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

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

August 28, 2014

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 \textit{reduction theorem} that allows us to verify algorithms in the coarse-grained semantics (where proofs are easier) and infer their correctness for the fine-grained one (which corresponds to actual executions). We instantiate the framework by verifying six Consensus algorithms that differ in the underlying algorithmic mechanisms and the kinds of faults they tolerate.

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

1 Introduction 4

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

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

4 Utility Lemmas About Majorities 30

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

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

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

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

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

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

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

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

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


2 Heard-Of Algorithms

2.1 The Consensus Problem

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

More formally, we represent runs of algorithms as $\omega$-sequences of configurations (vectors of process states). Hence, a run is modeled as a function of type $\text{nat} \Rightarrow \text{proc} \Rightarrow \text{pst}$ where type variables $\text{proc}$ and $\text{pst}$ represent types of processes and process states, respectively. The Consensus property is expressed with respect to a collection $\text{vals}$ of initially proposed values (one per process) and an observer function $\text{dec}::\text{pst} \Rightarrow \text{val} \text{ option}$ 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**

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

**Definition**

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

**Where**

\[\text{consensus vals dec rho} \equiv (\forall n p v. \text{dec} (\rho n p) = \text{Some} v \rightarrow v \in \text{range vals}) \land (\forall m n p q v w. \text{dec} (\rho m p) = \text{Some} v \land \text{dec} (\rho n q) = \text{Some} w \rightarrow v = w) \land (\forall p. \exists n. \text{dec} (\rho n p) \neq \text{None})\]

A variant of the Consensus problem replaces the Integrity requirement by

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

**Definition**

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

\[\text{weak-consensus vals dec rho} \equiv (\forall v. (\forall p. \text{vals} p = v) \rightarrow (\forall n p w. \text{dec} (\rho n p) = \text{Some} w \rightarrow w = v)) \land (\forall m n p q v w. \text{dec} (\rho m p) = \text{Some} v \land \text{dec} (\rho n q) = \text{Some} w \rightarrow v = w) \land (\forall p. \exists n. \text{dec} (\rho n p) \neq \text{None})\]

Clearly, 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., \(\text{card (range vals)} \leq 2\)).

**Lemma** binary-weak-consensus-then-consensus:

**Assumes** bc: weak-consensus (vals::'proc => bool) dec rho

**Shows** consensus vals dec rho

**Proof**

\{ — Show the Integrity property, the other conjuncts are the same.\]

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

have $v \in \text{range} \ \text{vals}$

proof (cases $\exists \ w. \ \forall \ p. \ \text{vals} \ p = w$)

  case True
  then obtain $w$ where $w : \forall \ p. \ \text{vals} \ p = w$ ..

with $\text{be} \ \text{have} \ \text{dec} (\rho n p) \in \{\text{Some} \ w, \ \text{None}\}$ by (auto simp: weak-consensus-def)

with $\text{dec} \ w$ show $\text{thesis}$ by (auto simp: image-def)

next

  case False

  — In this case both possible values occur in $\text{vals}$, and the result is trivial.

  thus $\text{thesis}$ by (auto simp: image-def)

qed

} note $\text{integrity} = \text{this}$

from $\text{be} \ \text{show} \ \text{thesis}$

unfolding $\text{consensus-def \ weak-consensus-def}$ by (auto elim!: $\text{integrity}$)

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” $\text{HO}(p, r)$ denotes the set of processes from which $p$ receives a message in round $r$. Since every process is assumed to send a message to all processes in each round, the complement of $\text{HO}(p, r)$ represents the set of faults that may affect $p$ in round $r$ (messages that were not received, e.g. because the sender crashed, because of a network problem etc.).

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

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

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

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

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

- a finite type $\text{'proc}$ of processes,
- a type $\text{'pst}$ of local process states,
- a type $\text{'msg}$ 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 $msgs :: \text{'proc} \Rightarrow \text{'msg option}$ of messages that $p$ received at round $r$ ($msgs q = \text{None}$ 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.

```isabelle
record (\text{'proc}, \text{'pst}, \text{'msg}) CHOAlgorithm =
  CinitState :: \text{'proc} \Rightarrow \text{'pst} \Rightarrow \text{'proc} \Rightarrow \text{bool}
  sendMsg :: \text{nat} \Rightarrow \text{'proc} \Rightarrow \text{'proc} \Rightarrow \text{'pst} \Rightarrow \text{'msg}
  CnextState :: \text{nat} \Rightarrow \text{'proc} \Rightarrow \text{'pst} \Rightarrow (\text{'proc} \Rightarrow \text{'msg option}) \Rightarrow \text{'proc} \Rightarrow \text{'pst} \Rightarrow \text{bool}
```

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

\[
\text{definition}\ \text{isNCAlgorithm}\ \text{where} \\
isNCAlgorithm\ alg \equiv \\
\quad \forall p\ st\ crd\ crd' . \ CinitState\ alg\ p\ st\ crd = CinitState\ alg\ p\ st\ crd' \\
\quad \land \quad \forall r\ p\ st\ msgs\ crd\ crd'\ st'. \ CnextState\ alg\ r\ p\ st\ msgs\ crd\ st' \equiv CnextState\ alg\ r\ p\ st\ msgs\ crd\ crd'\ st' \\
\]

\[
\text{definition}\ initState\ \text{where} \\
\quad initState\ alg\ p\ st \equiv CinitState\ alg\ p\ st\ undefined \\
\]

\[
\text{definition}\ nextState\ \text{where} \\
\quad nextState\ alg\ r\ p\ st\ msgs\ st' \equiv CnextState\ alg\ r\ p\ st\ msgs\ undefined\ st' \\
\]

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.

\[
\text{type-synonym}\ \\
\quad '\text{proc} \ HO = '\text{proc} \Rightarrow \ '\text{proc} \ set \\
\]

\[
\text{type-synonym}\ \\
\quad '\text{proc} \ coord = '\text{proc} \Rightarrow \ '\text{proc} \\
\]

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 \( CSHO\text{initConfig} \) describes the possible initial configurations for algorithm \( A \) (remember that a configuration is a function that assigns local states to every process).

\[
\text{definition}\ CSHO\text{initConfig}\ \text{where} \\
\quad CSHO\text{initConfig} A\ \text{cfg} (\text{coord}::'\text{proc} \ coord) \equiv \forall p . \ CinitState A\ p\ (\text{cfg}\ p)\ (\text{coord}\ p) \\
\]
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

$$SHOmsgVectors A r p cfg HOp SHOp \equiv \{ \mu. \ (\forall q. q \in HOp \leftrightarrow \mu q \neq None) \land (\forall q. q \in SHOp \cap HOp \rightarrow \mu q = Some (sendMsg A r q p (cfg q))) \}$$

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

$$CSHORun A rho HOs SHOs coords \equiv \ CHOinitConfig A \ (rho 0) (coords 0) \land (\forall r. CSHOnextConfig A r (rho r) (HOs r) (SHOs r) (coords (Suc r)) (rho (Suc r)))$$

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

**definition** $HOinitConfig$ 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** $SHOnextConfig-eq$:

$$SHOnextConfig A r cfg HO SHO cfg' = \ (\forall p. \exists \mu \in SHOmsgVectors A r p cfg (HO p) (SHO p). \ CnextState A r p (cfg p) \mu (cfg' p))$$

**by** (auto simp: SHOnextConfig-def CSHOnextConfig-def SHOmsgVectors-def nextState-def)
Definition SHORun where
SHORun A rho HOs SHOs ≡
CSHORun A rho HOs SHOs (λ r q. undefined)

Lemma SHORun-eq:
SHORun A rho HOs SHOs =
(\HOpInitConfig A (rho 0)
∧ (\forall r. \SHOnextConfig A r (rho r) (\HOs r) (\SHOs r) (rho (Suc r))))
by (auto simp: SHORun-def CSHORun-def HOpInitConfig-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 ≡
λ q. 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 coord cfg'

Lemma CHOnextConfig-eq:
CHOnextConfig A r cfg HO coord cfg' =
(\forall 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)
∧ (\forall 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
lemma \text{HOnextConfig-eq}:
\text{HOnextConfig A r cfg HO cfg'} \equiv \text{SHOnextConfig A r cfg HO HO cfg'}

\begin{align*}
\text{definition HORun where} \\
\text{HORun A rho HOs} & \equiv \text{SHORun A rho HOs HOs}
\end{align*}

lemma \text{HORun-eq}:
\text{HORun A rho HOs} = \\
( ~ \text{HOinitConfig A (rho 0)} \\
\wedge (\forall r. \text{HOnextConfig A r (rho r) (HOs r) (rho (Suc r)))})

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

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

lemma \text{CHORun-0}:
\text{assumes CHORun A rho HOs coords} \\
\text{and } \forall \text{cfg}. \text{CHOinitConfig A cfg (coords 0)} \Rightarrow P \text{cfg} \\
\text{shows } P (\text{rho 0})

\text{using assms unfolding CHORun-eq by blast}

lemma \text{CHORun-Suc}:
\text{assumes CHORun A rho HOs coords} \\
\text{and } \forall r. \text{HOnextConfig A r (rho r) (HOs r) (coords (Suc r)) (rho (Suc r))} \\
\Rightarrow P r \\
\text{shows } P n

\text{using assms unfolding CHORun-eq by blast}

lemma \text{CHORun-induct}:
\text{assumes run: CHORun A rho HOs coords} \\
\text{and init: CHOinitConfig A (rho 0) (coords 0)} \Rightarrow P 0 \\
\text{and step: } \forall r. [ P r; \text{HOnextConfig A r (rho r) (HOs r) (coords (Suc r))} (rho (Suc r)) ] \Rightarrow P (Suc r)

\text{shows } P n

\text{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 \textit{communication predicate}. For convenience, we split this predicate into a \textit{per Round} part that is expected to hold at every round and a \textit{global} part that must hold of the sequence of (S)HO assignments and may thus express liveness assumptions.

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

\begin{verbatim}
record ('proc, 'pst, 'msg) HOMachine = ('proc, 'pst, 'msg) CHOAlgorithm +
\end{verbatim}
3 Reduction Theorem

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

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

As a corollary, interesting properties remain valid in the fine-grained semantics if they hold in the coarse-grained semantics. It is therefore enough to verify such properties in the coarse-grained semantics, which is much eas-
ier to reason about. The essential restriction is that properties may not
depend on states of different processes occurring simultaneously. (For ex-
ample, 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.

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

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

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

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

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

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

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

\text{definition} \ fg\text{-send\text{-msg}} \text{ where}

\begin{align*}
\text{fg-send-msg } A p q \text{ config config'} &\equiv \\
q \notin (\text{sent config } p) \\
\land \text{config'} = \text{config} \| \\
\text{sent} := (\text{sent config})(p := (\text{sent config } p) \cup \{q\}), \\
\text{network} := \text{network config} \cup \\
\{(p, \text{round config } p, q, \text{sendMsg } A (\text{round config } p) p q (\text{state config } p))\} \\
\end{align*}

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

\text{definition} \ fg\text{-rev\text{-msg} where}

\begin{align*}
\text{fg-rev-msg } p q \text{ HO SHO config config'} &\equiv \\
\exists m m'. (q, (\text{round config } p), p, m) \in \text{network config} \\
\land q \in \text{HO} \\
\land \text{config'} = \text{config} \| \\
\text{rcvd} := (\text{rcvd config})(p := (\text{rcvd config } p)(q := \\
\quad \text{if } q \in \text{SHO then Some } m \text{ else Some } m'), \\
\text{network} := \text{network config} \setminus \{(q, (\text{round config } p), p, m)\} \\
\end{align*}

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

**definition fg-local where**

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

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

**definition fg-next-config where**

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

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**

\[
\text{fg-run } A \ rho \ HOs \ SHOs \ coords \equiv \\
\text{fg-init-config } A \ (\rho \ 0) \ (coords \ 0) \\
\land (\forall i. \ \exists p. \ \text{fg-next-config } A \ p \ (Ho \ (\rho \ i) \ p) \ (Ho \ (\rho \ i) \ p) \\
(\rho \ i) \ (\rho \ (\text{Suc } i)) \\
\land (\forall p \ r. \ \exists n. \ \text{round} \ (\rho \ n) \ p = r)
\]

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**

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

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

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

**lemma fg-state-change:**

**assumes** \( \rho : \text{fg-run} \ A \ \rho \ \text{HOs} \ \text{SHOs} \ \text{coords} \)

**and** \( \text{rd} : \text{round} (\rho (\text{Suc} \ n)) \ p = \text{round} (\rho \ n) \ p \)

**shows** \( \text{state} (\rho (\text{Suc} \ n)) \ p = \text{state} (\rho \ n) \ p \)

**proof** –

**from** \( \rho \) **have** \( \exists \ p'. \ \text{fg-next-config} \ A \ p' (\text{HOs} (\text{round} (\rho \ n) \ p') \ p') \)

\( \text{(SHOs} (\text{round} (\rho \ n) \ p') \ p') \)

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

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

by (**auto simp: fg-run-def**)

with \( \text{rd} \) **show** \( \text{thesis} \)

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** \( \rho : \text{fg-run} \ A \ \rho \ \text{HOs} \ \text{SHOs} \ \text{coords} \) **and** \( n : n \leq m \)

**shows** \( \text{round} (\rho \ n) \ p \leq \text{round} (\rho \ m) \ p \)

**proof** –

**from** \( n \) **obtain** \( k \) **where** \( k : m = n+k \) by (**auto simp: le-iff-add**)

\{

fix \( i \)

**have** \( \text{round} (\rho \ n) \ p \leq \text{round} (\rho (n+i)) \ p \) (**is \ ?P \ i**)

**proof** (**induct \( i \)**)

**show** \( \text{thesis} \)

next

fix \( j \)

**assume** \( \text{ih:} \ ?P \ j \)

**from** \( \rho \) **have** \( \exists \ p', \ \text{fg-next-config} \ A \ p' (\text{HOs} (\text{round} (\rho (n+j)) \ p') \ p') \)

\( \text{(SHOs} (\text{round} (\rho (n+j)) \ p') \ p') \)

\( \text{(coords} (\text{round} (\rho (\text{Suc} \ (n+j))) \ p') \ p') \)

\( (\rho \ n+j) (\rho (\text{Suc} \ (n+j))) \)

by (**auto simp: fg-run-def**)

**hence** \( \text{round} (\rho (n+j)) \ p \leq \text{round} (\rho (n+ \text{Suc} \ j)) \)

by (**auto simp: fg-next-config-def fg-send-msg-def fg-rcv-msg-def fg-local-def**)

**with** \( \text{ih} \) **show** \( ?P \ (\text{Suc} \ j) \)

by (**auto**)

**qed**

\}

**with** \( k \) **show** \( \text{thesis} \)

by (**simp**)

**qed**

Combining the two preceding lemmas, it follows that the local states of
Since every process executes every round, function $fg\text{-}start\text{Round}$ is well-defined. We also list a few facts about $fg\text{-}start\text{Round}$ that will be used to show that it is a “stuttering sampling function”, a notion introduced in the theories about stuttering equivalence.

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

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

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

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

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

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

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

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

qed

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

lemma \( \text{fg-local-transition-from-round} \):
assumes \( \rho : \text{fg-run} A \rho \_HOS \_SHOs \_coords \)
obtains \( n \) where \( \text{round}(\rho n) p = r \)
and \( \text{fg-start-round} \rho p (\text{Suc} r) = \text{Suc} n \)
and \( \text{fg-local} A p (\_HOS r p) (\_coords (\text{Suc} r) p) (\rho n) (\rho (\text{Suc} n)) \)

proof –
- have \( \text{fg-start-round} \rho p (\text{Suc} r) \neq 0 \) (is \( \?start \neq 0 \))
  - proof
    - assume contr: \( \?start = 0 \)
    - from rho have \( \text{round}(\rho \?start) p = \text{Suc} r \) by (rule \( \text{fg-start-round} \rho \))
      - with contr rho show False by (auto simp: \( \text{fg-run-def \_fg-init-config-def} \))
  qed
- then obtain \( n \) where \( n: \?start = \text{Suc} n \) by (auto simp: \( \text{gr0-conv-Suc} \))
  - with \( \text{fg-start-round}[\text{OF} \rho, \_of p \text{ Suc} r] \)
    - have 0: \( \text{round}(\rho (\text{Suc} n)) p = \text{Suc} r \) by simp
    - have 1: \( \text{round}(\rho n) p = r \)
      - proof (rule \( \text{fg-round-between-start-rounds}[\text{OF} \rho] \))
        - have \( \text{fg-start-round} \rho p r < \text{fg-start-round} \rho p (\text{Suc} r) \)
          - by (rule \( \text{fg-start-round-strict-mono}[\text{OF} \rho, THEN \text{strict-monoD}] \)) simp
        - with \( n \) show \( \text{fg-start-round} \rho p r \leq n \) by simp
      qed
- next
  - from \( n \) show \( n < \?start \) by simp
  qed
- from rho obtain \( p' \) where
  - \( \text{fg-next-config} A p' (\_HOS (\text{round} (\rho n) p') p') \)
    - \( (\_SHOs (\text{round} (\rho n) p') p') \)
    - \( (\_coords (\text{round} (\rho (\text{Suc} n)) p') p') \)
    - \( (\rho n) (\rho (\text{Suc} n)) \)
(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
  — impossible because round changes
  with 0 1 show ?thesis by (auto simp: fg-recv-msg-def)

next
  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 \( r \)th local transition.

**Lemma fg-invariant1:**

**Assumes**: \( \rho : \text{fg-run} \ A \ \rho \ \	ext{HOs} \ \text{SHOs} \ \text{coords} \)

and \( m : (p.r.q.m) \in \text{network} \ (\rho n) \ (\text{is} \ ?msg \ n) \)

**Shows**: \( m = \text{sendMsg} A r p q \ (\text{state} (\rho (\text{fg-start-round} \ \rho \ p \ r)) \ p) \)

**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 \( \text{ih:} \ ?msg \ n \Rightarrow \ ?thesis \)
  from \( \rho \) obtain \( p' \) where
    \( \text{fg-next-config} \ A \ p' (\text{HOs} (\text{round} (\rho n) \ p') \ p') \)
    (SHOs (\text{round} (\rho n) \ p') \ p')
    (coords (\text{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 \( \text{fg-send-msg} \) 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: \( \text{fg-send-msg} A \ p' q' \ ?cfg \ ?cfg' \)
show ?thesis

21
proof (cases ?msg n)
  case True
  with ih show ?thesis .
next
  case False
  with send m' have 1: p' = p round ?cfg p = r
  and 2: m = sendMsg A r p q (state ?cfg p)
    by (auto simp: fg-send-msg-def)
  from rho 1 have state ?cfg p = state (rho (fg-start-round rho p r)) p
    by (auto simp: fg-start-round fg-same-round-same-state)
  with 1 2 show ?thesis by simp
qed
next
  fix q'
  with m' have ?msg n by (auto simp: fg-rcv-msg-def)
  with ih show ?thesis .
next
  with m' have ?msg n by (auto simp: fg-local-def)
  with ih show ?thesis .
qed
qed

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

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

Except for \(\text{fg-rcv-msg}\) steps of process \(q\), the proof is immediately reduced to the induction hypothesis.

fix \(q'\)
assume \(\text{rcvmsg}: \text{fg-rcv-msg} \; \text{p'} \; \text{q'} \; \HO \; \text{SHO} \; \text{cfg} \; \text{cfg}'\)
hence \(\text{rd}: \; \text{rd} \; \text{(Suc n)} = ?\text{rd} \; 
by \; \text{(auto simp: fg-rcv-msg-def)}\)
show \(?P \; \text{(Suc n)}\)
proof (cases \(?\text{rcvd} \; \text{n}\))
  case True
  with \(\text{ih} \; \text{rd}\)
  show \(?\text{thesis by simp}\)
next
  case False
  with \(\text{rcvd} \; \text{rcvmsg} \; \text{rd}\)
  show \(?\text{thesis by simp}\)
qed

next
fix \(q'\)
assume \(\text{fg-send-msg} \; \text{A} \; \text{p'} \; \text{q'} \; \text{cfg} \; \text{cfg}'\)
with \(\text{rcvd}\)
  have \(?\text{rcvd} \; \text{n} \; \text{and} \; ?\text{rd} \; \text{(Suc n)} = ?\text{rd} \; 
by \; \text{(auto simp: fg-send-msg-def)}\)
  with \(\text{ih}\)
  show \(?P \; \text{(Suc n)} \; \text{by simp}\)
next
assume \(\text{fg-local} \; \text{A} \; \text{p'} \; \HO \; \text{crd} \; \text{cfg} \; \text{cfg}'\)
with \(\text{rcvd}\)
  have \(?\text{rcvd} \; \text{n} \; \text{and} \; ?\text{rd} \; \text{(Suc n)} = ?\text{rd} \; 
— in fact, \(p' = q\) is impossible because the \(\text{rcvd}\) field of \(p'\) is cleared
by \; \text{(auto simp: fg-local-def)}\)
  with \(\text{ih}\)
  show \(?P \; \text{(Suc n)} \; \text{by simp}\)
qed

lemma \(\text{fg-invariant2b}\):
assumes \(\text{rho}: \text{fg-run} \; \text{A} \; \text{rho} \; \HOs \; \text{SHOs} \; \text{coords}\)
  and \(m: \text{rcvd} \; \text{(rho n)} \; \text{q} \; \text{p} = \text{Some} \; \text{m} \; \text{(is} \; \text{?rcvd} \; \text{n})\)
  and \(\text{sho}: \; \text{p} \in \text{SHOs} \; \text{(round} \; \text{(rho n)} \; \text{q} \; \text{q} \; \text{is} \; \text{p} \in \text{SHOs} \; \text{(?rd} \; \text{n}) \; \text{q}\)
shows \(m = \text{sendMsg} \; \text{A} \; \text{(?rd} \; \text{n}) \; \text{p} \; \text{q} \)
(is \; ?P \; \text{n})
using \(m \; \text{sho}\)
proof (induct \(n\))
  — The base case is trivial because \(q\) has not received any message initially
  assume \(?\text{rcvd} \; 0 \; \text{with rho} \; \text{show} \; ?P \; 0\)
  by \; \text{(auto simp: fg-run-def fg-init-config-def)}\)
next
fix \(n\)
assume \(\text{rcvd}: \; ?\text{rcvd} \; \text{(Suc n)} \; \text{and} \; \text{p} \; \in \text{SHOs} \; \text{(?rd} \; \text{(Suc n}) \; \text{q}\)
  and \(\text{ih}: \; ?\text{rcvd} \; n \; \Longrightarrow \; \text{p} \; \in \text{SHOs} \; \text{(?rd} \; \text{n}) \; \text{q} \; \Longrightarrow \; ?P \; \text{n}\)
  — For the inductive step we again distinguish the possible transitions
from \(\text{rho}\) obtain \(p'\) where
  \(\text{fg-next-config} \; \text{A} \; \text{p'} \; \text{(HOs} \; \text{(round} \; \text{(rho n)} \; \text{p'}) \; \text{p'})\)
  \(\text{(SHOs} \; \text{(round} \; \text{(rho n)} \; \text{p'}) \; \text{p'})\)
(coords (round (rho (Suc n)) p') p')
(rho n) (rho (Suc n))
(is fg-next-config - - ?HO ?SHO ?crd ?cfg ?cfg')
by (force simp: fg-run-def)
thus ?P (Suc n)
proof (auto simp: fg-next-config-def)

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

fix q'
hence rd: ?rd (Suc n) = ?rd n by (auto simp: fg-rcv-msg-def)
show ?P (Suc n)
proof (cases ?rcvd n)
case True
with ih p rd show ?thesis by simp
next
case False
from rcvmsg obtain m' m'' where
  (q', round ?cfg p', p', m') ∈ network ?cfg
rcvd ?cfg' = (rcvd ?cfg)(p' := (rcvd ?cfg p')(q' :=
  if q' ∈ ?SHO then Some m' else Some m''))
by (auto simp: fg-rcv-msg-def split del: split-if-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
case False

next
case False

next
fix q'
assume fg-send-msg A p' q' ?cfg ?cfg'
with rcvd have ?rcvd n and ?rd (Suc n) = ?rd n
by (auto simp: fg-send-msg-def)
with p ih show ?P (Suc n) by simp

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

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.**

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

**Theorem (Reduction).**

- **Assumes:** $\rho$: $fg$-run $A \rho$ HOs SHOs coords
- **Shows:** CSHORun $A$ (coarse-run $\rho$) HOs SHOs coords
- **Proof:**

  - **From:** $\rho$ show CHOinitConfig $A$ ($?cr$ 0) (coords 0)
  - **By:** (auto simp: fg-run-def fg-init-config-def CSHORun-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 $r$) (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 $?cfg p \in$ SHOmsgVectors $A r p$ ($?cr r$) ?HO (SHOs $r$ $p$)
- **Proof:** (auto simp: SHOmsgVectors-def)

  - **Fix:** $q$
  - **Assume:** $q \in$ ?HO
  - **With:** $n$ loc show $\exists m$. 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 $q \in$ ?HO **By:** (auto simp: $fg$-invariant2a)
  - **Next**
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 ⇒ 'proc ⇒ 'pst) τ ≡
  ∀ p :: 'proc. (λ n. σ n p) ≈ (λ n. τ n p)

definition local-property where
  local-property P ≡
  ∀ σ τ. locally-similar σ τ → P σ → P τ

Local similarity is an equivalence relation.

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

lemma locally-similar-sym: locally-similar σ τ → locally-similar τ σ
  by (simp add: locally-similar-def stutter-equiv-sym)

lemma locally-similar-trans [trans]:
  locally-similar g σ → locally-similar σ τ → locally-similar g τ
  by (force simp add: locally-similar-def elim: stutter-equiv-trans)

lemma local-property-eq:
  local-property P = (∀ σ τ. locally-similar σ τ → P σ = P τ)
  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 \(A\) \(\rho\) HO\(s\) SHO\(s\) coords

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

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

**Fix** \(p\)

**Show** \(\lambda n.\ state\ (\rho\ n)\ p\) \(\approx\) \(\lambda n.\ coarse-run\ \rho\ n\ p\) (is \(?\)gr \(\approx\) \(?\)cgr)

**Proof** (rule stutter-equivI)

**Show** stutter-sampler (fg-start-round \(\rho\) \(p\)) \(?\)gr

**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\([\text{OF } \rho]\))

**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 \(\rho\) \(r\) \(p\))) \(p\)

**By** (simp add: fg-start-round fg-round-between-start-rounds)

**With** \(\rho\) **Show** state (\(\rho\) \(n\)) \(p\) \(=\) state (\(\rho\) (fg-start-round \(\rho\) \(r\) \(p\))) \(p\)

**By** (rule fg-same-round-same-state)

**Qed**

**Next**

**Show** stutter-sampler id \(?\)cgr

**By** (rule id-stutter-sampler)

**Next**

**Show** \(?\)gr \(\circ\) fg-start-round \(\rho\) \(p\) \(=\) \(?\)cgr \(\circ\) 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 \(A\) \(\rho\) HO\(s\) SHO\(s\) coords

**And** \(P\): local-property \(P\)

**And** coarse-correct:

\[
\bigwedge crho. \ [\ \text{CSHORun } A\ crho\ HO\(s\) SHO\(s\) coords; crho\ 0\ =\ state\ (\rho\ 0)\] \implies P\ crho
\]

**Shows** \(P\) (state \(\circ\) \(\rho\))

**Proof**

**Have** coarse-run \(\rho\) \(0\) \(=\) state (\(\rho\) \(0\))

**By** (rule ext, simp add: coarse-run-def fg-start-round-0\([\text{OF } \rho]\))

27
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:
assems sim: locally-similar σ τ
and val: \( \forall n. \text{dec}(σ \ n p) = \text{Some} \ v \implies v \in \text{range vals} \)
and dec: dec (τ \ n p) = Some v
shows v \in range vals
proof -
from sim have (λr. σ r p) ≈ (λr. τ r p) by (simp add: locally-similar-def)
then obtain m where σ m p = τ n p by (rule stutter-equiv-element-left)
from sym[OF this] dec show thesis by (auto elim: val)
qed

lemma validity-is-local:
assems sim: locally-similar σ τ
and val: \( \forall n. \text{dec}(σ \ n p) = \text{Some} \ w \implies w = v \)
and dec: dec (τ \ n p) = Some w
shows w = v
proof -
from sim have (λr. σ r p) ≈ (λr. τ r p) by (simp add: locally-similar-def)
then obtain m where σ m p = τ n p by (rule stutter-equiv-element-left)
from sym[OF this] dec show thesis by (auto elim: val)
qed

lemma agreement-is-local:
assems sim: locally-similar σ τ
and agr: \( \forall m \ n. [[\text{dec}(σ \ m p) = \text{Some} \ v; \ \text{dec}(σ \ n q) = \text{Some} \ w]] \implies v=w \)
and v: dec (τ \ m p) = Some v and w: dec (τ \ n q) = Some w
shows v = w
proof -
from sim have (λr. σ r p) ≈ (λr. τ r p) by (simp add: locally-similar-def)
then obtain m’ where m’: σ m’ p = τ m p by (rule stutter-equiv-element-left)
from sim have (λr. σ r q) ≈ (λr. τ r q) by (simp add: locally-similar-def)
then obtain n’ where n’: σ n’ q = τ 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:
assems sim: locally-similar σ τ
and trm: dec (σ \ m p) = Some v
shows ∃ n. dec (τ \ n p) = Some v

28
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 \(\text{trm}\) show \(?\text{thesis}\) by auto
qed

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

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

end
theory Majorities
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**
  - $\text{card } E < \text{card } S + \text{card } T$
  - $s: S \subseteq E$ and $t: T \subseteq E$ and $e: \text{finite } E$

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

- **Proof**
  - Assume $s t e$
  - From $s t e$ have $\text{finite } S$ and $\text{finite } T$ by (auto simp: finite-subset)
  - With $\text{card } S \cap T \neq \{\}$ by (auto simp add: card-Un-Int)
  - Ultimately
  - Show $\text{False }$ by simp

- **Qed**

**Lemma abs-majoritiesE:**

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

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

- **Proof**
  - From $s t e$ have $S \cap T \neq \{\}$ by (rule abs-majorities-intersect)
  - Then obtain $p$ where $p \in S \cap T$ by blast
  - With that show $\text{thesis }$ by auto

- **Qed**

Special case: both sets $S$ and $T$ are majorities.

**Lemma abs-majoritiesE\':**

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

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

- **Proof** (rule abs-majoritiesE[OF $s t e$])
  - From $\text{Smaj Tmaj show } \text{card } E < \text{card } S + \text{card } T$ by auto

- **Qed**

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

**Lemma majorities-intersect:**

30
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.

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

abbreviation
N ≡ card (UNIV::Proc set)

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

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

definition \(\text{OTR-initState}\) where
\(\text{OTR-initState} \ p \ st \equiv \text{decide} \ st = \text{None}\)

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

definition \(\text{HOV} : (\text{Proc} \Rightarrow \text{'val option}) \Rightarrow \text{'val} \Rightarrow \text{Proc set}\) where
\(\text{HOV} \ \text{msgs} \ v \equiv \{q . \text{msgs} \ q = \text{Some} \ v\}\)

\(\text{MFR} \ \text{msgs} \ v\) (“most frequently received”) holds for vector \(\text{msgs}\) 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 \(\text{HOV} \ \text{msgs} \ v\).

definition \(\text{MFR} : (\text{Proc} \Rightarrow \text{'val option}) \Rightarrow \text{'val} \Rightarrow \text{bool}\) where
\(\text{MFR} \ \text{msgs} \ v \equiv \forall \ w. \ \text{card} (\text{HOV} \ \text{msgs} \ w) \leq \text{card} (\text{HOV} \ \text{msgs} \ v)\)

lemma \(\text{MFR-exists}: \exists v. \ \text{MFR} \ \text{msgs} \ v\)
proof –
let \(?\text{cards} = \{\ \text{card} (\text{HOV} \ \text{msgs} \ v) | v . \text{True}\}\\)
let \(?\text{mfr} = \text{Max} \ ?\text{cards}\)
have \(\forall \ v. \ \text{card} (\text{HOV} \ \text{msgs} \ v) \leq N\) by (auto intro: card-mono)
hence \(?\text{cards} \subseteq \{0 .. N\}\) by auto
hence \(\text{fin: finite} \ ?\text{cards}\) by (metis atLeast0AtMost finite-atMost finite-subset)
hence \(?\text{mfr} \in ?\text{cards}\) by (rule Max-in) auto
then obtain \(v\) where \(?\text{mfr} = \text{card} (\text{HOV} \ \text{msgs} \ v)\) by auto
have \(\text{MFR} \ \text{msgs} \ v\)
proof (auto simp: MFR-def)
fix \(w\)
from \(\text{fin}\) have \(\text{card} (\text{HOV} \ \text{msgs} \ w) \leq ?\text{mfr}\) by (rule Max-ge) auto
thus \(\text{card} (\text{HOV} \ \text{msgs} \ w) \leq \text{card} (\text{HOV} \ \text{msgs} \ v)\) by (unfold \(v\))
qued
thus \(?\text{thesis} .\) ..
qued

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

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

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

**definition** TwoThirds where

TwoThirds \( \text{msgs} \text{v} \) \( \equiv \) \( (2 \times N) \text{ div } 3 < \text{card}(\text{HOV} \text{msgs} \text{v}) \)

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 \( \text{v} \) if it received \( \text{v} \) from more than 2/3 of all processes. If \( \text{p} \) 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 \( \epsilon \) is Hilbert’s choice operator.) We require the type of values to be linearly ordered so that the minimum is guaranteed to be well-defined.

**definition** OTR-nextState where

\[
\text{ORT-nextState} \text{r} \text{p} \text{(st::(val::linorder) pstate) msgs st'} \equiv \\
\text{if } (2 \times N) \text{ div } 3 < \text{card} \{q. \text{msgs} q \neq \text{None}\} \text{then st'} = (\{ x = \text{Min} \{v. \text{MFR msgs} v\}, \text{decide} = (\text{if } (\exists v. \text{TwoThirds} \text{msgs} v) \text{then Some } (\epsilon v. \text{TwoThirds} \text{msgs} v) \text{else decide st}) \}) \text{else st'} = st
\]

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

\[
\text{ORT-sendMsg} \text{r} \text{p} \text{q st} \equiv x \text{st}
\]

### 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 \( \Pi \) of processes where \( \Pi \)
contains more than two thirds of all processes. The “per-round” part of the communication predicate is trivial.

definition \( \text{OTR-commPerRd} \) where
\( \text{OTR-commPerRd} \) \( \text{HOrs} \equiv \text{True} \)

definition \( \text{OTR-commGlobal} \) where
\( \text{OTR-commGlobal} \) \( \text{HOs} \equiv \forall r. \exists r_0 \Pi. r_0 \geq r \land (\forall p. \text{HOs} r_0 p = \Pi) \land \text{card} \Pi > (2 \times N) \div 3 \)

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 \( \text{crd} \) arguments of the initial- and next-state predicates are unused.

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

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

end

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

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

5.4 Proof of Integrity

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

definition \( \text{VInv} \) where
\( \text{VInv} \) \( \text{rho} n \equiv \)
let \( \text{xinit} = (\text{range} (x \circ (\text{rho} 0))) \)
in range \( (x \circ (\text{rho} n)) \subseteq \text{xinit} \)
\& range \( (\text{decide} \circ (\text{rho} n)) \subseteq \{\text{None}\} \cup (\text{Some}' \text{xinit}\)
lemma \texttt{vinv-invariant}:

\begin{description}
\item[assumes] \texttt{run: HORun OTR-M rho HOs}
\item[shows] \texttt{VInv rho n}
\end{description}

\textbf{proof (induct \texttt{n})}

\begin{description}
\item[from \texttt{run} show \texttt{VInv rho 0}]
  \begin{itemize}
  \item by (simp add: \texttt{HORun-eq HOinitConfig-eq OTR-HOMachine-def initState-def OTR-initState-def VInv-def image-def})
  \end{itemize}
\item[next fix \texttt{m}]
  \begin{description}
  \item[assume \texttt{ih: VInv rho m}]
  \item[let \texttt{?xinit = range (x o (rho 0))}]
  \item[have range (x o (rho (Suc m))) \subseteq ?xinit]
  \item[proof (clarsimp)]
    \begin{itemize}
      \item fix \texttt{p}
      \item from \texttt{run}
      \item have \texttt{nxt: OTR-nextState m p (rho m p)}
        \begin{itemize}
          \item \texttt{(HOrcvdMsgs OTR-M m p (HOs m p) (rho m))}
          \item \texttt{(rho (Suc m) p)}
          \item (is \texttt{OTR-nextState - - ?st ?msgs ?st'})
        \end{itemize}
      \item by (simp add: \texttt{HORun-eq HOinitConfig-eq OTR-HOMachine-def nextState-def})
      \item show \texttt{x ?st \in ?xinit}
        \begin{itemize}
          \item cases (2*N) div 3 < \texttt{card (HOs m p)}
          \item case \texttt{True}
          \item hence \texttt{HO: HOs m p \neq \{\}} by auto
          \item let \texttt{?MFRs = \{v. MFR ?msgs v\}}
          \item have \texttt{Min ?MFRs \in ?MFRs}
            \begin{itemize}
              \item proof (rule Min-in)
              \item from \texttt{HO} have \texttt{?MFRs \subseteq (the o ?msgs)\{HOs m p\}}
                \begin{itemize}
                  \item by (auto simp: image-def intro: MFR-in-msgs)
                \end{itemize}
              \item thus \texttt{finite ?MFRs by (auto elim: finite-subset)}
            \end{itemize}
          \item next
          \item from \texttt{MFR-exists} show \texttt{?MFRs \neq \{\}} by auto
        \end{itemize}
      \item qed
    \end{itemize}
  \end{description}
\item[with \texttt{HO} have \texttt{\exists q \in HOs m p. Min ?MFRs = the (?msgs q)}}
  \begin{itemize}
    \item by (intro MFR-in-msgs) auto
  \end{itemize}
\item[hence \texttt{\exists q \in HOs m p. Min ?MFRs = x (rho m q)}]
  \begin{itemize}
    \item by (auto simp: \texttt{HOrcvdMsgs-def OTR-HOMachine-def OTR-sendMsg-def})
  \end{itemize}
\item[moreover]
  \begin{description}
    \item from \texttt{True nxt} have \texttt{x ?st' = Min ?MFRs}
      \begin{itemize}
        \item by (simp add: \texttt{OTR-nextState-def HOrcvdMsgs-def})
      \end{itemize}
  \end{description}
\item[ultimately]
  \begin{description}
    \item show \texttt{?thesis using \texttt{ih}} by (auto simp: \texttt{VInv-def image-def})
  \end{description}
\item[next case \texttt{False}]
  \begin{description}
    \item with \texttt{nxt \texttt{ih show \texttt{?thesis}}}
      \begin{itemize}
        \item by (auto simp: \texttt{OTR-nextState-def VInv-def HOrcvdMsgs-def Let-def})
      \end{itemize}
  \end{description}
\item[qed]
\item[qed]
\end{description}
moreover
have \( \forall p. \text{decide} ((\rho (\text{Suc} \, m)) \, p) \in \{\text{None}\} \cup (\text{Some} \, \, ?xinit) \)

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

Integrity is an immediate consequence.

theorem \( \text{OTR-integrity} \):
  assumes run:\( \text{HORun} \, \text{OTR-M} \, \rho \, \text{HOs} \) and \( \text{dec}: \text{decide} \, (\rho \, n \, p) = \text{Some} \, v \)
  shows \( \exists \, q. \, v = x \, (\rho \, 0 \, q) \)
proof –
  let \( \text{?xinit} = \text{range} \, (x \circ (\rho \, 0)) \)
  from run have \( \text{VInv} \, \rho \, n \) by (rule vine-invariant)
  hence \( \text{range} \, (\text{decide} \, (\rho \, n)) \) \( \subseteq \{\text{None}\} \cup (\text{Some} \, \, ?xinit) \)
    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 add: 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 chg
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)
sshows (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 \( \{ q . ?\text{msgs} q = \text{Some} \; v \} \subseteq \{ q . x (\rho \; n \; q) = v \} \)
by (auto simp: OTR-HOMachine-def OTR-sendMsgs-def OTRrcvdMsgs-def)
hence \( \text{card} \{ q . ?\text{msgs} q = \text{Some} \; v \} \leq \text{card} \{ q . x (\rho \; n \; q) = v \} \)
by (simp add: card-mono)
ultimately
show \( \text{thesis} \) 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 \( \text{run} : \text{HORun} \; \text{OTR-M} \; \rho \; \text{HOs} \)
and \( \text{HO} : (2 \ast N) \div 3 < \text{card} \{ q . \text{HOrcvdMsgs} \; \text{OTR-M} \; n \; p \; (\text{HOs} \; n \; p) \; (\rho \; n) \neq \text{None} \} \)
and \( \text{maj} : (2 \ast N) \div 3 < \text{card} \{ q . x (\rho \; n \; q) = v \} \)
shows \( x (\rho \; (\text{Suc} \; n) \; p) = v \)
proof –
from \( \text{run} \\
\text{have} \; \text{nxt} : \text{OTR-nextState} \; n \; p \; (\rho \; n \; p) \;
(\text{HOrcvdMsgs} \; \text{OTR-M} \; n \; p \; (\text{HOs} \; n \; p) \; (\rho \; n)) \\
(\rho \; (\text{Suc} \; n) \; p) \\
\quad \text{(is} \; \text{OTR-nextState} - - \; ?\text{st} \; ?\text{msgs} \; ?\text{st'}) \)
by (simp add: HORun-eq HOnextConfig-eq OTR-HOMachine-def nextState-def)
let \( ?\text{HOVothers} = \bigcup \{ HOV \; ?\text{msgs} \; w \mid w . w \neq v \} \)
— processes from which \( p \) received values different from \( v \)
\( \text{have} \; w : \text{card} \; ?\text{HOVothers} \leq N \div 3 \)
proof –
\( \text{have} \; \text{card} \; ?\text{HOVothers} \leq \text{card} \; (\text{UNIV} - \{ q . x (\rho \; n \; q) = v \}) \)
by (auto simp: HOV-def)
also have \( \ldots = N - \text{card} \{ q . x (\rho \; n \; q) = v \} \)
by (auto simp: card-Diff-subset)
also from \( \text{maj} \) have \( \ldots \leq N \div 3 \) by auto
finally show \( \text{thesis} \).
qed

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

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

Show that \( v \) has been received from more than \( N/3 \) processes.
from \( \text{HO} \) have \( N \div 3 < \text{card} \{ q . ?\text{msgs} \; q \neq \text{None} \} - (N \div 3) \)
by auto
also from \( \text{w} \) have \( \ldots \leq \text{card} \{ q . ?\text{msgs} \; q \neq \text{None} \} - \text{card} \; ?\text{HOVothers} \)
by auto
also from hov othHO have \ldots = card (HOV ?msgs v)
   by (auto simp: card-Diff-subset)
finally have HOV: \( N \div 3 < \text{card} (HOV ?msgs v) \).
All other values are received from at most \( N/3 \) processes.

have \( \forall w. w \not= v \rightarrow \text{card} (HOV ?msgs w) \leq \text{card} \ ?HOVothers \)
   by (force intro: card-mono)
with \( w \) have cardw: \( \forall w. w \not= v \rightarrow \text{card} (HOV ?msgs w) \leq N \div 3 \) by auto
In particular, \( v \) is the single most frequently received value.

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

have \( \text{card} \ \{ q. ?msgs q = \text{Some} v \} \leq \text{card} \ \{ q. ?msgs q \not= \text{None} \} \)
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: \((2*N) \div 3 < \text{card} \ \{ q. (rho n q) = v \}\) (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 \( \forall q. x (rho (n+m) q) = v \rightarrow x (rho (n + Suc m) q) = v \)
proof (rule+)
fix q
assume q: \( x ((rho (n+m)) q) = v \)
let ?msgs = 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 < \text{card} \ \{ q. ?msgs q \not= \text{None} \})
case True

39
from m have \((2N) \div 3 < \text{card} \{ q . x (\rho(n+m)) \} = v \) by simp
next
  case False
  with run q show ?thesis by (auto dest: nextState-change)
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 \(\text{run}: \text{HORun} \; \text{OTR-M} \; \rho \; \text{HOs} \)
and \(\text{dec}: \text{decide} (\rho n p) = \text{Some} \; v \; (\text{is} \; ?\text{dec} \; n) \)
shows \(\forall k. (2N) \div 3 < \text{card} \{ q . x (\rho(n+k)) \} = v \)
  
  (is \(\forall k. \; ?\text{twothird} (n+k)) \)
using \(\text{dec}\) proof (induct \(n\))

— The base case is trivial since no process has decided

assume \(?\text{dec} \; 0\) with \(\text{run} \; \forall k. \; ?\text{twothird} \; (0+k)\)
  
  by (simp add: \text{HORun-eg} \; \text{HOinitConfig-eg} \; \text{OTR-HOMachine-def}
           \text{initState-def} \; \text{OTR-initState-def})
next
— For the inductive step, we assume that process \(p\) has decided on \(v\).

fix \(m\)

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

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

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

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 ?\text{dec} \; m)
  case True with \(ih\) show ?thesis by blast
next
  case False
  with \(\text{run} \; m\) have \(?\text{twothird} \; m\) by (auto elim: \(A1\))
  with \(\text{run} \; \text{show} \; ?\text{thesis}\) by (blast dest: \(A3\))
  qed
  thus \(?\text{twothird} \; ((Suc \ m) + k)\) 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 $\rho$ HO

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

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

shows $v = w$

**Proof**

- from run $p$
  have $(2\ast N) \ div \ 3 < \ card \ \{ q. \ x \ (\rho \ (n+k) \ q) = v \} \ (is - < \ card \ ?V)$
  by (blast dest: A4)

  moreover
  from run $p'$
  have $(2\ast N) \ div \ 3 < \ card \ \{ q. \ x \ (\rho \ ((n+k)+0) \ q) = w \} \ (is - < \ card \ ?W)$
  by (blast dest: A4)

  ultimately
  have $N < \ card \ ?V + card \ ?W$ by auto

  then obtain proc where proc $\in ?V \cap ?W$ by (auto dest: majorities-intersect)

  thus $\vdash \thesis$ by auto

**Theorem OTR-agreement:**

assumes run: HORun OTR-M $\rho$ HO

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

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

shows $v = w$

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

- case True
  then obtain $k$ where $m = n+k$ by (auto simp add: le-iff-add)

  with run $p \ p'$ show $\vdash \thesis$ by (auto elim: A5)

  next

  - case False
    hence $m \leq n$ by auto

    then obtain $k$ where $n = m+k$ by (auto simp add: le-iff-add)

    with run $p \ p'$ have $w = v$ by (auto elim: A5)

    thus $\vdash \thesis$ ..

**Qed**

5.6 Proof of Termination

We now show that every process must eventually decide.

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

Lemma A2 is instrumental in this proof.

**Theorem OTR-termination:**

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

**Proof:**

- from \( \text{commG} \) obtain \( r_0 \) \( II \) where
  - \( \text{pi: } \forall q. \ \text{HOs r0 q} = II \) and \( \text{pic: } \text{card} \ II > (2 N) \div 3 \)
  - by (auto simp: \text{OTR-HOMachine-def OTR-commGlobal-def})
- let \( \text{\modelsmsgs q r = HORcvdMsgs OTR-M r q (HOs r q (rho r))} \)

**Proof:**

- from \( \text{run pi} \) have \( \forall p q. \ \text{\modelsmsgs \ q \ r0 = \modelsmsgs \ p \ r0} \)
  - by (auto simp: \text{HORun-eq OTR-HOMachine-def \text{HOrcvdMsgs-def OTR-sendMsg-def}})
- then obtain \( \mu \) where \( \forall q. \ \text{\modelsmsgs \ q \ r0 = \mu} \) by auto

**Moreover:**

- from \( \text{pi pic} \) have \( \forall p. \ (2 N) \div 3 < \text{card} \ \{q. \ \text{\modelsmsgs p r0 \ q} \neq \text{None} \} \)
  - by (auto simp: \text{HORun-eq HOnextConfig-eq HORcvdMsgs-def})
- with \( \text{run have} \ \forall \ q. \ x (\text{rho (Suc r0) q}) = \text{Min \ \{v. \ MFR (\modelsmsgs q r0) \ v\}} \)
  - by (auto simp: \text{HORun-eq HOnextConfig-eq \text{OTR-HOMachine-def OTR-nextState-def}})

**Ultimately:**

- have \( \forall q. \ x (\text{rho (Suc r0) q}) = \text{Min \ \{v. \ MFR \mu \ v\}} \) by auto
- then obtain \( \nu \) where \( \forall q. \ \forall x (\text{rho (Suc r0) q}) = \nu \) by auto

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

**Proof:**

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

42
qed

from commG obtain \( r_0' \Pi' \)
where \( r_0' : r_0' \geq \text{Suc} \ r_0 \)
and \( \pi' : \forall q. \text{HOs} \ r_0' \ q = \Pi' \)
and \( \pi' : \text{card} \ \Pi' > (2 \ast N) \div 3 \)
by (force simp: OTR-HOMachine-def OTR-commGlobal-def)

from \( r_0' \ P \) have \( \forall q \ x (\text{rho} \ r_0' \ q) = \Pi' \) by (auto simp: le_iff_add)
moreover
from \( \pi' \pi' \ ) have \( (2 \ast N) \div 3 < \text{card} \ \{ q. \ (\text{msgs} \ p \ r_0') \ q \neq \text{None} \} \)
by (auto simp: HOrcvdMsgs-def OTR-sendMsg-def)
moreover
from \( \pi' \pi' \ ) have \( \text{TwoThirds} \ (\text{msgs} \ p \ r_0') \ v \)
by (simp add: TwoThirds-def HOrcvdMsgs-def OTR-HOMachine-def OTR-sendMsg-def HOV-def)
ultimately
have \( \text{decide} \ (\text{rho} \ (\text{Suc} \ r_0') \ p) = \text{Some} \ (\epsilon \ v. \ \text{TwoThirds} \ (\text{msgs} \ p \ r_0') \ v) \)
by (auto simp: OTR-nextState-def)
thus \( \text{thesis} \) 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: \( \text{HORun} \ OTR-M \text{ rho} \ \text{HOs} \) and \( \text{commG} : \text{HOcommGlobal} \ OTR-M \text{ HOs} \)
shows \( \text{consensus} \ (x \circ (\text{rho} \ 0)) \ \text{decide} \ \text{rho} \)
using \( \text{OTR-integrity}[\text{OF} \ run] \ OTR-agreement[\text{OF} \ run] \ OTR-termination[\text{OF} \ run \ \text{commG}] \)
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: \( \text{fg-run} \ OTR-M \text{ rho} \ \text{HOs} \) \( (\lambda r \ q. \ \text{undefined}) \)
and \( \text{commG} : \text{HOcommGlobal} \ OTR-M \text{ HOs} \)
shows \( \text{consensus} \ (\lambda p. \ x \ (\text{state} \ (\text{rho} \ 0) \ p)) \ \text{decide} \ (\text{state} \circ \text{rho}) \)
(is \( \text{consensus} \ ?\text{inits} \ - - \))
proof (rule local-property-reduction[\text{OF} \ run \ \text{consensus-is-local}])
fix \ crun

43
assume crun: CSHORun OTR-M crun HOs HOs (λr q. undefined)
    and init: crun 0 = state (rho 0)
from crun have HORun OTR-M crun HOs by (unfold HORun-def SHORun-def)
from this commG have consensus (x o (crun 0)) decide crun by (rule OTR-consensus)
with init show consensus ?inits decide crun by (simp add: o-def)
qed

end
theory UeDefs
imports ../HOModel
begin

6 Verification of the Uniform Voting Consensus Algorithm

Algorithm Uniform Voting 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 ≡ 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 ≡ 2

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

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

The following record models the local state of a process.

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

```
| datatype 'val msg = Val 'val | Val Vote 'val 'val option | Null — dummy message in case nothing needs to be sent
| definition isValVote where isValVote m ≡ ∃ z v. m = ValVote z v
| definition isVal where isVal m ≡ ∃ v. m = Val v
```

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

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

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

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

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

```
| definition msgRcvd where — processes from which some message was received msgRcvd (msgs::Proc ↦ 'val msg) = { q . msgs q ≠ None}
| definition smallestValRcvd where smallestValRcvd (msgs::Proc ↦ ('val::linorder) msg) ≡ Min { v. ∃ q. msgs q = Some (Val v)}
```

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

```
| definition send0 where send0 r p q st ≡ Val (x st)
| definition next0 where next0 r p st (msgs::Proc ↦ ('val::linorder) msg) st' ≡
   (∃ v. (∀ q ∈ msgRcvd msgs. msgs q = Some (Val v)) ∧ st' = st [‘ vote := Some v, x := smallestValRcvd msgs []]) ∨ ~ (∃ v. ∀ q ∈ msgRcvd msgs. msgs q = Some (Val v)) ∧ 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 ($msgs :: Proc \rightarrow 'val\ msg$) \equiv

\{ q . \exists v.\ msgs\ q = Some\ (ValVote\ v\ None)\}$

**definition** smallestValNoVoteRcvd where

smallestValNoVoteRcvd ($msgs :: Proc \rightarrow 'val\ linorder\ msg$) \equiv

$Min\ \{ v.\ \exists q.\ msgs\ q = Some\ (ValVote\ v\ None)\}$

**definition** someVoteRcvd where

— set of processes from which some vote was received

someVoteRcvd ($msgs :: Proc \rightarrow 'val\ msg$) \equiv

$\{ q .\ q \in msgRcvd\ msgs\ \land\ isValVote\ (the\ (msgs\ q))\ \land\ getvote\ (the\ (msgs\ q)) \neq None\ }$

**definition** identicalVoteRcvd where

identicalVoteRcvd ($msgs :: Proc \rightarrow 'val\ msg$) \equiv

$\forall q \in msgRcvd\ msgs.\ isValVote\ (the\ (msgs\ q))\ \land\ getvote\ (the\ (msgs\ q)) = Some\ v$

**definition** x-update where

$x$-update $st\ msgs\ st' \equiv

$(\exists q \in someVoteRcvd\ msgs\ .\ x\ st' = the\ (getvote\ (the\ (msgs\ q))))\ \lor\ someVoteRcvd\ msgs = \{}\ \land\ x\ st' = smallestValNoVoteRcvd\ msgs$

**definition** dec-update where

dec-update $st\ msgs\ st' \equiv

$(\exists v.\ identicalVoteRcvd\ msgs\ v\ \land\ decide\ st' = Some\ v)\ \lor\ \neg(\exists v.\ identicalVoteRcvd\ msgs\ v)\ \land\ decide\ st' = decide\ st$

**definition** next1 where

next1 $r\ p\ st\ msgs\ st' \equiv

$x$-update $st\ msgs\ st'$

$\land\ dec-update\ st\ msgs\ st'$

$\land\ vote\ st' = None$

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-sendMsg ($r :: nat$) \equiv if step $r = 0$ then send0 $r$ else send1 $r$

**definition** UV-nextState where

UV-nextState $r \equiv if\ step\ r = 0\ then\ next0\ r\ else\ next1\ r$
6.2 Communication Predicate for *Uniform Voting*

We now define the communication predicate for the *Uniform Voting* 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** \( UV\text{-commPerRd} \) where

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

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** \( UV\text{-commGlobal} \) where

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

6.3 The *Uniform Voting* 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 *Uniform Voting* is not a coordinated algorithm.

**definition** \( UV\text{-HOMachine} \) where

\[
UV\text{-HOMachine} = ()
\]

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

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

end
theory UvProof
imports UvDefs ../Reduction
begin

6.4 Preliminary Lemmas

At any round, given two processes \( p \) and \( q \), there is always some process which is heard by both of them, and from which \( p \) and \( q \) have received the same message.
lemma some-common-msg:
assumes HOcommPerRd UV-M (HOs r)
shows \( \exists p, q \in \text{msgRcvd} \ (\text{HOrcvdMsgs UV-M r p} \ (\text{HOs r p}) \ (\rho r)) \)
\( \land \ p \in \text{msgRcvd} \ (\text{HOrcvdMsgs UV-M r q} \ (\text{HOs r q}) \ (\rho r)) \)
\( \land \ (\text{HOrcvdMsgs UV-M r p} \ (\text{HOs r p}) \ (\rho r)) \ p q \)
\( = (\text{HOrcvdMsgs UV-M r q} \ (\text{HOs r q}) \ (\rho r)) \ p q \)
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) (r))
\in \{ v, \exists p. (HOrcvdMsgs UV-M r q (HOs r q) (r)) \ p = Some (Val v) \}
(is smallestValRcvd ?msgs \in {?vals})
unfolding smallestValRcvd-def proof (rule Min-in)
have ?vals \subseteq \text{getval} \ ((\text{the} \ ？msgs) \ (\text{HOs r q}))
by (auto simp: HOrcvdMsgs-def image-def)
thus finite ?vals by (auto simp: finite-subset)
next
from some-common-msg[of HOs, OF com]
obtain p where p \in msgRcvd ?msgs by blast
with s0 show ?vals \neq {} by (auto simp: finite-subset)
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 \neq 0
and nov: someVoteRcvd (HOrcvdMsgs UV-M r q (HOs r q) (r)) = {}
shows smallestValNoVoteRcvd (HOrcvdMsgs UV-M r q (HOs r q) (r))
\in \{ v, \exists p. (HOrcvdMsgs UV-M r q (HOs r q) (r)) \ p = Some (ValVote v None) \}
(is smallestValNoVoteRcvd ?msgs \in {?vals})
unfolding smallestValNoVoteRcvd-def proof (rule Min-in)
have ?vals \subseteq \text{getval} \ ((\text{the} \ ？msgs) \ (\text{HOs r q}))
by (auto simp: HOrcvdMsgs-def image-def)
thus finite ?vals by (auto simp: finite-subset)
next
from some-common-msg[of HOs, OF com]
obtain p where p \in msgRcvd ?msgs by blast
with s1 nov show ?vals \neq {} by (auto simp: finite-subset)
qed

48
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'`)

  *assumee* `r' = 0`

  with `run` show `?thesis`

  by (auto simp: `UV-HOMachine-def HORun-eq HOinitConfig-eq`
   `initState-def UV-initState-def`)

**next**

fix `r`

*assumes* `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: \forall 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` have `?thesis` — no vote in initial state

  by (auto simp: `UV-HOMachine-def HORun-eq HOinitConfig-eq`
   `initState-def UV-initState-def`)

  *next*

  fix `r'`

  *assumes* `r: r = Suc r'`

  let `msgs = HOrcvdMsgs UV-M r' p (HOs r' p) (rho r')`

  from `run` have `nxt: 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: nxt0 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: \forall q \in 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 \in msgRcvd ?msgs`
by (force dest: some-common-msg)
with idv have \{ x . \exists q q = Some (Val x) \} = \{ v \}
  by (auto simp: msgRcvd-def)

hence smallestValRcvd ?msgs = v
  by (auto simp: smallestValRcvd-def)

ultimately

show ?thesis by simp

qed

\[ \text{6.5 Proof of Irrevocability, Agreement and Integrity} \]

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

\begin{proof}

\begin{enumerate}
\item \textbf{lemma decide-step:}
  \begin{enumerate}
  \item \textbf{assumes run: HORun UV-M rho HOs}
  \item and decide: decide (rho (Suc r) p) \neq decide (rho r p)
  \item \textbf{shows step r = 1}
  \end{enumerate}

\item \textbf{proof –}

\begin{enumerate}
\item \textbf{let ?msgs = HOrcvdMsgs UV-M r p (HOs r p) (rho r)}
\item \textbf{from run have nextState UV-M r p (rho r p) ?msgs (rho (Suc r) p)}
\item by (auto simp: HORun-eq HOnextConfig-eq nextState-def)
\item \textbf{with decide show ?thesis}
\item by (auto simp: nextState-def UV-HOMachine-def UV-nextState-def)
\end{enumerate}

\end{enumerate}

\end{proof}

No process ever decides None.

\begin{proof}

\begin{enumerate}
\item \textbf{lemma decide-nonnull:}
  \begin{enumerate}
  \item \textbf{assumes run: HORun UV-M rho HOs}
  \item and decide: decide (rho (Suc r) p) \neq None
  \item \textbf{shows decide (rho (Suc r) p) \neq None}
  \end{enumerate}

\item \textbf{proof –}

\begin{enumerate}
\item \textbf{let ?msgs = HOrcvdMsgs UV-M r p (HOs r p) (rho r)}
\item \textbf{from assms have s1: step r = 1 by (rule decide-step)}
\item \textbf{with run have next1 r p (rho r p) ?msgs (rho (Suc r) p)}
\item by (auto simp: UV-HOMachine-def HORun-eq HOnextConfig-eq nextState-def)
\item \textbf{with decide show ?thesis}
\item by (auto simp: next1-def dec-update-def)
\end{enumerate}

\end{enumerate}

\end{proof}

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.

\begin{proof}

\begin{enumerate}
\item \textbf{lemma msgs-unanimity:}
  \begin{enumerate}
  \item \textbf{assumes run: HORun UV-M rho HOs}
  \item and vote: vote (rho (Suc r) p) = Some v
  \item and q: q \in msgRcvd (HOrcvdMsgs UV-M r p (HOs r p) (rho r))
  \item (is - \in msgRcvd ?msgs)
  \item \textbf{shows getval (the (?msgs q)) = v}
  \end{enumerate}

\end{enumerate}

\end{proof}
proof –
have \( s_0 \): step \( r = 0 \)
proof (rule ccontr)
  assume \( r \neq 0 \)
  hence \( \text{step } (\text{Suc } r) = 0 \) 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 \( UV-M r p \ (\rho r p) \ ?msgs \ (\rho (\text{Suc } r) p) \)
  by (auto simp: HORun-eq HOnextConfig-eq nextState-def)
with \( s_0 \) have nxt: \( \text{next0 } r p \ (\rho r p) \ ?msgs \ (\rho (\text{Suc } r) p) \)
  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 \( \text{vote-agreement}: \)
  assumes \( \text{run: } \text{HORun } UV-M \rho \text{ HO} \)
  \( \text{and } \text{com: } \forall r. \text{HOcommPerRd } UV-M \text{ } (\text{HO} r) \)
  \( \text{and } p: \text{vote } (\rho r p) = \text{Some } v \)
  \( \text{and } q: \text{vote } (\rho r q) = \text{Some } w \)
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 \( r: r = \text{Suc } r' \)
let \( \text{?msgs } p = \text{HORcdMsgs } UV-M \text{ } r' p \ (\text{HO} r' p) \ (\rho r') \)
from \( \text{com} \) obtain \( pq \)
  where \( \text{?msgs } p \ pq = \text{?msgs } q \ pq \)
    and \( \text{smq: } pq \in \text{msgRcvd } (\text{?msgs } p) \text{ and } \text{smq: } pq \in \text{msgRcvd } (\text{?msgs } q) \)
  by (force dest: some-common-msg)
moreover
from run p \( \text{getval } (\text{the } (\text{?msgs } p \ pq)) = v \)
  by (simp add: msgs-unanimity)
moreover
from run q \( \text{getval } (\text{the } (\text{?msgs } q \ pq)) = w \)
  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 \( \text{decide-equals-x}: \)
  assumes \( \text{run: } \text{HORun } UV-M \rho \text{ HO} \)
    \( \text{and } \text{com: } \forall r. \text{HOcommPerRd } UV-M \text{ } (\text{HO} r) \)

and decide: decide (\rho (\text{Suc } r) \ p) \neq \text{decide } (\rho r p)
and decval: decide (\rho (\text{Suc } r) \ p) \trianglerighteq \text{Some } v
shows x (\rho (\text{Suc } r) \ q) = v

proof
let \ ?msgs p' = \text{HOrcvdMsgs } UV-M r p' (\text{HOs } r p') (\rho r)
from run decide have s1: step r = 1 by (rule decide-step)
from run have nextState UV-M r p (\rho r p) (\ ?msgs p) (\rho (\text{Suc } r) \ p)
  by (auto simp: \text{HORun-eq } \text{HOnextConfig-eq } nextState-def)
with s1 have nxtp: next1 r p (\rho r p) (\ ?msgs p) (\rho (\text{Suc } r) \ p)
  by (auto simp: \text{UV-HOMachine-def } nextState-def \text{UV-nextState-def})
from run have nextState UV-M r q (\rho r q) (\ ?msgs q) (\rho (\text{Suc } r) \ q)
  by (auto simp: \text{HORun-eq } \text{HOnextConfig-eq } nextState-def)
with s1 have nxtq: next1 r q (\rho r q) (\ ?msgs q) (\rho (\text{Suc } r) \ q)
  by (auto simp: \text{UV-HOMachine-def } nextState-def \text{UV-nextState-def})

from \text{com obtain } pq where
  pq: pq \in \text{msgRcvd } (\ ?msgs p) \ p q \in \text{msgRcvd } (\ ?msgs q)
  by (force dest: \text{some-common-msg})
with decide decval have vote: isValVote (\text{the } (\ ?msgs p p q))
  by (auto simp: \text{nextState-def dec-update-def } \text{identicalVoteRcvd-def})
with nxtq pq obtain q' where
  q': q' \in \text{someVoteRcvd } (\ ?msgs q)
  by (auto simp: \text{nextState-def } x-update-def \text{someVoteRcvd-def})
with s1 pq vote show \ ?thesis
  by (auto simp: \text{HOrcvdMsgs-def } \text{UV-HOMachine-def } \text{UV-sendMsg-def } \text{send1-def } \text{someVoteRcvd-def } \text{vote-agreement}[OF run \text{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 \text{same-x-stable:}
assumes \text{run}: \text{HORun } UV-M \ rho \text{ HOs}
  and \text{comm}: \forall r. \text{HOcommPerRd } UV-M (\text{HOs } r)
and x: \forall p. x (\rho r p) = v
shows x (\rho (\text{Suc } r) \ q) = v

proof
let \ ?msgs = \text{HOrcvdMsgs } UV-M r q (\text{HOs } r q) (\rho r)
from \text{comm obtain } p \text{ where } \ p \in \text{msgRcvd } \ ?msgs
  by (force dest: \text{some-common-msg})
from run have nextState UV-M r q (\rho r q) (\ ?msgs) (\rho (\text{Suc } r) \ q)
  by (auto simp: \text{HORun-eq } \text{HOnextConfig-eq } nextState-def)
hence next0 r q (\rho r q) \ ?msgs (\rho (\text{Suc } r) \ q) \land step r = 0
  \lor next1 r q (\rho r q) \ ?msgs (\rho (\text{Suc } r) \ q) \land step r \neq 0
  by (auto simp: \text{HORun-eq } \text{HOnextConfig-eq } nextState-def)
with \text{com obtain } p \text{ such that } p \trianglerighteq \text{Some } v
  by (auto simp: \text{UV-HOMachine-def } nextState-def \text{UV-nextState-def})

52
thus \(?thesis\)

proof
  assume \(?nxt0\): \(?nxt0\)
  hence \(x (\text{rho} (\text{Suc} r) q) = \text{smallestValRcvd} \, \text{?msgs}\)
      by (auto simp: \(\text{nxt0-def}\))
  moreover
  from \(?nxt0 \, x\) have \(\forall p \in \text{msgRcvd} \, \text{?msgs}. \, \text{?msgs} p = \text{Some} \, (\text{Val} \, v)\)
      by (auto simp: \text{UV-HOMachine-def} \text{HOrcvdMsgs-def} \text{UV-sendMsg-def}
      \text{msgRcvd-def} \text{send0-def})
  from this \(p\) have \(\{x . \exists p. \, \text{?msgs} p = \text{Some} \, (\text{Val} \, x)\} = \{v\}\)
      by (auto simp: \text{msgRcvd-def})
  hence \(\text{smallestValRcvd} \, \text{?msgs} = v\)
      by (auto simp: \text{smallestValRcvd-def})
  ultimately
  show \(?thesis\) by simp
next
  assume \(?nxt1\): \(?nxt1\)
  show \(?thesis\)
    proof (cases \text{someVoteRcvd} \, \text{?msgs} = \{\})
      case True
      with \(?nxt1 \, x\) True
      have \(x (\text{rho} (\text{Suc} r) q) = \text{smallestValNoVoteRcvd} \, \text{?msgs}\)
        by (auto simp: \text{next1-def} \text{x-update-def})
      moreover
      from \(?nxt1 \, x\) True
      have \(\forall p \in \text{msgRcvd} \, \text{?msgs}. \, \text{?msgs} p = \text{Some} \, (\text{ValVote} \, v \, \text{None})\)
        by (auto simp: \text{UV-HOMachine-def} \text{HOrcvdMsgs-def} \text{UV-sendMsg-def}
        \text{msgRcvd-def} \text{send1-def} \text{someVoteRcvd-def} \text{isValVote-def})
      from this \(p\) have \(\{x . \exists p. \, \text{?msgs} p = \text{Some} \, (\text{ValVote} \, x \, \text{None})\} = \{v\}\)
        by (auto simp: \text{msgRcvd-def})
      hence \(\text{smallestValNoVoteRcvd} \, \text{?msgs} = v\)
        by (auto simp: \text{smallestValNoVoteRcvd-def})
      ultimately show \(?thesis\) by simp
    next
      case False
      with \(?nxt1\) obtain \(p' \, v'\) where
        \(p': p' \in \text{msgRcvd} \, \text{?msgs} \, \text{isValVote} \, (\text{the} \, (\text{?msgs} \, p'))\)
        getvote (the (\text{?msgs} \, p')) = Some \(v'x (\text{rho} (\text{Suc} r) q) = v'\)
        by (auto simp: \text{someVoteRcvd-def} \text{next1-def} \text{x-update-def})
      with \(?nxt1\) have \(x (\text{rho} (\text{Suc} r) q) = x (\text{rho} \, r \, p')\)
        by (auto simp: \text{UV-HOMachine-def} \text{HOrcvdMsgs-def} \text{UV-sendMsg-def}
        \text{msgRcvd-def} \text{send1-def} \text{isValVote-def}
        \text{x-vote-eq[OF \text{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: ∀ r. H0commPerRd UV-M (HOs r)
  and decide: decide (rho (Suc r) p) ≠ decide (rho r p)
  and decval: decide (rho (Suc r) p) = Some v
shows x (rho (Suc r+k) q) = v
proof (induct k arbitrary: q)
  fix q
  from decide-equals-x[OF assms] show x (rho (Suc r + 0) q) = v by simp
next
  fix k q
  assume \( \forall q. x (\rho (\text{Suc } r+k) q) = v \)
  with run com show x (rho (Suc r + Suc k) q) = v
    by (auto dest: same-x-stable)
qed

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

lemma decided-then-past-decision:
assumes run: HORun UV-M rho HOs
  and dec: decide (rho n p) = Some v
shows \( \exists m < n. \) decide (rho (Suc m) p) ≠ decide (rho m p)
  ∧ decide (rho (Suc m) p) = Some v
proof –
let \( \text{dec } k = \text{decide } (\rho k p) \)
have \( \forall m < n. \text{dec } (\text{Suc } m) \neq \text{dec } (\text{Suc } m) 
  
  \rightarrow \text{dec } n \neq \text{Some } v \)
  (is \( \text{?P } n \) is \( \text{?A } n \rightarrow - \))
proof (induct n)
  from run show \( \text{?P } 0 \)
    by (auto simp: HORun-eq UV-HOMachine-def H0initConfig-eq
      initState-def UV-initState-def)
next
  fix n
  assume ih: \( \text{?P } n \) thus \( \text{?P } (\text{Suc } n) \) by force
qed
with dec show \( \text{?thesis } \) by auto
qed

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

lemma x-values-initial:
assumes run: HORun UV-M rho HOs
  and com: ∀ r. H0commPerRd UV-M (H0s r)
shows \( \exists q. x (\rho r p) = x (\rho \theta q) \)
proof (induct r arbitrary: p)
  fix p
  show \( \exists q. x (\rho \theta p) = x (\rho \theta q) \) by auto
next
fix \ r \ p
assume ih: \exists 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 nextState UV-M r p (\rho r p) ?msgs (\rho (Suc r) p) ∧ step r = 0
  \lor nextState UV-M 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 \exists 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 \?nxt0 have ... ∈ \{v . \exists 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)
also with \?nxt1 True have ... ∈ \{v . \exists 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 \?nxt0 have x (\rho (Suc r) p) = x (\rho r q)
  by (rule x-vote-eq)
with ih show ?thesis by auto
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 \((auto \ \text{dest: decided-then-past-decision})\)

from \(\text{run } q \ \text{obtain } l \ \text{where}\)
\[ l: \ \text{decide} \ (\rho (\text{Suc } l) \ q) \neq \ \text{decide} \ (\rho \ l \ q) \]
\[ \ \text{decide} \ (\rho (\text{Suc } l) \ q) = \text{Some } w \]
by \((auto \ \text{dest: decided-then-past-decision})\)

show \(?\text{thesis}\)

proof \(\text{(cases } k \leq l)\)
  case True
  then obtain \(m\) where \(m: l = k + m\) by \((auto \ \text{simp: le-iff-add})\)
  from \(\text{run com } k \ l \ m \ \text{show } ?\text{thesis by blast dest: two-decisions-agree}\) next
  case False
  hence \(l \leq k\) by simp
  then obtain \(m\) where \(m: k = l + m\) by \((auto \ \text{simp: le-iff-add})\)
  from \(\text{run com } k \ l \ m \ \text{show } ?\text{thesis by blast dest: two-decisions-agree}\)
qed

qed

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

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

\textbf{6.6 Proof of Termination}

Two processes having the same\textit{ Heard-Of} set at some round will hold the same value in their \(x\) variable at the next round.

\textbf{lemma} \textit{hoeq-xeq}:
assumes \(\text{run: HORun UV-M rho HOs}\)
and \(\text{com: } \forall r. \ \text{HOcommPerRd UV-M (HOs } r)\)
and \(\text{hoeq: HOs } r \ p = \text{HOs } r \ q\)
shows $x \cdot (\rho \cdot (\text{Suc} \ r) \ p) = x \cdot (\rho \cdot (\text{Suc} \ r) \ q)$

**proof** –

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

from **hooeq** have msgeq: $?\text{msgs} \ p = $?\text{msgs} \ q
  by (auto simp: \text{UV-HOMachine-def} \text{HOrcvdMsgs-def} \text{UV-sendMsg-def} 
      \text{send0-def} \text{send1-def})

show $?\text{thesis}$

**proof** (cases step $r = 0$)
  case True
  with **run** have $\forall \ p. \ \text{next0} \ r \ p \ (\rho \ r \ p) \ (\text{?msgs} \ p) \ (\rho \cdot (\text{Suc} \ r) \ p) \ (\text{is \?next0} \ p)$
    by (force simp: \text{UV-HOMachine-def} \text{HORun-eq} \text{HOnextConfig-eq} 
        \text{nextState-def} \text{UV-nextState-def})
  hence $?\text{next0} \ p \ ?\text{next0} \ q$ by auto
  with msgeq have $?\text{thesis}$ by (auto simp: \text{next0-def})
next
assume stp: step $r \neq 0$

with **run** have $\forall \ p. \ \text{next1} \ r \ p \ (\rho \ r \ p) \ (\text{?msgs} \ p) \ (\rho \cdot (\text{Suc} \ r) \ p) \ (\text{is \?next1} \ p)$
  by (force simp: \text{UV-HOMachine-def} \text{HORun-eq} \text{HOnextConfig-eq} 
        \text{nextState-def} \text{UV-nextState-def})
  hence $x\cdot \text{update} \ (\rho \ r \ p) \ (\text{?msgs} \ p) \ (\rho \cdot (\text{Suc} \ r) \ p)$
    $x\cdot \text{update} \ (\rho \ r \ q) \ (\text{?msgs} \ q) \ (\rho \cdot (\text{Suc} \ r) \ q)$
  by (auto simp: \text{next1-def})
with msgeq have
  $x' \cdot \text{update} \ (\rho \ r \ p) \ (\text{?msgs} \ p) \ (\rho \cdot (\text{Suc} \ r) \ p)$
    $x' \cdot \text{update} \ (\rho \ r \ q) \ (\text{?msgs} \ p) \ (\rho \cdot (\text{Suc} \ r) \ q)$
  by auto
show $?\text{thesis}$

**proof** (cases someVoteRcvd (\?msgs \ p) = \{\})
  case True
  with $x' \cdot \text{show} \ \text{?thesis}$
    by (auto simp: \text{x-update-def})
next
case False
with $x' \cdot \text{stp \ obtain} \ qp \ qq$ \text{where}
  vote (\rho \ r \ qp) = Some (x (\rho \cdot (\text{Suc} \ r) \ p)) \text{ and}
  vote (\rho \ r \ qq) = Some (x (\rho \cdot (\text{Suc} \ r) \ q))
  by (force simp: \text{UV-HOMachine-def} \text{HOrcvdMsgs-def} \text{UV-sendMsg-def} 
      \text{x-update-def} \text{someVoteRcvd-def} \text{isValVote-def} 
      \text{msgRcvd-def} \text{send1-def})
with **run** \text{com \ show} \ \text{?thesis by (rule \text{vote-agreement})}

qed

We now prove that \text{Uniform Voting} terminates.

**theorem** uv-termination:
assumes run: HORun UV-M rho HOs
    and commR: ∀ r. H0commPerRd UV-M (HOs r)
    and commG: H0commGlobal UV-M HOs
shows ∃ r v. decide (rho r p) = Some v
proof –

First obtain a round where all x values agree.

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

Now obtain a round where all votes agree.

def r' ≡ if step (Suc r0) = 0 then Suc r0 else Suc (Suc r0)
have stp': step r' = 0
    by (simp add: r'-def step-def mod-Suc)
have x': ∀ q. x (rho r' q) = ?v
proof (auto simp: r'-def)
   fix q
   from xs show x (rho (Suc r0) q) = ?v ..
next
   fix q
   from run commR xs show x (rho (Suc (Suc r0)) q) = ?v
    by (rule same-x-stable)
qed
have vote': ∀ q. vote (rho (Suc r') q) = Some ?v
proof
   fix q
   let ?msgs = H0rcvdMsgs 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 ∀ q' ∈ msgRcvd ?msgs. ?msgs q' = Some (Val ?v)
    by (auto simp: UV-HOMachine-def H0rcvdMsgs-def UV-sendMsg-def
        send0-def msgRcvd-def)
moreover
from commR have msgRcvd ?msgs ≠ {}
    by (force dest: some-common-msg)
ultimately
show vote (rho (Suc r') q) = Some ?v
    by (auto simp: next0-def)
qed

At the subsequent round, process p will decide.

let r'' = Suc r'
let ?msgs'' = H0rcvdMsgs 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)
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 Uniform Voting Solves Consensus

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

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

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

theorem uv-consensus-fg:
  assumes run: fg-run UV-M rho HOs HOs (\lambda r q. undefined)
      and commR: \forall r. HOcommPerRd UV-M (HOs r)
      and commG: HOcommGlobal UV-M HOs
  shows consensus (\lambda p. x (state (rho 0) p)) decide (state o rho)
      (is consensus ?inits - -)
  proof (rule local-property-reduction[OF run consensus-is-local])
  fix crun
  assume crun: CSHORun UV-M crun HOs HOs (\lambda r 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 \( \text{init show consensus ?init decide crun} \)
by \( (\text{simp add: o-def}) \)
qed

end

theory LastVotingDefs
imports ../HOModel
begin

7 Verification of the \textit{LastVoting} Consensus Algorithm

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

typedcl \( \text{Proc} \) — the set of processes

axiomatization where \( \text{Proc-finite} : \text{OFCLASS}(\text{Proc, finite-class}) \)

instance \( \text{Proc} :: \text{finite} \) by \( (\text{rule Proc-finite}) \)

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

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

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

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

lemma \text{phase-zero} \ ([simp]; \text{phase} 0 = 0)
by \( (\text{simp add: phase-def}) \)

lemma \text{step-zero} \ ([simp]; \text{step} 0 = 0)
by \( (\text{simp add: step-def}) \)

lemma \text{phase-step}: \((\text{phase} r \ast 4) + \text{step} r = r\)
by \( (\text{auto simp add: phase-def step-def}) \)
The following record models the local state of a process.

```haskell
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
  coordΦ :: Proc — coordinator for current phase
```

Possible messages sent during the execution of the algorithm.

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

Characteristic predicates on messages.

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

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

```haskell
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.

```haskell
definition LV-initState where
  LV-initState p st crd ≡
  vote st = None
  ∧ ¬(commt st)
  ∧ ¬(ready st)
  ∧ timestamp st = 0
  ∧ decide st = None
  ∧ coordΦ 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
\textbf{definition} \texttt{valStampsRcvd} \textbf{where} \par
\texttt{valStampsRcvd (msgs :: Proc → ‘val msg) ≡} \par
\{ q . ∃ v ts. msgs q = Some (ValStamp v ts)\} \par
\textbf{definition} \texttt{highestStampRcvd} \textbf{where} \par
\texttt{highestStampRcvd msgs ≡} \par
\texttt{Max \{ ts . ∃ q v. (msgs::Proc → ‘val msg) q = Some (ValStamp v ts)\}} \par

In step 0, each process sends its current \(x\) and \(timestamp\) values to its coordinator. \par
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 \(commit\) field to true. \par
\textbf{definition} \texttt{send0} \textbf{where} \par
\texttt{send0 r p q st ≡} \par
\texttt{if q = coordΦ st then ValStamp (x st) (timestamp st) elseNull} \par
\textbf{definition} \texttt{next0} \textbf{where} \par
\texttt{next0 r p st msgs crd st′ ≡} \par
\texttt{if p = coordΦ st ∧ card (valStampsRcvd msgs) > N div 2} \par
\texttt{then (∃ p v. msgs p = Some (ValStamp v (highestStampRcvd msgs)))} \par
\texttt{∧ st′ = st (\{ vote := Some v, commit := True \})} \par
\texttt{else st′ = st} \par

In step 1, coordinators that have committed send their vote to all processes. \par
Processes update their \(x\) and \(timestamp\) fields if they have received a vote from their coordinator. \par
\textbf{definition} \texttt{send1} \textbf{where} \par
\texttt{send1 r p q st ≡} \par
\texttt{if p = coordΦ st ∧ commit st then Vote (the (vote st)) elseNull} \par
\textbf{definition} \texttt{next1} \textbf{where} \par
\texttt{next1 r p st msgs crd st′ ≡} \par
\texttt{if msgs (coordΦ st) \neq None \land isVote (the (msgs (coordΦ st)))} \par
\texttt{then st′ = st (\{ x := val (the (msgs (coordΦ st))), timestamp := Suc(phase r) \})} \par
\texttt{else st′ = st} \par

In step 2, processes that have current timestamps send an acknowledgement to their coordinator. \par
A coordinator sets its \(ready\) field to true if it receives a majority of acknowledgements. \par
\textbf{definition} \texttt{send2} \textbf{where} \par
\texttt{send2 r p q st ≡} \par
\texttt{if timestamp st = Suc(phase r) \land q = coordΦ st then Ack elseNull} \par

— processes from which an acknowledgement was received \par
\textbf{definition} \texttt{acksRcvd} \textbf{where}
acksRcvd (msgs :: Proc ⇒ 'val msg) ≡ 
{ q . msgs q ≠ None ∧ isAck (the (msgs q)) }

**definition next2 where**

next2 r p st msgs crd st' ≡
if p = coordΦ st ∧ card (acksRcvd msgs) > N div 2
then st' = st (¬ ready := False)
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 ≡
if p = coordΦ st ∧ ready then Vote (the (vote st)) else Null

**definition next3 where**

next3 r p st msgs crd st' ≡
(if msgs (coordΦ st) ≠ None ∧ isVote (the (msgs (coordΦ st))))
then decide st' = decide st
else decide st' = decide st
∧ (if p = coordΦ st
then ¬(ready st') ∧ ¬(commit st')
else ready st' = ready st ∧ commit st' = commit st)
∧ x st' = x st
∧ vote st' = vote st
∧ timestamp st' = timestamp st
∧ coordΦ 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 ⇒ Proc ⇒ Proc ⇒ 'val pstate ⇒ 'val msg where**

LV-sendMsg (r::nat) ≡
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-nextState :: nat ⇒ Proc ⇒ 'val pstate ⇒ (Proc ⇒ 'val msg) ⇒ Proc ⇒ 'val pstate ⇒ bool

where

LV-nextState r ≡
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**

\[ \text{LV-commPerRd} \] where
\[ \text{LV-commPerRd} \; r \; (\text{HO}::\text{Proc} \; \text{HO}) \; (\text{coord}::\text{Proc} \; \text{coord}) \equiv \text{True} \]

**definition**

\[ \text{LV-commGlobal} \] where
\[ \text{LV-commGlobal} \; \text{HOS} \; \text{coords} \equiv \exists \; \text{ph}::\text{nat} \cdot \exists \; \text{c}::\text{Proc} \cdot \]
\[ \forall \; \text{p} \cdot \; \text{coords} \left( 4 \ast \text{ph} \right) \; \text{p} = \text{c} \]
\[ \land \; \text{card} \left( \text{HOS} \left( 4 \ast \text{ph} \right) \; \text{c} \right) > N \div 2 \]
\[ \land \; \text{card} \left( \text{HOS} \left( 4 \ast \text{ph} + 2 \right) \; \text{c} \right) > N \div 2 \]
\[ \land \; \left( \forall \; \text{p} \cdot \; \text{c} \in \text{HOS} \left( 4 \ast \text{ph} + 1 \right) \; \text{p} \cap \text{HOS} \left( 4 \ast \text{ph} + 3 \right) \; \text{p} \right) \]

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** \[ \text{LV-CHOMachine} \] where
\[ \text{LV-CHOMachine} \equiv \]
\[ \emptyset \; \text{CinitState} = \text{LV-initState}, \]
\[ \text{sendMsg} = \text{LV-sendMsg}, \]
\[ \text{CnextState} = \text{LV-nextState}, \]
\[ \text{CHOcommPerRd} = \text{LV-commPerRd}, \]
\[ \text{CHOcommGlobal} = \text{LV-commGlobal} \] \}

**abbreviation**

\[ \text{LV-M} \equiv \text{LV-CHOMachine}::(\text{Proc}, \text{val psstate}, \text{val msg}) \; \text{CHOMachine} \]

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 . ∃ q v. (msgs::Proc → '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 ≠ {}
  obtains p v where msgs p = Some (ValStamp v (highestStampRcvd msgs))
proof –
  let ?ts = {ts . ∃ q v. msgs q = Some (ValStamp v ts)}
  from nempty have ?ts ≠ {} by (auto simp add: valStampsRcvd-def)
  with timeStampsRcvdFinite
  have highestStampRcvd msgs ∈ ?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 ≤ 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: ∀ p. CinitState LV-M p (rho 0 p) (coords 0 p) ⟹ P 0
and step0: \( \forall r. \)
\[
[ \text{step } r = 0; P r; \text{phase } (\text{Suc } r) = \text{phase } r; \text{step } (\text{Suc } r) = 1; \\
\forall p. \text{next0 } r p (\rho r p) \]
\( (\text{HOrcvedMsgs } LV-M r p (\text{HOs } r p) (\rho r)) \)
\( (\text{coords } (\text{Suc } r) p) \)
\( (\rho (\text{Suc } r) p) \]
\( \Rightarrow P (\text{Suc } r) \)

and step1: \( \forall r. \)
\[
[ \text{step } r = 1; P r; \text{phase } (\text{Suc } r) = \text{phase } r; \text{step } (\text{Suc } r) = 2; \\
\forall p. \text{next1 } r p (\rho r p) \]
\( (\text{HOrcvedMsgs } LV-M r p (\text{HOs } r p) (\rho r)) \)
\( (\text{coords } (\text{Suc } r) p) \)
\( (\rho (\text{Suc } r) p) \]
\( \Rightarrow P (\text{Suc } r) \)

and step2: \( \forall r. \)
\[
[ \text{step } r = 2; P r; \text{phase } (\text{Suc } r) = \text{phase } r; \text{step } (\text{Suc } r) = 3; \\
\forall p. \text{next2 } r p (\rho r p) \]
\( (\text{HOrcvedMsgs } LV-M r p (\text{HOs } r p) (\rho r)) \)
\( (\text{coords } (\text{Suc } r) p) \)
\( (\rho (\text{Suc } r) p) \]
\( \Rightarrow P (\text{Suc } r) \)

and step3: \( \forall r. \)
\[
[ \text{step } r = 3; P r; \text{phase } (\text{Suc } r) = \text{Suc } (\text{phase } r); \text{step } (\text{Suc } r) = 0; \\
\forall p. \text{next3 } r p (\rho r p) \]
\( (\text{HOrcvedMsgs } LV-M r p (\text{HOs } r p) (\rho r)) \)
\( (\text{coords } (\text{Suc } r) p) \)
\( (\rho (\text{Suc } r) p) \]
\( \Rightarrow P (\text{Suc } r) \)

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)
(\( \text{coords } (\text{Suc } r) (\rho (\text{Suc } r)) \))
have \( \text{step } r \in \{0,1,2,3\} \) by (auto simp add: step-def)
thus \( P (\text{Suc } r) \)
proof auto
assume stp: \( \text{step } r = 0 \)
  hence \( \text{step } (\text{Suc } r) = 1 \)
  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: \( \text{step } r = \text{Suc } 0 \)
hence \( \text{step} (\text{Suc} \ r) = 2 \)
by (auto simp add: step-def mod-Suc)
with \( \text{ih} \ \text{nxt} \ \text{stp} \ \text{show} \ \text{?thesis} \)
by (intro step1)
  (auto simp: LV-CHOMachine-def CHOnextConfig-eq
   LV-nextState-def LV-sendMsg-def phase-Suc)

next
assume \( \text{stp}: \text{step} \ r = 2 \)

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

next
assume \( \text{stp}: \text{step} \ r = 3 \)

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

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

\textbf{lemma \( \text{LV-Suc} \):}
\begin{itemize}
\item \textbf{assumes} \( \text{run}: \text{CHORun} \ \text{LV-M} \ \text{rho} \ \text{HOs} \ \text{coords} \)
\item \textbf{and \( \text{step}0: \) \[ \begin{array}{l}
\text{step} \ r = 0; \ \text{step} (\text{Suc} \ r) = 1; \ \text{phase} (\text{Suc} \ r) = \text{phase} \ r; \\
\forall \ p. \ \text{next0} \ r \ p \ (\text{rho} \ r \ p) \\
(\text{HOrcvdMsgs} \ \text{LV-M} \ r \ p \ (\text{HOs} \ r \ p) \ (\text{rho} \ r) \) \\
(\text{coords} (\text{Suc} \ r) \ p \ (\text{rho} \ (\text{Suc} \ r) \ p) \] \\
\implies \ P \ r
\end{array} \]}
\item \textbf{and \( \text{step}1: \) \[ \begin{array}{l}
\text{step} \ r = 1; \ \text{step} (\text{Suc} \ r) = 2; \ \text{phase} (\text{Suc} \ r) = \text{phase} \ r; \\
\forall \ p. \ \text{next1} \ r \ p \ (\text{rho} \ r \ p) \\
(\text{HOrcvdMsgs} \ \text{LV-M} \ r \ p \ (\text{HOs} \ r \ p) \ (\text{rho} \ r) \) \\
(\text{coords} (\text{Suc} \ r) \ p \ (\text{rho} \ (\text{Suc} \ r) \ p) \] \\
\implies \ P \ r
\end{array} \]}
\item \textbf{and \( \text{step}2: \) \[ \begin{array}{l}
\text{step} \ r = 2; \ \text{step} (\text{Suc} \ r) = 3; \ \text{phase} (\text{Suc} \ r) = \text{phase} \ r; \\
\forall \ p. \ \text{next2} \ r \ p \ (\text{rho} \ r \ p) \\
(\text{HOrcvdMsgs} \ \text{LV-M} \ r \ p \ (\text{HOs} \ r \ p) \ (\text{rho} \ r) \) \\
(\text{coords} (\text{Suc} \ r) \ p \ (\text{rho} \ (\text{Suc} \ r) \ p) \] \\
\implies \ P \ r
\end{array} \]}
\item \textbf{and \( \text{step}3: \) \[ \begin{array}{l}
\text{step} \ r = 3; \ \text{step} (\text{Suc} \ r) = 0; \ \text{phase} (\text{Suc} \ r) = \text{Suc} (\text{phase} \ r); \\
\forall \ p. \ \text{next3} \ r \ p \ (\text{rho} \ r \ p) \\
(\text{HOrcvdMsgs} \ \text{LV-M} \ r \ p \ (\text{HOs} \ r \ p) \ (\text{rho} \ r) \) \\
(\text{coords} (\text{Suc} \ r) \ p \ (\text{rho} \ (\text{Suc} \ r) \ p) \] \\
\implies \ P \ r
\end{array} \]}
\end{itemize}
shows $P \rho$
proof
  from run
  have $\text{nxt}: \text{CHO}_{\text{nextConfig}} \text{LV-M} \rho (\rho \rho) (\text{HOs} \rho)$
    (coords (Suc $\rho$)) (rho (Suc $\rho$))
    by (auto simp add: CHORun-eq)
  have $\text{step } r \in \{0,1,2,3\}$ by (auto simp add: step-def)
  thus $P \rho$
proof (auto)
  assume $\text{stp}: \text{step } r = 0$
  hence $\text{step} (\text{Suc } r) = 1$
  by (auto simp add: step-def mod-Suc)
  with $\text{nxt stp show } ?\text{thesis}$
  by (intro step0)
    (auto simp: LV-CHOMachine-def CHOnextConfig-eq
      LV-nextState-def LV-sendMsg-def phase-Suc)
next
  assume $\text{stp}: \text{step } r = \text{Suc 0}$
  hence $\text{step} (\text{Suc } r) = 2$
  by (auto simp add: step-def mod-Suc)
  with $\text{nxt stp show } ?\text{thesis}$
  by (intro step1)
    (auto simp: LV-CHOMachine-def CHOnextConfig-eq
      LV-nextState-def LV-sendMsg-def phase-Suc)
next
  assume $\text{stp}: \text{step } r = 2$
  hence $\text{step} (\text{Suc } r) = 3$
  by (auto simp add: step-def mod-Suc)
  with $\text{nxt stp show } ?\text{thesis}$
  by (intro step2)
    (auto simp: LV-CHOMachine-def CHOnextConfig-eq
      LV-nextState-def LV-sendMsg-def phase-Suc)
next
  assume $\text{stp}: \text{step } r = 3$
  hence $\text{step} (\text{Suc } r) = 0$
  by (auto simp add: step-def mod-Suc)
  with $\text{nxt stp show } ?\text{thesis}$
  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 $\text{LV-induct'}$:
  assumes $\text{run}: \text{CHORun LV-M } \rho \text{ HOs coords}$

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

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

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

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

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

shows \( P \ p \ n. \)

by (rule LV-induct[OF run])

(auto intro: init step0 step1 step2 step3)

\text{lemma LV-Suc':}

\text{assumes } \text{run: CHORun LV-M rho HOs coords}

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

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

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

\text{and step3:} \ [ \text{step} \ r = 3; \ \text{step} \ (\text{Suc} \ r) = 0; \ \text{phase} \ (\text{Suc} \ r) = \text{Suc} \ (\text{phase} \ r); \ \\
\text{next3} \ r \ p \ (\rho \ r \ p) \ \\
(\text{HOrcvdMsgs} \ LV-M \ r \ p \ (\text{HOs} \ r \ p) \ (\rho \ r)) \ \\
(\text{coords} \ (\text{Suc} \ r) \ p) \ (\rho \ (\text{Suc} \ r) \ p) \ ] \implies 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** \( \text{timestamp} (\rho n p) \leq (\text{if step } n < 2 \text{ then phase } n \text{ else Suc (phase } n)) \)
  
  \( (\text{is } ?P p n) \)

  **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** \( \text{timestamp} (\rho n p) \leq \text{timestamp} (\rho (\text{Suc } n) p) \)
  
  \( (\text{is } ?P p n \text{ is } ?ts \leq -) \)

  **Proof (rule LV-Suc[OF run, where P=?P])**

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

**Assume stp: step n = 1**

and **next: next1 n p (rho n p)**

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

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

**From stp have ?ts \leq \text{phase } n**

**Using LV-timestamp-bounded[OF run, where n=n, where p=p] by auto**

**With next 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** \( \text{timestamp} (\rho m p) \leq \text{timestamp} (\rho n p) \)
  
  \( (\text{is } ?ts m \leq -) \)

  **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)**

71
The following definition collects the set of processes whose timestamp is beyond a given bound at a system state.

**definition** \( \text{procsBeyondTS} \) where
\[
\text{procsBeyondTS } ts \ cfg \equiv \{ p . \ ts \leq \text{timestamp} (cfg \ p) \}
\]

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

**lemma** \( \text{procsBeyondTS-monotonic} \):

**assumes** run: \( \text{CHORun LV-M rho HOs coords} \)
and \( p: p \in \text{procsBeyondTS } ts (\rho \ m) \) and \( le: m \leq n \)
**shows** \( p \in \text{procsBeyondTS } ts (\rho \ n) \)
**proof**

- from \( p \) have \( ts \leq \text{timestamp} (\rho \ m \ p) \) (is \( - \leq \ ?ts \ m \))
  by (simp add: \( \text{procsBeyondTS-def} \))
moreover
- from \( \text{run} \) \( le \) have \( ?ts \ m \leq ?ts \ n \) by (rule \( \text{LV-timestamp-monotonic} \))
ultimately show \( ?thesis \)
  by (simp add: \( \text{procsBeyondTS-def} \))
**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** \( \text{notStep3EqualCoord} \):

**assumes** run: \( \text{CHORun LV-M rho HOs coords} \) and \( \text{stp:step} \) \( r \neq 3 \)
**shows** \( \text{coord} \ \Phi (\rho \ \text{Suc} \ r \ p) = \text{coord} \ \Phi (\rho \ r \ p) \) (is \( - \) \( \text{P} \ p \ r \))
**by** (rule \( \text{LV-Suc'}[OF \ run, \ where \ P=\text{?P}] \))
(auto simp: \( \text{stp next0-def next1-def next2-def} \))

**lemma** \( \text{coordinators} \):

**assumes** run: \( \text{CHORun LV-M rho HOs coords} \)
**shows** \( \text{coord} \ \Phi (\rho \ r \ p) = \text{coords} \ (4 \ast(\text{phase} \ r)) \ p \)
**proof**

- let \( \text{?r0} = (4 \ast(\text{phase} \ r)) - 1 \)
- let \( \text{?r1} = (4 \ast(\text{phase} \ r)) \)
**have** \( \text{coord} \ \Phi (\rho \ ?r1 \ p) = \text{coords} \ ?r1 \ p \)
**proof** (cases \( \text{phase} \ r \geq 0 \))
  - 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 p))
    (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 p))
    (coords (Suc ?r0) p) (rho (Suc ?r0) p)
  by (auto simp: LV-nextState-def)
  hence coord (rho (Suc ?r0) p) = coords (Suc ?r0) p
  by (auto simp: next3-def)

with True show ?thesis by auto

qed

moreover
  from run
  have coord (rho (Suc (Suc ?r1)) p) = coord (rho ?r1 p)
    (coords (Suc (Suc ?r1)) p) (rho (Suc ?r1) p)
    (coords (Suc ?r1) p) (rho (Suc ?r1) p)
  by (auto simp: notStep3EqualCoord step-def phase-def mod-Suc)

moreover
  have r ∈ {?r1, Suc ?r1, Suc (Suc ?r1), 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 ≠ 1 →→ timestamp (rho Suc r p) = timestamp (rho r p)
(by (rule LV-Suc[OF run, where P=??])
(auto simp: next0-def next1-def next2-def next3-def)

The $x$ field only changes at step 1.

**lemma** notStep1EqualX [rule-format]:
**assumes** run: CHORun LV-M rho HOs coords
**shows** step r ≠ 1 →→ $x$ (rho Suc r p) = $x$ (rho r p) (is $P$ p r)
(by (rule LV-Suc[OF run, where P=??])
(auto simp: next0-def next1-def next2-def next3-def)

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

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

**lemma** commitE:
**assumes** run: CHORun LV-M rho HOs coords and cmt: commit (rho r p)
and conds: \[ 1 \leq \text{step } r; \; \text{coord} \Phi (\text{rho } r \ p) = p; \; \text{vote} (\text{rho } r \ p) \neq \text{None}; \]
\[ \text{card} \{ \ q . \ \text{coord} \Phi (\text{rho } r \ q) = p \} > N \div 2 \]
**shows** $A$
**proof**
- **have** commit (rho r p) →→
  \[ 1 \leq \text{step } r \]
  \[ \land \text{coord} \Phi (\text{rho } r \ p) = p \]
  \[ \land \text{vote} (\text{rho } r \ p) \neq \text{None} \]
  \[ \land \text{card} \{ \ q . \ \text{coord} \Phi (\text{rho } r \ q) = p \} > N \div 2 \]
  (is $P$ p r is - -- $R$ r)
**proof** (rule LV-induct[OF run, where P=??])
— the only interesting step is step 0
fix $n$
**assume** next: 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: ¬ commt (rho n p) by simp
with nxt cm'
have coordΦ (rho n p) = p
  ∧ vote (rho (Suc n) p) ≠ None
  ∧ card (valStampsRcvd (HOrcvdMsgs LV-M n p (HOs n p) (rho n)))
    > N div 2
by (auto simp add: next0-def)
moreover
from stp
have valStampsRcvd (HOrcvdMsgs LV-M n p (HOs n p) (rho n))
  ⊆ {q . coordΦ (rho n q) = p}
by (auto simp: valStampsRcvd-def LV-CHOMachine-def
    HOrcvdMsgs-def LV-sendMsg-def send0-def)
hence card (valStampsRcvd (HOrcvdMsgs LV-M n p (HOs n p) (rho n)))
  ≤ card {q . coordΦ (rho n q) = p}
by (auto intro: card-mono)
moreover
note stp stp' run
ultimately
show ?R (Suc n) by (auto simp: notStep3EqualCoord)
qed
— the remaining cases are all solved by expanding the definitions
qed (auto simp: LV-CHOMachine-def LV-initState-def next1-def next2-def
    next3-def notStep3EqualCoord[OF run])
with cmt show ?thesis by (intro conds, auto)
qed

A process has a current timestamp only if:

- it is at step 2 or beyond,
- its coordinator has committed,
- its \( x \) value is the \( \text{vote} \) of its coordinator.

lemma \text{currentTimestampE}:
assumes run: CHORun LV-M rho HOs coords
and ts: \text{timestamp} (rho r p) = Suc (phase r)
and conds: \[ 2 \leq \text{step } r; \]
  \[ \text{commit } (rho r (coordΦ (rho r p))); \]
  \[ x (rho r p) = \text{the } (\text{vote } (rho r (coordΦ (rho r p)))) \]
\[ \] \[ \Rightarrow \] = A
shows A

proof
let ?ts n = \text{timestamp} (rho n p)
let ?crd n = coordΦ (rho n p)
have ?ts r = Suc (phase r) \[ \Rightarrow \] \[ 2 \leq \text{step } r \]

75
∧ 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.
asume 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

76
assume \( \text{stp}: \text{step } n = 2 \) and \( \text{stp}' : \text{step } (\text{Suc } n) = 3 \)
and \( \text{ph}: \text{phase } (\text{Suc } n) = \text{phase } n \)
and \( \text{ih}: \exists Q \ p \ n \)
and \( \text{nxt}: \text{next2 } n \ p \ (\rho n \ p) \ (\text{HOrcvdMsgs } \text{LV-M } n \ p \ (\text{HOs } n \ p) \ (\rho n)) \)
(coords \( (\text{Suc } n) \ p) \ (\rho \ (\text{Suc } n) \ p) \)

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

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

from \( \text{run } \text{stp} \)

have \( \forall t: \text{vote } (\rho n \ (\text{Suc } n) ((\text{\text{\text{\text{crd } Suc } n))}) = \text{vote } (\rho n (\text{\text{\text{\text{crd } n))}) \)
by (auto simp add: notStep3EqualCoord notStep0EqualVote)

from \( \text{run } \text{stp} \)

have \( \forall t: \text{commt } (\rho n \ (\text{Suc } n) ((\text{\text{\text{\text{crd } Suc } n))}) = \text{commt } (\rho n (\text{\text{\text{\text{crd } n))}) \)
by (auto simp add: notStep3EqualCoord notStep03EqualCommit)

with \( \forall t \ \text{ts } \text{ph } \text{stp} \ \text{stp}' \ \text{ih } \text{nxt} \)

show \( \exists R \ (\text{Suc } n) \)
by (auto simp add: next2-def)
qed

next

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

\[
\begin{align*}
\text{fix } n \\
\text{assume } \text{stp}' : \text{step } (\text{Suc } n) = 0 \\
\text{with } \text{run } \text{LV-timestamp-bounded}[\text{where } n=\text{Suc } n] \\
\text{have } \exists ts \ (\text{Suc } n) \leq \text{phase } (\text{Suc } n) \text{ by auto} \\
\text{thus } \exists Q \ p \ (\text{Suc } n) \text{ by simp} \\
\text{qed} \\
\text{with } \text{ts } \text{show } \exists \text{thesis by (intro cons) auto} \\
\text{qed}
\end{align*}
\]

If a process \( p \) has its \text{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.

\text{lemma readyE:}
\text{assumes run: CHORun LV-M rho HOs coords and rdy: ready } (\rho r p) 

\text{and cons:} \ \begin{aligned}
\text{step } r = 3; 
\text{coord} \Phi (\rho r p) = p; \\
\text{card } \{ q . \text{coord} \Phi (\rho r q) = p \\ \\
\wedge \text{timestamp } (\rho r q) = \text{Suc } (\text{phase } r) \} > N \text{ div } 2
\end{aligned}

\text{shows } P \\

\text{proof –}

\text{let } \exists qs n = \{ q . \text{coord} \Phi (\rho n q) = p \\
\wedge \text{timestamp } (\rho n q) = \text{Suc } (\text{phase } n) \}

77
have \( \text{ready} (\rho r p) \rightarrow \)

\[ \text{step } r = 3 \]
\[ \land \text{coord} \Phi (\rho r p) = p \]
\[ \land \text{card} (\?qs r > N \div 2) \]

(is \( \?Q p r \) is \( \rightarrow \?R p r \))

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

— the interesting case is step 2

fix \( n \)

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

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

and \( \text{nxt}: \text{next2 } n p (\rho n p) (\text{HOrcvdMsgs LV-M } n p (\text{HOs } n p) (\rho n)) \)

(coords (Suc n) p) (rho (Suc n) p)

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

proof

assume \( \text{rdy}: \text{ready } (\rho (\text{Suc } n) p) \)

from \( \text{stp } \text{ih} \) have \( \text{nrdy}: \neg \text{ready } (\rho n p) \) by simp

with \( \text{rdy } \text{nxt} \) have \( \text{coord} \Phi (\rho n p) = p \)

by (auto simp: next2-def)

with \( \text{run } \text{stp} \) have \( \text{coord} \Phi (\rho (\text{Suc } n) p) = p \)

by (simp add: notStep3EqualCoord)

let \( \?acks = \text{acksRcvd } (\text{HOrcvdMsgs LV-M } n p (\text{HOs } n p) (\rho n)) \)

from \( \text{nrdy } \text{rdy } \text{nxt} \) have \( \text{aRcvd}: \text{card } \?acks > N \div 2 \)

by (auto simp: next2-def)

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

proof (clarify)

fix \( q \)

assume \( q : q \in \?acks \)

with \( \text{stp} \)

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

by (auto simp: LV-CHOMachine-def HOrcvdMsgs-def LV-sendMsg-def

acksRcvd-def send2-def isAck-def)

with \( \text{run } \text{stp } \text{ph} \)

show \( \text{coord} \Phi (\rho (\text{Suc } n) q) = p \)

\[ \land \text{timestamp } (\rho (\text{Suc } n) q) = \text{Suc } (\text{phase } (\text{Suc } n)) \]

by (simp add: notStep3EqualCoord notStep1EqualTimestamp)

qed

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

by (intro card-mono) auto

with \( \text{stp'} \) coord \( \text{aRcvd} \) show \( \?R p (\text{Suc } n) \)

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 \( \text{rdy} \) show \( \?thesis \) by (blast intro: conds)

A process decides only if the following conditions hold:

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

**Lemma** decisionE:

**Assumes** run: CHORun LV-M rho HOs coords
**And** dec: decide (rho (Suc r) p) ≠ decide (rho r p)
**And** conds: [ ]
  
  step r = 3;
  
  decide (rho (Suc r) p) = Some (the (vote (rho r (coordΦ (rho r p)))))
  
  ready (rho r (coordΦ (rho r p))); commit (rho r (coordΦ (rho r p)))

[ ] ⟹ 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 commit field, the assertion can be proved directly from the next-state relation.

**Have** 1: step r = 3

∧ ?dec' = Some (the (vote (?cfg (?crd p))))
∧ ready (?cfg (?crd p))
(is ?Q p r)

**Proof** (rule LV-Suc[O F 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 ∧ 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 ∧ timestamp (?cfg q) = Suc (phase r)} ≠ 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 I show ?thesis by (blast intro: conds)
qed
```

### 7.7 Proof of Integrity

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

**lemma lv-integrityInvariant:**

assumes run: CHORun LV-M rho HOs coords
and inv: [ range (x ◦ (rho n)) ⊆ range (x ◦ (rho 0)); range (vote ◦ (rho n)) ⊆ {None} ∪ Some · range (x ◦ (rho 0)); range (decide ◦ (rho n)) ⊆ {None} ∪ Some · range (x ◦ (rho 0)) ] ⇒ A
shows A
proof –
let ?x0 = range (x ◦ rho 0)
let ?x0opt = {None} ∪ Some · ?x0
have range (x ◦ rho n) ⊆ ?x0
  ∧ range (vote ◦ rho n) ⊆ ?x0opt
  ∧ range (decide ◦ rho n) ⊆ ?x0opt
(is ?Inv n is ?X n ∧ ?Vote n ∧ ?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)

```
from run
show \( x (\rho (\text{Suc } n) p) \in \text{range}\ (\lambda q. x (\rho 0 q)) \) \((?P p n)\)
proof (rule LV-Suc [\textbf{where} \( P = ?P \)])
— only \(\text{step1}\) is of interest
assume \(\text{stp}: \text{step } n = 1\)
and \(\text{nxt}: \text{nxt1 } n p (\rho n p)\)
end \(\text{HOrcvdMsgs LV-M } n p (H\text{Os } n p) (\rho n)\)
(coords (Suc n) p) (rho (Suc n) p)
show \(?\text{thesis}\)
proof (cases \rho (\text{Suc } n) p = \rho n p)
case True
with \(x\) show \(?\text{thesis}\) by auto
next
case False
with \(\text{stp nxt} \) have \(\text{cnt: commt } (\rho n (\text{coord}\Phi (\rho n p)))\)
and \(xp: x (\rho (\text{Suc } n) p) = \text{the } (\text{vote } (\rho n (\text{coord}\Phi (\rho n p))))\))
by (auto simp: next1-def LV-CHOMachine-def HOrcvdMsgs-def
LV-sendMsg-def send1-def isVote-def)
from run \(\text{cnt}\) have \(\text{vote } (\rho n (\text{coord}\Phi (\rho n p))) \neq \text{None}\)
by (rule commitE)
moreover
from \(\text{vt}\) have \(\text{vote } (\rho n (\text{coord}\Phi (\rho n p))) \in \text{?x0opt}\)
by (auto simp add: image-def)
moreover
note \(xp\)
ultimately
show \(?\text{thesis}\) by (force simp add: image-def)
qed
— the other steps don’t change \(x\)
next
assume \(\text{step } n = 0\)
with run have \(x (\rho (\text{Suc } n) p) = x (\rho n p)\)
by (simp add: notStep1EqualX)
with \(x\) show \(?\text{thesis}\) by auto
next
assume \(\text{step } n = 2\)
with run have \(x (\rho (\text{Suc } n) p) = x (\rho n p)\)
by (simp add: notStep1EqualX)
with \(x\) show \(?\text{thesis}\) by auto
next
assume \(\text{step } n = 3\)
with run have \(x (\rho (\text{Suc } n) p) = x (\rho n p)\)
by (simp add: notStep1EqualX)
with \(x\) show \(?\text{thesis}\) by auto
qed
qed

Proof of second conjunct

have \(\text{vt’}: \ ?\text{Vote} (\text{Suc } n)\)
proof (clarsimp simp: image-def)
  fix p v
  assume v: vote (rho (Suc n) p) = Some v
  from run
  have vote (rho (Suc n) p) = Some v → v ∈ ?x0 (is ?P p n)
proof (rule LV-Suc[where P=?P])
  — here only step0 is of interest
  assume stp: step n = 0
  and nxt: next0 n p (rho n p)
    (HOrcvdMsgs LV-M n p (HOs n p) (rho n))
    (coords (Suc n) p) (rho (Suc n) p)
  show ?thesis
proof (cases rho (Suc n) p = rho n p)
  case True
  from vt have vote (rho n p) ∈ ?x0opt
    by (auto simp: image-def)
  with True show ?thesis by auto
next
  case False
  from nxt stp False v obtain q where v = x (rho n q)
    by (auto simp: next0-def send0-def LV-CHOMachine-def
         HOrcvdMsgs-def LV-sendMsg-def)
  with x show ?thesis by (auto simp: image-def)
qed
  — the other cases don’t change the vote
next
  assume step n = 1
  with run have vote (rho (Suc n) p) = vote (rho n p)
    by (simp add: notStep0EqualVote)
  moreover
  from vt have vote (rho n p) ∈ ?x0opt
    by (auto simp: image-def)
  ultimately
  show ?thesis by auto
next
  assume step n = 2
  with run have vote (rho (Suc n) p) = vote (rho n p)
    by (simp add: notStep0EqualVote)
  moreover
  from vt have vote (rho n p) ∈ ?x0opt
    by (auto simp: image-def)
  ultimately
  show ?thesis by auto
next
  assume step n = 3
  with run have vote (rho (Suc n) p) = vote (rho n p)
    by (simp add: notStep0EqualVote)
  moreover
  from vt have vote (rho n p) ∈ ?x0opt
ultimately show thesis by auto
qed
with \( v \) show \( \exists q. v = x (\rho_0 q) \) by auto
qed

Proof of third conjunct

have \( \text{dec}' : \exists \text{Decide} (\text{Suc} n) \)
proof (clarsimp simp: image-def)
  fix \( p \) \( v \)
  assume \( v : \exists q. v = x (\rho_0 q) \)
  show \( \exists q. v = x (\rho_0 q) \)
  proof (cases decide (\( \rho_0 (\text{Suc} n) p \) = decide (\( \rho_0 n p \)))
    case True
    with \( \text{dec True} v \) show thesis by (auto simp: image-def)
  next
    case False
    let \( ?\text{crd} = \text{coord} \Phi (\rho_0 n p) \)
    from False run
    have \( \text{d'} : \exists \text{Decide} (\text{Suc} n) p = Some (\text{vote} (\rho_0 n ?\text{crd})) \)
    and \( \text{cmt} : \text{commit} (\rho_0 n ?\text{crd}) \)
    by (auto elim: decisionE)
    from \( \text{vt} \) have \( \text{vtc} : \text{vote} (\rho_0 n ?\text{crd}) \in ?x0\text{opt} \)
    by (auto simp: image-def)
    from \( \text{run cmt} \) have \( \text{vote} (\rho_0 n ?\text{crd}) \neq \text{None} \)
    by (rule commitE)
    with \( \text{d'} v \) \( \text{vtc} \) show thesis by auto
  qed
qed
with \( \text{inv} \) show thesis by simp
qed

Integrity now follows immediately.

theorem lv-integrity:
  assumes run: \( \text{CHORun LV-M} \rho \text{ HOs coords} \)
  and \( \text{dec} : \exists q. v = x (\rho_0 q) \)
  shows \( \exists q. v = x (\rho_0 q) \)
proof –
  from run have \( \exists q. v = x (\rho_0 q) \in \{\text{None}\} \cup \text{range } (x \circ (\rho_0)) \)
  by (rule lv-integrityInvariant) (auto simp: image-def)
  with \( \text{dec} \) 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: commtE)`
- moreover
- from `run cmt' have card { q . coordΦ (rho r q) = p'} > N div 2` by `(blast elim: commtE)`
- ultimately
- obtain `q` where `coordΦ (rho r q) = p` and `p' = coordΦ (rho r q)`
- by `(auto elim: majoritiesE')`
- thus `?thesis by simp`

qed

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

**Lemma readyProcsEqual:**
assumes \( \text{run: CHORun LV-M rho HOs coords} \)
and \( \text{rdy: ready (rho r p) and rdy': ready (rho r p')} \)
shows \( p = p' \)
proof —
  let \( ?C p = \{ q . \text{coord}\Phi \ (rho r q) = p \land \text{timestamp} \ (rho r q) = \text{Suc} \ (\text{phase r}) \} \)
from \( \text{run rdy have card \ (?C p) > N \div 2} \)
  by \( \text{(blast elim: readyE)} \)
moreover
from \( \text{run rdy' have card \ (?C p') > N \div 2} \)
  by \( \text{(blast elim: readyE)} \)
ultimately
obtain \( q \) where \( \text{coord}\Phi \ (rho r q) = p \) and \( p' = \text{coord}\Phi \ (rho r q) \)
by \( \text{(auto elim: majoritiesE')} \)
thus \( ?\text{thesis by simp} \)
qed

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

lemma \text{commitThenVoteRecent}:  
assumes \( \text{run: CHORun LV-M rho HOs coords} \)
and \( \text{maj: card \ (procsBeyondTS ts (rho r)) > N \div 2} \)
and \( \text{cmt: commt (rho r p)} \)
shows \( \exists q \in \text{procsBeyondTS ts (rho r)}. \text{vote (rho r p)} = \text{Some} (x (rho r q)) \)
(\( \text{is } ?Q r \))
proof —
  let \( ?\text{bynd n} = \text{procsBeyondTS ts (rho n)} \)
  have \( \text{card \ (?bynd r) > N \div 2} \land \text{commt (rho r p)} \rightarrow ?Q r \ (\text{is } ?P p r) \)
  proof \( \text{(rule LV-induct[OF run])} \)
next0 establishes the property

  fix \( n \)
  assume \( \text{stp: step n = 0} \)
  and \( \text{nxt: } \forall q. \text{next0 n q (rho n q)} \)
  \( (\text{HOrcvdMsgs LV-M n q (HOs n q) (rho n)}) \)
  \( (\text{coords (Suc n) q}) \)
  \( (\text{rho (Suc n) q}) \)

  (\( \text{is } \forall q. \ ?\text{nxt q} \))
  from \( \text{nxt have } ?\text{nxt p ..} \)
  show \( ?P p \ (\text{Suc n}) \)
  proof \( \text{(clarify)} \)
  assume \( \text{mj: card \ (?bynd (Suc n)) > N \div 2} \)
  and \( \text{ct: commt (rho (Suc n) p)} \)
  show \( ?Q (\text{Suc n}) \)
  proof —
  let \( ?\text{msgs} = \text{HOrcvdMsgs LV-M n p (HOs n p) (rho n)} \)
  from \( \text{stp run have } \neg \text{commt (rho n p)} \) by \( \text{(auto elim: commitE)} \)
  with \( ?\text{msgs} \)
  obtain \( q v \) where
  \( v: ?\text{msgs q} = \text{Some} (\text{ValStamp v (highestStampRcvd ?msgs)}) \) and
vote; vote (\rho (Succ 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' \in \ ?bynd (Succ n) \ and \ q2': q' \in valStampsRcvd ?msgs
by (rule majoritiesE')
have timestamp (\rho n q') \leq 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' \leq 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 (Succ n) q') = timestamp (\rho n q')
  by (simp add: notStep1EqualTimestamp)
moreover
from run stp
have timestamp (\rho (Succ n) q) = timestamp (\rho n q)
  by (simp add: notStep1EqualTimestamp)
moreover
note q1'
ultimately
have q \in \ ?bynd (Succ n)
  by (simp add: procsBeyondTS-def)
moreover
from v vote stp
have vote (\rho (Succ 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 (Succ n) q) = x (\rho n q)
  by (simp add: notStep1EqualX)
ultimately
show \ ?thesis by force
qed
qed
We now prove that \(\text{next1}\) preserves the property. Observe that \(\text{next1}\) 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.

\[
\text{fix } n \\
\text{assume stp: } \text{step } n = 1 \\
\quad \text{and nxt: } \forall q. \text{next1 } n \ q (\rhoo n q) \\
\quad \quad \quad (\text{HOrcvdMsgs } \text{LV-M } n \ q (\text{HOs } n \ q) (\rhoo n)) \\
\quad \quad \quad (\text{coords } (\text{Suc } n) \ q) \\
\quad \quad \quad (\rhoo (\text{Suc } n) \ q) \\
\quad (\text{is } \forall q. \ \text{nxt } q) \\
\quad \text{and ih: } \exists P \ p \ n \\
\text{from } \text{nxt have } \exists P \ p \ .. \\
\text{show } \exists P \ p (\text{Suc } n) \\
\text{proof (clarify)} \\
\quad \text{assumemj': } \text{card } (?\text{bynd } (\text{Suc } n)) > N \text{ div } 2 \\
\quad \quad \text{and ct': } \text{commit } (\rhoo (\text{Suc } n) \ p) \\
\quad \text{from run stp ct'} \text{ have ct': commit } (\rhoo n p) \\
\quad \quad \quad \text{by (simp add: notStep03EqualCommit)} \\
\quad \text{from run stp have vote': } \text{vote } (\rhoo (\text{Suc } n) \ p) = \text{vote } (\rhoo n p) \\
\quad \quad \quad \text{by (simp add: notStep0EqualVote)} \\
\quad \text{show } \exists Q (\text{Suc } n) \\
\quad \text{proof (cases } \exists q \in (?\text{bynd } (\text{Suc } n)). \rhoo (\text{Suc } n) \ q \neq \rhoo n q) \\
\quad \quad \text{case True} \\
\text{in this case the property holds because q updates its x field to the vote} \\
\quad \text{then obtain q where} \\
\quad \quad q1: q \in (?\text{bynd } (\text{Suc } n)) \text{ and q2: } \rhoo (\text{Suc } n) \ q \neq \rhoo n q .. \\
\quad \text{from } \text{nxt have } \exists P \ q .. \\
\quad \text{with q2 stp} \\
\quad \text{have x': } x (\rhoo (\text{Suc } n) \ q) = \text{the } (\text{vote } (\rhoo n (\text{coord}\Phi (\rhoo n q)))) \\
\quad \quad \text{and coord: } \text{commit } (\rhoo n (\text{coord}\Phi (\rhoo n q))) \\
\quad \quad \quad \text{by (auto simp: next1-def send1-def LV-CHOMachine-def HOrcvdMsgs-def} \\
\quad \quad \quad \text{LV-sendMsg-def isVote-def)} \\
\quad \text{from run ct have vote: } \text{vote } (\rhoo n p) \neq \text{None} \\
\quad \quad \text{by (rule commitE)} \\
\quad \text{from run coord ct have coord}\Phi (\rhoo n q) = p \\
\quad \quad \text{by (rule committedProcsEqual)} \\
\quad \text{with q1 x' vote vote' show } \exists P \ p \ .. \\
\text{next} \\
\quad \text{case False} \\
\text{if no relevant process moves then } \text{procsBeyondTS} \text{ doesn't change and we invoke the induction hypothesis} \\
\quad \text{hence bynd: } ?\text{bynd } (\text{Suc } n) = ?\text{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
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** \texttt{XOfTimestampBeyondDecision}: 

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

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

**shows** \( \forall q \in \text{procsBeyondTS \ (Suc \ (phase \ r)) \ (rho \ (r+k))}. \)

\[ x \ (rho \ (r+k) \ q) = \text{the \ (decide \ (rho \ (Suc \ r) \ p))} \]

\( (\text{is \ } \forall q \in \ ?bynd \ k. \ - \ = \ ?v \ \text{is \ } ?P \ p \ k) \)

**proof** (\texttt{induct \ k})

— base step

show \( ?P \ p \ 0 \)

proof (\texttt{clarify})

fix \( q \)

assume \( q: q \in \ ?bynd \ 0 \)

use preceding lemmas about the decision value and the \( x \) field of processes with fresh timestamps.

\( \text{from \ run \ dec} \)

\( \text{have \ stp: \ step \ r = 3} \)

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

\( \text{and \ cmt: \ commit \ (rho \ r \ (coordΦ \ (rho \ r p)))} \)

by (auto elim: decisionE)

from \( \text{stp \ LV-timestamp-bounded[OF \ run, \ where \ n=r]} \)

have \( \text{timestamp \ (rho \ r \ q) \leq \ Suc \ (phase \ r)} \) by simp

with \( q \) have \( \text{timestamp \ (rho \ r \ q) = \ Suc \ (phase \ r)} \)

by (simp add: \text{procsBeyondTS-def})

with \( \text{run} \)

have \( x: x \ (rho \ (r \ q) = \text{the \ (vote \ (rho \ r \ (coordΦ \ (rho \ r \ q))))} \)

\( \text{and \ cmt': \ commit' \ (rho \ r \ (coordΦ \ (rho \ r \ q)))} \)

by (auto elim: currentTimestampE)

from \( \text{run \ cmt' \ have \ coordΦ \ (rho \ r \ p) = coordΦ \ (rho \ r \ q)} \)

by (rule committedProcsEqual)

with \( x \ v \) show \( x \ (rho \ (r+0) \ q) = \ ?v \) by simp

qed

next

— induction step

fix \( k \)

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

show \( ?P \ p \ (Suc \ k) \)

proof (\texttt{clarify})

fix \( q \)

assume \( q: q \in \ ?bynd \ (Suc \ k) \)

— distinguish the kind of transition—only \texttt{step1} is interesting

have \( x \ (rho \ (Suc \ (r + k)) \ q) = \ ?v \ (\text{is \ ?X \ q \ (r+k)}) \)

proof (\texttt{rule LV-Suc[OF \ run, \ where \ P=\ ?X)})

assume \( \text{stp: \ step \ (r + k) = 1} \)

89
and \( \text{nxt: next1} \) \( (r+k) \) \( q \) \( (\text{rho} \ (r+k)) \) \( q \) 
\( (\text{HOrcvdMsgs} \ \text{LV-M} \ (r+k) \) \( q \) \( (\text{HOs} \ (r+k)) \) \( q \) \( (\text{rho} \ (r+k)) \) \( q \) 
\( (\text{coords} \ (\text{Suc} \ (r+k)) \) \( q \) 
\( (\text{rho} \ (\text{Suc} \ (r+k)) \) \( q \) 

\text{show} \ \text{?thesis} 
\text{proof} \ (\text{cases} \ \text{rho} \ (\text{Suc} \ (r+k)) \ q = \text{rho} \ (r+k) \ q) 
\text{case} \ True 
\text{with} \ q \ \text{ih} \ \text{show} \ \text{?thesis} \ \text{by} \ (\text{auto simp: procsBeyondTS-def}) 
\text{next} 
\text{case} \ False 
\text{from} \ \text{run dec} \ \text{have} \ \text{card} \ (\text{?bynd} \ 0) > \ N \ \text{div} \ 2 
\text{by} \ (\text{simp add: decisionThenMajorityBeyondTS}) 
\text{moreover} 
\text{have} \ \text{?bynd} \ 0 \ \subseteq \ ?bynd \ k 
\text{by} \ (\text{auto elim: procsBeyondTS-monotonic[OF run]}]) 
\text{hence} \ \text{card} \ (\text{?bynd} \ 0) \ \leq \ \text{card} \ (\text{?bynd} \ k) 
\text{by} \ (\text{auto intro: card-mono}) 
\text{ultimately} 
\text{have} \ \text{maj: card} \ (\text{?bynd} \ k) > \ N \ \text{div} \ 2 \ \text{by simp} 
\text{let} \ ?\text{crd} = \text{coord}\Phi \ (\text{rho} \ (r+k) \ q) 
\text{from False stp nxt have} 
\text{cmt: commt} \ (\text{rho} \ (r+k) \ ?\text{crd}) \ \text{and} 
\text{x: x} \ (\text{rho} \ (\text{Suc} \ (r+k)) \ q) = \text{the} \ (\text{vote} \ (\text{rho} \ (r+k) \ ?\text{crd})) 
\text{by} \ (\text{auto simp: next1-def LV-CHOMachine-def HOOrcvdMsgs-defLV-sendMsg-def send1-def isVote-def}) 
\text{from run maj cmt stp obtain} \ q' 
\text{where} \ q1': \ q' \in \ ?bynd \ k 
\text{and q2': vote} \ (\text{rho} \ (r+k) \ ?\text{crd}) = \text{Some} \ (x \ (\text{rho} \ (r+k) \ q')) 
\text{by} \ (\text{blast dest: commitThenVoteRecent}) 
\text{with} \ x \ \text{ih} \ \text{show} \ \text{?thesis} \ \text{by} \ \text{auto} 
\text{qed} 
\text{next} 
\text{— all other steps hold by induction hypothesis} 
\text{assume step} \ (r+k) = 0 
\text{with run have} \ x: x \ (\text{rho} \ (\text{Suc} \ (r+k)) \ q) = x \ (\text{rho} \ (r+k) \ q) 
\text{and ts: timestamp} \ (\text{rho} \ (\text{Suc} \ (r+k)) \ q) = \text{timestamp} \ (\text{rho} \ (r+k) \ q) 
\text{by} \ (\text{auto simp: notStep1EqualX notStep1EqualTimestamp}) 
\text{from ts q have} \ q \in \ ?bynd \ k 
\text{by} \ (\text{auto simp: procsBeyondTS-def}) 
\text{with} \ x \ \text{ih} \ \text{show} \ \text{?thesis} \ \text{by} \ \text{auto} 
\text{next} 
\text{assume step} \ (r+k) = 2 
\text{with run have} \ x: x \ (\text{rho} \ (\text{Suc} \ (r+k)) \ q) = x \ (\text{rho} \ (r+k) \ q) 
\text{and ts: timestamp} \ (\text{rho} \ (\text{Suc} \ (r+k)) \ q) = \text{timestamp} \ (\text{rho} \ (r+k) \ q) 
\text{by} \ (\text{auto simp: notStep1EqualX notStep1EqualTimestamp}) 
\text{from ts q have} \ q \in \ ?bynd \ k 
\text{by} \ (\text{auto simp: procsBeyondTS-def}) 
\text{with} \ x \ \text{ih} \ \text{show} \ \text{?thesis} \ \text{by} \ \text{auto} 
\text{next}
assume step \((r+k)\) = 3
with run have \(x \cdot (rho \cdot (Suc \cdot (r+k))) q = x \cdot (rho \cdot (r+k)) q\)
    and ts: timestamp \((rho \cdot (Suc \cdot (r+k))) q = timestamp \((rho \cdot (r+k)) q\)
    by (auto simp: notStep1EqualX notStep1EqualTimestamp)
from ts q have \(q \in ?bynd k\)
    by (auto simp: procsBeyondTS-def)
with x ih show ?thesis by auto
qed
thus \(x \cdot (rho \cdot (r + Suc \cdot k) q) = ?v\) by simp
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 \cdot (Suc \cdot r) p) \neq decide \((rho \cdot r) p)\)
and q: decide \((rho \cdot (Suc \cdot (r+k)) q) \neq decide \((rho \cdot (r+k)) q)\)
shows decide \((rho \cdot (Suc \cdot (r+k))) q = decide \((rho \cdot (Suc \cdot r)) p)\)
proof —
let \(?bynd k = procsBeyondTS \((Suc \cdot (phase r)) \cdot (rho \cdot (r+k))\)
let \(?qcrd = coordΦ \((rho \cdot (r+k)) q)\)
from run p have notNone: decide \((rho \cdot (Suc \cdot r) p) \neq None\)
    by (auto elim: decisionE)
— process \(q\) decides on the vote of its coordinator
from run q have dec: decide \((rho \cdot (Suc \cdot (r+k)) q) = Some \((the \cdot (vote \cdot (rho \cdot (r+k)) ?qcrd)))\)
    and cmt: comm \((rho \cdot (r+k)) ?qcrd)\)
    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 \(?bynd 0 \subseteq ?bynd k\)
    by (auto elim: procsBeyondTS-monotonic)
hence card \((?bynd 0) \leq 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 \(q'1: q' \in ?bynd k\)
    and \(q'2: vote \cdot (rho \cdot (r+k)) ?qcrd = Some \((x \cdot (rho \cdot (r+k)) \cdot q'))\)
    by (auto dest: commitThenVoteRecent)
— the \(x\) field of process \(q'\) is the value \(p\) decided on
from run p \(q'1\)
have \(x \cdot (rho \cdot (r+k)) \cdot q' = the \cdot (decide \cdot (rho \cdot (Suc \cdot r)) p))\)
    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

\[ \text{and } \text{dec} \text{ decide} (\rho \ n \ p) = \text{Some } \ v \]

shows \( \exists m < n. \text{dec} (\rho \ Suc \ m \ p) \neq \text{dec} (\rho \ m \ p) \)

\[ \land \text{dec} (\rho \ Suc \ m \ p) = \text{Some } \ v \]

**proof** –

let \( \text{dec} \ k = \text{dec} \text{ decide} (\rho \ k \ p) \)

have \((\forall \ m < n. \ \text{dec} \ (\rho \ Suc \ m) \neq \text{dec} \ (\rho \ m)) \rightarrow \text{dec} \ (\rho \ Suc \ m) \neq \text{dec} \ (\rho \ m) \)

\[ (\text{is } \text{dec} \ n \ \text{is } \text{dec} \ (\rho \ Suc \ n) \rightarrow \_ ) \]

**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 \( \text{ih} : ?P \ n \)

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

**proof** (clarify)

assume \( p : ?A \ (\text{Suc} \ n) \text{ and } v : \text{dec} \ (\rho \ Suc \ n) = \text{Some } \ v \)

from \( p \) have \( ?A \ n \text{ by simp} \)

with \( \text{ih} \) have \( ?\text{dec} \ n \neq \text{Some } \ v \text{ by simp} \)

moreover

from \( p \)

have \( ?\text{dec} \ (\rho \ Suc \ n) \neq ?\text{dec} \ n \rightarrow ?\text{dec} \ (\rho \ Suc \ n) \neq \text{Some } \ v \text{ by simp} \)

ultimately

have \( ?\text{dec} \ (\rho \ Suc \ n) \neq \text{Some } \ v \text{ by auto} \)

with \( v \) show \( False \text{ by simp} \)

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

\[ \text{and } \text{dec} \text{ decide} (\rho \ m \ p) = \text{Some } \ v \]

shows \( \text{dec} \text{ decide} (\rho \ (m+k) \ p) = \text{Some } \ v \)

**proof** –

from run p obtain \( n \) where

\[ \text{n1: } n < m \text{ and } \]

\[ \text{n2: } \text{dec} \text{ decide} (\rho \ Suc \ n \ p) \neq \text{dec} \text{ decide} (\rho \ n \ p) \text{ and } \]

\[ \text{n3: } \text{dec} \text{ decide} (\rho \ Suc \ n \ p) = \text{Some } \ v \]

by (auto dest: decisionNonNullThenDecided)
have \( \forall i. \text{decide} (\rho (\text{Suc} (n+i)) p) = \text{Some} \; v \) (is \( \forall i. \text{decide} \; i \))

proof

fix \( i \).
show \( \text{decide} (\rho (\text{Suc} (n+i)) p) = \text{Some} \; v \)

proof (induct \( i \))
from \( n3 \) show \( \text{decide} 0 \) by simp

next
fix \( j \).
assume \( \text{ih: \text{decide} (\rho (\text{Suc} (n+j)) p)} \)
show \( \text{decide} (\rho (\text{Suc} (n+j+1)) p) \)
proof (rule ccontr)
assume \( \text{ctr: \neg (\text{decide} (\rho (\text{Suc} (n+j+1)) p)} \)
with \( \text{ih} \)
have \( \text{decide} (\rho (\text{Suc} (n+j)) p) \neq \text{decide} (\rho (\text{Suc} (n+j)) p) \)
by simp
with \( \text{run n2} \)
have \( \text{decide} (\rho (\text{Suc} (n+j)) p) = \text{decide} (\rho (\text{Suc} n) p) \)
by (rule laterProcessDecidesSameValue)
with \( \text{ctr n3} \) show \( \text{False} \) by simp
qed
qed
qed

moreover
from \( n4 \) obtain \( j \) where \( m+k = \text{Suc}(n+j) \)
by (auto dest: less-imp-Suc-add)
ultimately
show \( \text{thesis} \) by auto
qed

theorem lv-agreement:
assumes run: \( \text{CHO\text{Run} LV-M \rho HOs coords} \)
and \( \text{p: \text{decide} (\rho m p) = \text{Some} \; v} \)
and \( \text{q: \text{decide} (\rho n q) = \text{Some} \; w} \)
shows \( v = w \)
proof –
from \( \text{run p obtain k} \)
where \( k1: \text{decide} (\rho (\text{Suc} k) p) \neq \text{decide} (\rho k p) \)
and \( k2: \text{decide} (\rho (\text{Suc} k) p) = \text{Some} \; v \)
by (auto dest: decisionNonNullThenDecided)
from \( \text{run q obtain l} \)
where \( l1: \text{decide} (\rho (\text{Suc} l) q) \neq \text{decide} (\rho l q) \)
and \( l2: \text{decide} (\rho (\text{Suc} l) q) = \text{Some} \; w \)
by (auto dest: decisionNonNullThenDecided)
show \( \text{thesis} \)
proof (cases \( k \leq l \))
case \( \text{True} \)
then obtain \( m \) where \( m: l = k+m \) by (auto simp: le-iff-add)
from \( \text{run k1 l1 m} \)
have \( \text{decide} (\rho (\text{Suc} l) q) = \text{decide} (\rho (\text{Suc} k) p) \)
by (auto elim: laterProcessDecidesSameValue)
with k2 l2 show thesis by simp
next
case False
hence \( l \leq k \) by simp
then obtain m where \( m = k + m \) by (auto simp: le_iff_add)
from run l1 k1 m
have decide (\( \rho (\mathsf{Suc} k) p \)) = decide (\( \rho (\mathsf{Suc} l) q \))
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: \( \text{CHO} \text{Run LV-M} \ \rho \ \text{HOs coords} \)
  and commG: \( \text{CHO} \text{commGlobal LV-M} \ \text{HOs coords} \)
  shows \( \exists r. \ \forall p. \ \text{decide} (\rho r p) \neq \text{None} \)
proof –

The communication predicate implies the existence of a “successful” phase \( ph \), coordinated by some process \( c \) for all processes.

from commG obtain ph c
  where c: \( \forall p. \ \text{coords} (\mathsf{Suc} \ \mathsf{ph}) p = c \)
  and maj0: card (\( \text{HOs (} \mathsf{Suc} \ \mathsf{ph}) c \)) > N div 2
  and maj2: card (\( \text{HOs (} \mathsf{Suc} \ \mathsf{ph} + 2) c \)) > N div 2
  and rcv1: \( \forall p. \ c \in \text{HOs (} \mathsf{Suc} \ \mathsf{ph} + 1) p \)
  and rcv3: \( \forall p. \ c \in \text{HOs (} \mathsf{Suc} \ \mathsf{ph} + 3) p \)
  by (auto simp: LV-CHOMachine-def LV-commGlobal-def)
let \( \mathsf{r0} = \mathsf{Suc} \ \mathsf{ph} \)
let \( \mathsf{r1} = \mathsf{Suc} \ \mathsf{r0} \)
let \( \mathsf{r2} = \mathsf{Suc} \ \mathsf{r1} \)
let \( \mathsf{r3} = \mathsf{Suc} \ \mathsf{r2} \)
let \( \mathsf{r4} = \mathsf{Suc} \ \mathsf{r3} \)

Process \( c \) is the coordinator of all steps of phase \( ph \).

from run c have c\'\( i. \forall p. \ \text{coord} (\rho \ ?r p) = c \)
  by (auto simp add: phase-def coordinators)
with run have c1: \( \forall p. \ \text{coord} (\rho \ ?r1 p) = c \)
  by (auto simp add: step-def mod-Suc notStep3EqualCoord)
with run have c2: \( \forall p. \ \text{coord} (\rho \ ?r2 p) = c \)
  by (auto simp add: step-def mod-Suc notStep3EqualCoord)
with run have c3: \( \forall p. \) coord\( \Phi \) (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: \( \forall p. \) timestamp (rho ?r2 p) = Suc \( ph \)
proof
fix p
let ?msgs = HOrcvdMsgs LV-M ?r1 p (HOs ?r1 p) (rho ?r1)
let ?crd = coord\( \Phi \) (rho ?r1 p)
from run 1 c1 rcv1
have cnd: \( ?\)msgs ?crd \( \neq \) 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: \( \forall p. \) decide (rho ?r4 p) \( \neq \) None
proof
fix p
let ?crd = coordΦ (rho ?r3 p)
from run 3 c3 recv3
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 (λp. x (state (rho 0) p)) decide (state ◦ rho)
     (is consensus ?inits - -)
proof (rule local-property-reduction[OF run consensus-is-local])
fix crun
assume crun: CSHORun 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 o (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 $U_{T,E,\alpha}$ Consensus Algorithm

Algorithm $U_{T,E,\alpha}$ 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 $\alpha$ appear as thresholds of the algorithm and in the communication predicates. Their values can be chosen within certain bounds in order to adapt the algorithm to the characteristics of different systems.

We formalize in Isabelle the correctness proof of the algorithm that appears in [3], using the framework of theory 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 ≡ 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 ≡ 2

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

definition step where step (r::nat) ≡ 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 $UT,E,\alpha$ algorithm and their constraints [3].

**locale** ute-parameters =
  fixes α :: nat and T :: nat and E :: nat
  assumes majE : 2 * E ≥ N + 2 * α
  and majT : 2 * T ≥ N + 2 * α
  and EltN : E < N
  and TltN : T < N

begin

Simple consequences of the above parameter constraints.

**lemma** alpha-lt-N: α < N
*using* EltN majE by auto

**lemma** alpha-lt-T: α < T
*using* majT alpha-lt-N by auto

**lemma** alpha-lt-E: α < E
*using* majE alpha-lt-N by auto

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

In step 0, each process sends its current x. If it receives the value v more than T times, it votes for v, otherwise it doesn’t vote.
In step 1, each process sends its current *vote*. If it receives more than \(\alpha\) votes for a given value \(v\), it sets its \(x\) field to \(v\), else it sets \(x\) to a default value. If the process receives more than \(E\) votes for \(v\), it decides \(v\), otherwise it leaves its decision unchanged.

The overall send function and next-state relation are simply obtained as the composition of the individual relations defined above.
8.2 Communication Predicate for $\mathcal{U}_{T,E,\alpha}$

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 $\alpha$ corrupted messages, and
- every process should receive more than $\max(T, N + 2\cdot \alpha - E - 1)$ correct messages.

[3] also requires that every process should receive more than $\alpha$ correct messages, but this is implied, since $T > \alpha$ (cf. lemma alpha-lt-T).

definition $\text{Ute-commPerRd}$ where
$\text{Ute-commPerRd} \ HOrs \ SHOrs \equiv$
\[\forall p. \ \text{card} (\text{HOrs} p - \text{SHOrs} p) \leq \alpha\]
\[\land \ \text{card} (\text{SHOrs} p \cap \text{HOrs} p) > N + 2\cdot \alpha - E - 1\]
\[\land \ \text{card} (\text{SHOrs} p \cap \text{HOrs} p) > T\]

The global communication predicate requires there exists some phase $\Phi$ such that:

- all HO and SHO sets of all processes are equal in the second step of phase $\Phi$, i.e. all processes receive messages from the same set of processes, and none of these messages is corrupted,
- every process receives more than $T$ correct messages in the first step of phase $\Phi+1$, and
- every process receives more than $E$ correct messages in the second step of phase $\Phi+1$.

The predicate in the article [3] requires infinitely many such phases, but one is clearly enough.

definition $\text{Ute-commGlobal}$ where
$\text{Ute-commGlobal} \ HOs \ SHOs \equiv$
\[\exists \Phi. \ \text{let } r = \text{Suc} (n\text{Steps} \cdot \Phi)\]
\[\ \text{in} \ (3 \pi. \ \forall p. \ \pi = \text{HOs} r p \land \pi = \text{SHOs} r p)\]
\[\land (\forall p. \ \text{card} (\text{SHOs} (\text{Suc} r) p \cap \text{HOs} (\text{Suc} r) p) > T)\]
\[\land (\forall p. \ \text{card} (\text{SHOs} (\text{Suc} (\text{Suc} r)) p \cap \text{HOs} (\text{Suc} (\text{Suc} r)) p) > E))\]

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 $\text{Ute-SHOMachine}$ where
$\text{Ute-SHOMachine} = []$
\( CinitState = (\lambda p \, st \, crd. \ Ute-initState \, p \, st), \)
\( sendMsg = Ute-sendMsg, \)
\( CnextState = (\lambda r \, p \, st \, msgs \, st'. \ Ute-nextState \, r \, p \, st \, msgs \, st'), \)
\( SHOcommPerRd = Ute-commPerRd, \)
\( SHOcommGlobal = Ute-commGlobal \)

abbreviation
\( Ute-M \equiv (Ute-SHOMachine::(Proc, 'val \, pstate, 'val \, msg) \, SHOMachine) \)

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.

\textbf{lemma vote-step:}
\textbf{assumes} \( \text{next: nextState} \ Ute-M \ r \ p \ (\rho \ r \ p) \ \mu \ (\rho \ (\text{Suc} \ r) \ p) \)
\textbf{and} \( \text{vote} \ (\rho \ (\text{Suc} \ r) \ p) \ \neq \ \text{None} \)
\textbf{shows} \( \text{step} \ r = 0 \)
\textbf{proof} (rule ccontr)
\textbf{assume} \( \text{step} \ r \neq 0 \)
\textbf{with} assms \textbf{have} \( \text{vote} \ (\rho \ (\text{Suc} \ r) \ p) = \text{None} \)
\textbf{by} (auto simp:Ute-SHOMachine-def nextState-def Ute-nextState-def next1-def)
\textbf{with} assms \textbf{show} False \textbf{by} auto
\end{proof}

Processes can make a new decision only at second round of each phase.

\textbf{lemma decide-step:}
\textbf{assumes} \( \text{run: SHORun} \ Ute-M \rho \ HOs \ SHOs \)
\textbf{and} \( \text{d1: decide} \ (\rho \ r \ p) \neq \text{Some} \ v \)
\textbf{and} \( \text{d2: decide} \ (\rho \ (\text{Suc} \ r) \ p) = \text{Some} \ v \)
\textbf{shows} \( \text{step} \ r \neq 0 \)
\textbf{proof}
\textbf{assume} \( \text{sr:step} \ r = 0 \)
\textbf{from} \( \text{run} \ \text{obtain} \ \mu \ \text{where} \ Ute-nextState \ r \ p \ (\rho \ r \ p) \ \mu \ (\rho \ (\text{Suc} \ r) \ p) \)
\textbf{unfolding} Ute-SHOMachine-def nextState-def SHORun-eq SHOnextConfig-eq
\textbf{by} force
\textbf{with} \( \text{sr} \ \text{have} \ \text{next0} \ r \ p \ (\rho \ r \ p) \ \mu \ (\rho \ (\text{Suc} \ r) \ p) \)
\textbf{unfolding} Ute-nextState-def \textbf{by} auto
\textbf{hence} \( \text{decide} \ (\rho \ r \ p) = \text{decide} \ (\rho \ (\text{Suc} \ r) \ p) \)
\textbf{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 - α
  and majw: card \{ qq::Proc. F qq = m' \} > E - α
  shows m = m'
proof -
  from majE alpha-lt-N majv majw
  have card \{ qq::Proc. F qq = m \} > N div 2
    and card \{ qq::Proc. F qq = m' \} > N div 2
    by auto
  then obtain qq
    where qq \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

102
No two processes may vote for different values in the same round.

\textbf{lemma} common-vote:

\textbf{assumes} unsafe: \(\text{SHOcommPerRd\ Ute-M\ HO\ SHO}\)

\textbf{and} nxtp: nextState Ute-M \(r\ p\ (\rho\ r\ p)\ \mu p\ (\rho\ (\text{Suc}\ r)\ p)\)

\textbf{and} mup: \(\mu p \in \text{SHOmsgVectors\ Ute-M\ r\ p\ (\rho\ r)\ (HO\ p)\ (SHO\ p)}\)

\textbf{and} nxtpq: nextState Ute-M \(r\ q\ (\rho\ r\ q)\ \mu q\ (\rho\ (\text{Suc}\ r)\ q)\)

\textbf{and} muq: \(\mu q \in \text{SHOmsgVectors\ Ute-M\ r\ q\ (\rho\ r)\ (HO\ q)\ (SHO\ q)}\)

\textbf{and} \(\text{vote\ (\rho\ (\text{Suc}\ r)\ p) = Some\ vp}\)

\textbf{and} \(\text{vote\ (\rho\ (\text{Suc}\ r)\ q) = Some\ vq}\)

\textbf{shows} \(vp = vq\) using \(\text{assms}\)

\textbf{proof} –

\textbf{have} gtn: \(\text{card\ \{qq. sendMsg\ Ute-M\ r\ qq\ p\ (\rho\ r\ qq) = Val\ vp\}}\)

\hspace{1em}+ \(\text{card\ \{qq. sendMsg\ Ute-M\ r\ qq\ q\ (\rho\ r\ qq) = Val\ vq\} > N}\)

\textbf{proof} –

\textbf{have} \(\text{card\ \{qq. sendMsg\ Ute-M\ r\ qq\ p\ (\rho\ r\ qq) = Val\ vp\} > T - \alpha}\)

\hspace{1em}\land \(\text{card\ \{qq. sendMsg\ Ute-M\ r\ qq\ q\ (\rho\ r\ qq) = Val\ vq\} > T - \alpha}\)

\textbf{proof} –

\textbf{from} nxtp \(\text{vp}\) \textbf{have} stp: step \(r = 0\) by (auto simp: vote-step)

\textbf{from} mup

\textbf{have} \(\text{\{qq. }\mu p qq = Some\ (Val\ vp)\}\ \land \ (HO\ p - SHO\ p)\ \subseteq \text{\{qq. sendMsg\ Ute-M\ r\ qq\ p\ (\rho\ r\ qq) = Val\ vp\}}\)

\hspace{1em}\text{(is card ?vrcvd\ - ?ahop \(\subseteq \) ?sentp)}

\hspace{1em}\text{by (auto simp: \text{SHOmsgVectors-def})}

\textbf{hence} \(\text{card\ \{?vrcvd\ - ?ahop\} \leq card\ ?sentp}\)

\hspace{1em}\text{and card \(\{?vrcvd\ - ?ahop\} \geq card\ ?vrcvd - card\ ?ahop\ by (auto simp: card-mono diff-card-le-card-Diff)\}

\textbf{hence} \(\text{card\ ?sentp \geq card\ ?vrcvd - card\ ?ahop by auto}\)

\textbf{moreover}

\textbf{from} nxtp \(\text{stp}\) \textbf{have} next0 \(r\ p\ (\rho\ r\ p)\ \mu p\ (\rho\ (\text{Suc}\ r)\ p)\)

\hspace{1em}\text{by (auto simp: Ute-SHOMachine-def nextState-def Ute-nextState-def)}

\textbf{with} \(\text{vp}\) \textbf{have} \(\text{card\ ?vrcvd}\ > T\)

\textbf{unfolding} next0-def \textbf{by auto}

\textbf{moreover}

\textbf{from} mup

\textbf{have} \(\text{\{qq. }\mu q qq = Some\ (Val\ vq)\}\ \land \ (HO\ q - SHO\ q)\ \subseteq \text{\{qq. sendMsg\ Ute-M\ r\ qq\ q\ (\rho\ r\ qq) = Val\ vq\}}\)

\hspace{1em}\text{(is \?vrcvdq\ - \?ahoq \(\subseteq \) \?sentq)}

\hspace{1em}\text{by (auto simp: \text{SHOmsgVectors-def})}

\textbf{hence} \(\text{card\ \{?vrcvdq\ - ?ahoq\} \leq card\ ?sentq}\)

\hspace{1em}\text{and card \(\{?vrcvdq\ - ?ahoq\} \geq card\ ?vrcvdq - card\ ?ahoq\ by (auto simp: card-mono diff-card-le-card-Diff)\}

\textbf{hence} \(\text{card\ ?sentq \geq card\ ?vrcvdq - card\ ?ahoq by auto}\)

\textbf{moreover}

\textbf{from} nxtp \(\text{stp}\) \textbf{have} next0 \(r\ q\ (\rho\ r\ q)\ \mu q\ (\rho\ (\text{Suc}\ r)\ q)\)

\hspace{1em}\text{by (auto simp: Ute-SHOMachine-def nextState-def Ute-nextState-def)}

\textbf{with} \(\text{vq}\) \textbf{have} \(\text{card\ \{qq. }\mu q qq = Some\ (Val\ vq)\} > T\)

103
by (unfold next0-def, auto)
moreover
from usafe have card ?ahop ≤ α and card ?ahoq ≤ α
  by (auto simp: Ute-SHOMachine-def Ute-commPerRd-def)
ultimately
show ?thesis using alpha-lt-T by auto
qed
thus ?thesis using majT by auto
qed

show ?thesis
proof (rule ccontr)
  assume vpq:vp ≠ vq
  have ∀ qq. sendMsg Ute-M r qq p (rho r qq)
    = sendMsg Ute-M r qq q (rho r qq)
    by (auto simp: Ute-SHOMachine-def Ute-sendMsg-def
       step-def send0-def send1-def)
  with gpq
  have {qq. sendMsg Ute-M r qq p (rho r qq) = Val vp}\n    ∩ {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}\n    ∪ {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}\n    ∪ {qq. sendMsg Ute-M r qq q (rho r qq) = Val vq}) ≤ N
    by (auto simp: card-mono)
  ultimately
  show False by auto
qed
qed

No decision may be taken by a process unless it received enough messages
holding the same value.

lemma decide-with-threshold-E:
assumes run: SHORun Ute-M rho HOs SHOs
and usafe: SHOcommPerRd Ute-M (HOs r) (SHOs r)
and d1: decide (rho r p) ≠ Some v
and d2: decide (rho (Suc r) p) = Some v
shows card {q. sendMsg Ute-M r q p (rho r q) = Vote (Some v)}
   > E − α
proof –
from run obtain μp
  where nxt:nextState Ute-M r p (rho r p) μp (rho (Suc r) p)
  and ∀ qq. qq ∈ HOs r p ←→ μp qq ≠ None
  and ∀ qq. qq ∈ SHOs r p ∩ HOs r p
       → μp qq = Some (sendMsg Ute-M r qq p (rho r qq))
unfolding Ute-SHOMachine-def SHORun-eq SHOnextConfig-eq SHOmsgVectors-def
by blast

hence \{ qq, \mu p qq = Some (Vote (Some v)) \} \subseteq \{ qq, sendMsg Ute-M r qq p (rho r qq) = Vote (Some v) \}
(is \( ?\text{rcvd}p - ?\text{ahop} \subseteq ?\text{sent}p \)) by auto

hence card \((?\text{rcvd}p - ?\text{ahop})\) \leq card ?\text{sent}p
and card \((?\text{rcvd}p - ?\text{ahop})\) \geq card ?\text{rcvd}p - card ?\text{ahop}
by (auto simp: card-mono diff-card-le-card-Diff)

hence card ?\text{sent}p \geq card ?\text{rcvd}p - card ?\text{ahop} by auto

moreover
from unsafe have card \((\text{HOs r p} - \text{SHOs r p})\) \leq \alpha
by (auto simp: Ute-SHOMachine-def Ute-commPerRd-def)

moreover
from run d1 d2 have step r \neq 0 by (rule decide-step)
with nat have next1 r p (rho r p) \mu p (rho (Suc r) p)
by (auto simp: Ute-SHOMachine-def nextState-def Ute-nextState-def)

with run d1 d2 have card \{ qq, \mu p qq = Some (Vote (Some v)) \} > E
unfolding next1-def by auto
ultimately

show \?thesis using alpha-lt-E by auto

qed

8.5 Proof of Agreement and Validity

If more than \( E - \alpha \) messages holding \( v \) are sent to some process \( p \) at round \( r \), then every process \( pp \) correctly receives more than \( \alpha \) such messages.

lemma common-x-argument-1:
assumes unsafe:SHOcommPerRd Ute-M (HOs (Suc r)) (SHOs (Suc r))
and threshold: card \{ q, sendMsg Ute-M (Suc r) q p (rho (Suc r) q) = Vote (Some v) \} > E - \alpha
(is card (?msgs p v) > -)
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 \( \forall q, sendMsg Ute-M (Suc r) q p (rho (Suc r) q) = sendMsg Ute-M (Suc r) q pp (rho (Suc r) q) \)
by (auto simp: Ute-SHOMachine-def Ute-sendMsg-def step-def send0-def send1-def)

moreover
from unsafe
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))
If more than $E - \alpha$ messages holding $v$ are sent to $p$ at some round $r$, then any process $pp$ will set its $x$ to value $v$ in $r$.

**Lemma** common-x-argument-2:
- **Assumes** run: SHORun Ute-M rho HOs SHOs
- and unsafe: $\forall r$. SHOrandPerRd Ute-M (HOs r) (SHOs r)
- and nxtpp: nextState Ute-M (Suc r) pp (rho (Suc r) pp) $\mu_{pp} (rho (Suc (Suc r))) pp$
- and mupp: $\mu_{pp} \in SHOmsgVectors Ute-M (Suc r) pp (rho (Suc r)) (HOs (Suc r) pp) (SHOs (Suc r) pp)$
- and threshold: $\text{card} \{ q. sendMsg Ute-M (Suc r) q p (rho (Suc r) q) = Vote (Some v) \} > E - \alpha$
- shows $x (rho (Suc (Suc r))) pp = v$

**Proof**
- **Have** stp: step (Suc r) $\neq 0$
  - **Proof**
    - **Assume** sr: step (Suc r) = 0
    - **Hence** $\forall q. sendMsg Ute-M (Suc r) q p (rho (Suc r) q) = Val (x (rho (Suc r) q))$
      - **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: $\text{card} \{ qq. \mu_{pp} qq = Some (Vote (Some v)) \} > \alpha$
- **Is card** $(?msgs v) > \alpha$

**Proof**
- **From** mupp
  - **Have** SHOs (Suc r) pp $\cap$ HOs (Suc r) pp $\subseteq \{ qq. \mu_{pp} qq = Some (sendMsg Ute-M (Suc r) qq pp (rho (Suc r) qq)) \}$
    - **Unfolding** SHOmsgVectors-def by auto
  - **Moreover**
    - **Hence** $(?msgs v) \supseteq (?sent pp v) \cap (SHOs (Suc r) pp \cap HOs (Suc r) pp)$
      - **By** auto
    - **Hence** $\text{card} \{ (?msgs v) \}$
      - $\geq \text{card} \{ (?sent pp v) \cap (SHOs (Suc r) pp \cap HOs (Suc r) pp) \}$

106
by (auto intro: card-mono)
moreover
from usafe threshold
have alph:card ((?sent pp v) ∩ (SHOs (Suc r) pp ∩ HOs (Suc r) pp)) > α
  by (blast dest: common-x-argument-1)
ultimately
show ?thesis by auto
qed
moreover
from nxtpp stp
have next1 (Suc r) pp (rho (Suc r) pp) µpp (rho (Suc (Suc r)) pp)
  by (auto simp: Ute-SHOMachine-def nextState-def Ute-nextState-def)
ultimately
obtain w where wa:card (?msgs w) > α and xw:x (rho (Suc (Suc r)) pp) = w
  unfolding next1-def by auto
have v = w
proof –
  note usafe
moreover
obtain qv where qv ∈ SHOs (Suc r) pp and µpp qv = Some (Vote (Some v))
proof –
  have ¬ (?msgs v ⊆ HOs (Suc r) pp - SHOs (Suc r) pp)
    proof
    assume ?msgs v ⊆ HOs (Suc r) pp - SHOs (Suc r) pp
    hence card (?msgs v) ≤ card ((HOs (Suc r) pp) - (SHOs (Suc r) pp))
      by (auto simp: card-mono)
    moreover
    from usafe
    have card (HOs (Suc r) pp - SHOs (Suc r) pp) ≤ α
      by (auto simp: Ute-SHOMachine-def Ute-commPerRd-def)
    moreover
    note va
    ultimately
    show False by auto
    qed
    then obtain qv
    where qv ∉ HOs (Suc r) pp - SHOs (Suc r) pp
      and qv:µpp qv = Some (Vote (Some v))
      by auto
    with mupp have qv ∈ SHOs (Suc r) pp
      unfolding SHOmsgVectors-def by auto
    with qv that show ?thesis by auto
    qed
  with stp mupp have vote (rho (Suc r) qv) = Some v
    by (auto simp: Ute-SHOMachine-def SHOmsgVectors-def
      Ute-sendMsg-def send1-def)
moreover
obtain qw where
\(qw \in \text{SHOs} (\text{Suc } r) \text{ pp and } \mu \text{pp } qw = \text{Some } (\text{Vote } (\text{Some } w))\)

**proof**

**have** \(\neg (\text{msgs } w \subseteq \text{HOs} (\text{Suc } r) \text{ pp} - \text{SHOs} (\text{Suc } r) \text{ pp})\)

**proof**

**assume** \(\text{msgs } w \subseteq \text{HOs} (\text{Suc } r) \text{ pp} - \text{SHOs} (\text{Suc } r) \text{ pp}\)

**hence** \(\text{card }(\text{msgs } w) \leq \text{card } ((\text{HOs} (\text{Suc } r) \text{ pp}) - (\text{SHOs} (\text{Suc } r) \text{ pp}))\)

by (auto simp: card-mono)

**moreover**

**have** \(\text{card } (\text{HOs} (\text{Suc } r) \text{ pp} - \text{SHOs} (\text{Suc } r) \text{ pp}) \leq \alpha\)

by (auto simp: Ute-SHOMachine-def Ute-commPerRd-def)

**moreover**

**note** \(wa\)

ultimately

**show** \(\text{False}\) by auto

**qed**

then obtain \(qw\)

where \(qw \notin \text{HOs} (\text{Suc } r) \text{ pp} - \text{SHOs} (\text{Suc } r) \text{ pp}\)

and \(qw: \mu \text{pp } qw = \text{Some } (\text{Vote } (\text{Some } w))\)

by auto

with \(mupp\) have \(qw \in \text{SHOs} (\text{Suc } r) \text{ pp}\)

unfolding \(\text{SHOmsgVectors-def}\) by auto

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

**qed**

with \(stp \ mupp\) have \(\text{vote } (\rho (\text{Suc } r) \text{ qw}) = \text{Some } w\)

by (auto simp: Ute-SHOMachine-def \(\text{SHOmsgVectors-def}\)

\(\text{Ute-sendMsg-def send1-def}\))

**moreover**

from \(\text{run}\) obtain \(\muqv \muqw\)

where \(\text{nextState } Ute-M r \text{ kv } ((\rho (\text{Suc } r) \text{ kv}) \muqv (\rho (\text{Suc } r) \text{ kv}))\)

and \(\muqv \in \text{SHOmsgVectors } Ute-M r \text{ kv } (\rho (\text{Suc } r) \text{ kv}) (\text{SHOs } r \text{ kv})\)

and \(\text{nextState } Ute-M r \text{ kv } ((\rho (\text{Suc } r) \text{ kv}) \muqv (\rho (\text{Suc } r) \text{ kv}))\)

and \(\muqv \in \text{SHOmsgVectors } Ute-M r \text{ kv } (\rho (\text{Suc } r) \text{ kv}) (\text{SHOs } r \text{ kv})\)

by (auto simp: Ute-SHOMachine-def SHORun-eq SHOnextConfig-eq) blast

ultimately

**show** \(?\text{thesis}\) using \(\text{usafe}\) by (auto dest: common-vote)

**qed**

with \(xw\) show \(x (\rho (\text{Suc } (\text{Suc } r)) \text{ pp}) = v\) by auto

**qed**

Inductive argument for the agreement and validity theorems.

**lemma** \(\text{safety-inductive-argument:}\)

**assumes** \(\text{run}: \text{SHORun } Ute-M \rho \text{ HOs SHOs}\)

and \(\text{comm}: \forall r. \text{SHOcommPerRd } Ute-M (\text{HOs } r) (\text{SHOs } r)\)

and \(\text{sh}: E - \alpha < \text{card } \{q. \text{sendMsg } Ute-M r' q p (\rho (r' q)) = \text{Vote } (\text{Some } v)\}\)

and \(\text{stp1}: \text{step } r' = \text{Suc } 0\)

**shows** \(E - \alpha < \text{card } \{q. \text{sendMsg } Ute-M (\text{Suc } (r')) q p (\rho (\text{Suc } (r')) q) = \text{Vote } (\text{Some } v)\}\)

108
proof –

from \( stp1 \) have \( r' > 0 \) by (auto simp: step-def)
with \( stp1 \) obtain \( r \) where \( rr',r' = Suc r \) and \( stp:step (Suc r) = Suc 0 \)
by (auto dest: gr0-implies-Suc)

have \( \forall pp. x \ (\rho (Suc r)) \ pp = v \)
proof
  fix \( pp \)
from \( run \) obtain \( \mu pp \)
where \( \mu pp \in SHOmsgVectors Ute-M r' \ pp \ (\rho r') \ (HOs r' \ pp) \ (SHOs r' \ pp) \)
  and \( nextState Ute-M r' pp (\rho r' pp) \ \mu pp \ (\rho (Suc r') pp) \)
by (auto simp: Ute-SHOMachine-def SHORun-eq SHOnextConfig-eq)
with \( run \) comm ih \( rr' \) show \( x \ (\rho (Suc r)) \ pp = v \)
by (auto dest: common-x-argument-2)
qed

with \( \mu pp \)
from \( run \) obtain \( \mu pp \)
where \( \mu pp \in SHOmsgVectors Ute-M r' \ pp (\rho r') \ (HOs r' \ pp) \ (SHOs r' \ pp) \)
  and \( nextState Ute-M r' pp (\rho r' pp) \ \mu pp \ (\rho (Suc r') pp) \)
by (auto simp: Ute-SHOMachine-def SHORun-eq SHOnextConfig-eq
Ute-sendMsg-def send0-def mod-Suc step-def)
with \( rr' \)

have \( \lambda p \ \mu pp'. \ \mu pp' \in SHOmsgVectors Ute-M (Suc r') p (\rho (Suc r')) \)
  \( (HOs (Suc r') p) \ (SHOs (Suc r') p) \)
  \( \subseteq \{ q. \ \mu pp' \ q = Some (Val v) \} \)
by (auto simp: SHOmsgVectors-def)

hence \( \lambda p \ \mu pp'. \ \mu pp' \in SHOmsgVectors Ute-M (Suc r') p (\rho (Suc r')) \)
  \( (HOs (Suc r') p) \ (SHOs (Suc r') p) \)
  \( \subseteq \{ q. \ \mu pp' \ q = Some (Val v) \} \)
by (auto simp: card-mono)
moreover
from \( \mu pp \) have \( \lambda p. T < \text{card} (HOs (Suc r') p) \cap HOs (Suc r') p \)
by (auto simp: Ute-SHOMachine-def Ute-commPerRd-def)
ultimately

have \( \mu pp. \ \mu pp' \in SHOmsgVectors Ute-M (Suc r') p (\rho (Suc r')) \)
  \( (HOs (Suc r') p) \ (SHOs (Suc r') p) \)
  \( \subseteq \{ q. \ \mu pp' \ q = Some (Val v) \} \)
by (auto dest: le-les-trans)

show \( ?thesis \)

proof –

have \( \forall pp. \ \rho (Suc (Suc r')) \ pp = Some v \)
proof
  fix \( pp \)
from \( run \) obtain \( \mu pp \)
where \( nxtpp: nextState Ute-M (Suc r') pp (\rho (Suc r') pp) \ \mu pp \ (\rho (Suc (Suc r')) pp) \)
  and \( mupp: \ \mu pp \in SHOmsgVectors Ute-M (Suc r') pp (\rho (Suc r')) \)
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 \( \exists m < n. \, decide (rho (Suc m) p) \neq decide (rho m p) \land decide (rho (Suc m) p) = Some v \)

**Proof**
- let \(?dec k = decide ((rho k) p)\)
- have \((\forall m < n. \, ?dec (Suc m) \neq ?dec m \implies ?dec (Suc m) \neq Some v)\)
- \(\implies ?dec n \neq Some v\)
- \(\text{(is ?P n is } ?A n \implies \_}\)
- **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 \( r1 r < Suc r \)
and \( dn1: \) decide (rho \( r1 \) p1) \( \neq \) Some v1
and \( dv1': \) decide (rho (Suc \( r1 \)) p1) = Some v1
by (auto dest: decisionNonNullThenDecided)

from \( r1 r \) obtain \( s \) where \( r1 Suc r = Suc (r1 + s) \)
by (auto dest: less-imp-Suc-add)
then obtain \( k' \) where \( kk': r + k = r1 + k' \)
by auto
with \( dn2 dv2 \)
have \( dv2': \) 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 \( step (r1 + k') = step (Suc 0 + k) \)
unfolding step-def by (rule 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) \)
\( = Vote (Some v1) \} > E - \alpha \)

**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 v_1 \} > E - \alpha 

by \( \text{intro decide-with-threshold-E} \) auto

next

fix \( k'' \)

assume \( \text{ih: } E - \alpha < \)

\[
\text{card} \{ q. \text{sendMsg Ute-M} (r_1 + k'' * nSteps) q p_1 (\rho (r_1 + k'' * nSteps)) q \} = \text{Vote} \{ \text{Some v_1} \}
\]

from \( rs0 \) have \( stps: \text{step} (r_1 + k'' * nSteps) = \text{Suc} 0 \)

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

with \( \text{run \ comm ih} \)

have \( E - \alpha < \)

\[
\text{card} \{ q. \text{sendMsg Ute-M} (\text{Suc} (r_1 + k'' * nSteps)) q p_1 (\rho (\text{Suc} (r_1 + k'' * nSteps))) q \} = \text{Vote} \{ \text{Some v_1} \}
\]

by \( \text{rule safety-inductive-argument} \)

thus \( E - \alpha < \)

\[
\text{card} \{ q. \text{sendMsg Ute-M} (r_1 + \text{Suc} k'' * nSteps) q p_1 (\rho (r_1 + \text{Suc} k'' * nSteps)) q \} = \text{Vote} \{ \text{Some v_1} \}
\]

by \( \text{auto} \)

qed

moreover

from \( \text{run} \)

have \( \forall q. \text{sendMsg Ute-M} (r_1 + k'' * nSteps) q p_1 (\rho (r_1 + k'' * nSteps)) q \) \( = \text{sendMsg Ute-M} (r_1 + k'' * nSteps) q p_2 (\rho (r_1 + k'' * nSteps)) q \)

by \( \text{auto simp: Ute-SHOMachine-def Ute-sendMsg-def step-def send0-def send1-def} \)

moreover

from \( \text{run \ comm dn2'' \ dv2''} \)

have \( E - \alpha < \)

\[
\text{card} \{ q. \text{sendMsg Ute-M} (r_1 + k'' * nSteps) q p_2 (\rho (r_1 + k'' * nSteps)) q \} = \text{Vote} \{ \text{Some v_2} \}
\]

by \( \text{auto dest: decide-with-threshold-E} \)

ultimately

show \( v_2 = v_1 \) by \( \text{auto dest: unique-majority-E-\alpha} \)

qed

The Agreement property is an immediate consequence of the two preceding lemmas.

**theorem ute-agreement:**

assumes \( \text{run: SHORun Ute-M rho HOs SHOs} \)

and \( \text{comm: } \forall r. \text{SHOCcommPerRd Ute-M (HOs r) (SHOs r)} \)

and \( p: \text{decide (rho m p) = Some v} \)

and \( q: \text{decide (rho n q) = Some w} \)

shows \( v = w \)

proof –

from \( \text{run p obtain } k \)

- **where** \( k1: \text{decide (rho (Suc k) p) \# decide (rho k p)} \)
and \( k_2 \): \( \text{decide} (\rho (\text{Suc } k)) p \) = Some \( v \) by \((\text{auto dest: decisionNonNullThenDecided})\)

\text{from run } q \text{ obtain } l
\begin{align*}
\text{where } l_1 & : \text{decide} (\rho (\text{Suc } l) q) \neq \text{decide} (\rho l q) \\
\text{and } l_2 & : \text{decide} (\rho (\text{Suc } l) q) = \text{Some } w \\
\text{by } & (\text{auto dest: decisionNonNullThenDecided})
\end{align*}

\text{show } \text{?thesis}

\text{proof } (\text{cases } k \leq l)
\begin{align*}
\text{case } \text{True} \\
& \text{then obtain } m \text{ where } m: l = k + m \text{ by } (\text{auto simp add: le-iff-add}) \\
& \text{from run } \text{comm } k_2 l_1 l_2 m \text{ have } w = v \\
& \text{thus } \text{?thesis } \text{by simp}
\end{align*}

\text{next}
\begin{align*}
\text{case } \text{False} \\
& \text{hence } l \leq k \text{ by simp} \\
& \text{then obtain } m \text{ where } m: k = l + m \text{ by } (\text{auto simp add: le-iff-add}) \\
& \text{from run } \text{comm } l_2 k_1 k_2 m \text{ show } \text{?thesis} \\
& \text{by } (\text{auto elim!: laterProcessDecidesSameValue})
\end{align*}

\text{qed}

Main lemma for the proof of the Validity property.

\text{lemma } \text{validity-argument:}
\begin{align*}
\text{assumes } & \text{run: } \text{SHORun } \text{Ute-M } \text{rho } \text{HOs } \text{SHOs} \\
\text{and } & \text{comm: } \forall r. \text{SHOcommPerRd } \text{Ute-M } (\text{HOs } r) (\text{SHOs } r) \\
\text{and } & \text{init: } \forall x. (\rho 0 p) = v \\
\text{and } & \text{dw: } \text{decide} (\rho r p) = \text{Some } w \\
\text{and } & \text{stp: } \text{step } r' = \text{Suc } 0 \\
\text{shows } & \text{card } \{q. \text{sendMsg } \text{Ute-M } r' q p (\rho r' q) = \text{Vote } (\text{Some } v)\} > E - \alpha
\end{align*}

\text{proof --}
\begin{align*}
& \text{from stp obtain } k \text{ where } \text{stp};r' = \text{Suc } 0 + k * \text{nSteps} \\
& \quad \text{unfolding } \text{step-def using mod-Suc mod-nD by blast}
\end{align*}

\text{moreover}
\begin{align*}
& \text{have } E - \alpha < \\
& \quad \text{card } \{q. \text{sendMsg } \text{Ute-M } (\text{Suc } 0 + k*nSteps) q p ((\rho (\text{Suc } 0 + k*nSteps)) q) = \text{Vote } (\text{Some } v)\}
\end{align*}

\text{proof } (\text{induct } k)
\begin{align*}
& \text{have } \forall pp. \text{vote } ((\rho (\text{Suc } 0)) pp) = \text{Some } v \\
& \text{proof}
\end{align*}

\text{fix } pp
\begin{align*}
& \text{from run obtain } \mu pp \\
& \quad \text{where } \text{nxtpp;nextState } \text{Ute-M } 0 pp (\rho 0 pp) \mu pp (\rho (\text{Suc } 0) pp) \\
& \quad \text{and } \text{mapp};\mu pp \in \text{SHOmsgVectors } \text{Ute-M } 0 pp (\rho 0 ) (\text{HOs } 0 pp) (\text{SHOs } 0 pp) \\
& \quad \text{by } (\text{auto simp: Ute-SHOMachine-def SHORun-eq SHOnextConfig-eq})
\end{align*}

\text{have } \text{majv:card } \{q. \mu pp q = \text{Some } (\text{Val } v)\} > T

\text{proof --}

113
from run init have \( \forall q. \text{sendMsg Ute-M} 0 q \ pp (\rho 0 q) = \text{Val} v \)
by (auto simp: Ute-SHOMachine-def SHORun-eq SHOnextConfig-eq
Ute-sendMsg-def send0-def step-def)

moreover from comm have shoT:card (SHOs 0 pp \( \cap \) HOs 0 pp) > T
by (auto simp: SHOmsgVectors-def)

moreover from mupp have SHOs 0 pp \( \cap \) HOs 0 pp \( \subseteq \) \{q. \mu pp q = \text{Some} (\text{sendMsg Ute-M} 0 q \ pp (\rho 0 q))\}
by (auto simp: SHOmsgVectors-def)

hence card (SHOs 0 pp \( \cap \) HOs 0 pp) \( \leq \) card \{q. \mu pp q = \text{Some} (\text{sendMsg Ute-M} 0 q \ pp (\rho 0 q))\}
by (auto simp: card-mono)

ultimately show \(?thesis\) by (auto simp: less-le-trans)

qed

moreover from nxtpp have next0 0 pp ((\rho 0) pp) \mu pp (\rho (\text{Suc} 0) pp)
by (auto simp: Ute-SHOMachine-def nextState-def Ute-nextState-def step-def)

ultimately obtain \( w \) where majw:card \{q. \mu pp q = \text{Some} (\text{Val} w)\} > T
and votew:vote ((\rho (\text{Suc} 0)) pp) = Some \( w \)
by (auto simp: next0-def)

from majw majw have \( v = w \) by (auto dest: unique-majority-T)
with votew show vote ((\rho (\text{Suc} 0)) pp) = Some \( v \) by simp

qed

with run have card \{q. \text{sendMsg Ute-M} (\text{Suc} 0) q p (\rho (\text{Suc} 0) q) = \text{Vote} (\text{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. \text{sendMsg Ute-M} (\text{Suc} 0 + 0 * nSteps) q p (\rho (\text{Suc} 0 + 0 * nSteps) q) = \text{Vote} (\text{Some} \( v \))\}
using majE EltN by auto

next
fix \( k \)
assume ih:E - \alpha <
card \{q. \text{sendMsg Ute-M} (\text{Suc} 0 + k * nSteps) q p (\rho (\text{Suc} 0 + k * nSteps) q)
= \text{Vote} (\text{Some} \( v \))\}

have step (\text{Suc} 0 + k * nSteps) = \text{Suc} 0
by (auto simp: mod-Suc step-def)
from run comm ih this
have E - \alpha <
card \{q. \text{sendMsg Ute-M} (\text{Suc} (\text{Suc} 0 + k * nSteps))) q p

114
\[(\rho (\text{Suc} (\text{Suc} (\text{Suc} 0 + k \cdot n\text{Steps})))) q) = \text{Vote} (\text{Some} v)\]

by (rule safety-inductive-argument)

thus \(E - \alpha < \)

\[\text{card} \{q. \text{sendMsg Ute-M (Suc} 0 + \text{Suc} k \cdot n\text{Steps}) q p (\rho (\text{Suc} 0 + \text{Suc} k \cdot n\text{Steps}) q) = \text{Vote} (\text{Some} v)\} \text{ by simp}\]

qed

ultimately

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

qed

The following theorem shows the Validity property of algorithm \(U_{T,E,\alpha}\).

**theorem ute-validity:**

assumes run: \(\text{SHORun Ute-M rho HOs SHOs}\)

and comm: \(\forall r. \text{SHOcommPerRd Ute-M (HOs} r) (\text{SHOs} r)\)

and init: \(\forall p, x (\rho 0 p) = v\)

and dw: decide \((\rho r p) = \text{Some} w\)

shows \(v = w\)

**proof** –

from run dw obtain r1

where dnr1: decide \(((\rho r1) p) \neq \text{Some} w\)

and dnr1: decide \(((\rho (\text{Suc} r1)) p) = \text{Some} w\)

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 asms

have \(E - \alpha < \)

\[\text{card} \{q. \text{sendMsg Ute-M r1 q p (rho r1 q) = Vote} (\text{Some} v)\} \text{ by (rule validity-argument)}\]

moreover

from run comm dnr1 dnr1

have \(\text{card} \{q. \text{sendMsg Ute-M r1 q p (rho r1 q) = Vote} (\text{Some} w)\} > E - \alpha\)

by (auto dest: decide-with-threshold-E)

ultimately

show \(v = w\) by (auto dest: unique-majority-E-\(\alpha\))

qed

### 8.6 Proof of Termination

At the second round of a phase that satisfies the conditions expressed in the global communication predicate, processes update their \(x\) variable with the value \(v\) they receive in more than \(\alpha\) messages.

**lemma set-x-from-vote:**

assumes run: \(\text{SHORun Ute-M rho HOs SHOs}\)

and comm: \(\text{SHOcommPerRd Ute-M (HOs} r) (\text{SHOs} r)\)

and stp: step \((\text{Suc} r) = \text{Suc} 0\)

and \(\pi: \forall p. \text{HOs} (\text{Suc} r) p = \text{SHOs} (\text{Suc} r) p\)
and \( \text{nxt: nextState Ute-M (Suc r) p (rho (Suc r) p) \( \mu \) (rho (Suc (Suc r)) p) \) }
and \( \mu u: \mu \in \text{SHOmsgVectors Ute-M (Suc r) p (rho (Suc r))} \)
\( \text{(HOs (Suc r) p) (SHOs (Suc r) p) \) }
and \( \text{wp: } \alpha < \text{card \{qq. } \mu \text{ qq = Some (Vote (Some v))\}} \)
shows \( x ((\rho (\text{Suc (Suc r)})) p) = v \)

proof –
from \( \text{nxt stp wp} \text{ obtain wp} \)
where \( \text{wp: } \alpha < \text{card \{qq. } \mu \text{ qq = Some (Vote (Some wp))\}} \)
and \( \text{wp: } x (\rho (\text{Suc (Suc r)})) p) = wp \)
by \( \text{(auto simp: Ute-SHOMachine-def nextState-def Ute-nextState-def next1-def)} \)

have \( wp = v \)
proof –
from \( \text{wp obtain pp where smw: } \mu \text{ pp = Some (Vote (Some wp))} \)
by force
have \( \text{vote (rho (Suc r) pp) = Some wp} \)
proof –
from \( \text{smw mu} \pi \)
have \( \mu \text{ pp} = \text{Some (sendMsg Ute-M (Suc r) pp p (rho (Suc r) pp))} \)
unfolding \( \text{SHOmsgVectors-def by force} \)
with \( \text{stp have } \mu \text{ pp = Some (Vote (vote (rho (Suc r) pp)))} \)
by \( \text{(auto simp: Ute-SHOMachine-def Ute-sendMsg-def send1-def)} \)
with \( \text{smw show ?thesis by auto} \)
qed
moreover
from \( \text{wp obtain} \text{ qq where smw: } \mu \text{ qq = Some (Vote (Some v))} \)
by force
have \( \text{vote (rho (Suc r) qq) = Some v} \)
proof –
from \( \text{smw mu} \pi \)
have \( \mu \text{ qq} = \text{Some (sendMsg Ute-M (Suc r) qq p (rho (Suc r) qq))} \)
unfolding \( \text{SHOmsgVectors-def by force} \)
with \( \text{stp have } \mu \text{ qq = Some (Vote (vote (rho (Suc r) qq))} \)
by \( \text{(auto simp: Ute-SHOMachine-def Ute-sendMsg-def send1-def)} \)
with \( \text{smw show ?thesis by auto} \)
qed
moreover
from \( \text{run obtain } \mu pp \mu qq \)
where \( \text{nextState Ute-M r pp (rho r pp) } \mu pp (\rho (\text{Suc r) pp}) \)
and \( \mu pp \in \text{SHOmsgVectors Ute-M r pp (rho r) (HOs r pp) (SHOs r pp)} \)
and \( \text{nextState Ute-M r qq ((rho) r qq) } \mu qq (\rho (\text{Suc r) qq}) \)
and \( \mu qq \in \text{SHOmsgVectors Ute-M r qq (rho r) (HOs r qq) (SHOs r qq)} \)
unfolding \( \text{Ute-SHOMachine-def SHORun-eq SHOnextConfig-eq by blast} \)
ultimately
show ?thesis using \( \text{comm by (auto dest: common-vote)} \)
qed
with \( \text{xp show ?thesis by simp} \)
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** \( \text{run} : \text{SHORun} \ Ute-M \ \rho \ \text{HOs} \ \text{SHOs} \)
- **And** \( \text{comm} : \text{SHOcommPerRd} \ Ute-M \ (\text{HOs} \ r) \ (\text{SHOs} \ r) \)
- **And** \( \pi : \forall p. \ \pi 0 = \text{HOs} \ (\text{Suc} \ r) \ p \land \pi 0 = \text{SHOs} \ (\text{Suc} \ r) \ p \)

**Obtains** \( v \) where

\[
\bigwedge p \mu p' q.
\]

\[
\begin{cases}
q \in \text{SHOs} \ (\text{Suc} \ (\text{Suc} \ r)) \ p \cap \text{HOs} \ (\text{Suc} \ (\text{Suc} \ r)) \ p; \\
\mu' \in \text{SHOmsgVectors} \ Ute-M \ (\text{Suc} \ (\text{Suc} \ r)) \ p \ (\rho \ (\text{Suc} \ (\text{Suc} \ r))) \ (\text{HOs} \ (\text{Suc} \ (\text{Suc} \ r)) \ p) \ (\text{SHOs} \ (\text{Suc} \ (\text{Suc} \ r)) \ p)
\end{cases}
\implies \mu' q = (\text{Some} \ (\text{Val} \ v))
\]

**Proof**

- **From** \( \pi \) **have** \( \text{hosho} : \forall p. \ \text{SHOs} \ (\text{Suc} \ r) \ p = \text{SHOs} \ (\text{Suc} \ r) \ p \cap \text{HOs} \ (\text{Suc} \ r) \ p \)
  **by simp**

  **Have** \( \bigwedge p \ q. \ x \ (\rho \ (\text{Suc} \ (\text{Suc} \ r)) \ p) = x \ (\rho \ (\text{Suc} \ (\text{Suc} \ r)) \ q) \)

  **Proof**
  - **Fix** \( p \ q \)
  - **From** \( \text{run} \) **obtain** \( \mu p \)
    **Where** \( \text{nxt} : \text{nextState} \ Ute-M \ (\text{Suc} \ r) \ p \ (\rho \ (\text{Suc} \ r) \ p) \)
    \( \mu p \ (\rho \ (\text{Suc} \ (\text{Suc} \ r)) \ p) \)
    **And** \( \mu u : \mu p \in \text{SHOmsgVectors} \ Ute-M \ (\text{Suc} \ r) \ p \ (\rho \ (\text{Suc} \ r)) \ (\text{HOs} \ (\text{Suc} \ r) \ p) \ (\text{SHOs} \ (\text{Suc} \ r) \ p) \)
    **By** (auto simp: Ute-SHOMachine-def SHORun-eq SHOnextConfig-eq)

  **From** \( \text{run} \) **obtain** \( \mu q \)
  **Where** \( \text{nxtq} : \text{nextState} \ Ute-M \ (\text{Suc} \ r) \ q \ (\rho \ (\text{Suc} \ r) \ q) \)
  \( \mu q \ (\rho \ (\text{Suc} \ (\text{Suc} \ r)) \ q) \)
  **And** \( \mu q : \mu q \in \text{SHOmsgVectors} \ Ute-M \ (\text{Suc} \ r) \ q \ (\rho \ (\text{Suc} \ r)) \ (\text{HOs} \ (\text{Suc} \ r) \ q) \ (\text{SHOs} \ (\text{Suc} \ r) \ q) \)
  **By** (auto simp: Ute-SHOMachine-def SHORun-eq SHOnextConfig-eq)

  **Have** \( \forall qq. \ \mu p \ qq = \mu q \ qq \)
  **Proof**
  - **Fix** \( qq \)
  - **Show** \( \mu p \ qq = \mu q \ qq \)
  **Proof** (cases \( \mu p \ qq = \text{None} \))
    - **Case** False
      **With** \( \mu u \ \pi \) **have** \( 1 : qq \in \text{SHOs} \ (\text{Suc} \ r) \ p \) **and** \( 2 : qq \in \text{SHOs} \ (\text{Suc} \ r) \ q \)
      **Unfolding** SHOmsgVectors-def **by auto**
      **From** \( \mu u \ \pi \) **1**
      **Have** \( \mu p \ qq = \text{Some} \ (\text{sendMsg} \ Ute-M \ (\text{Suc} \ r) \ qq \ p \ (\rho \ (\text{Suc} \ r) \ qq)) \)
      **Unfolding** SHOmsgVectors-def **by auto**
      **Moreover**
      **From** \( \mu u \ \pi \) **2**
      **Have** \( \mu q \ qq = \text{Some} \ (\text{sendMsg} \ Ute-M \ (\text{Suc} \ r) \ qq \ q \ (\rho \ (\text{Suc} \ r) \ qq)) \)
      **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 \notin HOs (Suc r) p unfolding SHOMsgVectors-def by auto

with \pi muq have \mu qq = None unfolding SHOMsgVectors-def by auto

with True show \(?thesis by simp

qed

qed

hence \(\forall vsets. \forall v. \{qq. \mu p qq = Some (Vote (Some v))\} = \{qq. \mu q qq = Some (Vote (Some v))\}

by auto

show \((\rho (Suc (Suc r)) p) = x (\rho (Suc (Suc r)) q)\)

proof (cases \(\exists v. \alpha < \text{card} \{qq. \mu p qq = Some (Vote (Some v))\}, \text{clarify})

fix v

assume vp: \(\alpha < \text{card} \{qq. \mu p qq = Some (Vote (Some v))\}

with run comm stp \pi nxt mu have \(x (\rho (Suc (Suc r)) p) = v\)

by (auto dest: set-x-from-vote)

moreover

from vsets vp have \(\alpha < \text{card} \{qq. \mu q qq = Some (Vote (Some v))\} by auto

with run comm stp \pi nxtq muq have \(x (\rho (Suc (Suc r)) q) = v\)

by (auto dest: set-x-from-vote)

ultimately

show \((\rho (Suc (Suc r)) p) = x (\rho (Suc (Suc r)) q)\)

by auto

next

assume nov: \(\neg (\exists v. \alpha < \text{card} \{qq. \mu p qq = Some (Vote (Some v))\})\)

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 \(\neg (\exists v. \alpha < \text{card} \{qq. \mu q 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 \(\forall q. x (\rho (Suc (Suc r)) q) = v\ by blast

moreover

from stp have \(\text{step} (Suc (Suc r)) = 0\)

by (auto simp: step-def mod-Suc

118
hence $\bigwedge p \mu p' q$.  
\[ q \in \text{SHOs} (Suc (Suc r)) p \cap \text{HOs} (Suc (Suc r)) p; \]
\[ \mu p' \in \text{SHOmsgVectors Ute-M} (Suc (Suc r)) p (\rho (Suc (Suc r))) \]
\[ (\text{HOs} (Suc (Suc r)) p) (\text{SHOs} (Suc (Suc r)) p) \]
\[ \implies \mu p' q = \text{Some} (Val (x (\rho (Suc (Suc r)) q))) \]
\[ \text{by (auto simp: Ute-SHOMachine-def SHOmsgVectors-def Ute-sendMsg-def send0-def)} \]
ultimately
have $\bigwedge p \mu p' q$.  
\[ q \in \text{SHOs} (Suc (Suc r)) p \cap \text{HOs} (Suc (Suc r)) p; \]
\[ \mu p' \in \text{SHOmsgVectors Ute-M} (Suc (Suc r)) p (\rho (Suc (Suc r))) \]
\[ (\text{HOs} (Suc (Suc r)) p) (\text{SHOs} (Suc (Suc r)) p) \]
\[ \implies \mu p' q = (\text{Some} (Val v)) \]
\[ \text{by auto} \]
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** $\mu p \in \text{SHOmsgVectors Ute-M} (Suc (Suc r)) p (\rho (Suc (Suc r)))$

\[ (\text{HOs} (Suc (Suc r)) p) (\text{SHOs} (Suc (Suc r)) p) \]

**and** $\text{nxtq: nextState Ute-M r q (\rho r q) \mu (\rho (Suc r) q)}$

**and** $\text{vq: vote (\rho (Suc r) q) = Some v}$

**and** $\text{qsho: q \in \text{SHOs} (Suc r) p \cap \text{HOs} (Suc r) p}$

**shows** $\mu p q = \text{Some} (\text{Vote} (\text{Some} v))$

**proof**

\[ \text{from nxtq vq have step r = 0 by (auto simp: vote-step)} \]

**with** $\text{map qsho have \mu p q = Some (Vote (vote (\rho (Suc r) q))}$

**by (auto simp: Ute-SHOMachine-def SHOmsgVectors-def Ute-sendMsg-def step-def send1-def mod-Suc)$$

**with** $\text{vq show \mu p q = Some (Vote (Some v)) by auto}$

**qed**

We now prove the Termination property.

**theorem** ute-termination:

**assumes** $\text{run: SHORun Ute-M rho HOs SHOs}$

**and** $\text{commR: \forall r. SHOcommPerRd Ute-M (HOs r) (HOs r)}$

**and** $\text{commG: SHOcommGlobal Ute-M HOs SHOs}$

**shows** $\exists r v. \text{decide (rho r p) = Some v}$

**proof**

\[ \text{from commG} \]

**obtain** $\Phi \pi r0$

**where** $\text{rr: r0 = Suc (nSteps * \Phi)}$

\[ \text{and}\ \pi: \forall p. \pi = \text{HOs r0 p} \wedge \pi = \text{SHOs r0 p} \]

**and** $\text{t: \forall p. card (SHOs (Suc r0) p \cap \text{HOs (Suc r0) p}) > T}$

**and** $\text{e: \forall p. card (SHOs (Suc (Suc r0)) p \cap \text{HOs (Suc (Suc r0)) p}) > E}$

**by (auto simp: Ute-SHOMachine-def Ute-commGlobal-def Let-def)$$

**from** $\text{rr have stp:step r0 = Suc 0 by (auto simp: step-def)}$
obtain \( w \) where \( \forall p. (\text{vote} (\text{rho} \ (\text{Suc} \ (\text{Suc} \ r0))) \ p) = \text{Some} \ w \)

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

  have \( \exists v. \ T < \text{card} \ \{qq. \ \mu p \ qq = \text{Some} \ (\text{Val} \ v)\} \)
  proof –
    from \( t \) have \( \text{card} \ (\text{SHOs} \ (\text{Suc} \ r0) \ p) \cap \text{HOs} \ (\text{Suc} \ r0) \ p) > T \) ..
    moreover
    from run commR stp \( \pi \ rr \)
    obtain \( v \) where \( \forall q. q \in \text{SHOs} \ (\text{Suc} \ r0) \ p \cap \text{HOs} \ (\text{Suc} \ r0) \ p \)
    \( \mu p \ q = \text{Some} \ (\text{Val} \ v) \)
    using termination-argument-1 by blast

    with \( \mu p \) obtain \( v \) where \( \land p. \ \mu p \ q q \)
    \( \land q. \ q q \in \text{SHOs} \ (\text{Suc} \ r0) \ p \cap \text{HOs} \ (\text{Suc} \ r0) \ p \)
    \( \mu p q q = \text{Some} \ (\text{Val} v) \)
    by auto
    hence \( \text{SHOs} \ (\text{Suc} \ r0) \ p \cap \text{HOs} \ (\text{Suc} \ r0) \ p \ \subseteq \ \{qq. \ \mu p \ qq = \text{Some} \ (\text{Val} v)\} \)
    by auto
    hence \( \text{card} \ (\text{SHOs} \ (\text{Suc} \ r0) \ p) \cap \text{HOs} \ (\text{Suc} \ r0) \ p) \)
    \( \leq \ \text{card} \ \{qq. \ \mu p \ qq = \text{Some} \ (\text{Val} v)\} \)
    by (auto intro: card_mono)
    ultimately
    have \( \ T < \text{card} \ \{qq. \ \mu p \ qq = \text{Some} \ (\text{Val} v)\} \) by auto
    thus \( \text{thesis} \) by auto
  qed

  with \( \mu p \) show \( \exists w. (\text{vote} ((\text{rho} \ (\text{Suc} \ (\text{Suc} \ r0)))) \ p) = \text{Some} \ w \)
  by (auto simp: Ute-SHOMachine-def nextState-def Ute-nextState-def step-def mod-Suc next0-def)
  qed

then obtain \( qq \) \( w \) where \( qq w : \text{vote} ((\text{rho} \ (\text{Suc} \ (\text{Suc} \ r0)))) \ qq) = \text{Some} \ w \)
  by blast

have \( \forall pp. \text{vote} ((\text{rho} \ (\text{Suc} \ (\text{Suc} \ r0)))) \ pp) = \text{Some} \ w \)
  proof
    fix \( pp \)
    from abc obtain \( wp \) where \( wp w : \text{vote} ((\text{rho} \ (\text{Suc} \ (\text{Suc} \ r0)))) \ pp) = \text{Some} \ wp \)
  qed
by blast
from run obtain \( \mu pp \mu qq \)
where \( \text{nxtp: nextState Ute-M (Suc r0)} \) \( pp \) \( (\rho (Suc r0)) pp \)
\( \mu pp (\rho (Suc (Suc r0)) pp) \)
and \( \text{map: } \mu pp \in \text{SHOMsgVectors Ute-M (Suc r0)} \) pp \( (\rho (Suc r0)) pp \)
\( (\text{HOs (Suc r0)} pp) (\text{SHOs (Suc r0)} pp) \)
and \( \text{nxtq: nextState Ute-M (Suc r0)} \) \( qq \) \( (\rho (Suc r0)) qq \)
\( \mu qq (\rho (Suc (Suc r0)) qq) \)
and \( \text{map: } \mu qq \in \text{SHOMsgVectors Ute-M (Suc r0)} \) \( qq \) \( (\rho (Suc r0)) qq \)
\( (\text{HOs (Suc r0)} qq) (\text{SHOs (Suc r0)} qq) \)
unfolding \( \text{Ute-SHOMachine-def SHORun-eq SHOnextConfig-eq by blast} \)
from \( \text{commR this pwp qqw have wp = w} \)
by (auto dest: common-vote)
with \( \text{pwp show vote (} (\rho (Suc (Suc r0))) pp) = \text{Some w} \)
by auto
qed
with that show ?thesis by auto
qed

from run obtain \( \mu pp' \)
where \( \text{nxtp: nextState Ute-M (Suc (Suc r0))} \) \( p \) \( (\rho (Suc (Suc r0)) p) \)
\( \mu pp' (\rho (Suc (Suc (Suc r0))) p) \)
and \( \text{map': } \mu pp' \in \text{SHOMsgVectors Ute-M (Suc (Suc r0))} \) \( p \) \( (\rho (Suc (Suc r0))) \)
\( (\text{HOs (Suc (Suc r0)) p}) (\text{SHOs (Suc (Suc r0)) p}) \)
by (auto simp: \( \text{Ute-SHOMachine-def SHORun-eq SHOnextConfig-eq} \))
have \( \forall qq. \ qq \in \text{SHOs (Suc (Suc r0))} \) \( p \cap \text{HOs (Suc (Suc r0)) p} \)
\( \Longrightarrow \mu pp' qq = \text{Some (Vote (Some w))} \)
proof
fix \( \text{qq} \)
assume \( \text{qqsho: qq} \in \text{SHOs (Suc (Suc r0))} \) \( p \cap \text{HOs (Suc (Suc r0)) p} \)
from run obtain \( \mu qq \) where
\( \text{nxtqq:nextState Ute-M (Suc r0)} \) \( qq \) \( (\rho (Suc r0)) qq \)
\( \mu qq (\rho (Suc (Suc r0)) qq) \)
by (auto simp: \( \text{Ute-SHOMachine-def SHORun-eq SHOnextConfig-eq} \))
from \( \text{commR map' nxtqq votew qqsho show } \mu pp' qq = \text{Some (Vote (Some w))} \)
by (auto dest: termination-argument-2)
qed

hence \( \text{SHOs (Suc (Suc r0))} \) \( p \cap \text{HOs (Suc (Suc r0)) p} \)
\( \subseteq \{ qq. \ \mu pp' qq = \text{Some (Vote (Some w))} \} \)
by auto

hence \( \text{wsho: card (SHOs (Suc (Suc r0))} \) \( p \cap \text{HOs (Suc (Suc r0)) p} \)
\( \leq \text{card } \{ qq. \ \mu pp' qq = \text{Some (Vote (Some w))} \} \)
by (auto simp: card-mono)

from \( \text{stp} \) have \( \text{step (Suc (Suc r0)) = Suc 0} \)
unfolding \( \text{step-def by auto} \)
with \( \text{nxtp have next1 (Suc (Suc r0))} \) \( p \) \( (\rho (Suc (Suc r0)) p) \) \( \mu pp' \)
\( (\rho (Suc (Suc (Suc r0))) p) \)
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. pp' qq = Some (Vote (Some w))} > E
  by auto
ultimately
  show ?thesis by (auto simp: next1-def)
qed

8.7 \(U_{T,E,\alpha}\) Solves Weak Consensus

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

\textbf{theorem} ute-weak-consensus:
  \begin{align*}
  &\text{assumes} \run: \text{SHORun Ute-M rho HOs SHOs} \\
  &\quad \text{and} \commR: \forall r. \text{SHOcommPerRd Ute-M (HOs r) (SHOs r)} \\
  &\quad \text{and} \commG: \text{SHOcommGlobal Ute-M HOs SHOs} \\
  &\text{shows} \text{weak-consensus} \ (x \circ (\rho \ 0)) \ \text{decide} \ \rho \\
  &\text{unfolding} \text{ weak-consensus-def} \\
  &\text{using} \text{ ute-validity[OF run commR]} \\
  &\quad \text{ute-agreement[OF run commR]} \\
  &\quad \text{ute-termination[OF run commR commG]} \\
  &\text{by auto}
  \end{align*}

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

\textbf{theorem} ute-weak-consensus-fg:
  \begin{align*}
  &\text{assumes} \run: \lambda\run Ute-M rho HOs SHOs (\lambda r. \text{undefined}) \\
  &\quad \text{and} \commR: \forall r. \text{SHOcommPerRd Ute-M (HOs r) (SHOs r)} \\
  &\quad \text{and} \commG: \text{SHOcommGlobal Ute-M HOs SHOs} \\
  &\text{shows} \text{weak-consensus} \ (\lambda p. x \circ (\rho \ 0)) \ \text{decide} \ (\text{state} \circ \rho) \\
  &\quad \text{(is weak-consensus ?inits - -)} \\
  &\text{proof} \ (\text{rule local-property-reduction[OF run weak-consensus-is-local]}) \\
  &\text{fix} \ \crun \\
  &\text{assume} \crun: \text{CSHORun Ute-M crun HOs SHOs (\lambda r. \text{undefined})} \\
  &\quad \text{and} \init: \crun \ 0 = \text{state} (\rho \ 0) \\
  &\text{from} \crun \ \text{have SHORun Ute-M crun HOs SHOs by (unfold SHORun-def)} \\
  &\text{from} \ \text{this} \commR \ \text{commG} \\
  &\text{have} \text{weak-consensus} \ (x \circ (\crun \ 0)) \ \text{decide} \ \crun \\
  &\quad \text{by (rule ute-weak-consensus)} \\
  &\text{with} \ \init \ \text{show} \text{ weak-consensus ?inits decide crun} \\
  &\quad \text{by (simp add: o-def)}
  \end{align*}

qed
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 $\alpha$ 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 '\texttt{proc}' of the generic HO model.

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

abbreviation N ≡ card (UNIV::Proc set) — number of processes
\end{verbatim}

The following record models the local state of a process.

\begin{verbatim}
record 'val pstate =
  x :: 'val — current value held by process
  decide :: 'val option — value the process has decided on, if any
\end{verbatim}

The $x$ field of the initial state is unconstrained, but no decision has yet been taken.

\begin{verbatim}
definition Ate-initState where
  Ate-initState p st ≡ (decide st = None)
\end{verbatim}

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

\begin{verbatim}
locale ate-parameters =
  fixes α::nat and T::nat and E::nat
  assumes TNaE:T ≥ 2*(N + 2*α - E)
  and TltN:T < N
  and EltN:E < N
\end{verbatim}
The following are consequences of the assumptions on the parameters.

**Lemma majE**: \(2 \times (E - \alpha) \geq N\)

*using* \(TNaE \ 	ext{LltN by auto})*

**Lemma Egl\(\alpha\)**: \(E > \alpha\)

*using* \(majE \ 	ext{EltN by auto})*

**Lemma Tge2\(\alpha\)**: \(T \geq 2 \times \alpha\)

*using* \(TNaE \ 	ext{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

\[
\text{mostOftenRcvd} (\text{msgs}::\text{Proc} \Rightarrow '\text{val option}) \equiv \\
\{v. \ \forall w. \ \text{card} \{qq. \text{msgs} qq = \text{Some} w\} \leq \text{card} \{qq. \text{msgs} qq = \text{Some} v\}
\]

**Definition Ate-sendMsg** :: nat \(\Rightarrow\) Proc \(\Rightarrow\) Proc \(\Rightarrow\) 'val pstate \(\Rightarrow\) 'val

*where*

\(\text{Ate-sendMsg} r p q st \equiv x st\)

**Definition Ate-nextState** :: nat \(\Rightarrow\) Proc \(\Rightarrow\) ('val::linorder) pstate \(\Rightarrow\) (Proc \(\Rightarrow\) 'val option) \(\Rightarrow\) 'val pstate \(\Rightarrow\) bool

*where*

\(\text{Ate-nextState} r p st \text{msgs} st' \equiv \\
(\text{if card} \{q. \text{msgs} q \neq \text{None}\} > T \\
\text{then} x st' = \text{Min} (\text{mostOftenRcvd} \text{msgs}) \\
\text{else} x st' = x st) \\
\land (\exists v. \ \text{card} \{q. \text{msgs} q = \text{Some} v\} > E \land \text{decide} st' = \text{Some} v) \\
\lor \neg (\exists v. \ \text{card} \{q. \text{msgs} q = \text{Some} v\} > E) \\
\land \text{decide} st' = \text{decide} st)\)

### 9.2 Communication Predicate for \(A_{T,E,\alpha}\)

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

**Definition Ate-commPerRd** where

\(\text{Ate-commPerRd} \ 	ext{HOrs} \ 	ext{SHOrs} \equiv \\
\forall p. \ \text{card} \ (\text{HOrs} p - \text{SHOrs} p) \leq \alpha\)

The global communication predicate stipulates the three following conditions:
• for every process $p$ there are infinitely many rounds where $p$ receives more than $T$ messages,

• for every process $p$ there are infinitely many rounds where $p$ receives more than $E$ uncorrupted messages,

• and there are infinitely many rounds in which more than $E - \alpha$ processes receive uncorrupted messages from the same set of processes, which contains more than $T$ processes.

definition
Ate-commGlobal where
Ate-commGlobal HOs SHOs ≡
(\forall r. \exists r'. r' > r. \text{card}(HOs r' p) > T)
\land (\forall r. \exists r'. r' > r. \text{card}(SHOs r' p \cap HOs r' p) > E)
\land (\forall r. \exists r'. r' > r. \exists \pi_1 \pi_2.
\text{card}\pi_1 > E - \alpha
\land \text{card}\pi_2 > T
\land (\forall p \in \pi_1. HOs r' p = \pi_2 \land SHOs r' p \cap HOs r' p = \pi_2))

9.3 The $A_{T,E,\alpha}$ Heard-Of Machine

We now define the non-coordinated SHO machine for the $A_{T,E,\alpha}$ algorithm by assembling the algorithm definition and its communication-predicate.

definition Ate-SHOMachine where
Ate-SHOMachine =
(\lambda CinitState CnextState sendMsg SHOcommPerRd SHOcommGlobal.
CinitState = (\lambda p st crd. Ate-initState p (st::(\text{val::linorder}) pstate)),
sendMsg = Ate-sendMsg,
CnextState = (\lambda r p st msgs crd st'. Ate-nextState r p st msgs st'),
SHOcommPerRd = (\lambda HO r p msgs s. Ate-commPerRd:: Proc HO \Rightarrow Proc HO \Rightarrow \text{bool}),
SHOcommGlobal = Ate-commGlobal)

abbreviation
Ate-M ≡ (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) = $v$\} > $E - \alpha$

**proof**

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

- from mu have \{qq. $\mu q qq = \text{Some } v\} - (HOs r p - SHOs r p) \subseteq \{qq. sendMsg Ate-M r qq p (rho r qq) = v\}
  - (is ?vrcvd - ?ahop $\subseteq$ ?vsentp)
  - by (auto simp: SHOmsgVectors-def)

- hence card (?vrcvd - ?ahop) $\leq$ card ?vsentp
  - and card (?vrcvd - ?ahop) $\geq$ card ?vrcvd - card ?ahop
  - by (auto simp: card-mono diff-card-le-card-Diff)

- hence card ?vsentp $\geq$ card ?vrcvd - card ?ahop by auto

moreover

- from nxt nvp vp have card ?vrcvd > $E$
  - by (auto simp: Ate-SHOMachine-def nextState-def Ate-nextState-def)

moreover

- from comm have card (HOs r p - SHOs r p) $\leq \alpha$
  - by (auto simp: Ate-SHOMachine-def Ate-commPerRd-def)

ultimately

**show** ?thesis using Egta by auto

qed

If more than $E - \alpha$ processes 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) $\mu q$ ((rho (Suc r)) q)

**and** muq: $\mu q \in$ SHOmsgVectors Ate-M r q (rho r) (HOs r q) (SHOs r q)

**and** vsent: card \{qq. sendMsg Ate-M r qq q (rho r qq) = $v$\} > $E - \alpha$
  - (is card ?vsent $>$ -)

**shows** card \{qq. $\mu q qq \neq \text{Some } v\} \cap HOs r q\} $\leq N + 2*\alpha - E$

**proof**

- from nxt muq have \{qq. $\mu q qq \neq \text{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 $\subseteq$ ?notsent)
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) \( \mu_q ((\rho (Suc r)) q) \)

**and** muq: \( \mu_q \in \text{SHOmsgVectors Ate-M r q (rho r q)} \) (SHOs r q)

**and** threshold-T: card \( \{qq. \mu_q qq \neq \text{None}\} \geq T \)

**and** threshold-E: card \( \{qq. \text{sendMsg Ate-M r qq q (rho r qq)} = v\} > E - \alpha \)

**shows** mostOftenRcvd \( \mu_q = \{v\} \)

**proof** –

**from** muq **have** hodef:HOs r q = \( \{qq. \mu_q qq \neq \text{None}\} \)

**unfolding** SHOmsgVectors-def **by** auto

**from** nxt muq threshold-E **have** card \( \{qq. \mu_q qq \neq \text{Some v}\} \cap \text{HOs r q} \leq N + 2*\alpha - E \)

(is card \( \text{?heardnotv} \leq \cdot \cdot \cdot \))

**by** (rule other-values-received)

**moreover**

**have** card \( \text{?heardnotv} \geq T + 1 - \text{card} \{qq. \mu_q qq = \text{Some v}\} \)

**proof** –

**from** muq

**have** \( \text{?heardnotv} = (\text{HOs r q}) - \{qq. \mu_q qq = \text{Some v}\} \)

**and** \( \{qq. \mu_q qq = \text{Some v}\} \subseteq \text{HOs r q} \)

**unfolding** SHOmsgVectors-def **by** auto

**hence** card \( \text{?heardnotv} = \text{card} (\text{HOs r q}) - \text{card} \{qq. \mu_q qq = \text{Some v}\} \)

**by** (auto simp: card-Diff-subset)

**with** hodef threshold-T **show** ?thesis **by** auto

**qed**

ultimately

**have** card \( \{qq. \mu_q qq = \text{Some v}\} > \text{card} \text{?heardnotv} \)

**using** TNaE **by** auto
moreover

\{ 
  \begin{align*}
    & \text{fix } w \\
    & \text{assume } w: w \neq v \\
    & \text{with } hodef \ \text{have } \{ qq, \mu q qq = \text{Some } w \} \subseteq \text{heardnot} v \ \text{by auto} \\
    & \text{hence } \text{card } \{ qq, \mu q qq = \text{Some } w \} \leq \text{card } \text{heardnot} v \ \text{by } (\text{auto simp: card-mono}) \\
  \end{align*}
\}

ultimately

\begin{align*}
  \begin{alignat*}{3}
    & \text{have } \{ w, \text{card } \{ qq, \mu q qq = \text{Some } w \} \geq \text{card } \{ qq, \mu q qq = \text{Some } v \} \} = \{ v \} \\
    & \text{by force} \\
    & \text{thus } \text{thesis unfolding } \text{mostOftenRcvd-def } \text{by auto} \\
  \end{alignat*}
\end{align*}

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.

\textbf{lemma} common-x-induct:

\begin{align*}
  \text{assumes } & \text{ran: SHORun Ate-M rho HOs SHOs} \\
  \text{and } & \text{comm: SHOcommPerRd Ate-M (HOs (r+k)) (SHOs (r+k))} \\
  \text{and } & \text{ih: card } \{ qq, x \ (\rho \ (r + k) \ qq) = v \} > E - \alpha \\
  \text{shows } & \text{card } \{ qq, x \ (\rho \ (r + \text{Suc } k) \ qq) = v \} > E - \alpha \\
\end{align*}

\textbf{proof} – \\

\textbf{from } \text{ih} \\

\begin{align*}
  \text{have } \text{thrE}: \forall pp. \text{card } \{ qq, \text{sendMsg Ate-M} \ (r + k) \ qq pp (\rho \ (r + k) \ qq) = v \} \\
  & > E - \alpha \\
  \text{by } (\text{auto simp: Ate-SHOMachine-def Ate-sendMsg-def})
\end{align*}

\{ 
  \begin{align*}
    & \text{fix } qq \\
    & \text{assume } \text{kv:x} (\rho \ (r + k) \ qq) = v \\
    & \text{from } \text{run obtain } \mu qq \\
    & \text{where } \text{nxt: nextState Ate-M} \ (r + k) \ qq (\rho \ (r + k) \ qq) \ \mu qq ((\rho \ (\text{Suc } (r + k))) \ qq) \\
    & \text{and } \mu qq: \mu qq \in \text{SHOmsgVectors Ate-M} \ (r + k) \ qq (\rho \ (r + k)) \ (HOs (r + k) \ qq) \ (SHOs (r + k) \ qq) \\
    & \text{by } (\text{auto simp: SHORun-eq SHOnextConfig-eq})
  \end{align*}
\}

\begin{align*}
  \text{have } x \ (\rho \ (r + \text{Suc } k) \ qq) = v \\
  \text{proof (cases card } \{ pp, \mu qq pp \neq \text{None} \} > T) \\
  \text{case } \text{True} \\
  & \text{with } \text{comm nxt } \mu qq \text{ thrE } \text{have } \text{mostOftenRcvd } \mu qq = \{ v \} \\
  & \text{by } (\text{auto dest: mostOftenRcvd-v}) \\
  \text{with } \text{nxt True show } x \ (\rho \ (r + \text{Suc } k) \ qq) = v \\
  & \text{by } (\text{auto simp: Ate-SHOMachine-def nextState-def Ate-nextState-def}) \\
  \text{next} \\
  \text{case } \text{False} \\
  & \text{with } \text{nxt } \text{have } x \ (\rho \ (r + \text{Suc } k) \ qq) = x \ (\rho \ (r + k) \ qq) \\
  & \text{by } (\text{auto simp: Ate-SHOMachine-def nextState-def Ate-nextState-def}) \\
  & \text{with } \text{kv show } x \ (\rho \ (r + \text{Suc } k) \ qq) = v \text{ by simp} \\
  \text{qed}
\end{align*}
\[
\begin{align*}
\text{hence } \{\text{qq. } x (\rho (r + k)) \text{ qq} = v\} \subseteq \{\text{qq. } x (\rho (r + \text{Suc } k)) \text{ qq} = v\} \\
\text{by auto}
\end{align*}
\]

\[
\begin{align*}
\text{hence } \text{card } \{\text{qq. } x (\rho (r + k)) \text{ qq} = v\} \leq \text{card } \{\text{qq. } x (\rho (r + \text{Suc } k)) \text{ qq} = v\} \\
\text{by (auto simp: card-mono)}
\end{align*}
\]

\[
\begin{align*}
\text{with } \text{ih show } \text{thesis by auto}
\end{align*}
\]

\text{qed}

Whenever some process newly decides value \(v\), then any process that updates its \(x\) variable will set it to \(v\).

\text{lemma common-x:}

\text{assumes run: SHORun Ate-M \rho \text{HOs SHOs}}

\text{and \text{comm: } \forall r. SHOcommPerRd (Ate-M ::(Proc, 'val::linorder pstate, 'val) SHOMachine)}

\[
\begin{align*}
(\text{HOs } r) (\text{SHOs } r)
\end{align*}
\]

\text{and d1: } \text{decide } (\rho r p) \neq \text{Some } v

\text{and d2: } \text{decide } (\rho (\text{Suc } r) p) = \text{Some } v

\text{and qupdate: } x (\rho (r + \text{Suc } k) q) \neq x (\rho (r + k) q)

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

\text{proof –}

\text{from \text{comm}}

\text{have SHOcommPerRd (Ate-M ::(Proc, 'val::linorder pstate, 'val) SHOMachine)}

\[
\begin{align*}
(\text{HOs } (r+k)) (\text{SHOs } (r+k)) ..
\end{align*}
\]

\text{moreover}

\text{from \text{run obtain } \mu q}

\text{where nxt: nextState Ate-M (r+k) q (\rho (r+k) q) \mu q (\rho (r + \text{Suc } k) q)}

\text{and muq: } \mu q \in \text{SHOmsgVectors Ate-M (r+k) q (\rho (r+k))}

\[
\begin{align*}
(\text{HOs } (r+k)) (\text{SHOs } (r+k) q)
\end{align*}
\]

\text{by (auto simp: SHORun-eq SHOnextConfig-eq)}

\text{moreover}

\text{from \text{nxt qupdate}}

\text{have threshold-T: } \text{card } \{\text{qq. } \mu q \text{ qq} \neq \text{None}\} > T

\text{and xsmall: } x (\rho (r + \text{Suc } k) q) = \text{Min (mostOftenRcvd } \mu q)

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

\text{moreover}

\text{have } E - \alpha < \text{card } \{\text{qq. } x (\rho (r + k) q) \text{ qq} = v\}

\text{proof (induct } k)

\text{from \text{run comm d1 d2}}

\text{have } E - \alpha < \text{card } \{\text{qq. } \text{sendMsg Ate-M } r \text{ qq } p (\rho \text{ r qq}) = v\}

\text{by (auto dest: decide-sent-msgs-threshold)}

\text{thus } E - \alpha < \text{card } \{\text{qq. } x (\rho (r + 0) q) \text{ qq} = v\}

\text{by (auto simp: Ate-SHOMachine-def Ate-sendMsg-def)}

\text{next}

\text{fix } k

\text{assume } E - \alpha < \text{card } \{\text{qq. } x (\rho (r + k) q) \text{ qq} = v\}

\text{with \text{run comm show } E - \alpha < \text{card } \{\text{qq. } x (\rho (r + \text{Suc } k) q) \text{ qq} = v\}}

\text{by (auto dest: common-x-induct)}

\text{qed}
with \( \text{run} \)

\[
E - \alpha < \text{card} \{ qq, \text{sendMsg} \ Ate-M (r+k) \ qq \ qq (\rho (r+k) \ qq) = v \}
\]

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

qed

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

lemma decisionNonNullThenDecided:
- assumes \( \text{run}: \text{SHORun \ Ate-M \ rho \ HOs \ SHOs} \)
- and \( \text{dec: decide (rho n p) = Some v} \)
- obtains \( m \) where \( m < n \)
  - and \( \text{decide (rho m p) \neq Some v} \)
  - and \( \text{decide (rho (Suc m) p) = Some v} \)

proof –

let \( \? \text{dec} k = \text{decide (rho k p)} \)

have \( (\forall m < n. \ ? \text{dec (Suc m) \neq Some v} \longrightarrow \ ? \text{dec (Suc m) \neq Some v}) \longrightarrow \ ? \text{dec n \neq Some v} \)

(is \( ?P n \) is \( ?A n \longrightarrow - \))

proof (induct n)

from \( \text{run} \) show \( ?P 0 \)

by (auto simp: Ate-SHOMachine-def SHORun-eq HOinitConfig-eq

initState-def Ate-initState-def)

next

fix \( n \)

assume \( \text{ih}: \ ?P n \) thus \( ?P (Suc n) \) by force

qed

with \( \text{dec} \) that show \( ? \text{thesis} \) 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 \( \text{run}: \text{SHORun \ Ate-M \ rho \ HOs \ SHOs} \)

and \( \text{comm: \forall r. \ SHOcommPerRd \ Ate-M (HOs r) (SHOs r)} \)

and \( \text{initv: \forall q. \ x (rho 0 q) = v} \)

and \( \text{dp: decide (rho r p) = Some w} \)

shows \( w = v \)

proof –

\{

\{ 

fix \( r \)

have \( \forall qq. \ \text{sendMsg} \ Ate-M r qq p (\rho r qq) = v \)

proof (induct \( r \))

\{ 

from \( \text{run \ initv} \) show \( \forall qq. \ \text{sendMsg} \ Ate-M 0 qq p (\rho 0 qq) = v \)

by (auto simp: SHORun-eq SHOnextConfig-eq Ate-SHOMachine-def Ate-sendMsg-def)

\}

130
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 µ
  where nxt: nextState Ate-M r qq (rho r qq) µqq (rho (Suc r) qq)
  and µq: µqq ∈ SHOmsgVectors Ate-M r qq (rho r qq) (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
do not return a plain text representation of the image
qed

moreover
have \{pp. µqq pp ≠ None\} = \{pp. µqq pp = Some v\} ∪ \{pp. ∃w. w ≠ v ∧ µqq pp = Some w\}
and \{pp. µqq pp = Some v\} ∩ \{pp. ∃w. w ≠ v ∧ µqq pp = Some w\} =
{} by auto

hence \text{card}\ \{pp. µqq pp ≠ None\} = \text{card}\ \{pp. µqq pp = Some v\} + \text{card}\ \{pp. ∃w. w ≠ v ∧ µqq pp = Some w\}

by (auto simp: card-Un-Int

moreover
note threshold-T

ultimately
have \text{card}\ \{pp. µqq pp = Some v\} > \text{card}\ \{pp. ∃w. w ≠ v ∧ µqq pp = Some w\}

by auto

moreover
\{ fix w
assume w ≠ v
hence \{pp. µqq pp = Some w\} ⊆ \{pp. ∃w. w ≠ v ∧ µqq pp = Some w\}
by auto

hence \text{card}\ \{pp. µqq pp = Some w\} ≤ \text{card}\ \{pp. ∃w. w ≠ v ∧ µqq pp = Some w\}
= \text{Some w}\}
by (auto simp: card-mono)

} ultimately

have \text{zz:} w ≠ v \implies
\text{card}\ \{pp. µqq pp = Some v\} < \text{card}\ \{pp. µqq pp = Some v\}

by force

hence \text{zz:} w, \text{card}\ \{pp. µqq pp = Some v\} ≤ \text{card}\ \{pp. µqq pp = Some v\}

⇒ w = v

by force

with \text{zz:} have mostOftenRcvd µqq = \{v\}
by (force simp: mostOftenRcvd-def)

with \text{xsmall show } x (\text{rho} \ (Suc\ r)\ qq) = v by auto

qed

thus ∀ qq. \text{sendMsg Ate-M} (Suc r) qq p (\text{rho} (Suc r) qq) = v
by (auto simp: Ate-SHOMachine-def Ate-sendMsg-def)

qed

} note \text{P} = this

from \text{run dp obtain} rp

where \text{rp:} rp < r decide (\text{rho} rp p) ≠ Some w
decide (\text{rho} (Suc rp) p) = Some w

by (rule decisionNonNullThenDecided)
from run obtain \( \mu p \)
where nxt: \( \text{nxtState Ate-M \( \rho p \) (\( \rho \) (Suc \( \rho \)) \( p \))} \)
and \( \text{mu: } \mu p \in \text{SHOmsgVectors Ate-M \( \rho p \) (\( \rho \) (HOs \( \rho p \)) (SHOs \( \rho p \)))} \)
by (auto simp: SHORun-eq SHOnextConfig-eq)

{ 
fix \( w \)
assume \( w: w \neq v \)
from comm have \( \text{card (HOs \( \rho p \) - SHOs \( \rho p \))} \leq \alpha \)
by (auto simp: Ate-SHOMachine-def Ate-commPerRd-def)
moreover
from mu P
have \( \text{SHOs \( \rho p \) \cap HOs \( \rho p \)} \subseteq \{pp. \mu p pp = \text{Some} v\} \)
and \( \text{HOs \( \rho p \)} = \{pp. \mu p pp \neq \text{None}\} \)
by (auto simp: SHOmsgVectors-def)

hence \( \{pp. \mu p pp \neq \text{None}\} - \{pp. \mu p pp = \text{Some} v\} \)
\subseteq \( \text{HOs \( \rho p \) - SHOs \( \rho p \)} \)
by auto

hence \( \text{card (\{pp. \mu p pp \neq \text{None}\} - \{pp. \mu p pp = \text{Some} v\})} \)
\leq \( \text{card (HOs \( \rho p \) - SHOs \( \rho p \))} \)
by (auto simp: card-mono)

ultimately
have \( \text{card (\{pp. \mu p pp \neq \text{None}\} - \{pp. \mu p pp = \text{Some} v\})} < E \)
using Egta by auto
moreover
from \( w \) have \( \{pp. \mu p pp = \text{Some} w\} \)
\subseteq \( \{pp. \mu p pp \neq \text{None}\} - \{pp. \mu p pp = \text{Some} v\} \)
by auto

hence \( \text{card (\{pp. \mu p pp = \text{Some} w\}} \)
\leq \( \text{card (\{pp. \mu p pp \neq \text{None}\} - \{pp. \mu p pp = \text{Some} v\})} \)
by (auto simp: card-mono)

ultimately
have \( \text{card (\{pp. \mu p pp = \text{Some} w\}} < E \) by simp
}

hence \( \text{PP: } \forall w. \text{ card (\{pp. \mu p pp = \text{Some} w\}} \geq E \Rightarrow w = v \) by force

from \( \rho p \) \( \text{nxt mu} \)

have \( \text{card (\{q. \mu p q = \text{Some} w\} > E} \)
by (auto simp: SHOmsgVectors-def Ate-SHOMachine-def
nextState-def Ate-nextState-def)

with \( \text{PP} \)
show \( ?\text{thesis} \) 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: \( \text{SHORun Ate-M \( \rho \) HOs SHOs} \)
and comm: \( \text{SHOcommPerRd Ate-M (HOs \( r \)) (SHOs \( r \))} \)
and \( \text{nvp}: \text{decide}\ (\rho\ r\ p) \neq \text{Some}\ v \)
and \( \text{vp}: \text{decide}\ (\rho\ (\text{Suc}\ r)\ p) = \text{Some}\ v \)
and \( \text{nwp}: \text{decide}\ (\rho\ r\ q) \neq \text{Some}\ w \)
and \( \text{wp}: \text{decide}\ (\rho\ (\text{Suc}\ r)\ q) = \text{Some}\ w \)
shows \( w = v \)

**proof**

have \( gtn: \text{card}\ \{\text{qq}.\ \text{sendMsg}\ \text{Ate-M}\ r\ \text{qq}\ p\ (\rho\ r\ \text{qq}) = v\} \)
+ \( \text{card}\ \{\text{qq}.\ \text{sendMsg}\ \text{Ate-M}\ r\ \text{qq}\ q\ (\rho\ r\ \text{qq}) = w\} > N \)

**proof**

from \( \text{run comm nvp vp} \)
have \( \text{card}\ \{\text{qq}.\ \text{sendMsg}\ \text{Ate-M}\ r\ \text{qq}\ p\ (\rho\ r\ \text{qq}) = v\} > E - \alpha \)
by \((\text{rule decide-sent-msgs-threshold})\)

moreover
from \( \text{run comm nwp wq} \)
have \( \text{card}\ \{\text{qq}.\ \text{sendMsg}\ \text{Ate-M}\ r\ \text{qq}\ q\ (\rho\ r\ \text{qq}) = w\} > E - \alpha \)
by \((\text{rule decide-sent-msgs-threshold})\)

ultimately
show \(?\text{thesis using}\ \text{majE by auto}\)
qed

show \(?\text{thesis}\)

**proof** \((\text{rule ccontr})\)

assume \( \text{ww:w} \neq v \)

have \( \forall\ \text{qq}.\ \text{sendMsg}\ \text{Ate-M}\ r\ \text{qq}\ p\ (\rho\ r\ \text{qq}) = \text{sendMsg}\ \text{Ate-M}\ r\ \text{qq}\ q\ (\rho\ r\ \text{qq}) \)
by \((\text{auto simp: Ate-SHOMachine-def Ate-sendMsg-def})\)

with \( \text{ww} \)
have \( \{\text{qq}.\ \text{sendMsg}\ \text{Ate-M}\ r\ \text{qq}\ p\ (\rho\ r\ \text{qq}) = v\} \)
\( \cap\ \{\text{qq}.\ \text{sendMsg}\ \text{Ate-M}\ r\ \text{qq}\ q\ (\rho\ r\ \text{qq}) = w\} = \{\} \)
by \(\text{auto}\)

with \( gtn \)

have \( \text{card}\ \{\text{qq}.\ \text{sendMsg}\ \text{Ate-M}\ r\ \text{qq}\ p\ (\rho\ r\ \text{qq}) = v\} \)
\( \cup\ \{\text{qq}.\ \text{sendMsg}\ \text{Ate-M}\ r\ \text{qq}\ q\ (\rho\ r\ \text{qq}) = w\} > N \)
by \((\text{auto simp: card-Un-Int})\)

moreover

have \( \text{card}\ \{\text{qq}.\ \text{sendMsg}\ \text{Ate-M}\ r\ \text{qq}\ p\ (\rho\ r\ \text{qq}) = v\} \)
\( \cup\ \{\text{qq}.\ \text{sendMsg}\ \text{Ate-M}\ r\ \text{qq}\ q\ (\rho\ r\ \text{qq}) = w\} \leq N \)
by \((\text{auto simp: card-mono})\)

ultimately

show \(\text{False by auto}\)
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** \( \text{laterProcessDecidesSameValue} : \)

assumes \( \text{run: SHORun Ate-M rho HOs SHOs} \)
and \( \text{comm:}\ \forall\ r.\ \text{SHOcommPerRd Ate-M}\ (\text{HOs}\ r)\ (\text{SHOs}\ r) \)
and \( \text{ndl1:}\ \text{decide}\ (\rho\ r\ p) \neq \text{Some}\ v \)
and $d1$: decide $(\rho \ (\text{Suc} \ r) \ p) = \text{Some} \ v$

and $nd2$: decide $(\rho \ (r+k) \ q) \neq \text{Some} \ w$

and $d2$: decide $(\rho \ (\text{Suc} \ (r+k)) \ q) = \text{Some} \ w$

shows $w = v$

proof (rule ccontr)

assume $vdifw: w \neq v$

have $kgt0: k > 0$

proof (rule ccontr)

assume $\neg \ k > 0$

hence $k = 0$ by auto

with run comm $nd1 \ nd1 \ nd2 \ d2$ have $w = v$

by (auto dest: common-decision)

with $vdifw$ show False ..

qed

have 1: $\{qq. \ \text{sendMsg} \ At\text{-M} \ r \ qq \ (\rho \ r \ qq) = v\}

\cap \ \{qq. \ \text{sendMsg} \ At\text{-M} \ (r+k) \ qq \ (\rho \ (r+k) \ qq) = w\} = \{

(is ?sentv \ \text{and} \ ?sentw = \{\})

proof (rule ccontr)

assume $\neg \ ?thesis$

then obtain $qq$

where $xrv: x (\rho \ r \ qq) = v$ and $rkw: x (\rho \ (r+k) \ qq) = w$

by (auto simp: $At\text{-SHOMachine-def} \ At\text{-sendMsg-def}$)

have $\exists k' < k. \ x (\rho \ (r + k') \ qq) \neq w \ \text{and} \ x (\rho \ (r + \text{Suc} \ k') \ qq) = w$

proof (rule ccontr)

assume $f: \neg \ ?thesis$

\{

fix $k'$

assume $kk': k' < k$ hence $x (\rho \ (r + k') \ qq) \neq w$

proof (induct $k'$)

from $xrv \ vdifw$

show $x (\rho \ (r + 0) \ qq) \neq w$ by simp

next

fix $k'$

assume $ih: k' < k \Rightarrow x (\rho \ (r + k') \ qq) \neq w$

and $ksk': \text{Suc} \ k' < k$

from $ksk'$ have $k' < k$ by simp

with $ih \ f$ show $x (\rho \ (r + \text{Suc} \ k') \ qq) \neq w$ by auto

qed

\}

with $f$ have $\forall k' < k. \ x (\rho \ (r + \text{Suc} \ k') \ qq) \neq w$ by auto

moreover

from $kgt0$ have $k - 1 < k$ and $kk: \text{Suc} \ (k - 1) = k$ by auto

ultimately

have $x (\rho \ (r + \text{Suc} \ (k - 1)) \ qq) \neq w$ by blast

with $rkw \ kk$ show False by simp

qed

then obtain $k'$

where $k' < k$
and \( w: x \cdot (\rho (r + Suc k') qq) = w \)
and \( qquad: x \cdot (\rho (r + Suc k') qq) \neq x \cdot (\rho (r + k') qq) \)
by auto
from run comm nd1 d1 qquad
have \( x \cdot (\rho (r + Suc k') qq) = v \) by (rule common-x)
with \( w vdifw \) show False by simp
qed
from run comm nd1 d1 have \( sentv: \text{card } ?sentv > E - \alpha \)
by (auto dest: decide-sent-msgs-threshold)
from run comm nd2 d2 have \( card ?sentw > E - \alpha \)
by (auto dest: decide-sent-msgs-threshold)
with \( sentv majE \) have \( \text{card } ?sentv + \text{card } ?sentw > N \)
by simp
with \( 1 vdifw \) have \( 2: \text{card } (?sentv \cup ?sentw) > N \)
by (auto simp: card-Un-Int)
have \( \text{card } (?sentv \cup ?sentw) \leq N \)
by (auto simp: card-mono)
with \( 2 \) show False by simp
qed
The Agreement property is now an immediate consequence.

theorem ate-agreement:
\begin{itemize}
\item assumes \( \text{run: } SHO\text{run } Ate-M \rho HOs SHOs \)
\item and \( \text{comm: } \forall r. \text{SHOcommPerRd } Ate-M \ (HOS r) \ (SHOs r) \)
\item and \( p: \text{decide } (\rho m p) = \text{Some } v \)
\item and \( q: \text{decide } (\rho n q) = \text{Some } w \)
\item shows \( w = v \)
\end{itemize}
proof –
from run p obtain \( k \) where
\( k: k < m \) decide \( (\rho k p) \neq \text{Some } v \) decide \( (\rho (Suc k) p) = \text{Some } v \)
by (rule decisionNonNullThenDecided)
from run q obtain \( l \) where
\( l: l < n \) decide \( (\rho l q) \neq \text{Some } w \) decide \( (\rho (Suc l) q) = \text{Some } w \)
by (rule decisionNonNullThenDecided)
show \( \text{thesis} \)
proof (cases \( k \leq l \))
case \( \text{True} \)
then obtain \( i \) where \( l = k+i \) by (auto simp add: le-iff-add)
with run comm \( k \) \( l \) show \( \text{thesis} \)
by (auto dest: laterProcessDecidesSameValue)
next
case \( \text{False} \)
hence \( l \leq k \) by simp
then obtain \( i \) where \( m: k = l+i \) by (auto simp add: le-iff-add)
with run comm \( k \) \( l \) show \( \text{thesis} \)
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** at-termination:

**assumes** run: SHORun Ate-M rho HOs SHOs

and commR: \( \forall r. (SHOcommPerRd:((\text{Proc, 'val::linorder pstate, 'val}) SHOMachine) \Rightarrow (\text{Proc HO}) \Rightarrow \text{bool}) \)

and commG: SHOcommGlobal Ate-M HOs SHOs

**shows** \( \exists r v. \text{decide (rho r p) = Some v} \)

**proof**

from commG obtain \( r' \pi_1 \pi_2 \)

where \( \pi_1: \text{card} \pi_1 > E - \alpha \)

and \( \pi_2: \text{card} \pi_2 > T \)

and hosho: \( \forall p \in \pi_1. (HOs r' p = \pi_2 \land HOs r' p \cap HOs r' p = \pi_2) \)

by (auto simp: Ate-SHOMachine-def Ate-commGlobal-def)

obtain \( v \) where

\( P1: \forall pp. \text{card} \{qq. \text{sendMsg Ate-M} (Suc r') qq pp (rho (Suc r') qq) = v\} > E - \alpha \)

**proof**

from run obtain \( \mu p \)

where \( \text{nxtp: nextState Ate-M r' p (rho r' p) \mu p (rho (Suc r') p) \land mup: } \mu p \in \text{SHOmsgVectors Ate-M} r' p (rho r') (HOs r' p) (SHOs r' p) \)

by (auto simp: SHORun-eq SHOnextConfig-eq)

from run obtain \( \mu q \)

where \( \text{nxhq: nextState Ate-M r' q (rho r' q) \mu q (rho (Suc r') q) \land muq: } \mu q \in \text{SHOmsgVectors Ate-M} r' q (rho r') (HOs r' q) (SHOs r' q) \)

by (auto simp: SHORun-eq SHOnextConfig-eq)

from mup muq p q

have \( \{qq. \mu q qq \neq \text{None}\} = HOs r' q \)

and \( 2: \{qq. \mu q qq = \text{Some (sendMsg Ate-M} r' qq q (rho r' qq)\}\) \supset HOs r' q \cap HOs r' q

and \( \{qq. \mu p qq \neq \text{None}\} = HOs r' p \)

and \( 4: \{qq. \mu p qq = \text{Some (sendMsg Ate-M} r' qq p (rho r' qq)\}\) \supset HOs r' p \cap HOs r' p

by (auto simp: SHOmsgVectors-def)

with \( p q \text{ hosho} \)
have \( aa: \pi 2 = \{ qq. \mu p qq \neq \text{None} \} \)
and \( cc: \pi 2 = \{ qq. \mu p qq \neq \text{None} \} \) by auto
from \( p q \) hosho 2
have \( bb: \{ qq. \mu p qq = \text{Some} (\text{sendMsg Ate-M r'} qq q (rho r' qq )) \} \supseteq \pi 2 \)
by auto
from \( p q \) hosho 4
have \( dd: \{ qq. \mu p qq = \text{Some} (\text{sendMsg Ate-M r'} qq p (rho r' qq )) \} \supseteq \pi 2 \)
by auto
have \( \text{Min} (\text{mostOftenRcvd } \mu p) = \text{Min} (\text{mostOftenRcvd } \mu q) \)
proof
− have \( \forall qq. \text{sendMsg Ate-M} r' qq p (rho r' qq) = \text{sendMsg Ate-M} r' qq q (rho r' qq) \)
by (auto simp: Ate-SHOMachine-def Ate-sendMsg-def)
with \( aa bb cc dd \)
have \( \forall qq. \mu p qq \neq \text{None} \rightarrow \mu p qq = \mu q qq \)
by force
moreover
from \( aa bb cc dd \)
have \( \{ qq. \mu p qq \neq \text{None} \} = \{ qq. \mu q qq \neq \text{None} \} \) by auto
hence \( \forall qq. \mu p qq = \text{None} \leftrightarrow \mu q qq = \text{None} \) by blast
hence \( \forall qq. \mu p qq = \text{None} \rightarrow \mu p qq = \mu q qq \) by auto
ultimately
have \( \forall qq. \mu p qq = \mu q qq \) by blast
thus \(?thesis by (auto simp: mostOftenRcvd-def)\)
qed
with \( \pi t aa nxtq \pi t cc nxtp \)
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 \( \forall p \in \pi 1. \text{sendMsg Ate-M} (Suc r') p pp (rho (Suc r') p) = v \)
by (auto simp: Ate-SHOMachine-def Ate-sendMsg-def)
hence \( \text{card } \pi 1 \leq \text{card} \{ qq. \text{sendMsg Ate-M} (Suc r') qq pp (rho (Suc r') qq) = v \} \)
by (auto intro: card-mono)
with \( \pi e a \)
have \( E - \alpha < \text{card} \{ qq. \text{sendMsg Ate-M} (Suc r') qq pp (rho (Suc r') qq) = v \} \)
by simp
} with that show \(?thesis by blast\)
qed

{ fix \( k pp \)
have \( E - \alpha < \text{card} \{ qq. \text{sendMsg Ate-M} (Suc r' + k) qq pp (rho (Suc r' + k) qq) = v \} \)
(is \(?P k\)
proof (induct k)
  from P1 show ?P 0 by simp
next
  fix k
  assume ih: ?P k
  from commR
  have (SHOcommPerRd::((Proc, 'val::linorder pstate, 'val) SHOMachine) ⇒ (Proc HO) ⇒ (Proc HO) ⇒ bool)
    Ate-M (HOs (Suc r' + k)) (SHOs (Suc r' + k)) ..
moreover
  from ih have E - α < card {qq. x (rho (Suc r' + k) qq) = v}
    by (auto simp: Ate-SHOMachine-def Ate-sendMsg-def)
ultimately
  have E - α < card {qq. x (rho (Suc r' + Suc k) qq) = v}
    by (rule common-x-induct[OF run])
thus ?P (Suc k)
  by (auto simp: Ate-SHOMachine-def Ate-sendMsg-def)
qed

note P2 = this

{ 
  fix k pp
  assume ppupdateq: 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 ppupdateq
  have threshold-T: card {qq. µpp qq ≠ None} > T
    and zsmall: 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}
    by (auto simp: mostOftenRcvd-v)
ultimately
  have mostOftenRcvd µpp = {v} by (auto dest!: mostOftenRcvd-v)
  with zsmall
  have x (rho (Suc r' + Suc k) pp) = v by simp
have \( P4' \forall pp. \exists k. (\rho (Suc r' + Suc k) pp) = v \)
proof
fix \( pp \)
from \( \text{commG} \) have \( \exists r''. r' > r'. \text{card} (HOs r'' pp) > T \)
by (auto simp: \( \text{Ate-SHOMachine-def} \) \( \text{Ate-commGlobal-def} \))
then obtain \( k \) where \( Suc r' + k > r' \) and \( \text{t:card} (HOs (Suc r' + k) pp) > T \)
by (auto dest: \( \text{less-imp-Suc-add} \))
moreover
from \( \text{run} \) obtain \( \mu pp \)
where \( \text{nxt: nextState Ate-M} (Suc r' + k) pp (\rho (Suc r' + k) pp) \mu pp \)
and \( \mu: \mu pp \in \text{SHOmsgVectors Ate-M} (Suc r' + k) pp (\rho (Suc r' + k)) \)
\( (HOs (Suc r' + k) pp) (SHOs (Suc r' + k) pp) \)
by (auto simp: \( \text{SHORun-eq} \) \( \text{SHOnextConfig-eq} \))
moreover
have \( x (\rho (Suc r' + Suc k) pp) = v \)
proof 
from \( \text{commR} \) have \( (\text{SHOcommPerRd::} ((\text{Proc}, 'val::linorder pstate, 'val::linorder) \text{SHOMachine}) \)
\( \Rightarrow (\text{Proc HO}) \Rightarrow (\text{Proc HO}) \Rightarrow \text{bool} \)
\( \text{Ate-M} (HOs (Suc r' + k)) (SHOs (Suc r' + k)) .. \)
moreover
from \( \mu \) have \( HOs (Suc r' + k) pp = \{ q, \mu pp q \neq \text{None} \} \)
by (auto simp: \( \text{SHOmsgVectors-def} \))
with \( \text{nxt t} \)
have \( \text{threshold-T: card} \{ q. \mu pp q \neq \text{None} \} > T \)
and \( \text{xsmall: \( x (\rho (Suc r' + Suc k) pp) = \text{Min} (\text{mostOftenRcvd} \mu pp) \) \)
by (auto simp: \( \text{Ate-SHOMachine-def} \) \( \text{nextState-def Ate-nextState-def} \))
moreover
from \( P2 \)
have \( E - \alpha < \text{card} \{ qq, \text{sendMsg Ate-M} (Suc r' + k) qq pp (\rho (Suc r' + k) qq) = v \} . \)
ultimately
have \( \text{mostOftenRcvd} \mu pp = \{ v \} \)
using \( \text{nxt} \mu t \) by (auto dest!: \( \text{mostOftenRcvd-v} \))
with \( \text{xsmall show ?thesis by auto} \)
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 \( x (\rho \?rr pp) = v \) 

140
by blast
have \( \forall k. x (\rho (\bar{rr} + k) \ pp) = v \)
proof
  fix \( k \)
  show \( x (\rho (\bar{rr} + k) \ pp) = v \)
  proof (induct \( k \))
    from \( x \) show \( x (\rho (\bar{rr} + 0) \ pp) = v \) by simp
  next
    fix \( k \)
    assume ih: \( x (\rho (\bar{rr} + k) \ pp) = v \)
    obtain \( k' \) where \( \text{Suc } k' + k' = \bar{rr} + k \) by auto
    show \( x (\rho (\bar{rr} + \text{Suc } k) \ pp) = v \)
    proof (rule ccontr)
      assume \( \neg v \)
      with \( k' \) ih have \( x (\rho (\text{Suc } k' + \text{Suc } k) \ pp) \neq v \) by (rule P3)
      with \( k' \) \( \neg v \) show False by (simp add: ac-simps)
    qed
    qed
  qed
thus \( \exists \bar{rr}. \forall k. x (\rho (\bar{rr} + k) \ pp) = v \) by blast
qed

from \( P5a \) have \( \exists F. \forall pp k. x (\rho (F \ pp + k) \ pp) = v \) by (rule choice)
then obtain \( R': \forall pp k. x (\rho (R \ pp + k) \ pp) = v \)
  where \( \text{imgR}: R' \cdot (UNIV::\text{Proc set}) \neq {} \)
  and \( R': \forall pp k. x (\rho (R \ pp + k) \ pp) = v \)
  by blast
def \( \bar{rr} \equiv \text{Max } (R' \cdot UNIV) \)

have \( P5: \forall r' > \bar{rr}. \forall pp. x (\rho r' \ pp) = v \)
proof (clarify)
  fix \( r' \ pp \)
  assume \( r': r' > \bar{rr} \)
  hence \( r' > R \ pp \) by (auto simp: \( \bar{rr}-\text{def} \))
  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 \( \text{commG} \) have \( \exists r' > \bar{rr}. \text{card } (\text{SHOs r'} \ p \cap \text{HOs r'} \ p) > E \)
  by (auto simp: Ate-SHOMachine-def Ate-commGlobal-def)
with \( P5 \) obtain \( r' \)
  where \( r' > \bar{rr} \)
  and \( \text{card } (\text{SHOs r'} \ p \cap \text{HOs r'} \ p) > E \)
  and \( \forall pp. \text{sendMsg Ate-M r' pp } (\rho r' \ pp) = v \)
  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: µp ∈ SHOmsgVectors Ate-M r’ p (rho r’ p) (HOs r’ p) (SHOs r’ p)
by (auto simp: SHORun-eq SHOnextConfig-eq)
from mu
have card (SHOs r’ p ∩ HOs r’ p)
  ≤ card {q. µp 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. µp q = Some v} > E by auto
with nxt show ?thesis
  by (auto simp: Ate-SHOMachine-def nextState-def Ate-nextState-def)
qed

9.8 $A_{T,E,α}$ Solves Weak Consensus

Summing up, all (coarse-grained) runs of $A_{T,E,α}$ 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: ∀ r. SHOcommPerRd Ate-M (HOs r) (SHOs r)
and commG: SHOcommGlobal Ate-M HOs SHOs
shows weak-consensus ($x ◦ (rho 0)$) 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 (λr q. undefined)
and commR: ∀ r. SHOcommPerRd Ate-M (HOs r) (SHOs r)
and commG: SHOcommGlobal Ate-M HOs SHOs
shows weak-consensus ($λp. x$ (state (rho 0)) decide (state ◦ rho))
in weak-consensus ?inits - -
proof (rule local-property-reduction[OF run weak-consensus-is-local!])
fix crun
assume crun: CSHORun Ate-M crun HOs SHOs (λr 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 ($x ◦ (crun 0)$) decide crun
by (rule ate-weak-consensus)
with init show weak-consensus ?inits decide crun
by (simp add: o-def)
qed
10 Verification of the \textit{EIGByz}_f Consensus Algorithm

Lynch [12] presents \textit{EIGByz}_f, a version of the \textit{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 \( \texttt{proc} \) of the generic HO model.

\begin{verbatim}
typedecl Proc \text{ -- the set of processes}
axiomatization where Proc-finite: OFCLASS(Proc, finite-class)
instance Proc :: finite by (rule Proc-finite)

abbreviation
  \( N \equiv \text{card (UNIV::Proc set)} \) \text{ -- number of processes}
\end{verbatim}

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

\begin{verbatim}
axiomatization \( f::\text{nat} \)
where \( f: f < N \)
\end{verbatim}

10.1 Tree Data Structure

The algorithm relies on propagating information about the initially proposed values among all the processes. This information is stored in trees whose branches are labeled by lists of (distinct) processes. For example, the interpretation of an entry \([p,q] \mapsto \text{Some } v\) is that the current process heard from process \( q \) that it had heard from process \( p \) that its proposed value is \( v \). The value initially proposed by the process itself is stored at the root of the tree.

We introduce the type of \textit{labels}, which encapsulate lists of distinct process identifiers and whose length is at most \( f+1 \).

\begin{verbatim}
definition Label = \{xs::Proc list. length xs \leq Suc f \land distinct xs\}
typedef Label = Label
  by (auto simp: Label-def intro: exI[where x= []]) \text{ -- the empty list is a label}
\end{verbatim}

There is a finite number of different labels.

\begin{verbatim}
lemma finite-Label: finite Label
\end{verbatim}
proof

have Label ⊆ \{xs. set xs ⊆ (UNIV::Proc set) ∧ length xs ≤ Suc f\}
  by (auto simp: Label-def)
moreover
have finite \{xs. set xs ⊆ (UNIV::Proc set) ∧ length xs ≤ Suc f\}
  by (rule finite-lists-length-le) auto
ultimately
show ?thesis by (auto elim: finite-subset)
qed

lemma finite-UNIV-Label: finite (UNIV::Label set)
proof
  from finite-Label have finite (Abs-Label ` Label) by simp
moreover
  { fix l::Label
    have l ∈ Abs-Label ` Label
      by (rule Abs-Label-cases) auto
  }
hence (UNIV::Label set) = (Abs-Label ` Label) by auto
ultimately show ?thesis by simp
qed

lemma finite-Label-set [iff]: finite (S :: Label set)
  using finite-UNIV-Label by (auto intro: finite-subset)

Utility functions on labels.

definition root-node where
  root-node ≡ Abs-Label []

definition length-lbl where
  length-lbl l ≡ length (Rep-Label l)

lemma length-lbl [intro]: length-lbl l ≤ Suc f
  unfolding length-lbl-def using Label-def Rep-Label by auto

definition is-leaf where
  is-leaf l ≡ length-lbl l = Suc f

definition last-lbl where
  last-lbl l ≡ last (Rep-Label l)

definition butlast-lbl where
  butlast-lbl l ≡ Abs-Label (butlast (Rep-Label l))

definition set-lbl where
  set-lbl l = set (Rep-Label l)

The children of a non-leaf label are all possible extensions of that label.
**10.2 Model of the Algorithm**

The following record models the local state of a process.

```haskell
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

```haskell
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

```haskell
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

```haskell
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

```haskell
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 (λp st msgs st'. st' = st)

10.3 Communication Predicate for EIGByz

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

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**

\[
EIG\text{-commPerRd HO SHO} \equiv \text{card} (SKr HO SHO) > (N + f) \div 2
\]

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**

\[
EIG\text{-commGlobal HOs SHOs} \equiv \text{card} (SK HOs SHOs) \geq N - f
\]

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 \(EIG\text{-commGlobal}\) is satisfied. The standard correctness proof assumes that \(N > 3f\), and therefore \(N - f > (N + f) \div 2\). Since moreover, for any \(r\), we obviously have

\[
\left( \bigcap_{p \in \Pi, r' \in \mathbb{N}} \text{SHO}(p, r') \right) \subseteq \left( \bigcap_{p \in \Pi} \text{SHO}(p, r) \right),
\]

it follows that any execution of \(EIGByz_f\) where \(N > 3f\) also satisfies \(EIG\text{-commPerRd}\) at any round. The standard correctness hypotheses thus imply our communication predicates.

However, our proof shows that \(EIGByz_f\) 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 \(EIG\text{-commPerRd}\) always holds. Also, \(EIG\text{-commGlobal}\) is satisfied because there are three processes from which every process receives the correct messages at all rounds. By our correctness proof, it follows that \(EIGByz_f\) 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 assem-bling the algorithm definition and its communication-predicate.

**definition EIG-SHOMachine where**

\[
EIG\text{-SHOMachine} = \emptyset
\]

\[
\text{CinitState} = (\lambda p \ st \ \text{crd. EIG-initState p st}),
\]

147
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)
  end

  148
qed

  by (auto simp: Label-def elim: distinct-butlast)
qed

The label of a child is well-formed.

lemma Rep-Label-append:
  assumes l: ¬(is-leaf l)
  shows (Rep-Label l [] @ p ∈ Label) = (p ∉ set-lbl l)
    (is ?lhs = ?rhs is (?l' ∈ _) = _)
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' ≤ 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 ∈ children l
  shows ∃p. p ∉ set-lbl l ∧ Rep-Label c = Rep-Label l [] @ p
proof -
  from c obtain p
    where p: p ∉ set-lbl l and l: ¬(is-leaf l)
    and c: c = Abs-Label (Rep-Label l [] @ p)
    by (auto simp: children-def)
    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 ∈ children h
  shows length-lbl l = Suc (length-lbl h)
  using label-children[of l] by (auto simp: length-lbl-def)

The root node is never a child.

lemma children-not-root:
  assumes root-node ∈ children l
  shows P
by (auto simp: root-node-def)

The label of a child with the last element removed is the label of the parent.

lemma children-butlast-lbl:
assumes c ∈ children l
shows butlast-lbl c = l
using label-children[OF assms]
by (auto simp: butlast-lbl_def Rep-Label-inverse)

The root node is not a child, and it is the only such node.

lemma root-iff-no-child: (l = root-node) = (∀ l’. l /∈ children l’)
proof
  assume l = root-node
  thus ∀ l’. l /∈ children l’ by (auto elim: children-not-root)
next
  assume rhs: ∀ l’. l /∈ children l’
  show l = root-node proof (rule rev-exhaust[of Rep-Label l])
    assume Rep-Label l = []
    hence Rep-Label l = Rep-Label root-node by simp
    thus ?thesis by (simp only: Rep-Label-inject)
  next
    fix l’ q
    assume l’: Rep-Label l = l’ @ [q]
    let ?l’ = Abs-Label l’
    from Rep-Label[of l] l’ have l’ ∈ Label by (simp add: Label-def)
    hence repl’: Rep-Label ?l’ = l’ by (rule Abs-Label-inverse)
    from Rep-Label[of l] l’ have l’ @ [q] ∈ Label by (simp add: Label-def)
    with l’ have Rep-Label l = Rep-Label (Abs-Label (l’ @ [q]))
    by (simp add: Abs-Label-inverse)
    hence l = Abs-Label (l’ @ [q]) by (simp add: Rep-Label-inject)
  moreover
  from Rep-Label[of l] l’ have length l’ < Suc f q /∈ set l’
  by (auto simp: Label-def)
  moreover
  note repl’
  ultimately have l ∈ children ?l’
  by (auto simp: children-def is-leaf-def length-lbl-def set-lbl-def)
  with rhs show ?thesis by blast
qed
qed

If some label l is not a leaf, then the set of processes that appear at the end
of the labels of its children is the set of all processes that do not appear in l.

lemma children-last-set:
assumes l: ¬(is-leaf l)
shows last-lbl ‘ (children l) = UNIV − set-lbl l
proof
  show last-lbl '(children l) ⊆ UNIV − set-lbl l
    by (auto dest: label-children simp: last-lbl-def)
next
  show UNIV − set-lbl l ⊆ last-lbl '(children l)
proof (auto simp: image-def)
    fix p
    assume p: p ∉ set-lbl l
    with l have c: Abs-Label (Rep-Label l @ [p]) ∈ children l
      by (auto simp: children-def)
    with Rep-Label-append[OF l] p
    show ∃c ∈ 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 ∈ children l and c': c' ∈ 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 ¬(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
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 \( \text{run: SHORun EIG-M \rho HOs SHOs} \)

and \( l': l' \in \text{children } l \)

and \( \text{shop: last-lbl } l' \in \text{SHOs (length-lbl } l \) \( p \cap \text{HOs (length-lbl } l \) \( p \)

\((\text{is } ?q \in \text{SHOs (} ?\text{len } l \) \( p \cap \text{-})\))

shows \( \text{vals (rho (} ?\text{len } l' \) \( p \) \( l' = \text{vals (rho (} ?\text{len } l \) \( ?q \) \( l \)

proof –

let \( ?r = ?\text{len } l \)

from \( \text{run obtain } \mu p \)

where \( \text{nxt: nextState EIG-M } ?r p (\text{rho } ?r p \) \( \mu p (\text{rho (Suc } ?r \) \) \( p \)) \)

and \( \text{mu: } \mu p \in \text{SHOmsgVectors EIG-M } ?r p (\text{rho } ?r \) \( \text{(HOs } ?r p \) \( \text{(SHOs } ?r p \)

by \( \text{(auto simp: EIG-SHOMachine-def SHORun-eq SHOnextConfig-eq)} \)

with \( \text{shop} \)

have \( \text{mst: } \mu p ?q = \text{Some (vals (rho } ?r ?q)) \)

by \( \text{(auto simp: EIG-SHOMachine-def EIG-sendMsg-def SHOmsgVectors-def)} \)

from \( \text{nxt length-lbl[} \text{of } l' \text{] children-length[} \text{OF } l' \text{]} \)

have \( \text{extend-vals } ?r p (\text{rho } ?r p \) \( \mu p (\text{rho (Suc } ?r \) \) \( p \)) \)

by \( \text{(auto simp: EIG-SHOMachine-def nextState-def EIG-nextState-def next-main-def next-end-def)} \)

with \( \text{mst } l' \text{ show } \text{?thesis} \)

by \( \text{(auto simp: extend-vals-def children-length children-butlast-lbl)} \)

qed

A process fixes the value \( \text{vals } l \) of a label at state \( \text{length-lbl } l \), and then never modifies the value.

**Lemma keep vals:**

assumes \( \text{run: SHORun EIG-M \rho HOs SHOs} \)

shows \( \text{vals (rho (length-lbl } l + n \) \( p \) \( l = \text{vals (rho (length-lbl } l \) \( p \) \( l \)

\((\text{is } ?v n = ?vl)\)) \)

proof \( \text{(induct } n \text{)} \)

show \( ?v 0 = ?vl \) by simp

next

fix \( n \)

assume \( \text{ih: } ?v n = ?vl \)

let \( ?r = \text{length-lbl } l + n \)

from \( \text{run obtain } \mu p \)

where \( \text{nxt: nextState EIG-M } ?r p (\text{rho } ?r p \) \( \mu p (\text{rho (Suc } ?r \) \) \( p \)) \)

by \( \text{(auto simp: EIG-SHOMachine-def SHORun-eq SHOnextConfig-eq)} \)

with \( \text{ih show } ?v (\text{Suc } n) = ?vl \)

by \( \text{(auto simp: EIG-SHOMachine-def nextState-def EIG-nextState-def next-main-def next-end-def extend-vals-def)} \)
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 \( l': l' \in \text{children } l \)
and skr: last-lbl \( l' \in SKr (HOs (length-lbl l)) (SHOs (length-lbl l)) \)
shows vals (rho (length-lbl l') \( p \)) \( l' = \text{vals (rho (length-lbl l') } q \) \( l' \)
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 \( l: l \in \text{children } t \)
and skrl: last-lbl \( l \in SKr (HOs (length-lbl t)) (SHOs (length-lbl t)) \)
and \( l': l' \in \text{children } l \)
and skrl':last-lbl \( l' \in SKr (HOs (length-lbl l)) (SHOs (length-lbl l)) \)
shows vals (rho (length-lbl l') \( p \)) \( l' = \text{vals (rho (length-lbl l') } p \) \( l' \)
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 \( \text{length-lbl } l \).

**Lemma lynch-6-16-c:**

**Assumes:** commR: EIG-commPerRd (HOs (length-lbl l)) (SHOs (length-lbl l))
(is EIG-commPerRd (HOs ?r) -)
and \( l: \neg (\text{is-leaf } l) \)
shows card \{ \( l' \in \text{children } l \). last-lbl \( l' \in SKr (HOs ?r) (SHOs ?r) \}
> card (\text{children } l) \text{ div } 2
(is card ?lhs > -)
**Proof** –
let \( ?skr = SKr (HOs ?r) (SHOs ?r) \)
have last-lbl \( ?\text{lhs } ?\text{lhs } = ?\text{skr } \text{-set-lbl } l \)
**Proof**
from children-last-set[OF \( l \)]
show last-lbl \( ?\text{lhs } ?\text{lhs } \subseteq ?\text{skr } \text{-set-lbl } l \)
by (auto simp: children-length)
**Next**
\{ fix \( p \)
assume \( p: p \in ?\text{skr } p \notin \text{-set-lbl } l \)
with children-last-set[OF \( l \)]

qed
have \( p \in \text{last-lbl} \cdot \text{children} \ l \) by auto
with \( p \) have \( p \in \text{last-lbl} \cdot ?\text{lhs} \)
  by (auto simp: image-def children-length)
}

thus \( ?\text{skr} - \text{set-lbl} \ l \subseteq \text{last-lbl} \cdot ?\text{lhs} \) by auto
qed

moreover
from last-lbl-inj-on-children[of \ l]
have inj-on last-lbl ?\text{lhs} by (auto simp: inj-on-def)

ultimately
have \( \text{card} \ ?\text{lhs} = \text{card} \ (\text{skr} - \text{set-lbl} \ l) \) by (auto dest: card-image)

using \( \text{card} \ ?\text{lhs} \geq \text{card} \ (\text{skr} - \text{r}) \) by simp

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

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

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

lemma \( \text{newvals-skr-uniform} \):
assumes run: \( \text{SHORun} \ EIG-M \ \rho \ \text{HOs} \ \text{SHOs} \)
  and commR: \( \text{EIG-commPerRd} \ (\text{HOs} \ (\text{length-lbl} \ l)) \ (\text{SHOs} \ (\text{length-lbl} \ l)) \)
  (is EIG-commPerRd (HOs ?r) -)
and notleaf: \( \neg(\text{is-leaf} \ l) \)
and unif: \( \forall l'. \ l' \in \text{children} \ l; \)
  \( \text{last-lbl} \ l' \in \text{SKr} \ (\text{HOs} \ (\text{length-lbl} \ l)) \ (\text{SHOs} \ (\text{length-lbl} \ l)) \)
\( \implies \text{newvals} \ (\rho \ (\text{Suc} \ f) \ p) \ l' = v \)
shows \( \text{newvals} \ (\rho \ (\text{Suc} \ f) \ p) \ l = v \)

proof
  from unif
  have \( \text{card} \ \{l' \in \text{children} \ l. \ \text{last-lbl} \ l' \in \text{SKr} \ (\text{HOs} \ ?r) \ (\text{SHOs} \ ?r)\} \leq \text{card} \ \{l' \in \text{children} \ l. \ \text{newvals} \ (\rho \ (\text{Suc} \ f) \ p) \ l' = v\} \)
    by (auto intro: card-mono)
  with lynch-6-16-c[of HOs \ l \ SHOs, OF \ commR \ notleaf]
  have maj: \( \text{has-majority} \ v \ (\text{newvals} \ (\rho \ (\text{Suc} \ f) \ p)) \ (\text{children} \ l) \)
    by (simp add: has-majority-def)

  from run have \( \text{check-newvals} \ (\rho \ (\text{Suc} \ f) \ p) \)
A node whose label \( l \) ends with a process which is well heard at round \( \text{length-lbl } l \) will have its \( \text{newvals} \) field set (at round \( f + 1 \)) to the “fixed-up” value given by \( \text{vals} \).

**Lemma Lynch-6-16-d:**

**Assumes**
- \( \text{run} : \text{SHORun} \ EIG-M \ \rho \ \text{HOs} \ \text{SHOs} \)
- \( \text{commR} : \forall r. \ EIG-\text{commPerRd} (\text{HOs } r) (\text{SHOs } r) \)
- \( \text{notroot} : l \in \text{children } t \)
- \( \text{skr} : \text{last-lbl } l \in \text{SKr} (\text{HOs } (\text{length-lbl } t)) (\text{SHOs } (\text{length-lbl } t)) \)

**Shows**
- \( \text{newvals} (\rho (\text{Suc } f) p) l = \text{fixupval} (\text{vals} (\rho (\text{Suc } f) p) p) l \)
- \( \text{is } ?P l \)

**Using**
- \( \text{notroot} \text{ skr} \)

**Proof**

\[ \text{induct } \text{Suc } f - (?\text{len } l) \text{ arbitrary; } l \ t \]

**Fix** \( l \ t \)

**Assume**
- \( 0 = \text{Suc } f - ?\text{len } l \)

**With**
- \( \text{length-lbl}[?l] \)

**Have**
- \( \text{leaf} : \text{is-leaf } l \)

**By**
- \( \text{simp add: is-leaf-def} \)

**From**
- \( \text{run} \)
- \( \text{have} \)
- \( \text{check-newvals} (\rho (\text{Suc } f) p) \)

**By**
- \( \text{auto simp: EIG-SHOMachine-def SHORun-eq SHOnextConfig-eq nextState-def EIG-nextState-def next-end-def} \)

**With**
- \( \text{leaf} \)

**Show**
- \( ?P l \)

**By**
- \( \text{auto simp: check-newvals-def is-leaf-def} \)

**Next**
**Fix** \( k \ l \ t \)

**Assume**
- \( \forall l' t'. \)
- \( k = \text{Suc } f - \text{length-lbl } l'; l' \in \text{children } t'; \)
- \( \text{last-lbl } l' \in \text{SKr} (\text{HOs } (\text{length-lbl } t')) (\text{SHOs } (\text{length-lbl } t')) \)
- \( \Longrightarrow ?P l' \)
- \( \text{and flk} : \text{Suc } k = \text{Suc } f - ?\text{len } l \)

**And**
- \( \text{notroot} : l \in \text{children } t \)

**And**
- \( \text{skr} : \text{last-lbl } l \in \text{SKr} (\text{HOs } (\text{length-lbl } t)) (\text{SHOs } (\text{length-lbl } t)) \)

**Let**
- \( ?v = \text{fixupval} (\text{vals} (\rho (\text{Suc } f) p p) l) \)

**From**
- \( \text{flk} \)
- \( \text{have notlf} : \neg (\text{is-leaf } l) \)

**By**
- \( \text{simp add: is-leaf-def} \)

\[ \]
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 sh)
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-sk.r-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 \( l' \colon l \in \text{subtrees } l' \) and \( l'' \colon l \in \text{subtrees } l'' \)
shows \( l' \in \text{subtrees } l'' \lor l'' \in \text{subtrees } l' \)
using assms proof (auto simp: subtrees-def append-eq-append-conv2)
  fix \( xs \)
  assume Rep-Label \( l' @ xs = \text{Rep-Label } l' \)
  hence Rep-Label \( l' = \text{Rep-Label } l'' @ xs \) by (rule sym)
  thus \( \exists ys. \text{Rep-Label } l' = \text{Rep-Label } l'' @ ys .. \)
qed

lemma subtrees-cases:
  assumes \( l' \colon l' \in \text{subtrees } l \)
  and self \( \colon l' = l \implies P \)
  and child \( \colon \forall c. \[ c \in \text{children } l ; l' \in \text{subtrees } c \] \implies P \)
shows \( P \)
proof -
  from \( l' \) obtain \( t \) where \( t \colon \text{Rep-Label } l' = (\text{Rep-Label } l) @ t \)
  by (auto simp: subtrees-def)
  have \( l' = l \lor (\exists c \in \text{children } l, l' \in \text{subtrees } c) \)
  proof (cases \( t \))
    assume \( t = [] \)
    with \( t \) show \( ?thesis \) by (simp add: Rep-Label-inject)
  next
    fix \( p \) \( t' \)
    assume \( \text{cons } t = p \neq t' \)
    from \( \text{Rep-Label}[of } l' \) \( t \) have \( \text{length } (\text{Rep-Label } l @ t) \leq \text{Suc } f \)
    by (simp add: Label-def)
    with \( \text{cons } t \) have notleaf \( : \neg(\text{is-leaf } l) \)
    by (auto simp: is-leaf-def length-lbl-def)
    let \( \forall c = \text{Abs-Label } (\text{Rep-Label } l @ [p]) \)
    from \( \text{cons } t \text{Rep-Label}[of } l' \) have \( p \notin \text{set-lbl } l \)
    by (auto simp: Label-def set-lbl-def)
    with \( \text{notleaf } c \) \( : \forall c \in \text{children } l \)
    by (auto simp: children-def)
    moreover
    from \( \text{notleaf } p \) have \( \text{Rep-Label } l @ [p] \in \text{Label} \)
    by (simp add: Rep-Label-append)
    hence \( \text{Rep-Label } ?c = (\text{Rep-Label } l @ [p]) \)
    by (simp add: Abs-Label-inverse)
    with \( \text{cons } t \) have \( l' \in \text{subtrees } ?c \)
    by (auto simp: subtrees-def)
    ultimately show \( ?thesis \) by blast
  qed
thus \( ?thesis \) by (auto elim!: self child)
qed

lemma subtrees-leaf:
  assumes \( l \colon \text{is-leaf } l \) and \( l' \colon l' \in \text{subtrees } l \)
shows \( l' = l \)

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

lemma children-subtrees-equal:
  assumes $c: c \in \text{children } l$ and $c': c' \in \text{children } l$
  and $s: c' \in \text{subtrees } c$
  shows $c' = c$
proof —
  from assms have $\text{Rep-Label } c' = \text{Rep-Label } c$
  by (auto simp: subtrees-def dest: label-children)
  thus ?thesis by (simp add: Rep-Label-inject)
qed

A set $C$ of labels is a subcovering w.r.t. label $l$ if for all leaf subtrees $s$ of $l$
there exists some label $h \in C$ such that $s$ is a subtree of $h$ and $h$ is a subtree
of $l$.

definition subcovering where
  subcovering $C$ $l$ \equiv \forall s \in \text{subtrees } l. \text{is-leaf } s \rightarrow (\exists h \in C. h \in \text{subtrees } l \land s \in \text{subtrees } h)$

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

abbreviation covering where
  covering $C$ \equiv 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 $\rho$ HO$s$ SHOs
  and $\forall r. \text{EIG-commPerRd } \text{HO} s r$ (SHOs $r$)
  and $l \in \text{children } t$
  and last-lbl $l \in \text{SK} \text{HO}s$ (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 $f=\text{fixupval}$])

lemma lynch-6-18-b:
  assumes run: SHORun EIG-M $\rho$ HO$s$ SHOs
  and commG: EIG-commGlobal HO$s$ SHOs
  and commR: $\forall r. \text{EIG-commPerRd } \text{HO}s$ $r$ (SHOs $r$)
  shows covering \{ $l. \exists t. l \in \text{children } t \land \text{last-lbl } l \in (\text{SK } \text{HO}s \text{SHOs})$\}
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 \( N < \text{card} (SK \text{ HOs SHOs}) + \text{card} (\text{set-lbl } l) \)
by (simp add: EIG-commGlobal-def)
hence \( \exists q \in \text{set-lbl } l . q \in SK \text{ HOs SHOs} \)
by (auto dest: majorities-intersect)
then obtain \( l1 \ q \ l2 \) where
\( l : \text{Rep-Label } l = (l1 \ @ [q]) \ @ \ l2 \) and \( q \in SK \text{ HOs SHOs} \)
unfolding set-lbl-def by (auto intro: split-list-propE)

let \( ?h = \text{Abs-Label} (l1 \ @ [q]) \)
from Rep-Label[of \( l \)] \( l \) have \( l1 \ @ [q] \in \text{Label} \) by (simp add: Label-def)
hence length-lbl \( ?h \neq 0 \) by (simp add: length-lbl-inverse)
hence \( ?h \neq \text{root-node} \) by auto
then obtain \( t \) where \( t : ?h \in \text{children } t \)
by (auto simp: root-iff-no-child)
moreover
from reph \( q \) have last-lbl \( ?h \in SK \text{ HOs SHOs} \) by (simp add: last-lbl-def)
moreover
from reph \( l \) have \( l \in \text{subtrees } ?h \) by (simp add: subtrees-def)
ultimately
show \( \exists h . (\exists t . h \in \text{children } t) \land \text{last-lbl } h \in SK \text{ HOs SHOs} \land l \in \text{subtrees } h \)
by blast
qed

If \( C \) covers the subtree rooted at label \( l \) and if \( l \notin C \) then \( C \) also covers subtrees rooted at \( l \)'s children.

lemma lynch-6-19-a:
assumes cov: subcovering \( C \ l \)
and \( l : l \notin C \)
and \( e : e \in \text{children } l \)
shows subcovering \( C \ e \)
proof (clarsimp simp: subcovering-def)
fix \( s \)
assume \( s : s \in \text{subtrees } e \) and leaf: is-leaf \( s \)
from s children-in-subtree[OF \( e \)] have \( s \in \text{subtrees } l \)
by (rule subtrees-trans)
with leaf cov obtain \( h \) where \( h : h \in C \ h \in \text{subtrees } l \ s \in \text{subtrees } h \)
by (auto simp: subcovering-def)
with \( l \) obtain \( e' \) where \( e' : e' \in \text{children } l \ h \in \text{subtrees } e' \)
by (auto elim: subtrees-cases)
from \( s \in \text{subtrees } h \) \( h \in \text{subtrees } e' \) have \( s \in \text{subtrees } e' \)
by (rule subtrees-trans)
with \( s \) have \( e \in \text{subtrees } e' \lor e' \in \text{subtrees } e \)
by (rule subtrees-tree)
with \( e e' \) have \( e' = e \)
by (auto dest: children-subtrees-equal)
with \( e' \) show \( \exists h \in C . h \in \text{subtrees } e \land s \in \text{subtrees } h \) 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: \( \forall l' \in C. \text{common} \; \rho l' \)

**shows** common \( \rho l \)

**using** cov

**proof** (induct \( \text{Suc} \; f - \text{length-lbl} \; l \) arbitrary: \( l \))

**fix** \( l \)

**assume** \( 0: 0 = \text{Suc} \; f - \text{length-lbl} \; l \)

and \( C: \text{subcovering} \; C \; l \)

from \( 0 \; \text{length-lbl}[\text{of} \; l] \) have is-leaf \( l \)

by (simp add: is-leaf-def)

with \( C \)

obtain \( h \) where \( h: h \in C \; h \in \text{subtrees} \; l \; l \in \text{subtrees} \; h \)

by (auto simp: subcovering-def)

hence \( l \in C \) by (auto dest: subtrees-antisym)

with com

show common \( \rho l \) ..

**next**

**fix** \( k \; l \)

**assume** \( k: \text{Suc} \; k = \text{Suc} \; f - \text{length-lbl} \; l \)

and \( C: \text{subcovering} \; C \; l \)

and \( \text{ih}: \forall l'. \left[ k = \text{Suc} \; f - \text{length-lbl} \; l'; \text{subcovering} \; C \; l' \right] \implies \text{common} \; \rho \; l' \)

**show** common \( \rho l \)

**proof** (cases \( l \in C \))

**case** True

with com

show \( ?\text{thesis} \) ..

**next**

**case** False

with \( C \) have \( \forall e \in \text{children} \; l. \; \text{subcovering} \; C \; e \)

by (blast intro: lynch-6-19-a)

**moreover**

from \( k \) have \( \forall e \in \text{children} \; l. \; k = \text{Suc} \; f - \text{length-lbl} \; e \)

by (auto simp: children-length)

ultimately

have com-ch: \( \forall e \in \text{children} \; l. \; \text{common} \; \rho \; e \)

by (blast intro: ih)

**show** \( ?\text{thesis} \)

**proof** (clarsimp simp: common-def)

**fix** \( p \; q \)

from \( k \) have notleaf: \( \neg(\text{is-leaf} \; l) \) by (simp add: is-leaf-def)

let \( ?r = \text{Suc} \; f \)

from com-ch

have \( \forall e \in \text{children} \; l. \; \text{newvals} \; (\rho \; ?r \; p) \; e = \text{newvals} \; (\rho \; ?r \; q) \; e \)

by (auto simp: common-def)

**hence** \( \forall w. \left[ e \in \text{children} \; l. \; \text{newvals} \; (\rho \; ?r \; p) \; e = w \right] \)

= \{ e \in \text{children} \; l. \; \text{newvals} \; (\rho \; ?r \; q) \; e = w \}

by auto

moreover
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)
10.7 Proof of Agreement, Validity, and Termination

The Agreement property is an immediate consequence of lemma lynch-6-20.

**Theorem Agreement:**

**Assumes**
- \( \text{run}: \text{SHORun} \ EIG-M \rho HOs \ SHOs \)
- \( \text{commG}: \text{EIG-commGlobal} \ HOs \ SHOs \)
- \( \text{commR}: \forall r. \ EIG-commPerRd \ (HOs \ r) \ (SHOs \ r) \)
- \( p: \text{decide} \ (\rho m p) = \text{Some} \ v \)
- \( q: \text{decide} \ (\rho n q) = \text{Some} \ w \)

**Shows** \( v = w \)

**Using** \( p \ q \ \text{lynch-6-20}[\text{OF} \ \text{run} \ \text{commG} \ \text{commR}] \)

**By** (auto simp: \text{decide}[\text{OF} \ \text{run} \ \text{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**
- \( \text{run}: \text{SHORun} \ EIG-M \rho HOs \ SHOs \)
- \( \text{commR}: \forall r. \ EIG-commPerRd \ (HOs \ r) \ (SHOs \ r) \)
- \( \text{initv}: \forall q. \ \text{the} \ (\text{vals} \ (\rho 0 q) \ \text{root-node}) = v \)
- \( dp: \text{decide} \ (\rho r p) = \text{Some} \ w \)

**Shows** \( v = w \)

**Proof**

- **Have** \( v: \forall q. \ \text{vals} \ (\rho 0 q) \ \text{root-node} = \text{Some} \ v \)
- **Proof**
  - **Fix** \( q \)
  - **From** \( \text{run} \ \text{have} \ \text{vals} \ (\rho 0 q) \ \text{root-node} \neq \text{None} \)
    - **By** (auto simp: \text{EIG-SHOMachine-def} \text{SHORun-eq} \text{HOinitConfig-eq}
      \text{initState-def} \text{EIG-initState-def})
  - **Then obtain** \( w \) where \( w: \text{vals} \ (\rho 0 q) \ \text{root-node} = \text{Some} \ w \)
    - **By** auto
  - **From** \( \text{initv} \ \text{have} \ \text{the} \ (\text{vals} \ (\rho 0 q) \ \text{root-node}) = v \)
  - **With** \( w \) **Show** \( \text{vals} \ (\rho 0 q) \ \text{root-node} = \text{Some} \ v \) **By** simp

162
qed

let $?len = length-lbl
let $?r = Succ 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 $\text{EIGByz}_f$.

**Theorem** Termination:
- assumes $\text{SHORun EIG-M rho HOs SHOs}$
- shows $\exists r \cdot \text{decide} (rho r p) = \text{Some v}$
- using assms by (auto simp: decide)

### 10.8 $\text{EIGByz}_f$ Solves Weak Consensus

Summing up, all (coarse-grained) runs of $\text{EIGByz}_f$ for HO and SHO collections that satisfy the communication predicate satisfy the Weak Consensus property.

**Theorem** $\text{eig-weak-consensus}$:
- assumes run: $\text{SHORun EIG-M rho HOs SHOs}$
  and commR: $\forall r. \text{EIG-commPerRd} (\text{HOs } r) (\text{SHOs } r)$
  and commG: $\text{EIG-commGlobal} \text{HOs SHOs}$
- shows $\text{weak-consensus} (\lambda p. \text{the } (\text{vals } (\text{rho } \emptyset p) \text{root-node})) \text{ decide rho}$
- unfolding $\text{weak-consensus-def}$
- using $\text{Validity}[\text{OF run commR}]$
  $\text{Agreement}[\text{OF run commG commR}]$
  $\text{Termination}[\text{OF run}]$
- 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 (λ r q. undefined)

and commR: ∀ r. EIG-commPerRd (HOs r) (SHOs r)

and commG: EIG-commGlobal HOs SHOs

**Shows** weak-consensus (λp. the (vals (state (rho 0) p) root-node))

decide (state o rho)

(is weak-consensus ?inits - -)

**Proof** (rule local-property-reduction[OF run weak-consensus-is-local])

fix crun

assume crun: CSHORun EIG-M crun HOs SHOs (λ 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 (λp. 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

**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


