)

{
fix *l'*
assume *l'*: *l'* ∈ *children l*
and *skr'*: *last-lbl l'* ∈ *SKr* (*HOs* (*?len l*)) (*SHOs* (*?len l*))

from *run notroot skr* l' skr'
have *vals* (*rho* (*?len* l') *p*) l' = *vals* (*rho* (*?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* (*rho* (*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 rho l ≡
∀ *p q*. *newvals* (*rho* (*Suc f*) *p*) *l* = *newvals* (*rho* (*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': $l \in$ *subtrees* l' **and** l'': $l \in$ *subtrees* l''
shows $l' \in$ *subtrees* $l'' \vee l'' \in$ *subtrees* l'
using *assms* **proof** (*auto simp*: *subtrees-def append-eq-append-conv2*)
fix *xs*
assume *Rep-Label* l'' @ *xs* = *Rep-Label* l'
hence *Rep-Label* l' = *Rep-Label* l'' @ *xs* **by** (*rule sym*)
thus $\exists$ *ys*. *Rep-Label* l' = *Rep-Label* l'' @ *ys* **..**
qed

lemma *subtrees-cases*:
assumes l': $l' \in$ *subtrees* l
and *self*: $l' = l \Longrightarrow P$
and *child*: $\bigwedge c$. $[\![$ $c \in$ *children* l; $l' \in$ *subtrees* c $]\!] \Longrightarrow P$
shows P
proof $-$
from l' **obtain** t **where** t: *Rep-Label* l' = (*Rep-Label* l) @ t
by (*auto simp*: *subtrees-def*)
have $l' = l \vee (\exists c \in$ *children* l. $l' \in$ *subtrees* c)
proof (*cases* t)
assume $t = []$
with t **show** *?thesis* **by** (*simp add*: *Rep-Label-inject*)
next
fix p t'
assume *cons*: $t = p \# t'$
from *Rep-Label*[*of* l'] t **have** *length* (*Rep-Label* l @ t) $\leq$ *Suc f*
by (*simp add*: *Label-def*)
with *cons* **have** *notleaf*: $\neg$(*is-leaf* l)
by (*auto simp*: *is-leaf-def length-lbl-def*)

let *?c* = *Abs-Label* (*Rep-Label* l @ [p])
from t *cons* *Rep-Label*[*of* l'] **have** p: $p \notin$ *set-lbl* l
by (*auto simp*: *Label-def set-lbl-def*)
with *notleaf* **have** c: *?c* $\in$ *children* l
by (*auto simp*: *children-def*)
moreover
from *notleaf* p **have** *Rep-Label* l @ [p] $\in$ *Label*
by (*simp add*: *Rep-Label-append*)
hence *Rep-Label* *?c* = (*Rep-Label* l @ [p])
by (*simp add*: *Abs-Label-inverse*)
with *cons* t **have** $l' \in$ *subtrees* *?c*
by (*auto simp*: *subtrees-def*)
ultimately show *?thesis* **by** *blast*
qed
thus *?thesis* **by** (*auto elim*!: *self child*)
qed

lemma *subtrees-leaf*:
assumes l: *is-leaf* l **and** l': $l' \in$ *subtrees* l
shows $l' = l$

using l' **proof** (*rule subtrees-cases*)
  fix *c*
  assume $c \in$ *children l* — impossible
  with *l* **show** *?thesis* **by** (*simp add*: *children-def*)
qed

lemma *children-subtrees-equal*:
  assumes *c*: $c \in$ *children l* **and** c': $c' \in$ *children l*
    and *sub*: $c' \in$ *subtrees c*
  shows $c' = c$
proof −
  from *assms* **have** *Rep-Label* c' = *Rep-Label c*
    by (*auto simp*: *subtrees-def dest*!: *label-children*)
  thus *?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 $\equiv$
  $\forall\, s \in$ *subtrees l*. *is-leaf s* $\longrightarrow$ ($\exists\, h \in C$. $h \in$ *subtrees l* $\wedge$ $s \in$ *subtrees h*)

A *covering* is a subcovering w.r.t. the root node.

abbreviation *covering* **where**
  covering C $\equiv$ *subcovering C 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 *SHORun EIG-M rho HOs SHOs*
    and $\forall\, r$. *EIG-commPerRd* (*HOs r*) (*SHOs r*)
    and $l \in$ *children t*
    and *last-lbl l* $\in$ *SKr* (*HOs* (*length-lbl t*)) (*SHOs* (*length-lbl t*))
  shows *common rho l*
  using *assms*
  by (*auto simp*: *common-def lynch-6-16-d lynch-6-15*
      intro: *arg-cong*[**where** *f*=*fixupval*])

lemma *lynch-6-18-b*:
  assumes *run*: *SHORun EIG-M rho HOs SHOs*
    and *commG*: *EIG-commGlobal HOs SHOs*
    and *commR*: $\forall\, r$. *EIG-commPerRd* (*HOs r*) (*SHOs r*)
  shows *covering* $\{l.\ \exists\, t.\ l \in$ *children t* $\wedge$ *last-lbl l* $\in$ (*SK HOs SHOs*)$\}$
proof (*clarsimp simp*: *subcovering-def*)
  fix *l*
  assume *is-leaf l*
  with *card-set-lbl*[*of l*] **have** *card* (*set-lbl l*) = *Suc f*

by (*simp add*: *is-leaf-def*)
with *commG* **have** $N <$ *card* (*SK HOs SHOs*) + *card* (*set-lbl l*)
by (*simp add*: *EIG-commGlobal-def*)
hence $\exists\, q \in$ *set-lbl l* . $q \in$ *SK HOs SHOs*
by (*auto dest*: *majorities-intersect*)
then obtain *l1 q l2* **where**
l: *Rep-Label l* = (*l1* @ [*q*]) @ *l2* **and** *q*: $q \in$ *SK HOs SHOs*
unfolding *set-lbl-def* **by** (*auto intro*: *split-list-propE*)

let *?h* = *Abs-Label* (*l1* @ [*q*])
from *Rep-Label*[*of l*] *l* **have** *l1* @ [*q*] $\in$ *Label* **by** (*simp add*: *Label-def*)
hence *reph*: *Rep-Label ?h* = *l1* @ [*q*] **by** (*rule Abs-Label-inverse*)
hence *length-lbl ?h* $\neq$ *0* **by** (*simp add*: *length-lbl-def*)
hence *?h* $\neq$ *root-node* **by** *auto*
then obtain *t* **where** *t*: *?h* $\in$ *children t*
by (*auto simp*: *root-iff-no-child*)
moreover
from *reph q* **have** *last-lbl ?h* $\in$ *SK HOs SHOs* **by** (*simp add*: *last-lbl-def*)
moreover
from *reph l* **have** $l \in$ *subtrees ?h* **by** (*simp add*: *subtrees-def*)
ultimately
show $\exists\, h.$ ($\exists\, t.$ $h \in$ *children t*) $\wedge$ *last-lbl h* $\in$ *SK HOs SHOs* $\wedge$ $l \in$ *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 *cov*: *subcovering C l*
and *l*: $l \notin C$
and *e*: $e \in$ *children l*
shows *subcovering C e*
proof (*clarsimp simp*: *subcovering-def*)
fix *s*
assume *s*: $s \in$ *subtrees e* **and** *leaf*: *is-leaf s*
from *s children-in-subtree*[*OF e*] **have** $s \in$ *subtrees l*
by (*rule subtrees-trans*)
with *leaf cov* **obtain** *h* **where** *h*: $h \in C$ $h \in$ *subtrees l* $s \in$ *subtrees h*
by (*auto simp*: *subcovering-def*)
with *l* **obtain** *e′* **where** *e′*: $e' \in$ *children l* $h \in$ *subtrees e′*
by (*auto elim*: *subtrees-cases*)
from ‹$s \in$ *subtrees h*› ‹$h \in$ *subtrees e′*› **have** $s \in$ *subtrees e′*
by (*rule subtrees-trans*)
with *s* **have** $e \in$ *subtrees e′* $\vee$ $e' \in$ *subtrees e*
by (*rule subtrees-tree*)
with *e e′* **have** $e' = e$
by (*auto dest*: *children-subtrees-equal*)
with *e′ h* **show** $\exists\, h \in C.$ $h \in$ *subtrees e* $\wedge$ $s \in$ *subtrees h* **by** *blast*
qed

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 *run*: *SHORun EIG-M rho HOs SHOs*
    and *cov*: *subcovering C l*
    and *com*: $\forall\, l' \in C.$ *common rho* l'
  shows *common rho l*
using *cov* **proof** (*induct Suc f* $-$ *length-lbl l arbitrary*: *l*)
  fix *l*
  assume *0*: *0* $=$ *Suc f* $-$ *length-lbl l*
    and *C*: *subcovering C l*
  from *0 length-lbl*[*of l*] **have** *is-leaf l*
    by (*simp add*: *is-leaf-def*)
  with *C* **obtain** *h* **where** *h*: $h \in C$ $h \in$ *subtrees l* $l \in$ *subtrees h*
    by (*auto simp*: *subcovering-def*)
  hence $l \in C$ **by** (*auto dest*: *subtrees-antisym*)
  with *com* **show** *common rho l* **..**
next
  fix *k l*
  assume *k*: *Suc k* $=$ *Suc f* $-$ *length-lbl l*
    and *C*: *subcovering C l*
    and *ih*: $\bigwedge l'.$ $[\![k =$ *Suc f* $-$ *length-lbl* l'; *subcovering C* $l']\!] \Longrightarrow$ *common rho* l'
  show *common rho l*
  proof (*cases* $l \in C$)
    case *True*
    with *com* **show** *?thesis* **..**
  next
    case *False*
    with *C* **have** $\forall\, e \in$ *children l. subcovering C e*
      by (*blast intro*: *lynch-6-19-a*)
    moreover
    from *k* **have** $\forall\, e \in$ *children l.* $k =$ *Suc f* $-$ *length-lbl e*
      by (*auto simp*: *children-length*)
    ultimately
    have *com-ch*: $\forall\, e \in$ *children l. common rho e*
      by (*blast intro*: *ih*)

    show *?thesis*
    proof (*clarsimp simp*: *common-def*)
      fix *p q*
      from *k* **have** *notleaf*: $\neg$(*is-leaf l*) **by** (*simp add*: *is-leaf-def*)
      let *?r* $=$ *Suc f*
      from *com-ch*
      have $\forall\, e \in$ *children l. newvals* (*rho ?r p*) *e* $=$ *newvals* (*rho ?r q*) *e*
        by (*auto simp*: *common-def*)
      hence $\forall\, w.$ $\{e \in$ *children l. newvals* (*rho ?r p*) *e* $= w\}$
          $= \{e \in$ *children l. newvals* (*rho ?r q*) *e* $= w\}$
        by *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')
    qed
  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])
qed
```

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)
```

```
    by (auto simp: EIG-SHOMachine-def SHORun-eq SHOnextConfig-eq
               nextState-def)
  thus ?P (Suc r)
  proof (auto simp: EIG-nextState-def next-main-def next-end-def)
    assume ¬(r < f) r ≠ f
    with ih
    show decide (rho r p) = Some (newvals (rho (Suc f) p) root-node)
      by simp
  qed
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: ∀ 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: ∀ r. EIG-commPerRd (HOs r) (SHOs r)
      and initv: ∀ q. the (vals (rho 0 q) root-node) = v
      and dp: decide (rho r p) = Some w
  shows v = w
proof –

  have v: ∀ q. vals (rho 0 q) root-node = Some v
  proof
    fix q
    from run have vals (rho 0 q) root-node ≠ 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
```

```
  qed

  let ?len = length-lbl
  let ?r = Suc f

  {
    fix l'
    assume l': l' ∈ children root-node
       and skr: last-lbl l' ∈ SKr (HOs 0) (SHOs 0)
    with run v have vals (rho (?len l') p) l' = Some v
      by (auto dest: sho-correct-vals simp: SKr-def)

    moreover
    from run commR l' skr
    have newvals (rho ?r p) l' = fixupval (vals (rho (?len l') p) l')
      by (auto intro: lynch-6-16-d)

    ultimately
    have newvals (rho ?r p) l' = v by simp
  }
  with run commR root-node-not-leaf
  have newvals (rho ?r p) root-node = v
    by (auto intro: newvals-skr-uniform)
  with dp show ?thesis by (simp add: decide[OF run])
qed
```

Termination is trivial for $EIGByz_f$.

```
theorem Termination:
  assumes SHORun EIG-M rho HOs SHOs
  shows ∃ r v. decide (rho r p) = Some v
  using assms by (auto simp: decide)
```

10.8 $\boldsymbol{EIGByz_f}$ 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:
  assumes run: SHORun EIG-M rho HOs SHOs
      and commR: ∀ r. EIG-commPerRd (HOs r) (SHOs r)
      and commG: EIG-commGlobal HOs SHOs
  shows weak-consensus (λp. the (vals (rho 0 p) root-node)) decide rho
  unfolding weak-consensus-def
  using Validity[OF run commR]
        Agreement[OF run commG commR]
        Termination[OF run]
  by auto
```

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

theorem *eig-weak-consensus-fg*:
  assumes *run*: *fg-run EIG-M rho HOs SHOs* ($\lambda r\ q.$ *undefined*)
    and *commR*: $\forall\, r.$ *EIG-commPerRd* (*HOs r*) (*SHOs r*)
    and *commG*: *EIG-commGlobal HOs SHOs*
  shows *weak-consensus* ($\lambda p.$ *the* (*vals* (*state* (*rho 0*) *p*) *root-node*))
        decide (*state* $\circ$ *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* ($\lambda r\ q.$ *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* ($\lambda p.$ *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 $EIGByz_f$ 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

[1] A. Bar-Noy, D. Dolev, C. Dwork, and H. R. Strong. Shifting gears: Changing algorithms on the fly to expedite byzantine agreement. *Inf. Comput.*, 97(2):205–233, 1992.

[2] M. Ben-Or. Another advantage of free choice: completely asynchronous agreement protocols. In R. L. Probert, N. A. Lynch, and N. Santoro, editors, *Proc. 2nd Symp. Principles of Distributed Computing (PODC 1983)*, pages 27–30, Montreal, Canada, 1983. ACM.

[3] M. Biely, J. Widder, B. Charron-Bost, A. Gaillard, M. Hutle, and A. Schiper. Tolerating corrupted communication. In *Proc. 26th Annual ACM Symposium on Principles of Distributed Computing*, PODC '07, pages 244–253, New York, NY, USA, 2007. ACM.

[4] M. Chaouch-Saad, B. Charron-Bost, and S. Merz. A reduction theorem for the verification of round-based distributed algorithms. In O. Bournez and I. Potapov, editors, *Reachability Problems*, volume 5797 of *Lecture Notes in Computer Science*, pages 93–106, Palaiseau, France, 2009. Springer.

[5] B. Charron-Bost, H. Debrat, and S. Merz. Formal verification of consensus algorithms tolerating malicious faults. In X. Défago, F. Petit, and V. Villain, editors, *13th Intl. Symp. Stabilization, Safety, and Security of Distributed Systems (SSS 2011)*, volume 6976 of *LNCS*, pages 120–134, Grenoble, France, 2011. Springer.

[6] B. Charron-Bost and S. Merz. Formal verification of a Consensus algorithm in the Heard-Of model. *Intl. J. Software and Informatics*, 3(2-3):273–304, 2009.

[7] B. Charron-Bost and A. Schiper. The Heard-Of model: computing in distributed systems with benign faults. *Distributed Computing*, 22(1):49–71, 2009.

[8] C. Dwork, N. A. Lynch, and L. Stockmeyer. Consensus in the presence of partial synchrony. *J. ACM*, 35(2):288–323, Apr. 1988.

[9] M. J. Fischer, N. A. Lynch, and M. S. Paterson. Impossibility of distributed consensus with one faulty process. *J. ACM*, 32(2):374–382, Apr. 1985.

[10] M. Jaskelioff and S. Merz. Proving the correctness of DiskPaxos. *Archive of Formal Proofs*, 2005.

[11] L. Lamport. The part-time parliament. *ACM Trans. Comput. Syst.*, 16(2):133–169, 1998.

[12] N. Lynch. *Distributed Algorithms.* Morgan Kaufmann Publishers, San Mateo, CA, 1996.